A Coarse-Grained Parallel QR-Factorization Algorithm for Sparse Least Squares Problems

Tz. Ostromsky* P. C. Hansen† Z. Zlatev‡

Abstract

A sparse QR-factorization algorithm SPARQR for coarse-grained parallel computations is described. The coefficient matrix, which is assumed to be general sparse, is reordered in an attempt to bring as many zero elements in the lower left corner as possible. The reordered matrix is then partitioned into block rows, and Givens plane rotations are applied in each block-row. These are independent tasks and can be done in parallel. Row and column permutations are carried out within the diagonal blocks in an attempt to preserve better the sparsity of the matrix.

The algorithm can be used for solving least squares problems either directly or combined with an iterative method (preconditioned conjugate gradients are used). Small non-zero elements can optionally be dropped in the latter case. This leads to a better preservation of the sparsity and, therefore, to a faster factorization. The price which has to be paid is some loss of accuracy. The iterative method is used to regain the accuracy lost during the factorization.

Numerical results from several experiments with matrices from the well-known Harwell-Boeing collection as well as with some larger sparse matrices are presented in this work. An SGI Power Challenge computer with 16 processors has been used in the experiments.

Keywords: coarse-grained parallelism, least squares problem, QR-factorization, general sparse matrix, drop-tolerance, reordering, partitioning, block algorithm.

*Purdue University, Department of Computer Science,
West Lafayette, IN 47907, USA
e-mail: tto@cs.purdue.edu
†Institute of Mathematical Modelling, Technical University of Denmark,
Bldg. 305, DK-2800 Lyngby, Denmark
e-mail: pch@imm.dtu.dk
‡National Environmental Research Institute, Frederiksborgvej 399, DK-4000 Roskilde, Denmark
e-mail: luzz@sun2.dmu.dk

1 Introduction

The least squares problem is defined as follows: given the matrix $A \in \mathcal{R}^{m\times n}$ with $m \geq n$ and the right-hand side vector $b \in \mathcal{R}^m$, find $\bar{x} \in \mathcal{R}^n$ that minimizes the Euclidean norm of the *residual vector* $r = b - Ax$:

$$\|b - A\bar{x}\|_2 = \min_{x\in\mathcal{R}^n} \|b - Ax\|_2 \tag{1}$$

or equivalently, solve the system of linear algebraic equations

$$Ax = b - r \quad \wedge \quad A^T r = 0 \; . \tag{2}$$

Following Duff et al. [7] and Zlatev [30] it is assumed that the matrix A is **large**, **sparse** (most of its elements are equal to zero), and has **full column rank** (i.e., $\mathrm{rank}(A) = n$). Furthermore, it is also assumed that A has neither any special property (such as symmetry or positive definiteness) nor any special structure (banded, block upper/lower triangular, etc.). In other words, A is assumed to be *a general sparse matrix*.

Sparse linear least squares problems appear in various technical and scientific areas like tomography, photogrammetry, geodetic survey, cluster analysis, molecular structure, structural analysis, and so on. More details on these and other applications can be found, e.g., in Björck [4] or Rice [28].

In this paper we describe a coarse-grained parallel algorithm SPARQR for solving the sparse linear least squares problem (1) by an orthogonal factorization written as:

$$\Pi A P = QR \; , \tag{3}$$

where $Q \in \mathcal{R}^{m\times n}$ has orthonormal columns ($Q^T Q = I$), $R \in \mathcal{R}^{n\times n}$ is an upper triangular matrix and $\Pi \in \mathcal{R}^{m\times m}$, $P \in \mathcal{R}^{n\times n}$ are permutation matrices. Usually the matrix Q is obtained as a product of elementary orthogonal transformations and permutation matrices and is not calculated explicitly (Q is likely to be much denser than A). Equation (3) shows a simple relationship between matrix R, which is actually used in the solution and the original matrix A. In fact, we are calculating an orthogonal matrix $Q_1 \in \mathcal{R}^{m\times m}$, the first n columns of which form matrix Q, and a matrix $R_1 \in \mathcal{R}^{m\times n}$, the first n rows of which form matrix R (the last $m - n$ rows containing zeros only). However, matrix Q_1 is neither calculated explicitly nor stored; we are only working with matrix R.

The major advantage of the orthogonal factorization is its stability. Pivoting for stability is as a rule not used in the codes based on orthogonal decomposition techniques, although column interchanges are recommended when the matrix is very ill-conditioned, cf. Golub [17] and Powell & Reid [26]. The performance on parallel computers is normally more efficient when the orthogonal factorization is carried out without pivoting for numerical stability.

Column interchanges are also (explicitly or implicitly) needed if one wants to compute a rank-revealing QR-factorization; see, for example, Chan [5], Foster [10], or Pierce and Lewis [25]. Developing a parallel rank revealing code for sparse matrices will be one of the topics for future work.

Both column and row interchanges must be used in order to preserve better the sparsity of matrix A during the orthogonal decomposition. Finding a good pivoting strategy for the preservation of sparsity is important when sparse matrix algorithms are developed.

There are several ways for computing the QR-factorization. The most popular of them, the *Householder* and the *Givens* methods, are based on representing the orthogonal matrix Q as a product of *elementary transformations* (Householder reflections or Givens plane rotations respectively). The Householder method [20] is commonly used for dense QR-factorization, both sequential and parallel [29], because it is stable and requires minimum floating-point operations. It has been implemented in many standard numerical linear algebra packages such as LAPACK [2], LINPACK [6], and NAG [24]. The Householder method has also been successfully used in some multi-frontal techniques for parallel sparse matrix factorization; mainly by applying dense matrix computations for small frontal matrices, [1, 21, 27].

Our algorithm is based on the *Givens method*, i.e., on a series of elementary plane rotations. A single element of the active submatrix is annihilated in each plane rotation. This method is as stable as the Householder method (cf. [18]), but it preserves in general much better the sparsity of the original matrix A (cf. [30]). Only two rows take part in each rotation and a number of rotations can be performed concurrently (assuming that each row takes part in at most one rotation), which means that there is a lot of potential parallelism in this method, which may be an important factor in many situations, but it is not exploited in this paper (because we are interested in obtaining much bigger parallel tasks).

There are two versions of the Givens method, *classical Givens* [16] and *Gentleman-Givens* [12]. The classical Givens version was used by George and Heath in the package SPARSPAK-B [13, 14, 15]. The Gentleman-Givens version (also known as *fast Givens*) is more economic with regard to floating point operations, but has some overhead and may occasionally require rescaling in an attempt to prevent overflows. It has been applied in the code LLSS by Zlatev [30]. The classical Givens version is used in our code SPARQR. The pivotal strategy is the same as *Strategy 1* in Chapter 14 in [30], but it is restricted to the diagonal blocks discussed in §2.2 and in §2.3. Consider a diagonal block. Assume that the computations have been carried out to some major step $s-1$. This means that zero elements have been produced under the diagonal elements in the first $s-1$ columns. A column with a smallest number of non-zero elements in its active part (i.e. elements that has row number greater than or equal to s) has to be found in the set of columns with column number greater than or equal to s. If the column so found is not the column with column number s, then a column permutation is made to move this column to position s. Assume that there are q non-zero elements with row numbers greater than or equal to s in the pivotal column. Givens rotations are carried out to produce $q-1$ zero elements and to modify the non-zero elements in the involved rows (while the pivotal search is restricted to the diagonal blocks, the Givens rotations are performed for the whole involved rows). At each Givens rotation the pair of rows that have smallest number of non-zero elements is chosen and a zero is produced in one of rows. Assume that the last non-zero element in the active part of column s has a row number $p \geq s$. If $p > s$, then a row permutation is made to move the row p to position s. This completes major step s, and one can proceed with the next major step, major step $s+1$.

By using the QR-factorization (3), the linear least squares problem (2) can be transformed to the following equivalent and simpler system:

$$R(P^T x) = Q^T \Pi b\,. \tag{4}$$

The solution, obtained by back substitution in the triangular system (4) is called *the direct solution.*

Very often the amount of work during the factorization can be reduced significantly by considering as zeros all matrix elements and fill-ins that are relatively small in absolute value, according to a certain relative threshold τ called *the drop-tolerance.* The solution $\tilde{x}$, obtained in a similar way as the direct solution, but by using the approximate triangular factor $\tilde{R}$ instead of R in (4),

$$\tilde{R}(\tilde{P}^T \tilde{x}) = \tilde{Q}^T \tilde{\Pi} b\,, \tag{5}$$

will normally contain a certain error that depends on the size of the drop-tolerance τ. In SPARQR the solution obtained by applying (5) is used as an initial guess in an iterative method. It is called *the initial approximation.* The solution obtained when the iterative method converges sufficiently quickly (it will normally satisfy some accuracy requirement) will be called *the iterative solution.*

The *preconditioned conjugate gradients* (PCG) method is applied to the system $A^T Ax = A^T b$ of *semi-normal* equations [3], where the approximate factor $\tilde{R}$ is used as a preconditioner, in a similar way as in [30]. This means that the following preconditioned system of linear algebraic equations is treated:

$$Cz = d\,, \tag{6}$$

where

$$\begin{aligned} C &= (\tilde{R}^{-1})^T \tilde{P}^T A^T A\, \tilde{P}\, \tilde{R}^{-1} \\ z &= \tilde{R}\, \tilde{P}^T \tilde{x} \\ d &= (\tilde{R}^{-1})^T \tilde{P}^T A^T b\,. \end{aligned} \tag{7}$$

More details about the implementation of the PCG method are presented in §2.5 of the next section. It is important to emphasize here that the quantities C, z and d are never calculated explicitly; one works the whole time with the quantities A, $\tilde{R}$, $\tilde{P}$, $\tilde{x}$ and b (see Fig. 3).

We emphasize that in addition to improving the accuracy of the initial approximation, the PCG method yields a rather reliable error estimate, which is important when the matrix is ill conditioned and no pivoting for stability is used. The use of PCG methods in sparse codes for linear least squares is discussed in detail in Chapter 16 of [30].

SPARQR, which will be described in the following sections, is entirely based on sparse matrix computations (performed in one dimensional sparse arrays by trying to design pivotal strategies, restricted to the diagonal blocks, in order to keep the number of fill-ins low). One can also achieved parallel computations by using another technique, which is described in detail in [19]. The computations can be carried out by creating a sequence of dense blocks. The dense blocks are executed sequentially, and dense matrix kernels are called to treat each block. Parallel computations are performed by the dense kernels (LAPACK routines, [2], are used at present, but one can easily switch to other efficient dense subroutines). In this algorithm, let us call it here Method 1, all computations are carried out by using dense matrix techniques, sparse matrix technique is only used to move data from one-dimensional sparse arrays to two-dimensional dense arrays and vice versa. This means that the underlying techniques used in the two methods are quite different. The main objectives which one tries to achieve with these two techniques are also quite different. In SPARQR one tries to minimize the number of computations (the price being a lower computational speed). In Method 1 one tries to keep a high computational speed (the price that one has to pay is some unnecessary work with zero elements). This short analysis indicates that it is not very clear which method is the best one. For matrices that are very sparse and remain very sparse during the factorization SPARQR should be preferred (the low number of computations will be a sufficient compensation for the low computational speed in this situation). If the matrix is dense or if it becomes dense during the computations, then Method 1 will perform best (the high speed of computation will be a sufficient compensation for the extra operations with zero elements). A few results which confirm this statement are given in [19]. Very often the matrix is very sparse in the beginning, and then becomes denser and denser. This fact indicates that a hybrid methods, where one starts with SPARQR and switches to Method 1 when appropriate, may be a good choice. This is a good topic for a future research.

The SPARQR algorithm is described in Section 2. In Section 3 some results of our numerical experiments on a 16-processor SGI Power Challenge are presented. Some conclusions are drawn in Section 4.

2 SPARQR – a sparse parallel QR-factorization algorithm

The SPARQR algorithm consists of five stages as follows:

Stage 1. Reordering (which is preceded (i) by saving the non-zero elements of the original matrix, their column numbers and information about row starts and row ends in appropriate arrays and (ii) by dropping, optionally, all small elements in the copy that will be used during the orthogonal factorization).

Stage 2. Block partitioning.

Stage 3. Factorization using Givens rotations (dropping optionally all small elements during the computations).

Stage 4. Back substitution (computing the direct solution or an initial approximation).

Stage 5. Computing the iterative solution (using PCG).

It must be reiterated here that both the dropping procedures, applied in Stages 1 and 3, and the iterative method used in Stage 5 are optional. Without dropping small elements, we compute the triangular factor R, otherwise we compute the approximate factor $\tilde{R}$. By dropping relatively small elements (with respect to a given drop-tolerance τ), the cost of computing the least squares solution can often be reduced dramatically. However, the factorization and the solution of (5) will in general be inaccurate. Therefore the iterative Stage 5 should be applied in order to obtain a sufficiently accurate iterative solution.

Dropping small elements is the computer-oriented manner of exploiting the sparsity from Zlatev [30]. For unstructured matrices this device is often one of the most efficient ways to exploit the sparsity as much as possible. By varying the drop-tolerance τ, which is a user-specified parameter, one can obtain a more or less accurate QR-factorization that costs more or less computing time, respectively. There is a danger, however, that too many non-zero elements will be removed when τ is very large. If this happens, then the active sub-matrix becomes structurally singular at some stage of the factorization and, therefore, the calculations must be repeated by a smaller value of the drop-tolerance τ. It may be necessary to repeat this procedure several times. If the factorization is completed successfully (i.e. if a non-singular upper triangular factor $\tilde{R}$ is obtained), then $\tilde{R}$ is used during Stage 5 as a preconditioner in the system (6). An initial approximation of the solution vector is found in Stage 4. In most cases, the PCG algorithm used in Stage 5 converges reasonably fast and gives a solution with the required accuracy. Otherwise we must again reduce the drop-tolerance and restart the process from Stage 1, trying to obtain a more accurate preconditioner with a reduced τ.

The five stages of SPARQR are discussed in more detail in §2.1 – 2.5., while §2.6 contains a description and an analysis of the main stopping criteria for the iterative Stage 5.

The following brief description of the main data structures used in SPARQR – the *row-ordered list* and the *column-ordered list*, will be helpful for better understanding of our algorithm (see more details in [7] and [30]). The row-ordered list is the main dynamic data structure. In the beginning of the algorithm it contains the non-zero elements of the matrix A in an array AQR (the non-zero elements are ordered by rows by LORA). If an element of matrix A is stored in position K in array AQR, then its column number is stored in the same position in an integer array CNQR. Pointers for row starts and row ends in arrays AQR and CNQR are stored in an integer array HA1. In the end of Stage 3, the non-zero elements of the copy of matrix A, which is stored in array AQR, are overwritten by the non-zero elements of the triangular factor R or $\tilde{R}$. Again, if an element of the matrix R or $\tilde{R}$ is stored in position K in array AQR, then its column number is stored in the same position in the integer array CNQR. The new pointers associated with AQR and CNQR are again stored in the integer array HA1. This updated row ordered list is needed in Stage 4, and also in Stage 5 when the PCG method is used. The iterative Stage 5 requires also a copy of the original row-ordered list.

The row ordered list describes adequately the structure of the matrix. However, some searches (especially searches in the pivoting strategy) can be executed more efficiently if another structure, where the row numbers of the non-zero elements are ordered by columns, is also prepared. This is the *column-ordered list*. It is formed in a quite similar way as the row ordered list, but the non-zero elements themselves are not stored; one works only with their row numbers when the column ordered list is used. The row numbers of the non-zero elements (ordered by columns) are stored in an integer array RNQR. Information for column starts and column ends is stored in array HA2. The column-ordered list is only needed during the factorization phase; Stage 3. In fact, column-ordered lists are needed only for some parts of matrix A, namely, the diagonal blocks in the partitioned matrix; see §2.3.

The length of main arrays AQR, CNQR and RNQR is not known in advance. In Section 3, the length of the first two arrays is set to 10*NZ, while the length of RNQR is set to 5*NZ. If the matrix creates a lot of fill-ins, then this might be insufficient. If a positive drop-tolerance is used, then the storage requirement can normally be reduced (by reducing the length of the main arrays).

The information given above is quite sufficient for the purposes in this paper. Detailed overviews of the particular data structure which is applied here, and other commonly used data structures for sparse matrices, can be found in [7, 30]. Some specific features of the ordered lists that are implied by the requirements to obtain parallel computations will be discussed in §2.3 and in Section 3.

2.1 The reordering stage

This is the first stage of our algorithm (and of many other algorithms for sparse matrices). The purpose is to find proper initial column and row permutations such that the permuted matrix has a more appropriate structure for efficient parallel QR-factorization than A itself. The row permutation Π is not present in the semi-normal system $A^TAx = A^Tb$. The row interchanges are (both these needed in LORA and these needed during the factorization) are only used to permute the right-hand side b. As soon as this is done, one can get rid of them. On the other hand, the column permutation P must be saved to be used during the back substitution and throughout the iterative Stage 5, as shown in §2.4 and §2.5, respectively.

The *Locally Optimized Reordering Algorithm* (LORA) [9, 11] is used in SPARQR. LORA attempts to move as many zeros as possible to the lower left corner of the matrix. This has two positive effects. First, it tends to reduce both the number of non-zeros and the number of fill-ins below the main diagonal. This leads also to a reduction of the number of Givens rotations at Stage 3. Second, the structure obtained by LORA (see Fig. 1) facilitates the determination of a well balanced coarse-grained block partitioning and, consequently, the application of LORA leads to an efficient block-parallel factorization.

The version of LORA for square matrices is described in detail in [11], where it was used to reorder a square matrix before its LU-factorization by Gaussian elimination. The generalization for rectangular matrices is quite straightforward; the basic properties and the complexity estimate, $O(\mathtt{NZ}\log n)$, where NZ is the number of non-zeros, remain unchanged. Like many other reordering algorithms LORA is inherently sequential, but this is as a rule a small disadvantage, because its cost is normally very low (see Table 4). On the other hand, the use of LORA proved to be very advantageous for the parallel abilities of SPARQR. This reordering makes it easy to efficiently parallelize the most time-consuming Stage 3 (see Table 4) in terms of a well balanced coarse-grained partitioning. The numerical experiments with both SPARQR and the LU-factorization code confirm that LORA is a cheap and efficient reordering scheme. It is applicable to a wide class of problems in which general sparse matrices are involved.

Some other operations, not closely related to the reordering itself, but preceding

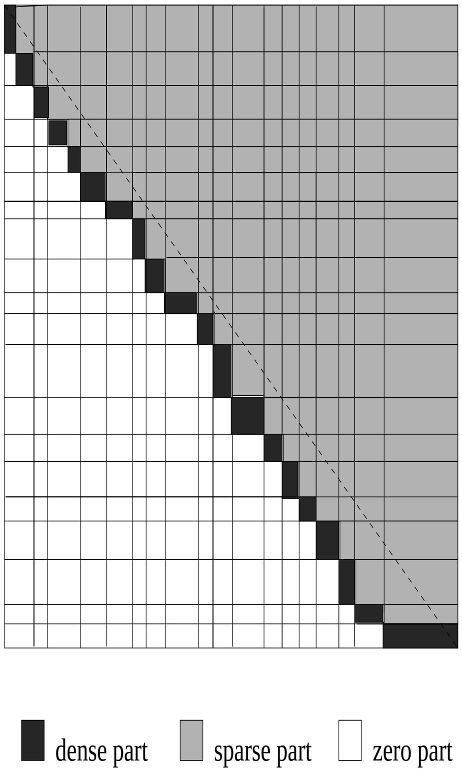

Figure 1: The structure of a matrix, reordered by LORA.

it or convenient to be executed together with it, are also included in Stage 1. We should mention here the creation of the row-ordered list as well as the optional initial dropping is preceded by saving a copy of the original row-ordered list (which is to be used in Stage 5). The initial dropping is applied to the original matrix when $\tau \neq 0$, and an element a_{ij} of A is dropped if

$$|a_{ij}| \leq \tau \max_{1 \leq l \leq m} |a_{lj}| \,.$$

When $\tau \neq 0$, then LORA is always applied to the matrix after the initial dropping of small non-zero elements.

2.2 Partitioning into blocks

During the second stage both the set of rows and the set of columns of the matrix (as ordered by LORA) are divided into k parts, called respectively *block-rows* and *block-columns*. The goal is to represent the reordered matrix in a $k \times k$ block upper triangular form with rectangular blocks, where each block is the intersection of a block-row and a block-column.

The pattern created by LORA (see Fig. 1) has already block upper triangular form with dense diagonal blocks (which are rectangular matrices). The corresponding block-rows and block-columns are usually too small to be useful as a partitioning for parallel computations. However, by gathering several such blocks we can easily obtain a partitioning in relatively large blocks which are more suitable for parallel computing.

The block-rows are processed in parallel during the most time-consuming Stage 3. The goal is to annihilate the subdiagonal non-zeros in the rectangular diagonal blocks by Givens rotations. Therefore both the number of rows and the number of columns in the diagonal blocks are important in our attempt to obtain well-balanced tasks during Stage 3. Once the matrix is ordered, if we fix the row blocking then the column blocking is also uniquely determined by the "diagonal" dense blocks, and vice versa (because of the requirement for block upper triangular form of the partitioning). This can be used either to fix the block-row partitioning or to fix the block-column partitioning of the matrix, determining either block-rows with approximately the same number of rows or block-columns with approximately the same number of columns. The first strategy is applied in the code discussed here. Other more sophisticated partitioning strategies are also possible but have not been tried yet.

The row-ordered list is physically partitioned in this stage according to the block-row partitioning. This means that the arrays AQR and CNQR, where the non-zero elements and their column numbers are kept, are divided into k parts, and for each part a separate row-ordered list is created for the corresponding block-row. In principle, the physical partitioning of the row-ordered list is not absolutely necessary on a shared memory machine. If it is not done, then one must carry out some complicated synchronization checks in order to parallelize the factorization stage. This will result both in a significant slowdown due to synchronization overhead and in some loss of portability of the parallel code. The partitioned row-ordered list that we use here is easy to update and provides completely data-independent tasks during the factorization stage, which implies efficient parallelization. In addition, the entire partitioning stage is rather inexpensive, as will be shown in the next section (see Table 4).

2.3 Factorization

The factorization stage consists of k steps, where k is the number of block-rows. The four factorization steps for an example with $k = 4$ are shown in Fig. 2. During each step the diagonal blocks of the active submatrix are triangularized by Givens rotations. The transformations are local within the corresponding block-row, therefore the block-rows can be processed in parallel.

At the end of the i-th step ($i = 1, 2, \ldots, k$), each block-row $B_j^{(i)}$, excluding the last one, $B_k^{(i)}$, can be divided into two parts: a trapezoidal *head* H_j (the rows that form the upper triangular part of the diagonal block) and a rectangular *tail* T_j (the rows that have only zeros in the diagonal block). The last block has obviously no tail, because it has been completely factorized as a usual rectangular matrix.

A rearrangement of the block-rows is performed before the next step, so that the head H_j ($j = i+1, i+2, \ldots, k$) of each block-row is gathered with the tail T_{j-1} of the previous block-row, thus forming a new block-row $B_j^{(i+1)}$. The heads of the first $i-1$ block-rows are already a part of the final triangular factor, so they are excluded from the active part of the matrix. The final blocks of the triangular factor are denoted R_j, $j = 1, 2, \ldots, k$, in Fig. 2 (one such block is created at the end of each step), and it should be mentioned that if small elements are dropped then it is more correct to use the notation $\tilde{R}_j$. The number of block-rows in the active part of the matrix (which are to be modified at the next step) decreases by one at the end of each step. This

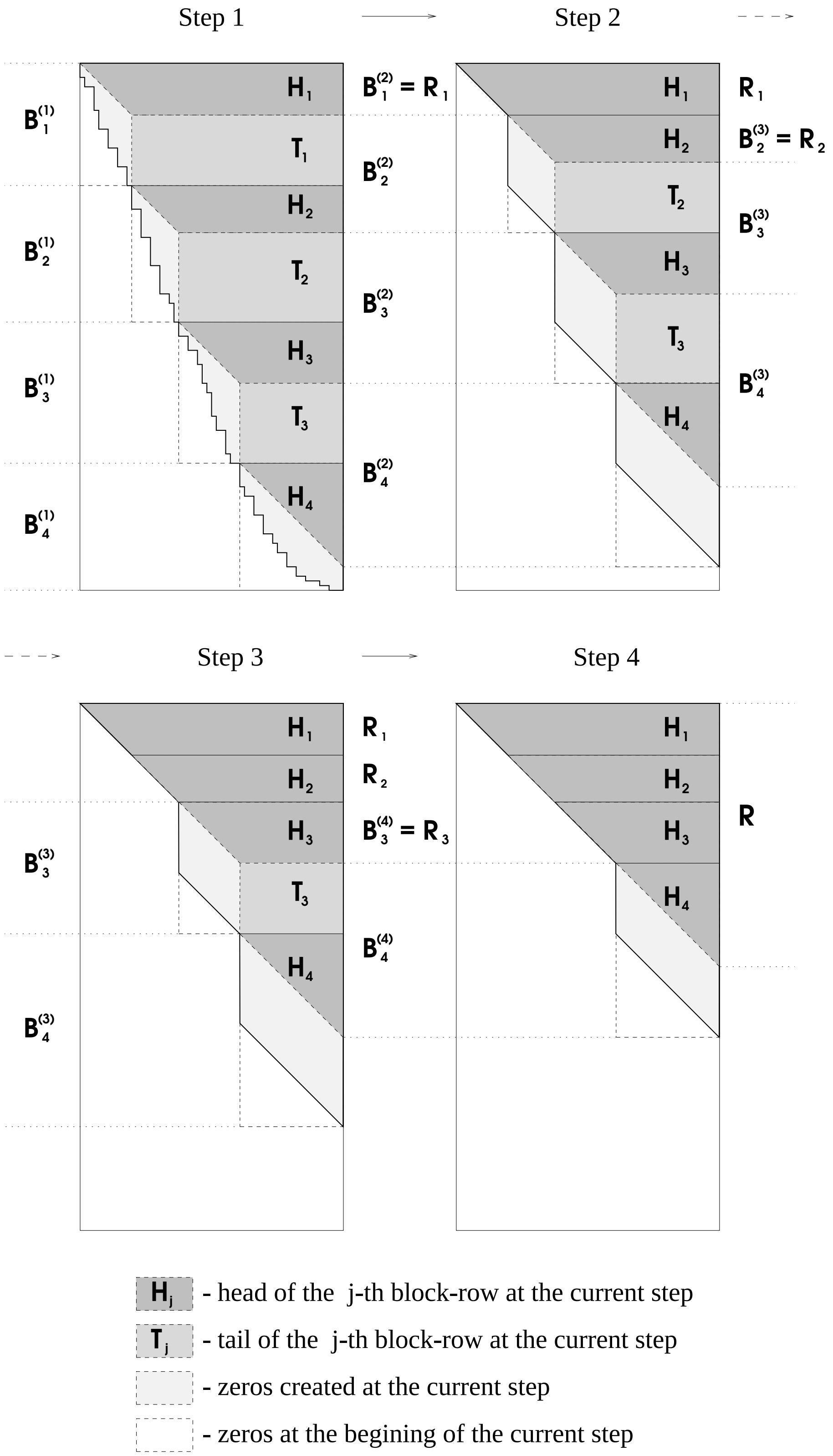

Figure 2: The four factorization steps for a matrix, partitioned into 4 block-rows.

means that the number of parallel tasks is k during the first step, $k-1$ during the second one, and so on. If the algorithm is run on a parallel machine with p processors ($p \leq k$), then during the last $p-1$ steps the average utilization of the processors will gradually decrease. Thus, the number of block-rows k should be considerably larger than p in order to ensure a high average utilization of the processors.

At each Step i, some of the diagonal blocks after the first one may be rank deficient, especially when k is large (even if A has full rank), but this does not cause any problem. If this happens, then the corresponding head H_j will have less rows then the number of columns in the corresponding block-column. The rank deficiency will disappear either during the performance of the next step or during the performance of some later step (unless the whole matrix is singular), when the head H_j is combined with a previous tail T_{j-s}, where $j-s>0$. If, as a result of the numerical dropping, the computed upper triangular matrix $\tilde{R}$ has become singular, then SPARQR should be restarted, as stated above, with a smaller drop-tolerance τ from Stage 1.

A *local* column-ordered list is created for each diagonal block in the active submatrix in order to support the factorization of this block. To be more precise, only the non-zeros in the active part of the corresponding block reside in this list, which becomes empty when the block is factorized. The column-ordered lists are used to provide direct access to the active part of the current pivotal column and to find easily the rows to be rotated in order to annihilate appropriate elements. The lists are also used for optimizing the pivotal choice with respect to sparsity preservation. Unlike the row-ordered list, only row indices are kept in a column-ordered list (no non-zero elements are stored in the column ordered lists), because these lists are used only for search and sort. When k is large, the maximal total number of elements kept in the column-ordered lists is normally much smaller than the corresponding number for the row-ordered list. Thus, the storage requirement for these local column-ordered lists is less than the storage for the row-ordered list and tends to decrease with increasing k. Outside Stage 3 this storage can be used for other purposes.

The drop-tolerance τ controls whether elements should be dropped throughout the factorization. A non-zero element is removed during the computations in Stage 3 when it becomes smaller in absolute value than the product of the drop-tolerance τ and the Euclidean norm of the active part of its column.

2.4 Back substitution

This is the last stage in our algorithm when a very small or zero drop-tolerance has been used during the factorization and, moreover, when no error estimation is required. In such a case, the direct solution calculated by (4) is accepted without any accuracy check. If the drop-tolerance τ is relatively large and/or if an error estimation is desirable, then the solution found in this stage is used only as an initial approximation for the iterative Stage 5. In both cases it is calculated by using the (possibly approximate) triangular factor, obtained in the previous stage.

Both the row permutation in Stage 1 and the orthogonal transformations in Stage 3 have been applied to the extended matrix $[A \,|\, b]$, in order to obtain $[R \,|\, Q^T \Pi b]$ or, in case of dropping small elements, $[\tilde{R} \,|\, \tilde{Q}^T \tilde{\Pi} b]$, where $\tilde{Q}$ and $\tilde{R}$ are the approximate factors corresponding to Q and R. The extra work for the transformation of the right-hand side b pays off, because neither Π (or $\tilde{\Pi}$) nor Q (or $\tilde{Q}$) will be needed anymore and there is no reason to keep them. In fact, the orthogonal factor Q (or $\tilde{Q}$) is never explicitly calculated (it is normally very dense), which is advantageous both from a storage-saving point of view and from a time-saving point of view. The solution (or the initial approximation of the solution) can be found by solving the triangular system (8) by back substitution, followed by a permutation of the solution vector:

$$(8) \qquad R\,y = Q^T \Pi\, b\;, \quad x = Py \qquad \text{or} \qquad \tilde{R}\,\tilde{y} = \tilde{Q}^T \tilde{\Pi}\, b\;, \quad \tilde{x} = \tilde{P}\tilde{y}\;\;.$$

If the method is to be used for solving several least squares problems with the same coefficient matrix A and different right-hand sides b, then it is enough to carry out the factorization only once, although neither $Q^T\Pi b$ (or respectively $\tilde{Q}^T\Pi b$) nor Q (or respectively $\tilde{Q}$) will be available after the factorization. In this case the direct solution (or the initial approximation) is calculated as the solution of either of the following systems (the semi-normal equations):

$$(9) \quad R^T R\, y = P^T A^T b\;, \quad x = Py \qquad \text{or} \qquad \tilde{R}^T \tilde{R}\, \tilde{y} = \tilde{P}^T A^T b\;, \quad \tilde{x} = \tilde{P}\tilde{y}\;\;.$$

It is clear that the direct solution (the initial approximation) x_0 can be obtained by (i) performing one matrix-vector multiplication, (ii) solving two systems with triangular coefficient matrices and (iii) permuting two vectors. The price of using the semi-normal system (9) is a certain loss of accuracy in the direct solution. However, this is usually not critical if the matrix is not extremely ill-conditioned; the iterative Stage 5 can always be used to compute an improved solution, provided a sufficiently small drop-tolerance τ has been used.

Both ways of calculating the direct solution, (8) and (9), are available in our code. We emphasize that for stability reasons (8) should always be used when possible.

2.5 Improving the solution by an iterative method

The upper triangular matrix $\tilde{R}$ is used as a preconditioner of the normal matrix $A^T A$; see (6) and (7). The PCG method is to be applied to the preconditioned system (6), in a similar way as in Zlatev [30], in an attempt to obtain a solution with the required accuracy. The solution computed in Stage 4 is taken as an initial approximation x_0. The PCG algorithm used in this stage is summarized in Fig. 3.

The number of iterations depends strongly on the drop-tolerance. A small drop-tolerance provides a more accurate preconditioner and, consequently, faster convergence. On the other hand, a too large drop-tolerance often results in a fast factorization, but a very slow convergence or even lack of convergence. The convergence criterion and other effective checks for stopping the iterations are discussed in §2.6.

An efficient implementation of the PCG algorithms is obtained by parallelizing the matrix-vector operations that appear in the initialization and in the first two lines of the iteration loop. There are three kinds of operations, namely, a matrix-vector multiplications with A and with A^T, both of complexity $\mathcal{O}(\mathtt{NZ}(A))$, two back substitutions, both of complexity $\mathcal{O}(\mathtt{NZ}(\tilde{R}))$, and two permutations of complexity $O(n)$.

A sparse matrix-vector multiplication with A is much more efficient than a sparse matrix-vector multiplication with A^T when only the row-ordered list of the matrix

1. $r_0 = (\tilde{R}^{-1})^T P^T A^T (b - Ax_0)$;
$\delta_0 = r_0^T r_0$; $p_1 = r_0$;
2. **do** $i = 1$, `ITMAX`
$q_i = P\,\tilde{R}^{-1} p_i$;
$s_i = (\tilde{R}^{-1})^T P^T A^T A\, q_i$;
$\gamma_i = p_i^T s_i$;
$\alpha_i = \delta_{i-1}/\gamma_i$;
$x_i = x_{i-1} + \alpha_i q_i$; $r_i = r_{i-1} - \alpha_i s_i$;
$\delta_i = r_i^T r_i$;
$\beta_i = \delta_i/\delta_{i-1}$;
$p_{i+1} = r_i + \beta_i p_i$;
if ({one of the termination checks is satisfied}) **go to 3.**
end do
3. {output the results} ;
stop

Figure 3: The Preconditioned CG algorithm, used in SPARQR. x_0 is the initial approximation. The capital latin leters represent matrices, the small latin leters (except the iteration index i) represent vectors, while the greek letters stand for scalar variables in the above description of the algorithm.

is available. However, multiplication with both A and A^T are performed in each iteration. Therefore an improvement of the performance can be achieved if also a row ordered list of A^T is prepared (the row ordered list of A^T is the column ordered list of A together with the non-zero elements).

2.6 Stopping criteria and error estimation in Stage 5

A variety of stopping criteria, discussed in detail in [30], are used to control the iterative process. The most important of them are the convergence criterion, which decides whether the method converges satisfactorily fast, and the accuracy check, which decides whether the desired accuracy is achieved. The convergence analysis is essential for this stage as well as for any iterative method. The PCG algorithm (Fig. 3) finds a series of approximate solutions

$$\{x_i\}_{i=0}^{\infty} \quad \text{with} \quad x_i = x_{i-1} + \alpha_i q_i\ ,\ \ \forall i > 0\ . \tag{10}$$

In exact arithmetic the series (10) must converge to the exact solution, because the matrix C in (7) is always symmetric and positive definite. In practice, however, when both α_i and q_i are calculated with rounding errors, this is in general not true and the analysis is very difficult. If computer arithmetic is used, then the convergence itself does not necessarily mean convergence to the exact solution (such an unclear situation may arise, for example, when $\alpha_i \to 0$). One should carefully check the behaviour of the parameter α_i throughout the iterative process [30]. The accuracy check, described in more detail below, should be used only if the α_i do not increase, decrease, or oscillate too much.

Let $f_i = \|\alpha_i q_i\|$. Assume that $f_i \neq 0$, $\forall i$, and consider $\rho_i = f_i / f_{i-1}$ (the analysis given below is valid for any vector norm, but it should be emphasized that Euclidean norms are used in the stopping criteria based on this analysis). It is well known (d'Alembert criterion) that

$$\text{(11)} \qquad \text{if} \quad \rho_i \le \rho < 1 \; , \quad \forall i \ge i_0 \quad \text{then (10) is convergent,}$$

$$\text{(12)} \qquad \text{if} \quad \rho_i \ge 1 \; , \quad \forall i \ge i_0 \quad \text{then (10) is divergent.}$$

In practical computations with the PCG method the main problem is that i_0 is not known. Therefore it is worthwhile to consider the arithmetic averages:

$$\tilde{\rho}_i \equiv \frac{\rho_i + \rho_{i-1} + \rho_{i-2} + \rho_{i-3}}{4} \quad , \quad \tilde{\rho}_i \equiv \varrho \;\text{ for }\; i \le 0$$

where $\varrho < 1$ is a given positive constant, as well as their maxima through the last 4 iterations

$$\hat{\rho}_i \equiv \max(\tilde{\rho}_i, \tilde{\rho}_{i-1}, \tilde{\rho}_{i-2}, \tilde{\rho}_{i-3})$$

Consider also two integer variables `NBAD` and `NBAD1`, initialized to zero and updated in each iteration as follows:

- If $\tilde{\rho}_i > \varrho$ then $\texttt{NBAD1} = \texttt{NBAD1} + 1$, $\texttt{NBAD} = \texttt{NBAD} + \lambda$, where λ depends on the size of $\tilde{\rho}_i$ and $1 \le \lambda \le 7$;
- If $\tilde{\rho}_i \le \varrho$ then $\texttt{NBAD1} = 0$;
- If $\hat{\rho}_i \le \varrho$ then $\texttt{NBAD} = 0$.

The series (10) is considered as convergent as long as

$$\text{(13)} \qquad (\tilde{\rho}_i \le \varrho) \vee ((\texttt{NBAD} \le 40) \wedge (\texttt{NBAD1} \le 5)).$$

Otherwise the series (10) is considered as non-convergent or very slowly convergent, and the algorithm restarts itself from Stage 1 with a decreased value of the drop-tolerance τ in an attempt to find a more accurate preconditioner. The constant ϱ, which controls the acceptable speed of convergence, is an user-driven parameter. In our numerical experiments, presented in the next section, $\varrho = 0.9$.

It is very important to obtain a reliable and tight error estimation in order to avoid unnecessary iterations. The following formula is used to estimate the relative error $\varepsilon(x_i) \equiv \|x - x_i\|_2 / \|x\|_2$ at the i-th iteration if $\hat{\rho}_i \le \varrho$ (otherwise the error estimation and the related accuracy check are skipped; it should be mentioned here that the accuracy check is also skipped when the behaviour of parameter α_i does not satisfy the requirements stated in the beginning of this paragraph):

$$\text{(14)} \qquad \tilde{\varepsilon}(x_i) = \frac{\max(f_i, \varrho f_{i-1}, \varrho^2 f_{i-2}, \varrho^3 f_{i-3})}{(1 - \hat{\rho}_i)\|x_i\|_2} \; .$$

If $\tilde{\varepsilon}(x_i) < \epsilon$ where ϵ is the desired accuracy specified by the user and if the last values of α_i satisfy the three conditions stated above, then the iterative process is stopped and the solution x_i from the last iteration is assumed to be sufficiently accurate. The following theorem indicates why the convergence criterion (13) and the accuracy check, based on the error estimation formula (14) should be expected to work well in practice.

Theorem 1 *Let $\{x_i\}_{i=1}^{\infty}$ be a sequence of vectors, $\Delta x_i = x_i - x_{i-1}$, $f_i = \|\Delta x_i\|$, $\rho_i \equiv f_i/f_{i-1}$ for $i \geq 1$, and let ℓ be a positive integer constant. Consider the arithmetic averages and their maxima (for given index $i_0 \geq \ell$):*

$$\tilde{\rho}_i \equiv \sum_{j=i-\ell+1}^{i} \frac{\rho_j}{\ell} \ , \forall i \geq \ell \ , \quad \hat{\rho} \equiv \max\{\tilde{\rho}_i\}_{i=i_0}^{\infty} \ . \tag{15}$$

If $\hat{\rho} < 1$ then the series $\{x_i\}$ is convergent and if $\chi \equiv \lim_{i\to\infty} x_i$, then:

$$\|\chi - x_i\| \leq \frac{\max\{f_{i-\lambda}\hat{\rho}^{\lambda}\}_{\lambda=0}^{\ell-1}}{(1-\hat{\rho})} \ , \quad \forall i \geq i_0 \ . \tag{16}$$

Proof: Let i, j, ξ and η be positive integers, such that $j > i \geq i_0$, $j = i + \xi\ell - \eta$ and $0 \leq \eta < \ell$. The following upper bound for f_j is obtained by using the inequality between the geometric and the arithmetic average and the conditions (15) of the theorem:

$$\begin{aligned} f_j &= f_{i-\eta} \prod_{\nu=1}^{\xi\ell} \rho_{i-\eta+\nu} = f_{i-\eta} \prod_{\mu=1}^{\xi} \prod_{\lambda=1}^{\ell} \rho_{i-\eta+(\mu-1)\ell+\lambda} \\ &\leq f_{i-\eta} \prod_{\mu=1}^{\xi} \Big(\sum_{\lambda=1}^{\ell} \frac{\rho_{i-\eta+(\mu-1)\ell+\lambda}}{\ell} \Big)^{\ell} = f_{i-\eta} \prod_{\mu=1}^{\xi} \Big(\tilde{\rho}_{i-\eta+\mu\ell}\Big)^{\ell} \\ &\leq f_{i-\eta} \prod_{\mu=1}^{\xi} \hat{\rho}^{\ell} = f_{i-\eta}\hat{\rho}^{\xi\ell} = f_{i-\eta}\hat{\rho}^{(j-i+\eta)} = \Big(f_{i-\eta}\hat{\rho}^{\eta}\Big)\hat{\rho}^{j-i} \\ &\leq M_i \hat{\rho}^{j-i} \ , \text{ where } M_i \equiv \max\{f_{i-\lambda}\hat{\rho}^{\lambda}\}_{\lambda=0}^{\ell-1} \end{aligned} \tag{17}$$

¿From (17) it is clear that

$$\hat{\rho} < 1 \quad \Longrightarrow \quad f_j \xrightarrow[j\to\infty]{} 0 \quad \Longrightarrow \quad M_i \xrightarrow[i\to\infty]{} 0 \ . \tag{18}$$

Consider the difference $x_j - x_i = \sum_{\nu=i}^{j} \Delta x_\nu$. ¿From the triangle inequality and (17) it follows:

$$\begin{aligned} \|x_j - x_i\| \leq \sum_{\nu=i}^{j} f_\nu \leq M_i \sum_{\nu=i}^{j} \hat{\rho}^{\nu-i} &= M_i \frac{1-\hat{\rho}^{j-i+1}}{1-\hat{\rho}} \\ &\leq M_i \frac{1}{(1-\hat{\rho})} \ \text{ for any } \ j > i \ . \end{aligned} \tag{19}$$

¿From (18) and (19) it follows that if $\hat{\rho} < 1$ then

$$\|x_j - x_i\| \xrightarrow[i\to\infty\ ,\ j>i]{} 0$$

which implies convergence of the series $\{x_i\}_{i=1}^{\infty}$ (Cauchy criterion). Let $\chi \equiv \lim_{i\to\infty} x_i$. As $\chi - x_i = \sum_{j=i}^{\infty} \Delta x_j$ then

$$\|\chi - x_i\| \le \sum_{\nu=i}^{\infty} f_\nu \le M_i \sum_{\nu=i}^{\infty} \hat{\rho}^{\nu-i} = M_i \sum_{j=0}^{\infty} \hat{\rho}^j = \max\{f_{i-\lambda}\hat{\rho}^{\lambda}\}_{\lambda=0}^{\ell-1} \frac{1}{(1-\hat{\rho})} \ ,$$

which is equivalent to (16). ∎

A sequence of convergence criteria can be obtained from the above theorem for different values of ℓ. In particular, for $\ell = 1$ we obtain d'Alembert criterion (11). Our stopping criteria and the error estimation formula (14) are derived from (16) with $\ell = 4$ and the approximation $\hat{\rho} \simeq \hat{\rho}_i$, since the exact maximum $\hat{\rho}$ in (15) cannot be calculated. It is important to emphasize here that the use of $\ell > 1$ has two main effects: (i) the convergence check is relaxed ($\tilde{\rho}_i$ may be less than one even if ρ_i is greater than one) and (ii) the acceptability check is made more stringent ($\hat{\rho}_i$ can be greater than one even if $\tilde{\rho}_i$ is less than one).

In other words, while we try to avoid the situation where the code declares too quickly the iterative process for divergent, we also are very careful in the attempts to avoid the situation where the code stops the iterative process too early (when the the calculated solution is not yet sufficiently accurate). This strategy is rather conservative, but it works very well in practice (especially when the matrices are ill-conditioned).

3 Numerical results

The algorithm SPARQR has been implemented and tested on an SGI Power Challenge computer with 16 processors. This is a shared memory machine, in which each processor has its own cache. The results of some our experiments, performed on this machine, are given in this section.

3.1 Test matrices, used in the experiments

In our experiments we have used rectangular test matrices from the Harwell-Boeing collection [8], type rra (real, rectangular, assembled), plus two matrices amoco1 and bellmedt kindly provided to us by M. Berry. It should be mentioned here that the transposed matrix of gemat1 is used in this section. The Harwell-Boeing matrices arise in practical problems and applications related to various technical and scientific areas; they are very well known, and they have been used extensively to test the performance of sparse algorithms. Unfortunately most of them are not sufficiently large when modern high-speed computers are used. That is why we included in our experiments a much larger matrix, obtained by using one of the sparse matrix generators described in [30]. This matrix is called matrf2. Information about the size of the matrices, used in our experiments, is given in Table 1. Several other matrices are used in Table 10. Some details about the size of these matrices will be given in Sub-section 3.8.

MATRIX		Number of rows	Number of columns	Number of nonzeros
1	illc1033	1033	319	4495
2	well1033	1033	319	4495
3	illc1850	1850	712	8758
4	well1850	1850	712	8758
5	amoco1	1436	330	35210
6	bellmedt	5831	1033	52012
7	gemat1	10595	4929	47369
8	matrf2	300000	25000	1500110

Table 1: Data for the test matrices.

3.2 Constructing the test problems

It is convenient to have test problems with a known solution. Therefore the following procedure has been used in all experiments. The right-hand side of the problem $b = A\bar{x}$ is calculated by using the chosen matrix and an exact solution, which either has all components equal to one or consists of random numbers (in the tables given in this section the former device is always used, however many experiments with the latter device gave very similar results). This procedure allows us (i) to calculate the norm of the exact error vector of the computed solution and (ii) to compare the norm of the exact error vector with the estimated norm computed in Stage 5.

3.3 Accuracy of the solution and error estimation

The accuracy ε of the direct solution depends on the condition number $\kappa(A)$ and on the machine precision ϵ_M, $\varepsilon = O(\epsilon_M \kappa(A))$. In all problems used in the experiments the accuracy of the direct solution was very high; see, for example, Table 2.

If (i) the PCG algorithm is convergent, (ii) double precision is used and (iii) the accuracy required is not too high, then the required accuracy is achievable when the number of allowed iterations (determined by a parameter MAXIT) is greater than the number of the iterations actually needed. Most of the supercomputers nowadays are designed to perform double precision arithmetic with maximal speed and, in general, it is as fast as single precision arithmetic. The only point to use single precision can be storage saving in case that the storage space is critical. All the results, presented in this paper, are obtained by operating and storing all floating-point data in double precision.

In Table 2 some results on the accuracy of the **direct solution** (when $\tau = 0$) are given. When high accuracy is required, we always recommend to use the iterative Stage 5 in order to estimate the error and to check if the required accuracy is achieved. The following three statements are typically true when $\tau = 0$: (i) one iteration is sufficient, (ii) the computing time spent to calculate it is negligible, and (iii) an updated (and slightly more accurate) solution is obtained together with the error

MATRIX		Direct sol. Exact error	Final sol. (after 1 it.) Estimated error	Exact error
1	illc1033	1.0E-13	1.0E-13	3.7E-14
2	well1033	7.7E-15	7.6E-15	3.0E-16
3	illc1850	3.5E-14	3.7E-14	2.5E-15
4	well1850	6.3E-15	6.4E-15	5.1E-16
5	amoco1	4.6E-14	4.1E-14	1.1E-14
6	bellmedt	5.1E-15	5.1E-15	2.0E-16
7	gemat1	1.8E-13	1.8E-13	2.9E-14

Table 2: The accuracy of the direct solution (obtained with $\tau = 0$) and the improvement, obtained by one iteration of the iterative stage.

estimate. The estimated error and the exact error of the updated solution are given in the last two columns of Table 2.

MATRIX		τ	Iter. number	Estimated error	Exact error
1	illc1033	10^{-3}	31	6.7E-08	8.7E-09
2	well1033	10^{-2}	19	4.6E-08	2.9E-09
3	illc1850	10^{-3}	18	8.2E-08	3.8E-09
4	well1850	10^{-2}	17	7.7E-08	1.2E-08
5	amoco1	10^{-2}	371	2.3E-08	9.4E-09
6	bellmedt	10^{-2}	19	4.3E-08	9.4E-09
7	gemat1	10^{-3}	250	3.1E-08	1.8E-08
8	matrf2	10^{-2}	6	8.8E-08	1.6E-10

Table 3: The number of iterations and the accuracy achieved in the iterative stage for the largest successful value of the drop-tolerance. The iterations are stopped when the estimated error becomes less than 10^{-7}.

Accuracy results obtained when the PCG algorithm is used and when the drop-tolerance is nonzero are presented in Table 3. The matrices listed in the table have been partitioned into $k = 128$ blocks and factorized with initial drop-tolerance $\tau = 10^{-2}$. If either the matrix becomes singular during the factorization stage or problems with the convergence appear in the iterative stage, then the algorithm automatically decreases the current value of the drop-tolerance by a factor of 10 and restarts the computation. This procedure continues until a successful exit from Stage 5. The final value of the drop-tolerance τ is given in the second column of the table. The total number of iterations for all trials (when restarting is used) is given in the third column of Table 3.

If the PCG algorithm used in Stage 5 converges sufficiently fast, then the iterations continue until the estimated error becomes smaller than the desired accuracy (10^{-7} in our experiments). The exact error of the final solution is given in the last column of Table 3. If the estimation is correct, the exact error should not exceed the estimated error (and the two numbers should not differ too much). In our experiments the estimated error of the iterative solution is always slightly larger and usually very close to the exact error (see Table 3). This is an indication that the error estimation used in the PCG algorithm is reliable and efficient.

3.4 Time cost of the different stages of SPARQR

In order to compare the relative cost of the five main stages of SPARQR, several linear least squares problems with matrices of different size were solved sequentially by using different values of the drop-tolerance τ. Table 4 contains the results of this experiment for three rectangular matrices: well1850, amoco1 and bellmedt. Partitioning into $k = 128$ blocks was used in all these runs. For the particular matrices used this is a good choice for parallel computations. The times for the five stages (both in seconds and in percentages of the total time) are given in Table 4.

The comparison of the computing times for the five stages shows that in all experiments the factorization stage is strongly time-dominant. That is why good parallelization of this stage is crucial for obtaining an efficient parallel algorithm.

The factorization time decreases when τ is increased, because the sparsity is preserved better (the number of fill-ins is sometimes reduced dramatically). At the same time, the iteration time increases because more iterations are needed to achieve the accuracy requirements with the poorer preconditioner. The number of iterations depends also on the desired accuracy ϵ, which has been set $\epsilon = 10^{-7}$ for all experiments. The factorization stage remains the most time-consuming part of the computations, but the time of the iterative Stage 5 becomes the second largest part and sometimes its contribution to the total time is quite significant (see the results for amoco1 with $\tau = 10^{-2}$ in Table 4).

In our experiments it was sometimes very advantageous to use a large drop-tolerance, because the reduction of the factorization time was larger than the increase of the iteration time. For some matrices that create a lot of fill-ins, the use of a large drop-tolerance may be extremely helpful. Such an example is the matrix bellmedt, which requires a lot of time if factorized directly. The use of a large drop-tolerance $\tau = 10^{-2}$, however, leads to a reduction of the computing time of more than 25 times.

3.5 Performance versus number of blocks

Results obtained in parallel runs of our code for factorization with $\tau = 0$ are presented in Tables 5, 6 and 7. Partitionings with various number of blocks were applied to each problem in order to examine the relations between the size of the matrix, the number of blocks and the smallest computing time (and, eventually, the speedup). The speedups are shown in parenthesis in Tables 5, 6 and 7. The speed up is measured by dividing the time obtained on one processor by the corresponding time (found by

	Stage	Time on 1 processor					
		well1850		amoco1		bellmedt	
		$\tau = 10^{-2}$					
1	Reordering	0.026	(1.9%)	0.072	(1.1%)	0.142	(0.4%)
2	Partitioning	0.005	(0.4%)	0.007	(0.2%)	0.021	(0.1%)
3	Factorization	1.241	(92.4%)	4.535	(70.6%)	31.930	(98.5%)
4	Back subst.	0.001	(0.1%)	0.002	(0.0%)	0.008	(0.0%)
5	Precond. CG	0.069	(5.2%)	1.804	(28.1%)	0.334	(1.0%)
	Total	**1.343**	(100%)	**6.428**	(100%)	**32.438**	(100%)
		$\tau = 10^{-4}$					
1	Reordering	0.026	(1.3%)	0.071	(0.5%)	0.154	(0.0%)
2	Partitioning	0.006	(0.3%)	0.014	(0.1%)	0.025	(0.0%)
3	Factorization	1.969	(97.5%)	13.962	(98.4%)	847.127	(99.9%)
4	Back subst.	0.001	(0.1%)	0.005	(0.0%)	0.067	(0.0%)
5	Precond. CG	0.017	(0.8%)	0.148	(1.0%)	0.691	(0.1%)
	Total	**2.020**	(100%)	**14.200**	(100%)	**848.187**	(100%)
		$\tau = 0$					
1	Reordering	0.026	(1.0%)	0.071	(0.5%)	0.151	(0.0%)
2	Partitioning	0.004	(0.1%)	0.014	(0.1%)	0.022	(0.0%)
3	Factorization	2.685	(98.7%)	14.285	(99.3%)	860.223	(99.9%)
4	Back subst.	0.001	(0.0%)	0.005	(0.0%)	0.070	(0.0%)
5	Precond. CG	0.003	(0.1%)	0.022	(0.1%)	0.209	(0.0%)
	Total	**2.720**	(100%)	**14.398**	(100%)	**860.799**	(100%)

Table 4: Times (in seconds and percentages) spent in the five stages for the solution of three Harwell-Boeing problems with three different values of the drop-tolerance. The results are obtained on one processor for partitioning on $k = 128$ blocks.

using the same value of blocks k) on p processors. One can also calculate speed-ups by using the best computing times obtained at one processor (these values are underlined in the tables). The speed-ups will become smaller when the latter definition is used. However, for the most time-consuming problems the differences are normally rather small; Table 6 is an illustration of this statement (see also Table 8 and Table 9 in §3.7).

The block partitioning in our algorithm was designed to allow effective coarse-grained parallelism. One should expect to improve the results when the number of blocks k is increased up to a certain point, which depends on the size of the matrix and the number of processors. Above that point the computing time increases, because the partitioning into a large number of very small blocks becomes too expensive.

It may look strange that often the sequential time also decreases with increasing k at the beginning, although with slighter reduction. In order to explain this phenomenon, we have to consider the effect of creating and using column-ordered lists. Only column-ordered lists for the elements in the diagonal blocks are needed in the

k	matrix amoco1 [1436 × 330] , nonzeros: 35210				
	Time (*Speed-up*)				
	1 proc.	4 proc.	8 proc.	12 proc.	16 proc.
1	10.09				
8	8.23	3.46 (*2.4*)	2.46 (*3.3*)	2.43 (*3.3*)	2.17 (*3.8*)
16	9.15	3.33 (*2.8*)	2.06 (*4.4*)	2.00 (*4.6*)	1.78 (*5.1*)
32	10.52	3.01 (*3.5*)	1.74 (*6.0*)	1.50 (*7.0*)	1.63 (*6.5*)
64	12.32	3.81 (*3.2*)	2.12 (*5.8*)	1.87 (*6.6*)	2.14 (*5.8*)
128	14.40	4.57 (*3.2*)	2.82 (*5.1*)	2.56 (*5.6*)	3.07 (*4.8*)

Table 5: Computing times (in seconds) and speedups (shown in parenthesis) for the matrix amoco1, with $\tau = 0$.

k	matrix gemat1 [10595 × 4929] , nonzeros: 47369				
	Time (*Speed-up*)				
	1 CPU	4 CPUs	8 CPUs	12 CPUs	16 CPUs
1	44.62				
8	42.39	19.92 (*2.1*)	17.98 (*2.4*)	19.06 (*2.2*)	15.20 (*2.8*)
16	48.79	17.29 (*2.8*)	14.44 (*3.4*)	13.78 (*3.5*)	14.06 (*3.5*)
32	65.17	19.66 (*3.3*)	12.41 (*5.3*)	11.95 (*5.5*)	13.13 (*5.0*)
64	97.00	27.96 (*3.5*)	17.14 (*5.7*)	13.45 (*7.2*)	10.86 (*8.9*)
128	135.20	36.23 (*3.7*)	22.96 (*5.9*)	17.13 (*7.9*)	16.12 (*8.4*)

Table 6: Computing times (in seconds) and speedups (shown in parenthesis) for the matrix gemat1, with $\tau = 0$.

block-partitioned algorithm (these lists can be created and updated in parallel). Suppose that (i) the non-zero elements of the matrix are evenly distributed and (ii) the matrix is partitioned into $k \times k$ approximately equal blocks, see §2.2. Then at the first factorization step (see §2.3 and Fig. 2) we have k column-ordered lists, each of size $\sim \mathtt{NZ}/k^2$. In total, about $1/k$ of the non-zero elements of the matrix reside in the column-ordered lists during the first factorization step and during the next factorization steps this part tends to decrease. If k becomes larger, then the lists become shorter. Thus, it is cheaper to create and update a column ordered list when k is larger. This explains the initial reduction of the sequential time with increasing k.

On the other hand, **the total number** of column ordered lists created during all the k factorization steps is $k(k-1)/2$. Creating such a list requires some overhead. Although insignificant for small values of k, the total work needed to create all column ordered lists is, roughly speaking, proportional to k^2 and sooner or later (depending on the size of the matrix) it becomes a significant part of the total time.

One should also take into account the pivotal interchanges for preservation of sparsity. These interchanges are restricted to the diagonal blocks. If k is small, then the diagonal blocks are large and the sparsity is normally preserved well. When k

k	matrix bellmedt [5831×1033] , nonzeros: 52012				
	Time (*Speed-up*)				
	1 proc.	4 proc.	8 proc.	12 proc.	16 proc.
1	853.68				
8	847.91	309.81 (*2.7*)	258.51 (*3.3*)	254.97 (*3.3*)	272.15 (*3.1*)
16	849.38	243.90 (*3.5*)	156.35 (*5.4*)	140.77 (*6.0*)	147.47 (*5.8*)
32	828.24	217.85 (*3.8*)	123.63 (*6.7*)	94.01 (*8.8*)	86.90 (*9.5*)
64	<u>825.81</u>	<u>212.92</u> (*3.9*)	<u>103.07</u> (*8.0*)	79.84 (*10.3*)	64.10 (*12.9*)
128	860.80	221.01 (*3.9*)	110.75 (*7.8*)	<u>76.91</u> (*11.2*)	<u>60.60</u> (*14.2*)
256	938.17	238.98 (*3.9*)	121.56 (*7.7*)	86.64 (*10.8*)	69.72 (*13.5*)

Table 7: Computing times and speedups (shown in parenthesis) for the matrix bellmedt, with $\tau = 0$.

becomes very large, however, the diagonal blocks become rather small. This fact will normally lead to a poor preservation of the sparsity.

These effects, connected with the column-ordered lists and with the pivotal interchanges, appear to be the major reason for the non-monotonous behaviour of the sequential computing time of SPARQR. The situation becomes much more complex when the code is run in parallel. The effects discussed above remain, but they are combined with several others, such as load balance, cache utilization, and so on. Nevertheless, the dependence on k remains, in principle, the same (see Tables 5, 6, 7).

Consider, for each test matrix, the smallest computing time, obtained by using any number of processors. The corresponding number of blocks k, for which the smallest computing time is obtained, will be called *the best number of blocks* for that matrix. Usually, the best number of blocks increases with increasing the number of non-zeros in the matrix. For example, for the smallest of these matrices, amocol (Table 5), the best number of blocks is 32, for gemat1 (Table 6) it is 64, and 128 for bellmedt (Table 7), which has less rows and columns than gemat1, but more non-zero elements. In addition, bellmedt creates much more fill-ins during the factorization. It should be mentioned here that for amocol the smallest computing time is obtained on only 12 processors because this matrix is so small.

3.6 Speedup on SGI Power Challenge

The speedups for the best number of blocks for the three matrices from the previous paragraph, amocol, gemat1 and bellmedt, with $\tau = 0$, are presented in Fig. 4. If the matrix is sufficiently large, then the speedup grows when the number of processors is increased. In our experiment this is true for gemat1 and bellmedt, while, as mentioned in the previous paragraph, amocol is a rather small matrix for 16 processors.

Among the three test matrices, the largest speedup is achieved when bellmedt is run on 16 processors. It is seen that the speed-up is nearly linear for this matrix. This is because this matrix has the greatest density $\mathtt{NZ}/(mn)$. Larger density will normally lead to more expensive factorization process. Thus, the factorization process, which

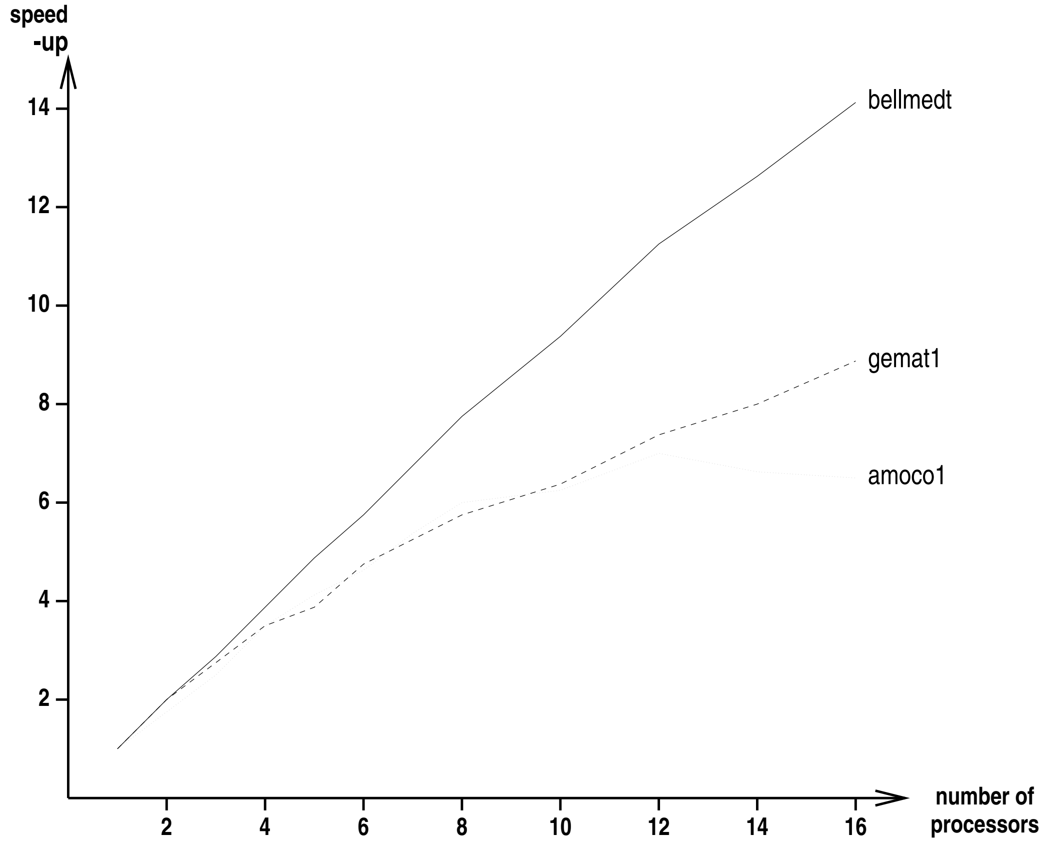

Figure 4: The speedup of SPARQR for solving directly (with $\tau = 0$) three test problems.

parallelizes best, is the most expensive part of the computational work for bellmedt, while the other parts of the computational work are not very essential for this matrix.

It should be noted, that the speed-ups will grow slower if the time on p processors is compared with the best time achieved (in general for some other value of k) on one processor. However, the pattern will remain the same. Furthermore, the speed-ups for the most time consuming matrix, bellmedt, will practically not change (see also Table 6).

Good speed-ups have also been achieved in [1] on an Alliant computer and in [23] on a Cray computer by using the multifrontal approach.

3.7 Using a large drop-tolerance

Results of parallel runs of our code with a large drop-tolerance, $\tau = 10^{-2}$, are presented in Tables 8 and 9.

For many matrices the solution of the problem obtained by using a large drop-tolerance is faster, and sometimes much faster (more than 25 times) in comparison with the direct method ($\tau = 0$), provided that an optimal value of the drop-tolerance is used. While finding the optimal drop-tolerance can be a difficult task, it is often rather easy to find a good value of τ, i.e. a value of the drop tolerance which gives an improvement of the computing time (compared with the case where $\tau = 0$ is used).

Table 8 shows the results for the matrix bellmedt with drop-tolerance 10^{-2}. The times are much better than the corresponding times for the direct method (see, for comparison, Table 7). The best number of blocks now is 64.

k	matrix bellmedt τ=10^{-2}				
	Time (*Speed-up*)				
	1 proc.	4 proc.	8 proc.	12 proc.	16 proc.
1	43.02				
8	40.43	16.50 (*2.5*)	14.38 (*2.8*)	16.72 (*2.4*)	15.89 (*2.5*)
16	35.74	12.94 (*2.8*)	9.17 (*3.9*)	9.07 (*3.9*)	9.73 (*3.7*)
32	33.53	10.22 (*3.3*)	6.91 (*4.9*)	5.44 (*6.2*)	5.95 (*5.6*)
64	<u>31.75</u>	<u>9.22</u> (*3.4*)	<u>5.79</u> (*5.5*)	<u>4.80</u> (*6.6*)	<u>5.65</u> (*5.6*)
128	32.43	9.60 (*3.4*)	5.89 (*5.5*)	5.49 (*5.9*)	6.48 (*5.0*)
256	36.84	11.83 (*3.1*)	7.18 (*5.1*)	7.57 (*4.9*)	7.90 (*4.6*)

Table 8: Computing times (in seconds) and speedups (shown in parenthesis) for the matrix bellmedt, with drop-tolerance $\tau = 10^{-2}$.

k	matrix matrf2 [300000 $\times$ 25000] , nonzeros: 1500110 ; τ=10^{-2}				
	Time (*Speed-up*)				
	1 proc.	4 proc.	8 proc.	12 proc.	16 proc.
1	107.21				
8	106.75	<u>34.18</u> (*3.1*)	22.83 (*4.7*)	22.67 (*4.7*)	22.66 (*4.7*)
16	<u>105.59</u>	34.47 (*3.1*)	22.57 (*4.7*)	21.35 (*4.9*)	16.82 (*6.3*)
32	106.79	34.57 (*3.1*)	<u>22.42</u> (*4.8*)	19.32 (*5.5*)	16.55 (*6.5*)
64	113.24	36.27 (*3.1*)	22.64 (*5.0*)	<u>18.83</u> (*6.0*)	<u>16.22</u> (*7.0*)
128	132.27	41.01 (*3.2*)	25.69 (*5.1*)	21.73 (*6.1*)	21.22 (*6.2*)
256	187.78	56.34 (*3.3*)	34.13 (*5.5*)	28.54 (*6.6*)	28.82 (*6.5*)

Table 9: Computing times (in seconds) and speedups (shown in parenthesis) for the randomly generated sparse matrix matrf2, with drop-tolerance $\tau = 10^{-2}$.

Table 9 contains the results for a matrix generated by the sparse matrix generator CLASSF, described in [30]. Although this matrix is much bigger than any of the Harwell-Boeing matrices, SPARQR solves this problem with large drop-tolerance, $\tau = 10^{-2}$, in a relatively short time. However, it must be mentioned that the computing times are increased very considerably when this matrix is run with $\tau = 0.0$; up to 10 times. As an illustration of this statement, let us mention that the computing time for $\tau = 0.0$ on 8 processors with 16 blocks is 194.12 sec.

If the accuracy requirement is relaxed (say, to 10^{-3}), then the number of iterations needed to achieve the accuracy required is normally decreased. If the accuracy requirement is made more stringent (say, 10^{-9}), then the number of iterations needed to achieve the accuracy required is normally increased. In both cases the computing time will normally not change too much. However, one should be careful. If the accuracy requirement is very high and/or if the matrix is ill-conditioned, then some difficulties may take place, because the stopping criteria will be affected by the rounding errors.

3.8 Comparing SPARQR with a multifrontal code

In the previous subsections it has been shown in the previous sub-sections that good speed-ups can be achieved when SPARQR is run on an SGI Power Challenge computer: see Table 7 and Fig. 4. However, a good speed-up is not sufficient. If, for example, a sequential code runs p times faster than SPARQR, then it will be better to use the sequential code when the number of processors is less than p. This shows that it is also necessary to show that SPARQR is at least comparable with a good sequential code. A multifrontal code, qr27 ([21], [22], [23]), has been chosen. This code has some parallel potentials, but on SGI Power Challenge it performs as a sequential code. An experiment with 20 matrices has been performed. The average number, r, of non-zero elements per row has been varied ($r = 5(5)100$). The results are given in Table 10.

If the matrix is very sparse, then qr27 is quite competitive (in fact, for $r < 5$ qr27 performs better than SPARQR). The situation changes dramatically when r is increased. It is clearly seen that the performance of SPARQR becomes much better that that of qr27 when the average number of non-zero elements per row is large.

We run SPARQR on 8 processors with partitioning into 16 blocks. The computing times can be further reduced either by increasing the number of processors or by trying to find the optimal blocking.

The speed-ups for 8 processors and 16 blocks were between 5 and 7. This means that the computing times will be multiplied by a factor between 5 and 7 if we run the code on one processor. The computing times for SPARQR will still be better for most of the matrices (and much better when the PCG option is used).

It must be emphasized here that much more experiments are necessary in order to determine the classes of matrices for which one of the two code should be preferred. Moreover, it may be possible to parallelize qr27 on SGI Silicon Graphics. However, the results in Table 10 give a very clear answer to the question stated in the beginning of this sub-section: for some classes of matrices SPARQR is at least not worse than a good sequential code.

4 Conclusions

SPARQR as a **coarse-grained parallel algorithm** reaches high speedup when the problem is sufficiently large. Starting parallel tasks requires a more or less constant overhead. Hence, with a fixed number of processors and blocks, the speedup increases with the size of the matrix. Furthermore, one should expect to obtain a good load balance when the number of blocks is greater than the number of processors. The larger the matrix, the more blocks of reasonable size can be created. Thus, there is a tendency to get better speedups for larger matrices. However, for general sparse matrices the situation is often rather complicated, and the efficiency will also depend on other parameter (e.g., the density of the non-zero elements in the original matrix, the amount of fill-in, the distribution of the non-zero elements, etc.).

r	qr27	Direct	PCG with $\tau = 0.01$
5	7.79	7.25	4.43 (9)
10	33.19	18.46	7.86 (9)
15	83.10	35.74	10.36 (9)
20	166.58	53.31	14.03 (10)
25	295.76	71.36	17.06 (10)
30	478.43	99.42	20.97 (10)
35	723.07	125.42	25.68 (11)
40	1027.13	165.16	29.10 (11)
45	1417.71	178.81	37.25 (11)
50	1920.86	246.84	39.16 (11)
55	2421.02	282.80	40.28 (12)
60	3022.68	335.90	42.49 (11)
65	3764.21	407.03	47.46 (12)
70	4585.82	461.37	47.66 (11)
75	5533.57	550.18	48.34 (12)
80	6613.16	697.38	55.69 (13)
85	7816.37	730.91	57.88 (12)
90	9161.99	772.64	60.99 (11)
95	10621.62	861.32	67.61 (16)
100	12189.31	949.31	62.51 (13)

Table 10: Computing times obtained by qr27 and two options of SPARQR when matrices with 50000 rows and 30000 columns are run. The average number of non-zeros per row, r, is varied in this experiment. The direct solution option and the PCG option of SPARQR were run on 8 processors with partitioning to 16 blocks. The number of iterations needed to achieve the accuracy required is given in brackets for the PCG option.

The number of blocks is essential. If the matrix is sufficiently large, then good performance can be achieved when the number of blocks is sp where p is the number of processors and s is a small constant (say, in the range 4–8).

In many cases a linear least squares problem is solved **faster when the drop-tolerance is larger**. Most of the well-conditioned matrices can be treated successfully with a drop-tolerance as large as 10^{-2}. In this case, the required accuracy can normally be achieved. Very ill-conditioned matrices, however, cannot be handled by using so large drop-tolerance (due to either removing too many non-zero elements during the factorization stage or problems with the convergence of the iterative stage). By gradually decreasing the value of the drop-tolerance, SPARQR is capable to obtain a good preconditioner and to solve successfully the problem, but this procedure might be more expensive than the solution with $\tau = 0$. Although it is difficult to find the optimal drop-tolerance, it is normally not difficult to find a good one, i.e., one that leads to an improvement of the computing time compared to that for $\tau = 0$.

The speedup for $\tau = 0$ is usually better than the speedup for large values of τ. This follows to a certain degree from the previous conclusions. The use of a large τ can dramatically reduce the factorization time for some matrices that create a lot of fill-ins. Dropping a big fraction of these fill-ins saves a lot of computing time in the factorization part (which parallelizes very well), while the savings in the other parts of the program are moderate. Another reason for the lower speedup when larger τ is used is the larger number of iterations performed in Stage 5. An example, which illustrates the above statements, is the matrix bellmedt, which requires a lot of time to be factorized completely, although the speedup is nearly optimal. A large τ makes the task much easier and although the speedup is not so impressive, the solution process is still more than 10 times faster on 16 processors.

Acknowledgements

This research was partially supported by BRA III Esprit project APPARC (# 6634), by research grant 9500764 from the Danish Natural Science Research Council and by grant I-505/95 from the Bulgarian Ministry of Science and Education.

References

[1] P. R. Amestoy, I. S. Duff and C. Puglisi, *Multifrontal QR-factorization in a multiprocessor environment*, Lin. Alg. with Appl., **Vol. 3, No. 4 (1996)**, 275-300.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen, *LAPACK: Users' guide*. SIAM (Society for Industrial and Applied Mathematics), Philadelphia, **1992**

[3] Å. Björck, *Stability analysis of the method of semi-normal equations for least squares problems*, Lin. Alg. Appl., Vol. 88/89 **1987**, pp. 31–48.

[4] Å. Björck, *Least squares methods*. In *Handbook of Numerical Analysis*, Vol.1: *Finite Difference Methods – Solution of Equations in R^n*, (P. G. Ciarlet and J. L. Lions, eds.). North-Holland, Amsterdam, **1990**.

[5] T. F. Chan, *Rank revealing QR factorizations*, Lin. Alg. Appl., Vol. 88/89 **1987**, pp. 67-82.

[6] J. J. Dongarra, J. R. Bunch, C. B. Moler and G. W. Stuwart, *LINPACk: Users' guide*, SIAM (Society for Industrial and Applied Mathematics), Philadelphia, **1979**.

[7] I. S. Duff, A. M. Erisman and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford-London, **1986**.

[8] I. S. Duff, R. G. Grimes and J. G. Lewis, *Sparse matrix test problems*, ACM Trans. Math. Software, Vol. 15, **1989**, pp. 1–14.

[9] A. C. N. van Duin, P. C. Hansen, Tz. Ostromsky, H. Wijshoff and Z. Zlatev, *Improving the numerical stability and the performance of a parallel sparse solver*, Computers and Mathematics with Applications, Vol.30, No.12, **1995**, pp. 81–96.

[10] L. V. Foster, *Rank and null space calculations using matrix decomposition without column interchanges*, Lin. Alg. Appl., Vol. 74 **1986**, pp. 47-71.

[11] K. A. Gallivan, P. C. Hansen, Tz. Ostromsky and Z. Zlatev, *A locally optimal reordering algorithm and its application to a parallel sparse linear system solver*, Computing, Vol.54, No.1, **1995**, pp. 39–67.

[12] W. M. Gentleman, *Least squares computations by Givens transformations without square roots*, J. Inst. Math. Appl., Vol. 12, **1973**, pp. 329–336.

[13] J. A. George and M. T. Heath, *Solution of sparse linear least squares problems using Givens rotations*, Lin. Alg. Appl., Vol. 34, **1980**, pp. 69–73.

[14] J. A. George, M. T. Heath and E. G. Y. Ng, *A comparison of some methods for solving sparse linear least squares problems*, SIAM J. Sci. Stat. Comput., Vol. 4, **1983**, pp. 177–187.

[15] J. A. George and E. G. Y. Ng, *SPARSPAK: Waterloo sparse matrix package user's guide for SPARSPAK-B*, Research Report CS-84-47, Department of Computer Science, University of Waterloo, Ontario, **1984**.

[16] J. W. Givens, *Computation of plane unitary rotations transforming a general matrix to a triangular form*, J. Soc. Ind. Appl. Math., Vol. 6, **1958**, pp. 26–50.

[17] G. H. Golub, *Numerical methods for solving linear least squares problems*, Numer. Math., Vol. 7, **1965**, pp. 206-216.

[18] G. H. Golub and C. F. Van Loan, *Matrix computations*, 2. Ed., John Hopkins University Press, Baltimore, Maryland, USA, **1989**.

[19] P. C. Hansen, Tz. Ostromsky, A. Sameh and Z. Zlatev, *Solving Sparse Linear Least Squares Problems on Some Supercomputers by Using a Sequence of Large Dense Blocks*, BIT, Vol. 37, **1997**, 535-558.

[20] A. S. Householder, *Unitary triangularization of a nonsymmetric matrix*, J. Assoc. Comput. Mach., Vol. 5, **1958**, pp. 339–342.

[21] P. Matstoms, *The Multifrontal Solution of Sparse Linear Least Squares Problems*, LIU-TEK-LIC-1991:33, Department of Mathematics, Linköping University, Linköping, Sweden, **1991**.

[22] P. Matstoms, *Sparse QR factorization with applications to linear least squares problems*, Ph D Thesis, Linköping University, Linköping, Sweden, **1994**.

[23] P. Matstoms, *Parallel sparse QR factorization on shared memory architectures*, Parallel Computing, **Vol. 21 (1995)**, 473-486.

[24] NAG LIBRARY FORTRAN MANUAL. *Mark 7*, Vol. 3-4. Numerical Algorithms Group, Banbury Road 7, Oxford, England, **1979**.

[25] D. J. Pierce and J. G. Lewis, *Sparse rank revealing QR factorizations*, Technical Report MEA-TR-193, Boeing Computer Services, May **1992**.

[26] M. J. D. Powell and J. K. Reid, *On applying Householder transformations to linear least squares problems*, In: "Information Processing" (A. J. H. Morrell, ed.), pp. 122-126. North-Holland, Amsterdam, 1969.

[27] C. Puglisi, *QR-factorization of large sparse overdetermined and square matrices using the multifrontal method in a multiprocessor environment*, Report TR/PA/93/33, CERFACS, **1993**.

[28] J. R. Rice, *PARVEC workshop on very large least squares problems and supercomputers*, Report CSD–TR 464, Purdue University, West Lafayette, IN, **1983**.

[29] G. W. Stewart, *A parallel implementation of the QR-algorithm*, Parallel Computing, Vol. 5, **1987**, pp.187–196.

[30] Z. Zlatev, *Computational Methods for General Sparse Matrices*, Kluwer Academic Publishers, Dordrecht-Toronto-London, **1991**.