A Block QR Factorization Algorithm Using Restricted Pivoting*†

Christian H. Bischof

Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439–4801
(312) 972-8875
bischof@mcs.anl.gov

CONF-891149--8

DE90 001921

April 29, 1989

Keywords: QR factorization, block algorithm, restricted pivoting, incremental condition estimation, high-performance computers.

Abstract. This paper presents a new algorithm for computing the QR factorization of a rank-deficient matrix on high-performance machines. The algorithm is based on the Householder QR factorization algorithm with column pivoting. The traditional pivoting strategy is not well suited for machines with a memory hierarchy since it precludes the use of matrix-matrix operations. However, matrix-matrix operations perform better on those machines than matrix-vector or vector-vector operations since they involve significantly less data movement per floating point operation. We suggest a restricted pivoting strategy which allows us to formulate a block QR factorization algorithm where the bulk of the work is in matrix-matrix operations. Incremental condition estimation is used to ensure the reliability of the restricted pivoting scheme. Implementation results on the Cray 2, Cray X-MP and Cray-Y-MP show that the new algorithm performs significantly better than the traditional scheme and can more than halve the cost of computing the QR factorization.

*This work was supported by NSF Grant ASC-8715728 and by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy under contract W-31-109-Eng-38.
†Submitted to Supercomputing '89.

---

1 Introduction

On most high-performance architectures memory access is quite slow compared to floating-point (in particular, vector) performance. To overcome this problem, most of today's high-performance machines incorporate a memory hierarchy (such as global memory, cache or local memory, and vector registers). For an overview of high-performance architectures employing memory hierarchies, see [21,30,37]. Data at low levels of the memory hierarchy can be accessed without delay, whereas data at higher levels is available only after some delay and (because of memory bank conflicts) may not be available at a rate fast enough to feed the arithmetic units. For this reason it is imperative to reuse data as much as possible to cut down on data movement overhead.

This goal can be achieved by expressing a computation in terms of matrix-matrix operations. If the matrices involved are of order n, matrix-matrix operations such as matrix-matrix multiplication require $O(n^3)$ floating-point operations; with proper implementation, however, the data movement overhead is only $O(n^2)$. In contrast, the order of magnitude of floating point operations and data movement overhead is the same for vector-vector or matrix-vector operations: $O(n)$ and $O(n^2)$, respectively. Hence, using matrix-matrix operations, we avoid excessive movement of data to and from memory and achieve a *surface-to-volume effect* for the ratio of operations to data movement. The Level 3 BLAS [18] provide the matrix-matrix operations needed for linear algebra. Together with the Level 1 and 2 BLAS [20,31] implementing vector-vector and matrix-vector operations, respectively, they provide a well-defined interface for the elementary matrix and vector operations.

In order to arrive at an algorithm rich in matrix-matrix operations, one usually must express the algorithm at the top level in terms of operations on submatrices (the so-called "blocks"). Block algorithms have been very successful on high-performance machines (for some examples see [2, 3, 6, 9, 11, 18, 22, 24, 35, 34]). As a result block algorithms will play an important role in the LAPACK project [4,10,15] that is currently underway to provide the functionality of EISPACK [25,36] and LINPACK [16] with algorithms better suited for today's high-performance architectures.

In this paper we develop an algorithm for computing the QR factorization

$$AP = QR \tag{1}$$

of a rank-deficient $m \times n$ matrix A. Here P is an $n \times n$ permutation matrix, Q is an $m \times m$

matrix orthogonal matrix, and R is an upper triangular $m \times n$ matrix. This factorization is typically used when we have to identify a basis for the range space of the columns of A. It arises for example as the so-called subset selection problem in statistics [26, 28] to identify redundant carriers in a linear model. Other applications are the solution of underdetermined or rank-deficient least-squares problems [1,28,32] and nullspace methods in optimization [14].

The goal is to find a permutation matrix P and a number of columns r such that R can be partitioned into

$$R = \left(\begin{array}{cc} R_1 & , \; R_2 \end{array} \right) \tag{2}$$

where the upper triangular $m \times r$ matrix R_1 is well-conditioned, but

$$\left(\begin{array}{cc} R_1 & , \; \tilde{R}_2 \end{array} \right)$$

is ill-conditioned for any subset $\tilde{R}_2$ of columns of R_2. This implies that the first r columns of AP are a basis for the range space $R(A)$ of A and hence the first r columns of Q are an orthogonal basis for $R(A)$. We mention that if $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{\min(m,n)}$ are the singular values of A, then a well-defined gap between σ_r and σ_{r+1} is necessary to find a sensible partition (2) [26].

The standard technique for determining such a factorization for a dense matrix A is the Householder QR factorization with column pivoting [12,28]. This algorithm is reliable in practice, but is computationally inherently limited to matrix-vector kernels. On the other hand, when A is of full rank and no pivoting is necessary (i.e. $P = I$ in (1)), block algorithms for the QR factorization can be designed by bundling a series of Householder updates using the so-called WY representation [11,35].

We suggest a new algorithm that combines the reliability of the column pivoting scheme with the computational advantages of block algorithms. By limiting our choice of pivot columns to a given block of the current matrix, we can delay updating remaining columns until a suitable block transformation has been computed. In order to make this strategy reliable, we use incremental condition estimation [7] to assess the effect that the selection of a pivot column would have on the condition number of the current triangular matrix R_1.

The outline of the paper is as follows: In Section 2 we briefly review the traditional Householder QR factorization algorithm with column pivoting. Section 3 review the WY

representation for products of Householder matrices. The next section introduces the restricted pivoting strategy and shows how incremental condition estimation is used to ensure its numerical reliability. Section 5 presents some implementation results on the Cray 2, Cray X-MP and Cray Y-MP. These results show that the new algorithm performs significantly better than the traditional scheme. Lastly we summarize our contributions and outline directions of future research.

2 The Householder QR Factorization Algorithm with Traditional Column Pivoting

The traditional technique for computing a QR factorization of a rank-deficient matrix is the Householder QR factorization with column pivoting [12,27]. Here Q is computed by a sequence of *Householder transformations*

$$H \equiv H(u) = I - 2uu^T, \|u\|_2 = 1. \tag{3}$$

Choosing

$$u = \frac{x + \text{sign}(x_1)\,\|x\|_2\,e_1}{\|x + \text{sign}(x_1)\,\|x\|_2\,e_1\|_2}, \tag{4}$$

we can reduce a given vector x to a multiple of the canonical unit vector e_1, since

$$(I - 2u\,u^T)\,x = -\,\text{sign}(x_1)\,\|x\|_2\,e_1.$$

To describe the Householder QR factorization algorithm we use the primitives *genhh* (generate Householder vector) and *apphh* (apply Householder matrix):

$$[u, y] \leftarrow genhh(x)$$

returns u as defined by (4) and $y = H(u)\,x$.

$$B \leftarrow apphh(u, A)$$

returns $H(u)\,A$.

Figure 1 describes the Householder QR factorization algorithm with traditional pivoting for computing the QR decomposition of an $m \times n$ matrix A. Here $a(i:j, k:l)$ refers to the submatrix of A consisting of row entries i to j and column entries k to l. A colon

```
foreach i ∈ {1, ..., n} do
   perm_i = i ; res_i = ||a(:,i)||_2
end foreach
for i = 1 to min(m,n) do
   Let pvt ∈ {i, ..., n} be such that res_pvt is maximal
   if (res_pvt < threshold) then
      break  { A has numerical rank i − 1 }
   else   { exchange columns pvt and i }
      perm_i ↔ perm_pvt ; a(:,i) ↔ a(:,pvt) ; res_pvt ← res_i ;
      [u_i, a(i:m,i)] ← genhh(a(i:m,i)) ;
         { apply H(u) and update residuals }
      a(i:m,i+1:n) ← apphh(u_i, a(i:m,i+1:n)) ;
      foreach j ∈ {i + 1, ..., n} do
         res_j ← sqrt(res_j^2 − a(i,j)^2) ;
      end foreach
   end if
end for
```

Figure 1: The QR Factorization Algorithm with Traditional Column Pivoting

(:) is used as shorthand to design a complete row or column. The vector *perm* is used to store the permutation matrix P. If $perm(i) = k$, then the kth column of A is permuted into the ith column of AP. After completing step i the values $res_j, j = i+1, \ldots, n$ are the length of the projections of the jth column of the currently permuted AP onto the orthogonal complement of the subspace spanned by the first i columns of AP. The pivoting strategy can be viewed as choosing at every step this column that is farthest away (in the two-norm sense) from the subspace spanned by the columns that were selected before [28, p.168, P.6.4-5]. Hence, if res_j is small for all $j > i$, then we can consider A to have rank i. Which *threshold* we choose for termination depends heavily on the application, but in general the computation will be terminated if the distance of the next pivot column from the already chosen subspace is $O(1/\epsilon)$ where ϵ is the machine precision. res_j can be easily updated and does not have to be recomputed at every step although roundoff errors may make it necessary to recompute $res_j = \|(a(i:m,j))\|_2, j = i+1, \ldots, n$ periodically [16, p. 9.17] (we suppressed this detail in Figure 1). Alternative pivoting strategies have been suggested by Chan [13] and Foster [23].

The bulk of the computational work in this algorithm is performed in the *apphh* kernel. Computing $B \leftarrow apphh(u, A)$ involves a matrix-vector product

$$z \leftarrow A^T u$$

and a rank-one update

$$B \leftarrow A - 2uz^T .$$

These operations require the same amount of data movement as floating point operations. To arrive at a block algorithm relying on matrix-matrix operations, it is necessary to avoid updating part of A until several Householder transformations have been computed. This is impossible to do for the traditional pivoting strategy, since we must update $a(i:m,j)$ and res_j before we can choose the next pivot column.

3 A Block Orthogonal Transformation

To arrive at a block formulation of the Householder QR algorithm, it is necessary to express a series of Householder reductions in a convenient closed form. Bischof and Van Loan [11] expressed the product

$$Q = H_1 \cdots H_{nb}$$

of a series of $m \times m$ Householder matrices (3) in the so-called *WY representation*

$$Q = I + WY^T \tag{5}$$

where W and Y are $m \times nb$ matrices. Schreiber and Van Loan [35] refined this representation by expressing $W = YT$ where T is a $nb \times nb$ upper triangular matrix. Schreiber and Van Loan called the resulting representation

$$Q = I + YTY^T \tag{6}$$

the *compact WY representation* since it requires only about half as much storage as the original WY representation (5) in the typical case where $m \gg nb$. To accumulate Y and T, observe that a Householder matrix is a special case of the compact WY representation and that we can write

$$\tilde{Q} \equiv QH = I + \tilde{Y}\tilde{T}\tilde{Y}^T$$

where

$$\begin{aligned} \tilde{Y} &= \begin{pmatrix} Y & , & u \end{pmatrix} \\ z &= -2TY^T v \\ \tilde{T} &= \begin{pmatrix} T & z \\ 0 & -2 \end{pmatrix}. \end{aligned} \tag{7}$$

Y is simply the collection of Householder vectors and in most applications where the dimension of Householder vectors decreases at every step Y will be lower trapezoidal. Compared to the traditional Householder algorithm the accumulation of T requires $O(mnb^2)$ extra flops and $\frac{nb^2}{2}$ extra words for storage. Since typically $m \gg nb$ this is a low-order term in the overall algorithmic complexity. The advantage of the compact WY representation is that the computation of $A \leftarrow Q^T A$ now involves two matrix-matrix multiplications

$$Z \leftarrow A^T YT \tag{8}$$

and a rank-nb update

$$A \leftarrow A + YZ^T \tag{9}$$

instead of a series of nb matrix-vector multiplications and rank-one updates. Although we are performing roughly the same amount of floating point computation, the data movement overhead has been reduced by a factor of nb.

We can now express the block Householder QR algorithm for computing a QR factorization $A = QR$ without pivoting in terms of the primitives *generate_Y* (compute Householder vectors), *accumulate_T* (generate compact WY factor), and *appcwy* (apply compact WY factor):

$$[Y, R] \leftarrow generate\_Y(A)$$

for an $m \times nb$ matrix A returns the Householder vectors in Y such that

$$H(Y)^T = H(Y(:,1)) \cdots H(Y(:,nb))^T A = R$$

using the traditional Householder QR algorithm without pivoting. This algorithm is obtained from Figure 1 by deleting all references to *res* and *pvt* and processing columns 1 through n of A in their natural order.

$$T \leftarrow accumulate\_T(Y)$$

accumulates the Householder vectors Y into a compact WY update such that

$$(I - YTY^T) = H(Y)$$

as described in (7).

$$A \leftarrow appcwy(Y, T, A)$$

performs the updates (8) and (9). Figure 2 shows the block Householder algorithm using the compact WY representation. Here A is partitioned as an $M \times N$ block matrix, and for simplicity we assume that all blocks are of the same size $mb \times nb$, so $m = Mmb$ and $n = Nnb$. We use the notation $A(i,j)$ to refer to block entry (i,j) and $A(i:j, k:l)$ to refer to the submatrix of A consisting of block row entries i to j and block column entries k to l.

Bischof and Van Loan [11], Harrod [29] and Mayes [33] used the WY representation to compute the QR factorization without pivoting on the FPS-164/MAX, the Alliant FX/8 and the IBM 3090, respectively. This algorithm is also currently implemented in the LAPACK package.

```
for i = 1 to N do
    Y ← generate_Y(A(i:M, i))
    T ← accumulate_T(Y)
    A(i:M, i:N) ← appcwy(Y, T, A(i:M, i:N))
end for
```

Figure 2: The Block QR Factorization Algorithm without Pivoting

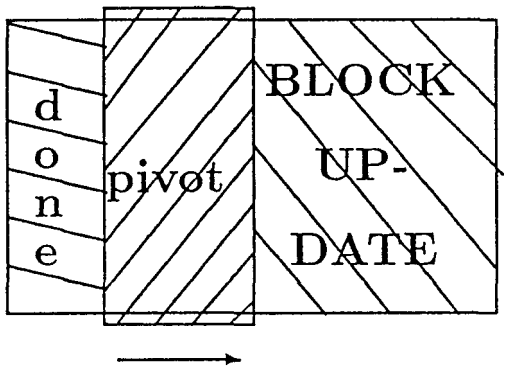

Figure 3: Restricting Pivoting for a Block Algorithm

4 Restricted Pivoting

We already mentioned that the traditional pivoting strategy prevents a block algorithm since it requires the update of all remaining columns at every step. To arrive at a block algorithm, we have to restrict pivoting: When we limit our choice of pivot columns, we do not have to update the remaining columns until we have computed enough Householder transformations to make a block update worthwhile.

The idea is graphically depicted in Figure 3: At a given stage we are done with the columns to the left of the pivot window. We then try to select the next pivot columns exclusively from the columns in the pivot window, not touching the part of A to the right of the pivot window. Only when we have combined the Householder vectors defined by the

next batch of pivot columns into a WY update, we apply this block update to the columns on the right.

We must however be able to guard against pivot columns that are in the span of columns already selected. That is, given the upper triangular matrix R_i defined by the first i columns of $Q^T AP$ and a new column $\begin{pmatrix} v \\ \gamma \end{pmatrix}$ determined by the new candidate pivot column, we must decide whether

$$R_{i+1} = \begin{pmatrix} R_i & v \\ 0 & \gamma \end{pmatrix}$$

is still of full rank. Since the smallest singular value $\sigma_{min}(A)$ of a matrix A measures the distance of A (in the two-norm sense) from the set of rank-deficient matrices [28, p. 19] it is natural to use $\sigma_{min}(R_{i+1})$ to decide whether to accept the new column. Computing $\sigma_{min}(R_{i+1})$ exactly is too expensive, but using incremental condition estimation [7] we can obtain a good estimate for $\sigma_{min}(R_{i+1})$ cheaply.

Given a good estimate $\hat{\sigma}_{min}(R_i) = 1/\|x\|_2$ defined by a large norm solution x to $R_i^T x = d, \|d\|_2 = 1$ and a new column $\begin{pmatrix} v \\ \gamma \end{pmatrix}$, incremental condition estimation allows us to obtain an estimate for $\sigma_{min}(R_{i+1})$ without accessing R_i again. Definining

$$\alpha = v^T x \ , \beta = \gamma^2 x^T x + \alpha^2 - 1 \ , \eta = \beta/(2\alpha) \text{ and } \mu = \alpha(\eta + \text{sign}(\alpha)\sqrt{\eta^2 + 1}),$$

the estimate for the smallest singular value of R_{i+1} is given by

$$\hat{\sigma}_{min}(R_{i+1}) = \frac{1}{\|z\|_2} \tag{10}$$

where

$$z = \begin{pmatrix} sx \\ (c - s\alpha)/\gamma \end{pmatrix} \tag{11}$$

and

$$s = \frac{\mu}{\sqrt{\mu^2 + 1}} \ , c = \frac{-1}{\sqrt{\mu^2 + 1}}. \tag{12}$$

The cost of determining $\hat{\sigma}_{min}(R_{i+1})$ is $3i$ flops for the inner product $v^T x$ and the scaling of x by c. Numerical experiments with this conditon estimation scheme [7] show that it is reliable in producing good estimates.

With the incremental condition estimator we now have the tool to ensure the reliability of the restricted pivoting strategy. By applying the incremental condition estimator to a candidate pivot column, we can cheaply decide whether this column is nearly dependent on the space spanned by the columns already chosen. This leads to the algorithm that is shown in simplified form in Figures 4 and 5. The outer loop steps through the block columns of A, whereas the inner loop tries to identify reasonable pivot columns in the current pivot window. k indicates the start of the pivot window, i.e. columns $1 : k-1$ are "done" in Figure 3. Columns that have been rejected as pivot columns are permuted to the end of the matrix and will never be part of a pivot window again. The rejected columns are in columns $sr : n$ ($sr \Leftrightarrow$ "start_rejected"). Columns $1 : acc$ ($acc \Leftrightarrow$ "accepted") have been accepted as pivot columns so far. Since the LAPACK implementation of the blocked algorithm of Figure 2 uses blocks of fixed width nb (except possibly for the last block) we may in the last pass through the inner loop compute Householder vectors based on both accepted and rejected columns. This was omitted for simplicity in Figure 5. kb is the width of the current block. $lacc$($\Leftrightarrow$ "locally accepted") is the number of columns that has been accepted in the current pivot window, whereas nrj($\Leftrightarrow$ "number rejected") is the number of columns rejected in the current pivot window. At any point, columns $k + lacc : sl$ ($sl \Leftrightarrow$ "search limit") are the columns available as pivot candidates, whereas columns $k + lacc : ul$ ($ul \Leftrightarrow$ "update limit") are the columns to the right of the accepted columns in the current pivot window. The reason that the update window is larger than the search window is that it may contain rejected columns that obviously need not be reconsidered as pivot candidates. If we run out of acceptable pivot columns in the current pivot window, we expand it to include columns $ul : nl$ ($nl \Leftrightarrow$ "new limit") and then try to find acceptable columns in the expanded window.

The subroutine

$$[\, \|z\|_2, s, c, \alpha] \leftarrow \text{cond\_est}(x, \|x\|_2, v, \gamma)$$

returns $\alpha = v^T x$, s and c as defined in (12) and $\|z\|_2$ with z defined as in (11). The incremental condition estimation procedure is inexpensive. $v = a(1 : k + lacc - 1, pvt)$ has already been computed and $\gamma = -\text{sign}(a(k + lacc, pvt))res(pvt)$ is also readily available. A call to cond_est costs $2(k + lacc)$ flops to compute, and only when we decide to accept the candidate pivot column, we have to update x at the cost of another $k + lacc$ flops. It is also important to note that incremental condition estimation is necessary for the reliability of the

$k \leftarrow 1$; $srj \leftarrow n+1$; $acc \leftarrow 0$;
$perm(i) \leftarrow i$, $i = 1, \ldots, n$
while $((k < sr)$ **and** $(k \leq \min(m,n)))$ **do**
$kb \leftarrow \min(\min(m,n) - k + 1, nb)$; $lacc \leftarrow 0$; $nrj \leftarrow 0$; $sl \leftarrow k + kb$; $ul \leftarrow k + kb$;
$res(i) \leftarrow \|a(k:m,i)\|_2$, $i = k, \ldots, k + kb - 1$
{ Find kb acceptable columns in the pivot window, starting with a pivot window of size kb. }
while $(lacc < kb)$ **do**
$pvt \leftarrow \{i \mid res_i = max_{k+lacc \leq j \leq sl} res_j\}$
$\gamma \leftarrow -\text{sign}(a(k + lacc, pvt))\, res(pvt)$;
$[\, \|z\|_2 \,, s \,, c \,, \alpha] \leftarrow$ cond_est$(x,\ \|x\|_2,\ a(1 : k + lacc - 1, pvt)\,, \gamma)$
see Figure 5
end while
{ accumulate block Householder transformation based on accepted pivot columns and apply them to columns to the right of pivot window}
$Y \leftarrow [y_1, \ldots, y_{kb}]$;
{compute block transformation determined by Y and apply it to } $T \leftarrow accumulate\_T(Y)$;
$a(k : m, k + kb : n) \leftarrow appcwy(Y,\ T,\ a(k : m, k + kb : n)$
{ move rejected columns to the end if there is space }
$ti \leftarrow sr$;
for $i = \max(ul - nrj, k + kb)$ **to** $\min(ul - 1, sr - nrj - 1)$ **do**
$ti \leftarrow ti - 1$;
$a(:,i) \leftrightarrow a(:,ti)$; $perm(:,i) \leftrightarrow perm(:,ti)$;
end for
$srj \leftarrow srj - nrj$; $acc \leftarrow acc + \min(kb, ul - k - nrj)$;
$k \leftarrow k + kb$;
end while
Compute QR factorization without pivoting of $a(k : m, k : n)$ using the algorithm of Figure 2. Columns $k : n$ are linearly dependent on the other columns.

Figure 4: The Householder Block QR Factorization algorithm with Restricted Pivoting: Top Level Loop.

if $(1/\|z\|_2 > threshold)$ **then**

{ candidate pivot column is acceptable. Exchange columns $k + lacc$ and pvt. }

$ti \leftarrow k + lacc$; $lacc \leftarrow lacc + 1$;

$perm(ti) \leftrightarrow perm(pvt)$; $a(:, ti) \leftrightarrow a(:, pvt)$;

$res(pvt) \leftarrow res(ti)$;

{ generate new Householder vector }

$[y_{lacc},\ a(ti : m, ti)] \leftarrow genhh(a(ti : m, ti))$;

{ apply Householder vector to leftover columns in pivot window }

$a(ti : m, ti + 1 : ul) \leftarrow apphh(y_{lacc}, a(ti : m, ti + 1 : ul))$;

{ update residuals and approximate singular vector }

$res(j) \leftarrow \sqrt{res(j)^2 - a(ti, j)^2},\ j = ti + 1, \ldots, ul$

$$x \leftarrow \begin{pmatrix} sx \\ \frac{c - s\alpha}{\gamma} \end{pmatrix};$$

else

{ all remaining columns in the pivot window have been rejected.
Expand the size of the pivot window. }

$nl \leftarrow \min(sr - 1, ul + kb)$; $nrj \leftarrow nrj + (sl - lacc)$;

{ apply Householder transformations determined by $y_1, \ldots, y_{lacc}$
to expanded pivoting window }

for $i = 1$ **to** $lacc$ **do**

$a(k : m, ul + 1 : nl) \leftarrow apphh(y_i,\ a(k : m, ul + 1 : nl))$;

end for

{ move rejected columns to the end of the expanded pivot window }

for $i = k + lacc + 1$ **to** $\min(ul, k + lacc + nl - ul)$ **do**

$a(:, i) \leftrightarrow a(:, nl + k + lacc - i)$; $perm(i) \leftrightarrow perm(nl + k + lacc - i)$;

end for

{ initialize residuals for new columns to be considered }

$res(i) \leftarrow \|a(k + lacc + 1, i)\|_2,\ i = k + lacc + 1, \ldots, nl$

$sl \leftarrow lacc + nl - ul$; $ul \leftarrow nl$;

end if

Figure 5: The Householder Block QR Factorization algorithm with Restricted Pivoting: Guarding pivot choice with incremental condition estimation and expansion of pivot window.

algorithm. If we were to use threshold pivoting, we would use $| \gamma |$ alone in deciding whether to accept or reject a pivot column and ignore v's contribution. In [8], Bischof developed a similar controlled pivoting scheme for computing a rank-revealing QR factorization on a distributed-memory machine and observed that threshold pivoting fails to identify the rank of A whereas incremental condition estimation produces reliable results.

5 Numerical Experiments

To assess the computational performance or our proposed scheme, we performed some experiments on the Cray 2, Cray X-MP and Cray Y-MP. We compared the performance of three different codes:

SQRDC: The LINPACK implementation of the Householder QR factorization algorithm with traditional column pivoting (see Figure 1). This implementation uses the BLAS 1 vector-vector kernels.

SQRDC2: This is the same algorithm as in SQRDC except that it uses BLAS 2 matrix-vector kernels whenever possible.

SGEQRF: The block Householder QR factorization algorithm without pivoting (see Figure 2) as currently implemented in LAPACK.

SGEQPF: The new block Householder QR factorization algorithm with restricted pivoting as described in Figures 4 and 5.

All codes were written in Fortran and linked with optimized versions of the level 1, 2 and 3 BLAS provided by Cray Research. As test case, we generated random matrices with $m = 400$ rows and $n = 200, 400, 600, 800$ columns with singular values

$$\sigma_1 = \ldots = \sigma_{\min(m,n)-10} = 1 \text{ and } \sigma_{\min(m,n)-9} = \ldots = \sigma_{\min(m,n)} = 1e-9.$$

Hence the rank of those matrices is $\min(m,n) - 9$. We used $threshold = 1e-7$ as our cutoff point for accepting or rejecting columns. As expected, the traditional as well as the restricted pivoting strategy correctly identified the rank of A and generated a well-conditioned R_1 in (2).

The run times observed for this problem are shown in Table 1, the corresponding execution rates are shown in Table 2. We want to stress that these performance results were

Table 1: Execution time (in seconds) of the QR factorization codes on a problem with $m = 400$ rows on one processor of a Cray-2, Cray X/MP and Cray Y/MP.

Cray-2					
n		200	400	600	800
SQRDC		0.56	2.17	3.85	5.44
SQRDC2		0.27	0.94	1.66	2.28
SGEQPF	$nb = 16$	0.191	0.47	0.75	1.02
	$nb = 32$	0.182	0.48	0.75	1.05
SGEQRF	$nb = 16$	0.167	0.49	0.71	0.95
	$nb = 32$	0.175	0.46	0.69	0.99

Cray X/MP					
n		200	400	600	800
SQRDC		0.33	1.24	2.33	3.40
SQRDC2		0.201	0.66	1.17	1.75
SGEQPF	$nb = 16$	0.174	0.50	0.84	1.12
	$nb = 32$	0.181	0.52	0.86	1.12
SGEQRF	$nb = 16$	1.160	0.48	0.81	1.14
	$nb = 32$	0.164	0.48	0.82	1.16

Cray Y/MP					
n		200	400	600	800
SQRDC		0.27	1.18	1.97	2.82
SQRDC2		0.154	0.51	0.97	1.43
SGEQPF	$nb = 16$	0.147	0.42	0.67	0.93
	$nb = 32$	0.141	0.40	0.73	0.92
SGEQRF	$nb = 16$	0.137	0.37	0.68	0.88
	$nb = 32$	0.126	0.37	0.63	0.92

Table 2: Execution rate (in MFlops) of the QR factorization codes on a problem with $m = 400$ rows on one processor of a Cray-2, Cray X/MP and Cray Y/MP.

Cray-2					
n		200	400	600	800
SQRDC		48	40	39	40
SQRDC2		97	92	90	94
SGEQPF	$nb = 16$	146	186	202	211
	$nb = 32$	158	184	203	206
SGEQRF	$nb = 16$	166	179	213	227
	$nb = 32$	164	193	220	217

Cray X/MP					
n		200	400	600	800
SQRDC		80	69	64	62
SQRDC2		133	130	128	123
SGEQPF	$nb = 16$	160	173	179	182
	$nb = 32$	159	171	177	180
SGEQRF	$nb = 16$	173	182	186	188
	$nb = 32$	175	184	185	186

Cray Y/MP					
n		200	400	600	800
SQRDC		99	73	76	76
SQRDC2		174	167	155	150
SGEQPF	$nb = 16$	189	205	227	234
	$nb = 32$	205	221	210	235
SGEQRF	$nb = 16$	203	234	222	246
	$nb = 32$	228	239	243	235

not obtained on a dedicated system. Although the average user will in all likelyhood be in a similar situation, these results do not represent the best possible performance of these codes and are subject to fluctuations depending on the current load of the system. In order to arrive at relevant performance results, each problem was executed as many times as was necessary to achieve an aggregate CPU time of 3 seconds. The figures reported here represent the average performance over these runs.

Even allowing for some inaccuracy in the observed performance figures, we can however make the following qualitative statements:

1. SGEQPF performs significantly better then SQRDC2.

2. SGEQPF performs only little worse than SGEQRF. That is, pivoting does not introduce much extra overhead.

We also see again (see [17,19] for example), that LINPACK performance can be improved dramatically by replacing loops with BLAS 1 calls with the corresponding BLAS 2 kernel. Table 3 shows in detail by which percentage of the execution time SQRDC2 performs worse than SGEQPF. The improvements of SGEQPF over SQRDC2 are greatest on the Cray-2. This is not surprising, since each processor on this machine has only one path to memory and a relatively long memory cycle time [21]. On the Cray X/MP and Y/MP, each processor has two load-pipes and one store-pipe into memory and as a result the improvements on those machines are not as spectacular. Nonetheless, SGEQPF performs significantly better than SQRDC2.

The choice of block sizes $nb = 16$ and $nb = 32$ was guided by previous experience with SGEQRF. As can be seen, there is some fluctuation in performance, but it is not dramatic on those machines. In general, there are some subtle issues involved in choosing the optimal block size. The block partitioning resulting in the fastest execution of the code (the "optimal" block partitioning) is problem-dependent (we can use larger blocks for larger matrices), but it is also depends on the architecture of a given machine. A discussion of these issues and a suggestion for a methodology to overcome this problem can be found in [5].

Table 3: Percentage by which the execution time of SQRDC2 is worse than that of SGEQPF

Cray-2				
n	200	400	600	800
$nb = 16$	41	100	121	124
$nb = 32$	48	96	121	117

Cray X/MP				
n	200	400	600	800
$nb = 16$	16	32	39	56
$nb = 32$	11	27	36	56

Cray Y/MP				
n	200	400	600	800
$nb = 16$	5	21	45	54
$nb = 32$	9	27	33	54

6 Conclusions

We presented a new algorithm for computing the QR factorization of rank-deficient matrices on high-performance machines. By using a restricted pivoting strategy guarded by incremental condition estimation, we were able to express the bulk of the computational work in terms of matrix-matrix operations. The traditional algorithm, on the other hand, is limited to matrix-vector operations and as a result suffers from performance degradation due to the cost of data movement. Compared to the traditional algorithm, the new approach more than halved the execution time on one processor of the Cray 2 and performed up to 50% better on one processor of the Cray X/MP and Cray Y/MP.

Although the results reported in this paper are preliminary, they add further evidence that block algorithms are preferable over algorithms based on matrix-vector kernels. We also mention that the same restricted pivoting strategy can be applied to computing the Cholesky factorization with column pivoting in a blocked fashion.

The reliability of these restricted pivoting strategies hinges on the reliability of incre-

mental condition estimation. Although experimental evidence indicates that incremental condition estimation is reliable in producing good estimates of the smallest singular value, one can construct examples where it fails. What would be ideal is an incremental condition estimation technique that computes both upper and lower bounds on the smallest singular value. Then one can easily monitor of the reliability of the condition estimator and apply countermeasures should degradation of the estimates occur. We are currently investigating this issue and will report on it later.

References

[1] Åke Björck. *Handbook of Numerical Analysis*, chapter Least Squares Methods. Volume 1, North Holland, 1987.

[2] Zhajoun Bai and Jim Demmel. *LAPACK Working Note #8: On a Block Implementation of Hessenberg Multishift QR Iteration.* Technical Report ANL–MCS–TM–127, Argonne National Laboratory, Mathematics and Computer Sciences Division, 1989.

[3] Michael Berry, Kyle Gallivan, William Harrod, William Jalby, Sy-Shin Lo, Ulrike Meier, Bernard Philippe, and Ahmed Sameh. Parallel algorithms on the Cedar system. In W. Händler, editor, *Proceedings of CONPAR 86*, pages 25–39, Springer Verlag, New York, 1986.

[4] Christian Bischof, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Danny Sorensen. *LAPACK Working Note #5: Provisional Contents.* Technical Report ANL–88–38, Argonne National Laboratory, Mathematics and Computer Sciences Division, September 1988.

[5] Christian H. Bischof. *Adaptive Blocking.* Technical Report ANL/MCS–P39–1288, Argonne National Laboratory, Mathematics and Computer Sciences Division, 1988.

[6] Christian H. Bischof. *Computing the Singular Value Decomposition on a Distributed System of Vector Processors.* Technical Report 87–869, Cornell University, Department of Computer Science, 1987. To appear in Parallel Computing.

[7] Christian H. Bischof. *Incremental Condition Estimation.* Technical Report ANL/MCS–P15–1088, Argonne National Laboratory, Mathematics and Computer Sciences Division, 1988.

[8] Christian H. Bischof. *A parallel QR factorization algorithm with controlled local pivoting.* Technical Report ANL/MCS–P21–1088, Argonne National Laboratory, Mathematics and Computer Sciences Division, 1988.

[9] Christian H. Bischof. A pipelined block QR decomposition algorithm. In Garry Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 3–7, SIAM Press, Philadelphia, 1989.

[10] Christian H. Bischof and Jack J. Dongarra. A project for developing a linear algebra library for high-performance computers. In Graham Carey, editor, *Parallel and Vector Supercomputing: Methods and Algorithms*, John Wiley & Sons, Somerset, NJ, 1989. to appear.

[11] Christian H. Bischof and Charles F. Van Loan. The WY representation for products of Householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8:s2–s13, 1987.

[12] P. A. Businger and G. H. Golub. Linear least squares solution by Householder transformation. *Numerische Mathematik*, 7:269–276, 1965.

[13] Tony F. Chan. Rank revealing QR factorizations. *Linear Algebra and Its Applications*, 88/89:67–82, 1987.

[14] Thomas F. Coleman. *Large Sparse Numerical Optimization.* Volume 165 of *Lecture Notes in Computer Science*, Springer Verlag, New York, 1984.

[15] Jim Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Danny Sorensen. *Prospectus for the Development of a Linear Algebra Library for High-Performance Computers.* Technical Report ANL–MCS–TM97, Argonne National Laboratory, Mathematics and Computer Sciences Division, September 1987.

[16] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide.* SIAM Press, 1979.

[17] J. J. Dongarra, F. G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26:91–112, 1984.

[18] Jack Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. *A Set of Level 3 Basic Linear Algebra Subprograms.* Technical Report MCS-P1-0888, Argonne National Laboratory, Mathematics and Computer Sciences Division, August 1988.

[19] Jack Dongarra, Sven Hammarling, and Linda Kaufman. Squeezing the most out of eigenvalue solvers on high-performance computers. *Linear Algebra and Its Applications*, 77:113–136, 1986.

[20] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[21] Jack J. Dongarra and Ian S. Duff. *Advanced Computer Architectures.* Technical Report ANL–MCS–TM57, Argonne National Laboratory, Mathematics and Computer Sciences Division, 1987. Revision 1.

[22] Jack J. Dongarra, Sven J. Hammarling, and Danny C. Sorensen. *Block Reduction of Matrices to Condensed Form for Eigenvalue Computations.* Technical Report ANL–MCS–TM99, Argonne National Laboratory, Mathematics and Computer Sciences Division, September 1987.

[23] L. V. Foster. Rank and null space calculations using matrix decomposition without column interchanges. *Linear Algebra and Its Applications*, 74:47–71, 1986.

[24] Kyle Gallivan, William Jalby, Ulrike Meier, and Ahmed Sameh. *The Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design.* Technical Report 625, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, September 1987.

[25] B. Garbow, J. Boyle, J. Dongarra, and C. Moler. *Matrix Eigensystem Routines – EISPACK Guide Extension.* Volume 51 of *Lecture Notes in Computer Science*, Springer Verlag, New York, 1977.

[26] G. H. Golub, V. Klema, and G. W. Stewart. *Rank Degeneracy and Least Squares Problems.* Technical Report TR–456, Dept. of Computer Science, University of Maryland, 1976.

[27] Gene H. Golub. Numerical methods for solving linear least squares problems. *Numerische Mathematik*, 7:206–216, 1965.

[28] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1983.

[29] William Harrod. *Solving Linear Least Squares Problems on an Alliant FX/8.* Technical Report, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1986.

[30] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.

[31] C. L. Lawson, R. J. Hanson, R. J. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[32] Charles L. Lawson and Richard J. Hanson. *Solving Least Squares Problems.* Prentice-Hall, 1974.

[33] Peter Mayes and Guiseppe Radicati di Brozolo. Block factorization algorithms on the IBM 3090/VF. In *Proceedings of the International Meeting on Supercomputing*, 1989.

[34] Robert Schreiber. *Block Algorithms for Parallel Machines.* Technical Report 87–5, Rensellaer Polytechnic Institute, Department of Computer Science, 1987.

[35] Robert Schreiber and Charles Van Loan. A storage efficient WY representation for products of Householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10(1):53–57, 1989.

[36] B. Smith, J. Boyle, J. Dongarra, B. Garbow, Y. Ikebe, V. Klema, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide.* Springer Verlag, New York, second edition, 1976.