SIAM J. SCI. STAT. COMPUT.
Vol. 12, No. 1, pp. 36–57, January 1991

A PARALLEL QR FACTORIZATION ALGORITHM WITH CONTROLLED LOCAL PIVOTING*

CHRISTIAN H. BISCHOF†

Abstract. This paper presents a new version of the Householder algorithm with column pivoting for computing a QR factorization that identifies rank and range space of a given matrix. The standard pivoting technique is not well suited for parallel computation, since it requires synchronization at every step in order to choose the next pivot column. In contrast, a restricted pivoting scheme that restricts the choice of pivot columns and avoids this synchronization constraint is employed. Incremental condition estimation is used to assess the effect that the addition of a candidate pivot column would have on the condition number of the matrix being generated. This safeguard ensures that this local strategy selects pivot columns that make sense in the global context of the computation. The resulting algorithm is well suited for implementation on a parallel machine, in particular, a MIMD machine with distributed memory. Simulations demonstrate that the numerical behavior of the restricted pivoting strategy is comparable to the traditional global pivoting strategy. Implementation results of the QR factorization algorithm without pivoting and with local and traditional pivoting on the Intel iPSC/1 and iPSC/2 hypercubes show that our scheme about halves the extra time required for pivoting.

Key words. QR factorization, column pivoting, controlled pivoting, incremental condition estimation, distributed architecture, parallel algorithms

AMS(MOS) subject classifications. 15A23, 65F20

1. Introduction.

An important problem in numerical linear algebra is the determination of a maximal set of linearly independent columns of a matrix A. In statistics this problem is often referred to as the *subset selection problem* [19], [20]. It arises in the context of identifying redundant carriers in a linear model. Other applications are the solution of underdetermined or rank-deficient least-squares problems [6], [20], [23] and nullspace methods in optimization [9].

In linear algebra terms this problem can be viewed as finding a basis for the range space of A. The common way to approach this problem is via a QR *factorization*

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

of 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. If A is a dense matrix, Q is usually computed by a sequence of *Householder transformations*

$$H = I - 2u\,u^T.$$

*Received by the editors December 27, 1988; accepted for publication (in revised form) October 3, 1989. This work was partially performed when the author was a graduate student at Cornell University and was supported by the U.S. Army Research Office through the Mathematical Science Institute of Cornell University, by the Office of Naval Research under contract N00014-83-K-0640, and by National Science Foundation contract CCR 86-02310. Support at Argonne was provided by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy under contract W-31-109-Eng-38. Computations were in part performed at the facilities of the Cornell Computational Optimization Project, which is supported by the National Science Foundation under contract DMS 87-06133, at the Cornell National Supercomputer Facility, and at the Advanced Research Computing Facility at Argonne National Laboratory.

†Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, Illinois 60439 (bischof@mcs.anl.gov).

Choosing

$$u = \frac{x + \operatorname{sign}(x_1)\,\|x\|_2\, e_1}{\|x + \operatorname{sign}(x_1)\,\|x\|_2\, e_1\|_2}, \tag{2}$$

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

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

The standard technique [7] for determining P can be viewed as choosing as the next column the one that is farthest away (in the two-norm sense) from the subspace spanned by the columns that were selected before [20, p. 168, P.6.4-5].

The hope is that in the resulting QR factorization (1) the rank of A will reveal itself by a small trailing subblock of R: if $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$ are the singular values of A and we partition R into

$$\begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix} \tag{3}$$

with an $r \times r$ lower right-hand block R_{22}, then it is easy to show [20, p. 19] that

$$\sigma_{n-r+1}(A) \leq \|R_{22}\|_2\,. \tag{4}$$

Hence if R_{22} is small, A can be considered to have numerical rank $n - r$ and the first $n - r$ columns of Q form an orthonormal basis for the range space of A. While we can construct counterexamples [20, p. 167] where the column pivoting strategy fails to reveal ill-conditioning of A, it works well in practice.

An alternative pivoting strategy has been suggested by Chan [8] and Foster [16]. It was pointed out originally in [19] that we can use the singular vector corresponding to the smallest singular value to find a permutation P that guarantees a small r_{nn} in (3) if $\sigma_n(A)$ is small. Chan [8] and Foster [16] extend this idea to higher dimensions. The idea is first to compute any QR factorization of A and then to "peel off" the small singular values of R one after the other by computing an appropriate singular vector at each step. Chan also shows how to compute upper and lower bounds $\hat{\sigma}_i$ and $\check{\sigma}_i$ of σ_i. This allows implementation of the algorithm in an adaptive fashion: it can be terminated if the lower bounds indicate that all the small singular values have been revealed. Let us from now on assume that A has r small singular values and that there is a well-defined gap between σ_{n-r} and σ_{n-r+1}. It is shown in [19] that a well-defined gap is necessary to make a sensible decision on the numerical rank of A. Then Chan proves that if r is not too large, his algorithm will compute a "rank-revealing QR factorization" in the sense that R_{22} in (3) is guaranteed to be small.

On a single processor the Householder QR factorization without pivoting (i.e., P in (1) is the identity) requires $O(mn^2)$ flops, column pivoting requires an additional n^2 flops, and the rank-revealing QR algorithm requires an additional $3rn^2$ flops on average [8]. So the computational complexity of these algorithms is comparable on a single-processor machine.

The situation is quite different on a multiprocessor machine, especially if it is based on a distributed architecture. The Householder QR algorithm without pivoting processes the columns of A in their natural order from left to right. Hence a natural approach to parallelizing such a scheme is to group the processors into a logical ring and deal out columns in a round-robin fashion. The resulting pipelined algorithm

staggers the computation across the processors and guarantees a load-balanced computation. It allows simple static assignment of data to processors and is for the most part synchronized by the flow of data between processors. The main reason for the efficiency of this algorithm is that a given processor can still be busy finishing a previous update while another processor is generating the next Householder vector. The introduction of column pivoting makes pipelining impossible since the order in which columns are processed is not known a priori. Furthermore, all processors have to synchronize to select the next pivot column. One can somewhat overlap the computation of a Householder vector with its application [11], but nonetheless global pivoting introduces considerable extra communication overhead and forces the program into a lockstep mode where essentially all processors wait for a Householder vector and then apply it at the same time. This can result in a serious loss of efficiency on MIMD machines that previously could have profited from the pipelining.

For the rank-revealing QR factorization algorithm the steps after the initial QR factorization are hard to parallelize. For each of the r small singular values of A, the algorithm computes an approximate singular vector via inverse iteration, which on the average [8] requires the solution of four triangular equation systems per small singular value. Although much progress has been made recently in solving triangular equation systems on distributed architectures [10], [15], [21], [25], [24], this problem can by no means be parallelized as efficiently as the initial QR factorization. In addition, the application of the permutation deduced from the singular vector destroys the upper triangular shape of R, which then has to be restored by a sequence of Givens rotations. Again this is essentially a sequential process that is hard to parallelize [11]. We also need $O(rn)$ extra storage to store these Givens rotations. Since r is not known beforehand, this requires either problem-specific knowledge about r or, alternatively, dynamic storage allocation. This complications does not arise in the column pivoting scheme.

Apart from their sequential nature, an inherent difficulty in parallelizing the equation solving and QR update steps is that the computational work is of the same order of magnitude as the amount of data it involves. That is, we have to perform $O(n^2)$ flops using $O(n^2)$ data. Since R is distributed throughout the system, it is hard to mask the communication overhead with the little arithmetic work to be performed. So the post-processing of R can end up being a good part of the overall computation time on a parallel machine.

In this paper we suggest a new parallel algorithm for computing a set of independent columns of A using a controlled local pivoting strategy. Each processor limits its choice of pivot columns to its local columns and thereby avoids the synchronization overhead associated with making a global pivot choice. To make this strategy numerically robust, we must decide whether a local pivot candidate is a reasonable choice in the context of the overall factorization. In other words, given the upper triangular matrix R_i already computed and a new column $\begin{pmatrix} v \\ \gamma \end{pmatrix}$ determined by the local pivot candidate 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 [20, 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 esti-

mation [4] we can obtain a good estimate for $\sigma_{\min}(R_{i+1})$ cheaply without reaccessing R_i and without disrupting the overall pipelining scheme. The resulting algorithm is well suited for a parallel machine, in particular, a MIMD machine with distributed memory.

The outline of the paper is as follows. In the next section we briefly review the pipelined algorithm for computing the QR factorization without column pivoting. In § 3 we review the QR factorization algorithm with column pivoting. Implementation results on the Intel iPSC/1 and iPSC/2 hypercubes show that—as expected—the global pivoting strategy results in considerable extra overhead. Section 4 then motivates the local pivoting scheme and shows how the reliability of this scheme can be ensured by controlling the selection of pivot columns using incremental condition estimation. Section 5 presents simulation results comparing the controlled local pivoting scheme with the traditional pivoting scheme and implementation results on the Intel iPSC/1 and iPSC/2 hypercubes. These data show that the local pivoting strategy substantially decreases the overhead associated with column pivoting while being as efficient in identifying the numerical rank of A. Lastly we summarize our contributions and outline directions for further research.

2. The pipelined Householder QR factorization algorithm. 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 (2) and $y = H(u)\,x$.

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

returns $H(u)\,A$.

Figure 1 describes the traditional Householder QR algorithm for computing the QR decomposition of an $m \times n$ matrix A $(m \geq n)$ without any pivoting. 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 (:) is used as shorthand to design a complete row or column.

for $i = 1$ **to** n **do**
  $[u_i, a(i{:}m, i)] \leftarrow genhh(a(i{:}m, i))$
  $a(i{:}m, i{+}1{:}n) \leftarrow apphh(u_i\,, a(i{+}1{:}m, i : n))$
end for

FIG. 1. *The Householder* QR *factorization algorithm without pivoting.*

A natural way to parallelize this algorithm is to distribute the columns of A to processors in a round-robin fashion. To be precise, let us assume that we have p processors $proc_0, \ldots, proc_{p-1}$ and that a_j is the jth column of A. Then processor $proc_i$ receives columns a_j where

$$i = (j - 1) \bmod p.$$

This is commonly referred to as the *column wrap mapping.* Each processor executes the algorithm given in simplified form in Fig. 2 (to save space we use the abbreviation "HH" for "Householder"). The array C is local to each processor and contains the $cols_k$ columns assigned to processor $proc_i (\sum_{i=0}^{p-1} cols_k = n)$. p_{left} and p_{right} designate

the left and right neighbor of $proc_k$, respectively. *lcnt* (local count) and *gcnt* (global count) are initially zero and count the Householder vectors generated in $proc_k$ and overall, respectively.

The application and generation of Householder vectors are staggered across processors so that the computation proceeds in what can be described as a "pipelined" fashion. To prevent processors from being idle waiting for the next Householder vector to arrive, new Householder vectors have to be generated as soon as possible. To that end a processor generating a new Householder vector applies the update $H(u)$ just received only to column $lcnt+1$, computes and sends out the new Householder vector $\hat{u}$, and only then completes the previous update $H(u)$ and applies the new update $H(\hat{u})$.

This algorithm has several attractive features. The wrap mapping is simple and makes it easy to interface the QR algorithm to other algorithms. The algorithm is also inherently load balanced since the computational work is completed in a round-robin fashion. Most important, however, is the fact that message passing overhead is low. On the average, each processor receives and sends every Householder vector once, resulting in n sends and receives and a total of $mn - n^2/2$ transmitted words per node. By computing a new Householder vector as soon as possible, we maximize the likelihood that a Householder vector will arrive at the next processor before it is actually needed, thereby avoiding processor idle time. Requiring only nearest-neighbor communication is also an important factor in decreasing communication overhead. If the parallel machine in question allows asynchronous message passing (i.e., a sender does not block while waiting for a message to be delivered to the receiver), then we can hope to further decrease communication overhead. For these reasons this pipelining technique has also been widely used [18], [21], [22], [25], [26], [28] for other factorizations. If special vector hardware can be exploited, several Householder matrices can be bundled together by using the WY factorization [5], [31] to arrive at a block pipelined algorithm [3].

3. The global pivoting strategy. In the one-processor setting column pivoting can be introduced into the Householder QR factorization algorithm at little additional cost. The resulting algorithm is shown in Fig. 3.

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 residuals of the jth column of the currently permuted AP with respect to the span of the first i columns of AP. As a result we can consider the numerical rank of A to be determined if res_{pvt} is small. 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 $O(\epsilon\|A\|_1)$ where ϵ is the machine precision. res_j can be easily updated and does not have to be recomputed at every step. Roundoff errors may make it necessary to recompute $res_j = \|(a(i:m,j))\|_2, j = i+1, \ldots, n$ periodically [13, p. 9.17] (we suppressed this detail in Fig. 3). In practice this is rarely the case, and so the additional cost for incorporating column pivoting into the QR factorization is $O(n^2)$ flops in addition to the overall flop count of $O(mn^2)$ flops for the QR factorization.

On a distributed memory machine, the determination of the pivot column introduces extra overhead. If the matrix is distributed by columns as in the pipelined QR factorization algorithm of Fig. 2, each processor can easily determine its local pivot candidate column, but to find the overall pivot column all processors have to syn-

```
processor proc_k
    lcnt ← 0; {counter for HH vectors generated in proc_k}
    gcnt ← 0; {counter for HH vectors generated globally}
    if (k == 0) then {generate initial HH vector}
      [u, c(:,1)] ← genhh(c(:,1)); send u to p_right;
      c(:,2:cols_0) ← apphh(u, c(:,2:cols_0)); lcnt ← gcnt ← 1;
    end if
    while (lcnt < cols_k) do {main loop}
      receive u from p_left; gcnt ← gcnt + 1;
      if (u not generated by p_right) then send u to p_right end if
      if (k == gcnt mod p) then { my turn to generate HH vector }
        lcnt ← lcnt + 1;
        c(gcnt:m, lcnt) ← apphh(u, c(gcnt:m, lcnt));
                          {update first column}
        [û, c(gcnt+1:m, lcnt)] ← genhh(c(gcnt+1:m, lcnt));
                                 { generate new HH vector }
        if (gcnt + 1 < n) then send û to p_right end if
        c(gcnt:m, lcnt+1:cols_k) ← apphh(u, c(gcnt:m, lcnt+1:cols_k));
                                   { complete previous HH update }
        gcnt ← gcnt + 1;
        c(gcnt:m, lcnt+1:cols_k) ← apphh(û, c(gcnt:m, lcnt+1:cols_k));
                                   { apply new HH update }
      else
        c(gcnt:m, lcnt+1:cols_k) ← apphh(u, c(gcnt:m, lcnt+1:cols_k));
                                   { simply apply HH update }
      end if
    end while
```

FIG. 2. *The pipelined Householder* QR *algorithm without pivoting.*

```
    foreach i ∈ {1, ..., n} do
      perm_i ← i; res_i ← ||a(:,i)||_2
    end foreach
    for i = 1 to 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
```

FIG. 3. *The* QR *factorization algorithm with traditional column pivoting.*

chronize. It should be noted that distributing A by rows is no quick way out either. Although the creation of a Householder vector and its application can be somewhat overlapped [11], the computation of the column norm in (2) requires collecting each processor's individual contribution necessitating $O(\log p)$ extra communication overhead.

Assuming column wrap mapping, a simplified version of the main loop of the QR factorization algorithm with column pivoting is shown in Fig. 4. To save space, we omitted the code for the generation of the first Householder vector, which is performed in an analogous fashion.

The arrays *perm* and *glblpos* encode the permutation matrix P. $perm_i = j$ if local column i was the jth column of A. $glblpos_i = j$ if local column i will be column j in AP. Hence P would permute column $perm_i$ of A into position $glblpos_i$ in AP. *myid* of $proc_i$ is i. The function *find_max* determines the processor that owns the maximal column by embedding a spanning tree [29] in the hypercube. Each processor merges packets containing a *myid* and res_{pvt} field for some processor (originally its own) so that at the end the *LEADER* node knows who possesses the new pivot column. Finding the overall pivot column in this fashion requires time about $t \log_2 p$ where t is the time to send a message containing two elements to a neighboring node. The broadcast primitive also uses the spanning tree to propagate messages and hence suffers a delay of $O(\log_2 p)$. The computation is terminated if either $\min(m, n)$ Householder vectors have been generated or the distance of the suggested new pivot column to the subspace spanned by the already selected columns is so small that the remaining columns can be considered linearly dependent.

As in the pipelined QR factorization algorithm without pivoting, we try to generate pivot columns as soon as possible. In the pipelined algorithm without pivoting, the column determining the next Householder vector is known a priori, and so it is sufficient to update just this column in order to be able to compute the next Householder vector. If we want to perform pivoting, we have to do more work before we can compute the next Householder vector. To illustrate, let $\hat{C}$ be the submatrix still left to process in a given node, let *res* be the vector of residuals that the columns defining $\hat{C}$ have with respect to the subspace already chosen, and let u be a Householder vector that is currently being received. Since the processor housing $\hat{C}$ has not seen u previously, *res* does not reflect the choice of u as a pivot column yet. So in order to be able to choose the next pivot column, we have to update *res* to reflect the choice of u as pivot column. As can be seen from Fig. 3, we need the first row of $H(u)\hat{C}$ to update the residuals. Now

$$H(u)\hat{C} = \hat{C} - 2uu^T\hat{C} = \hat{C} - 2uz^T$$

where

$$z = \hat{C}^T u.$$

So the first row of $H(u)\hat{C}$ is

$$(H(u)\hat{C})(1,:) = \hat{C}(1,:) - 2u(1)z^T.$$

Once this row has been computed, we can update the residuals, determine the local pivot column, and propagate our local pivot choice along the spanning tree in *find_max*. Nonetheless we cannot avoid the $O(\log n)$ overhead for the determination of the pivot column or the broadcasting of the Householder vector. Furthermore, the

```
processor proc_k
      { Initialization }
   foreach i ∈ {1, ..., cols_k} do
      perm_i ← myid + (i − 1)p + 1; res_i ← ||c_i||_2;
   end foreach
   { generate first Householder vector and broadcast it as below }
      { main loop }
   repeat
      repeat
         switch (message_type)
            case "Householder_vector":
                  receive u; gcnt ← gcnt + 1;
            case "You_are_winner":
                  gcnt ← gcnt + 1; lcnt ← lcnt + 1; glblpos(lcnt) ← gcnt;
                  [u, c(gcnt:m, pvt)] ← genhh(c(gcnt:m, pvt));
                  broadcast u;
                  c_lcnt ↔ c_pvt; perm_lcnt ↔ perm_pvt; res_pvt ← res_lcnt;
            case "Time_to_quit":
                  stop
         end switch
      until message is received
         { determine candidate pivot column }
      z ← c(gcnt:m, lcnt+1:cols_k)^T u;
      c(gcnt, lcnt+1:cols_k) ← c(gcnt, lcnt+1:cols_k) − 2u(1)z^T;
      res_i ← sqrt(res_i^2 − c(gcnt, i)^2), i ∈ {lcnt+1, ..., cols_k}
      Let pvt ∈ {lcnt+1, ..., cols_k} be such that res_pvt is maximal.
      [winner, res_of_winner] ← findmax(pvt, res_pvt, LEADER);
         { complete update }
      c(gcnt+1:m, lcnt+1:cols_k)
                  ← c(gcnt+1:m, lcnt+1:cols_k) − 2u(2: m−gcnt)z^T;
         { LEADER notifies the winner }
      if (myid = LEADER) then
        if ((gcnt > min(m, n)) or (res_of_winner < threshold)) then
          broadcast "Time_to_quit" message.
        else
          send "You_are_winner" message to winner.
        end if
      end if
   forever
```

FIG. 4. *The distributed Householder* QR *factorization algorithm with global pivoting.*

lockstep style of execution resulting from this synchronization point makes it impossible to mask the communication overhead as efficient as the pipelining technique of Fig. 2 allowed.

TABLE 1

The Householder QR *factorization algorithm with and without column pivoting applied to a* $500 \times n$ *matrix on a* 32*-node* iPSC/1 *hypercube.*

n	$t_{\max}$		$\bar{t}_{ohead}$	
	Global piv.	No piv.	Global piv.	No piv.
100	27.7	18.8	16.5	5.7
200	70.9	49.6	29.8	7.8
300	125	93.2	40.5	9.7
400	184	145	46.5	10.9
500	255	201	50.1	11.3
600	310	257	54.0	11.1
700	369	314	53.1	11.3
800	437	370	54.5	11.2
900	515	427	66.1	11.5

TABLE 2

The Householder QR *factorization algorithm with and without column pivoting applied to a* $500 \times n$ *matrix on a* 16*-node* iPSC/2 *hypercube.*

n	$t_{\max}$		$\bar{t}_{ohead}$	
	Global piv.	No piv.	Global piv.	No piv.
100	5.9	4.1	2.7	0.7
200	16.5	12.8	4.9	1.2
300	30.5	25.2	6.6	1.6
400	46.6	40.1	7.9	2.0
500	63.7	56.3	8.7	2.2
600	80.8	72.6	9.3	2.2
700	97.0	88.9	9.0	2.3
800	113	105	8.7	2.4
900	130	122	8.8	2.4

We compared the algorithms of Figs. 2 and 4 on a 32-node Intel iPSC/1 [14] and a 16-node iPSC/2 [1] hypercube. A Gray code mapping [29] was used to embed the ring of processors and the code was entirely written in Fortran for single precision. For the iPSC/1, we compiled with the Ryan–McFarland compiler (Version 2.20a) using the huge memory model and executed under the NX node operating system release 3.1.1. On the iPSC/2, we used the Green Hills compiler (Version 1.8.3a) and the NX/2 node operating system release 2.3. For $m = 500$ and $n = 100, 200, \ldots, 900$ we generated random matrices with singular values $1, 2, \ldots, \min(m, n)$ and reduced them to triangular form using $\min(m, n)$ Householder reductions. We observed the performance shown in Table 1 for the iPSC/1 and in Table 2 for the iPSC/2. Here $t_{\max}$ is the maximal execution time of any processor in seconds and $\bar{t}_{ohead}$ is the average time in seconds that a processor spends communicating and waiting idle.

We see that the algorithm for computing the QR factorization without pivoting is very efficient in that only a small portion of the overall execution time is spent on

communication. As predicted, we observe a noticeable performance degradation for the global pivoting scheme. In particular, for smaller problems not yet dominated by floating point work, the communication overhead increases dramatically compared to the pipelined QR factorization algorithm. This is unpleasant, especially in light of the fact that column pivoting costs only $O(n^2)$ extra flops.

4. Controlled local pivoting. The easiest way to avoid the extra overhead associated with global pivoting is to forego global pivoting altogether and have each processor limit its choice of pivot columns to the ones it houses. Whereas in the pipelined algorithm without pivoting, each processor simply picks the next available column to determine the next Householder vector, each processor now performs a *local column pivoting* step to determine the next Householder vector. This allows implementation of the algorithm in much the same fashion as in Fig. 2, and as a result we expect to reap much the same benefits as described in §2.

The problem with the strictly local pivoting strategy is obviously its reliability in identifying independent columns of A. As a pathological example, assume that all columns in processor $proc_0$ are nearly equal. As a result, processor $proc_0$ will make bad choices after it has generated the very first Householder vector. The resulting upper triangular matrix R will be nearly rank-deficient but will not necessarily have a small lower right-hand block to reveal that fact.

To guard against choosing nearly dependent pivot columns, it is natural to monitor $\sigma_{\min}(R)$ since the smallest singular value of R is the distance of R (in the two-norm sense) from the space of rank-deficient matrices. Computing $\sigma_{\min}(R)$ exactly via inverse iteration, for example, is too expensive since it would require us to reaccess R (which is distributed throughout the system) several times. In fact, it is not feasible to access the previously generated R even once when we want to decide on the suitability of a new pivot column.

Let R_i be the current upper triangular matrix R_i which is made up by the first i columns of Q^TAP after i Householder vectors have been generated and applied. Then we use the *incremental condition estimator* suggested by Bischof [4] to monitor $\sigma_{\min}(R_i)$. Given a good estimate $\hat{\sigma}_{\min}(R_i) = 1/\|x\|_2$ defined by a large norm solution x to $R_i^T x = d$ and a new column $\begin{pmatrix} v \\ \gamma \end{pmatrix}$, the incremental condition estimator allows us to obtain an estimate for $\sigma_{\min}(R_{i+1})$ where

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

without accessing R_i again. Defining

$$\alpha = v^T x \quad \text{and} \quad \beta = \gamma^2 x^T x + \alpha^2 - 1,$$

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

$$\hat{\sigma}_{\min}(R_{i+1}) = \frac{1}{\|y\|_2} \tag{5}$$

where

$$y = \begin{pmatrix} sx \\ (c - s\alpha)/\gamma \end{pmatrix} \tag{6}$$

solves

$$R_{i+1}^T y = \begin{pmatrix} sd \\ c \end{pmatrix}.$$

c and s are defined as the solutions of

$$\begin{aligned} &\max \Phi(c,s) = \|y\|_2 \\ &\text{subject to } c^s + s^2 = 1. \end{aligned}$$

c and s can easily be computed, and so 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 condition estimation scheme [4] show that it is reliable in producing good estimates. It overestimates the smallest singular value of a triangular matrix only by a small factor, and the results vary only a little with condition number, matrix size, and singular value distribution. Applying this condition estimator to upper triangular matrices generated by using the traditional column pivoting strategy somewhat increased its accuracy and we can confidently expect similar behavior when applying this estimator to matrices generated by the local pivoting strategy.

With the incremental condition estimator we now have the tool to ensure the reliability of the local pivoting strategy. By applying the incremental condition estimator to its local pivot candidate, a processor can decide whether its local choice is reasonable in the global context of the computation. Assuming that processor k knows the current estimate x as well as $\|x\|_2$ for the current upper triangular matrix R_{gcnt}, all that is needed for the next condition estimator step is the last column $\begin{pmatrix} v \\ \gamma \end{pmatrix}$ of R_{gcnt+1}. But

$$v = c(1 : gcnt, j)$$

has already been computed, and from the definition of u and res it follows immediately that

$$\gamma = -\text{sign}\,(c(gcnt+1, j))\, res_j.$$

So all the information for the next condition estimator step is readily at hand, and we can compute $\hat{\sigma}_{\min}(R_{gcnt+1})$ as described above. With

$$\omega = \max_{1 \le i \le n} \|a_i\|_2$$

being the norm of the largest column of A, we can then take

$$\hat{\kappa}\,(R_{gcnt+1}) = \eta\,\omega \|y\|_2 \tag{7}$$

as an estimate for the true condition number of R_{gcnt+1}. The scaling factor η reflects the trust we have in the accuracy of our estimates. A large η will result in estimates that are too pessimistic, while a small η might lead to underestimation. ω overestimates $\|A\|_2$ by at most a factor of $\sqrt{n}$, but heuristically it is a much more accurate estimate.

Comparing the estimates (5) or (7) against a chosen threshold, we will then accept or reject a candidate pivot column. If the candidate pivot column is rejected, processor k has exhausted its supply of "reasonable" columns, and from then on it will only apply Householder vectors generated by other processors to its remaining columns. If, on the other hand, we accept the candidate pivot column, then processor k will actually compute $\hat{u}$ and send $\hat{u}$ and $(y, \|y\|_2)$ to its right neighbor. So the additional work required for generating the ith Householder vector will on average be $3i$ floating point operations and the transmission of i words to a neighboring node. It is worth

emphasizing that on average y and $\|y\|_2$ have to be forwarded only to the processor that will generate the next Householder vector (which in most cases will be the right neighbor), while $\hat{u}$ will eventually be known to all processors. So the propagation of the condition estimator information will result in only a minor increase in data traffic.

This scheme continues until no processor has any acceptable pivot candidate left. Assuming that altogether we generated $\hat{n} = n - \hat{r}$ Householder vectors, we have at this point computed the incomplete QR factorization

$$AP = (Q_1, Q_2) \begin{pmatrix} R_{11} & R_{12} \\ 0 & \hat{A} \end{pmatrix} \tag{8}$$

where Q_1 is $m \times \hat{n}$, Q_2 is $m \times (m - \hat{n} + 1)$, and $Q = [Q_1, Q_2]$ is orthogonal. R_{11} is upper triangular of size $\hat{n} \times \hat{n}$ and $\hat{A}$ is of size $(m - \hat{n} + 1) \times \hat{r}$. Our controlled pivoting strategy gives us an estimate for $\sigma_{\min}(R_{11})$, and furthermore, we know that adding any of the leftover $\hat{r}$ columns of AP would result in a decrease of the smallest singular value below our chosen threshold. So we have good reason to assume that $\hat{r}$ is the dimension of the numerical null space of A.

A simplified version of the main loop of the resulting algorithm is shown in Figs. 5 and 6. We omitted the generation of the first Householder vector which is done in the same fashion as in Fig. 6 and in order not to bury the structure of the algorithm, we omitted the details concerning program termination. Each processor can be in one of three states as determined by *mystatus*. It is *ALIVE* if it still houses eligible columns; this is the initial state of every processor. Its status changes to *FINISHED* if it has processed all its local columns. Since other processors might still house eligible columns, it still must forward messages; this is its only action in this state. If a processor is *DEAD*, then all its remaining columns have been found to be unacceptable, but it still has to apply Householder vectors generated by other processors as well as forward them.

$$y \leftarrow cond\_est(x, \|x\|_2, v, \gamma)$$

returns y as defined by (6). *trustfactor* is used to adjust for the overestimation of the smallest singular value. Since experiments with the incremental condition estimator [4] indicate that in general the condition estimator does not overestimate $\sigma_{\min}$ by a factor of more than three, we used $1/(3\|y\|_2)$ as our estimate for $\sigma_{\min}$.

The overall structure of the main loop is described in Fig. 5. Notice that a processor checks halfway through a Householder update whether a singular vector has arrived. In this case it completes the $H(u)$ update only on the pivot candidate column in order to generate a new Householder vector as soon as possible. Only after the new Householder vector $\hat{u}$ has been sent out does it complete the $H(u)$ and $H(\hat{u})$ updates. These steps are described in more detail in Fig. 6 (ignoring Part b). Figure 6 (ignoring Part a) describes the easier case of generating a pivot column when the previous Householder update has already been completed.

Notice that compared to the algorithm with global pivoting we maintain the pipelined mode of operation. By generating Householder vectors as soon as possible we attempt to keep the pipeline filled with Householder vectors and minimize processor idle time. Furthermore there are no synchronization points in this algorithm that would add extra overhead or would prevent us from masking communication overhead. We would also like to mention that this algorithm is naturally load-balanced due to its pipelined nature. Load balancing is more difficult in the algorithm employing global pivoting, since it is possible that a processor may contain all the first pivot columns

processor $proc_k$

repeat
repeat
switch (message type)
case "Householder_vector":
{ complete enough of $H(u)$ application to update res }
receive u; $gcnt \leftarrow gcnt + 1$; forward u to p_{right}.
if ($mystatus \neq FINISHED$) **then**
$z \leftarrow c(gcnt{:}m, lcnt{+}1{:}cols_k)^T u$;
$c(gcnt, lcnt{+}1{:}cols_k) \leftarrow c(gcnt, lcnt{+}1{:}cols_k) - 2u(1)z^T$;
$res_i \leftarrow \sqrt{res_i^2 - c(gcnt, i)^2}\,, i \in \{lcnt{+}1, \ldots, cols_k\}$
{ check whether another message arrived in the mean time }
if (I received another message containing a singular value) **then**
Try to generate a new Householder vector before completing
the update determined by u. See Fig. 6 and ignore part b.
else
$c(gcnt{+}1, lcnt{+}1{:}cols_k)$
$\leftarrow c(gcnt{+}1, lcnt{+}1{:}cols_k) - 2u(2{:}m{-}gcnt)z^T$;
end if
else
break to outermost repeat-loop.
end if
case "singular_vector":
receive $(x, \|x\|_2)$;
if ($mystatus = ALIVE$) **then**
determine next pivot candidate. See Fig. 6 and ignore part a.
else
forward $(x, \|x\|_2)$ to p_{right}.
end if
case "time_to_quit":
send time_to_quit message to p_{right} and quit.
end switch
until a message has been received
forever

FIG. 5. *The pipelined Householder* QR *algorithm with controlled local pivoting: main loop.*

processor $proc_k$

{ Determine new pivot candidate and estimate $\hat{\sigma}_{min}(R_{gcnt+1})$. }
Let $pvt \in \{lcnt+1, \ldots, cols_k\}$ be such that res_{pvt} is maximal.
$y \leftarrow \text{cond\_est}(x, \|x\|_2, c(1\!:\!gcnt, pvt), -\text{sign}(c(gcnt+1, pvt))res_{pvt})$;
if $(\|y\|_2/trust\_factor > 1/threshold)$ **then**
{ new pivot candidate is not acceptable }
send $(x, \|x\|_2)$ to p_{right}. $mystatus \leftarrow DEAD$.
$c(gcnt+1, lcnt+1\!:\!cols_k) \leftarrow c(gcnt+1, lcnt+1\!:\!cols_k) - 2u(2\!:\!m-gcnt)z^T$;
else
$lcnt \leftarrow lcnt + 1; gcnt \leftarrow gcnt + 1; c_{lcnt} \leftrightarrow c_{pvt}$;
$perm_{lcnt} \leftrightarrow perm_{pvt}; res_{lcnt} \leftarrow res_{pvt}$;
begin of Part a
{ delay $H(u)$ update and generate new Householder vector $\hat{u}$ }
$z(1) \leftrightarrow z(pvt-lcnt+1)$;
$c(gcnt-1\!:\!m, lcnt) \leftarrow c(gcnt-1\!:\!m, lcnt) - 2z(1)u$;
$[\hat{u}, c(gcnt\!:\!m, lcnt)] \leftarrow genhh(c(gcnt\!:\!m, lcnt))$;
send $\hat{u}$ to p_{right}; send $(y, \|y\|_2)$ to p_{right}.
{ complete $H(u)$ update and then $H(\hat{u})$ update }
$c(gcnt-1\!:\!m, lcnt+1\!:\!cols_k)$
$\leftarrow c(gcnt-1\!:\!m, lcnt+1\!:\!cols_k) - 2u(2\!:\!m-gcnt)z^T$;
$c(gcnt\!:\!m, lcnt+1\!:\!cols_k) \leftarrow (I - 2\hat{u}\hat{u}^T)c(gcnt\!:\!m, lcnt+1\!:\!cols_k)$;
end of Part a
begin of Part b
{ generate new Householder vector }
$[u, c(gcnt\!:\!m, lcnt)] \leftarrow genhh(c(gcnt\!:\!m, lcnt))$;
send u to p_{right}; send $(y, \|y\|_2)$ to p_{right}.
{ complete Householder update }
$c(gcnt\!:\!m, lcnt+1\!:\!cols_k) \leftarrow (I - 2uu^T)c(gcnt\!:\!m, lcnt+1\!:\!cols_k)$;
end of Part b
$res_i \leftarrow \sqrt{res_i^2 - c(gcnt, i)^2}, i \in \{lcnt+1, \ldots, cols_k\}$
end if

FIG. 6. *Generation of a new Householder vector.*

and hence will be idle soon unless one redistributes columns dynamically. Our local pivoting scheme avoids those complications. In addition, the local pivoting scheme is—apart from the cost of startup and wind-down—insensitive to the number of processors, since communication is only between neighbors. The cost of the reduction and broadcast in the global scheme, on the other hand, depend on the diameter of the system and are likely to increase as the system grows.

5. Numerical experiments. To verify the merits of our proposed controlled local pivoting scheme, we tested its numerical reliability by comparing it with the QR factorization with traditional column pivoting. Furthermore, we tried to assess its computational performance on a distributed memory machine by implementing it on the Intel iPSC/1 and iPSC/2 hypercubes.

To assess the numerical behavior of the proposed local pivoting scheme, we simulated the parallel algorithm using PRO-MATLAB [27] and compared it with the traditional QR factorization algorithm with global column pivoting. Various 100×100 matrices were generated, and the local pivoting strategy was simulated on 8 and 32 processors.

For tests 1 to 3 we generated 50 random matrices with prescribed singular value distributions $\{\sigma_i\}$. Random orthogonal matrices U and V were generated using the method of Stewart [32], and then A was formed as $A = U\Sigma V^T$. For all matrices the largest and smallest singular values were 1 and 10^{-9}, respectively.

Break 1 *Distribution*: $\sigma_1 = \ldots = \sigma_{99} = 1;\ \sigma_{100} = 10^{-9}$.

Break 9 *Distribution*: $\sigma_1 = \ldots = \sigma_{91} = 1;\ \sigma_{92} = \ldots = \sigma_{100} = 10^{-9}$.

Exponential Distribution: $\sigma_1 = 1; \sigma_i = \alpha^{i-1} (i = 2, \ldots, 100);\ \alpha = (10^{-9})^{1/99}$.

Setting the rejection threshold for the smallest singular value to 10^{-7} and discounting the estimate for the smallest singular value (5) by a factor of 3 (i.e., *trustfactor* in Fig. 6 equals 3), we reject a candidate pivot column in the parallel algorithm if

$$\frac{1}{3\|y\|_2} = \hat{\sigma}_{min}(R_{gcnt+1}) \leq 10^{-7}.$$

For the traditional QR factorization algorithm we use the last diagonal entry of R_{gcnt+1} as an estimate for $\sigma_{\min}(R_{gcnt})$ and reject a candidate pivot column if

$$\frac{1}{3} \mid r_{gcnt+1,gcnt+1} \mid \leq 3 \cdot 10^{-7}.$$

Table 3 shows the condition numbers of the upper triangular matrices R generated by controlled local pivoting and by traditional column pivoting on those matrices. If we let σ_{cutoff} be the smallest singular value greater than 10^{-7}, then the optimal value we can theoretically achieve for $\kappa(R)$ is $\kappa_{opt}(R) = 1/\sigma_{cutoff}$. Furthermore, let $\kappa_{par}(R)$ be the condition number resulting from the parallel scheme and $\kappa_{trad}(R)$ the condition number resulting from the traditional column pivoting scheme. For $\kappa_{par}(R)$ and $\kappa_{trad}(R)$ observed minimum, average, and maximum values are displayed. These results show that guarded local pivoting is as effective as full column pivoting in generating a well-conditioned R—especially if there are more than just a few columns in each processor. Except for the break 9 example the transition from 8 to 32 processors had no noticeable effect. The break 9 example shows that the local pivoting strategy can deteriorate somewhat if the matrix is highly rank-deficient and the number of columns per processor is small. This fact is not surprising in that the pivoting choices of each processor are very limited. Due to the high dimension of the numerical null

TABLE 3
min/avg/max *values of the condition numbers of R using local and global pivoting.*

Distribution	Break 1	Break 9	Exponential
$\kappa_{par}(R)$, $p = 8$	2.7 /5.1 /8.0	6.4 /13 /46	7.2e6 /1.1e7 /1.6e7
$\kappa_{par}(R)$, $p = 32$	3.7 /7.7 /23	10 /180 /6.1e3	7.6e6 /1.3e7 /1.9e7
$\kappa_{trad}(R)$	2.8 /3.7 /4.8	4.3 /5.7 /7.8	7.2e6 /1.0e7 /1.9e7
$\kappa_{opt}(R)$	1.0	1.0	8.1e6

space, a processor may be forced to consider columns that are suboptimal but whose choice results in a matrix R_i where $\sigma_{\min}(R_i)$ is still well above the threshold.

The average values for the break 9 distribution on 32 processors in Table 3 are also somewhat misleading in that they make the local pivoting strategy look worse than it really is. If we ignore the one experiment where R had condition number 6.1e3 (which is still well below our rejection threshold), we obtain for the other 49 runs the condition number distribution shown in Fig. 7. We see that in the vast majority of cases (36 cases) the condition numbers of R were less than 50. We mention that this is a somewhat contrived example in that with only three columns per processor the pipelined algorithm would be dominated by startup and wind-down costs and as a result would not perform efficiently.

This example does, however, suggest the use of a variable threshold in accepting pivot columns if we are worried about generating as well-conditioned a matrix as possible. Starting with a conservative threshold and relaxing it as the computation proceeds, we shall in all likelihood prevent a processor from choosing a suboptimal column early on. The penalty is that we might have to consider the same candidate pivot column more than once. If we reject a candidate for the ith Householder vector, the added cost is $2i$ flops (since sx in (6) need not be computed) and the transmission of i words to a neighbor. Depending on the application this extra work might be worthwhile.

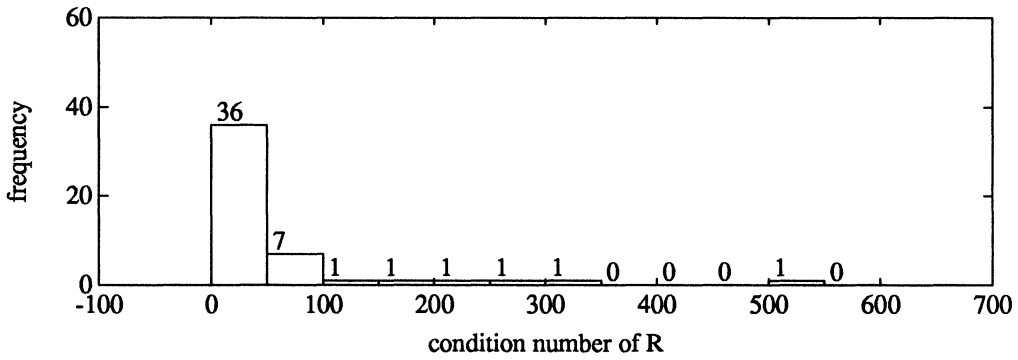

FIG. 7. *Condition number distribution of R for the "break 9" distribution with* $p = 32$.

For the sharp break distributions there is a well-defined gap between the singular values before and after the acceptance threshold, and both the local and global column pivoting strategies identify the numerical nullspace correctly for $p = 8$ and $p = 32$

TABLE 4
Frequency of accepting columns for the exponential distribution.

No. of columns accepted	72	73	74	75	76
Local pivoting, $p = 8$	2	3	12	24	9
Local pivoting, $p = 32$	5	11	23	9	2
Global pivoting	0	2	29	18	1

in all cases. As already pointed out, the determination of numerical rank becomes problematic if there is no well-defined gap between singular values that are considered "large" and "small". The exponential distribution is such a problematic case. There are 77 singular values that are larger than 10^{-7}, but there is no well-defined break. To be exact,

$$\sigma_{75} = 1.8 \cdot 10^{-7}, \quad \sigma_{76} = 1.5 \cdot 10^{-7}, \quad \sigma_{77} = 1.2 \cdot 10^{-7}, \quad \sigma_{78} = 1.0 \cdot 10^{-7}.$$

The column pivoting strategy reflects this difficulty in accepting less than 77 columns; the results are displayed in Table 4. So, for example, in 24 of the 50 runs we accepted 75 columns in the local pivoting scheme using 8 processors. These results show that even for an ill-defined problem the guarded local pivoting scheme is reliable in that it leans towards a small underestimate of the dimension of the numerical range space of A.

Finally, we give an example where the local pivoting strategy actually performs better than the global one. A well-known example (originally suggested by Kahan) where the traditional column pivoting strategy fails is

$$A_n = \operatorname{diag}(1, s, s^2, \ldots, s^{n-1}) \begin{pmatrix} 1 & -c & \cdots & \cdots & -c \\ 0 & 1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 1 & -c \\ 0 & \cdots & \cdots & 0 & 1 \end{pmatrix} + \Delta \tag{9}$$

where $\Delta = \operatorname{diag}(n\epsilon, (n-1)\epsilon, \ldots, \epsilon)$, $c^2 + s^2 = 1$ and ϵ is the machine precision. A_n is very ill-conditioned, but although each leading principal submatrix A_k $(k \leq n)$ is also ill-conditioned, there is a well-defined gap between σ_n and σ_{n-1}. As an example, in single precision for $n = 50$ and $c = 0.5$ we have $\sigma_{49} = 1.2 \cdot 10^{-3}$ and $\sigma_{50} = 3.7 \cdot 10^{-12}$. Even in floating-point arithmetic the matrix is its own QR factorization with pivoting, but no trailing block of R is small to reveal its ill-conditioning.

For this matrix both the local and global pivoting schemes select the columns in their natural order and fail. However, the condition estimator integrated into the parallel scheme detects the ill-conditioning of the leading principal submatrices A_k — in fact we observed that it never overestimates the smallest singular value by a factor of more than 1.5. So at least failure does not go unnoticed. Now let $\tilde{A}_{50}$ be the same matrix as A_{50} except that the order of columns has been reversed. For the global column pivoting scheme this permutation is without consequences and it fails. The parallel scheme on 8 or 32 processors, on the other hand, correctly identifies the numerical null space of $\tilde{A}_{50}$. While this is an exceptional occurrence resulting from the special structure of A_n, it is nonetheless surprising since intuitively we would expect the global pivoting strategy always to perform better than the local one.

To assess the computational performance of the proposed scheme, we implemented it on a 32-node iPSC/1 and 16-node iPSC/2 hypercube. When running the code on the problems used for Tables 1 and 2 and using the same notation we observed the performance shown in Table 5.

TABLE 5
Performance of the Householder QR *factorization algorithm with controlled local pivoting.*

	32-node iPSC/1		16-node iPSC/2	
n	$t_{\max}$	$\bar{t}_{ohead}$	$t_{\max}$	$\bar{t}_{ohead}$
100	25.1	13.0	4.8	1.4
200	60.2	17.2	13.9	2.0
300	108	20.6	26.9	2.6
400	165	24.0	42.4	3.1
500	227	28.1	59.4	3.8
600	280	23.6	75.2	3.2
700	336	21.0	91.6	3.1
800	392	19.3	108	3.0
900	454	20.0	125	2.9

Comparing these figures with those of Tables 1 and 2 we see that the local pivoting strategy does indeed result in a significant decrease in execution time. These gains would be even more pronounced on a bigger system as our pipelined strategy is essentially insensitive to the number of processors, whereas the broadcast and reduction operations needed in the global pivoting algorithm become more expensive. We also see that the local pivoting algorithm is efficient in that does not require significantly more flops than the QR factorization without pivoting and the communication overhead is in general a small portion of the overall execution time.

The advantages of local pivoting are even more apparent if we consider how much *more* local and standard pivoting cost us in comparison with the QR factorization without pivoting. Let $t_{\max}^{nopiv}(n)$ be the maximal execution time of the pipelined QR factorization algorithm without pivoting on a $500 \times 100 * n$ problem ($t_{\max}^{globalpiv}(n)$ and $t_{\max}^{localpiv}(n)$ analogous). Figure 8 displays

$$p_{total}^{strategy}(n) \equiv \frac{t_{\max}^{strategy}(n) - t_{\max}^{nopiv}(n)}{t_{\max}^{nopiv}(n)} * 100$$

for

$$strategy = \{globalpiv,\ nopiv\}$$

and varying n. $p_{total}^{strategy}(n)$ measures how much more time (in percent) the implementation of the two pivoting strategies costs compared to the QR factorization without pivoting. The dashed line represents the standard column pivoting strategy; the dash-dotted line represents controlled local column pivoting. Figure 9 displays

$$p_{comm}^{strategy}(n) \equiv \frac{\bar{t}_{ohead}^{strategy}(n) - \bar{t}_{ohead}^{nopiv}(n)}{\bar{t}_{ohead}^{nopiv}(n)} * 100$$

and shows the percentage increase in communication overhead due to pivoting.

These figures show that the extra running time required to implement local pivoting is on the average about half that required to implement global pivoting. This is

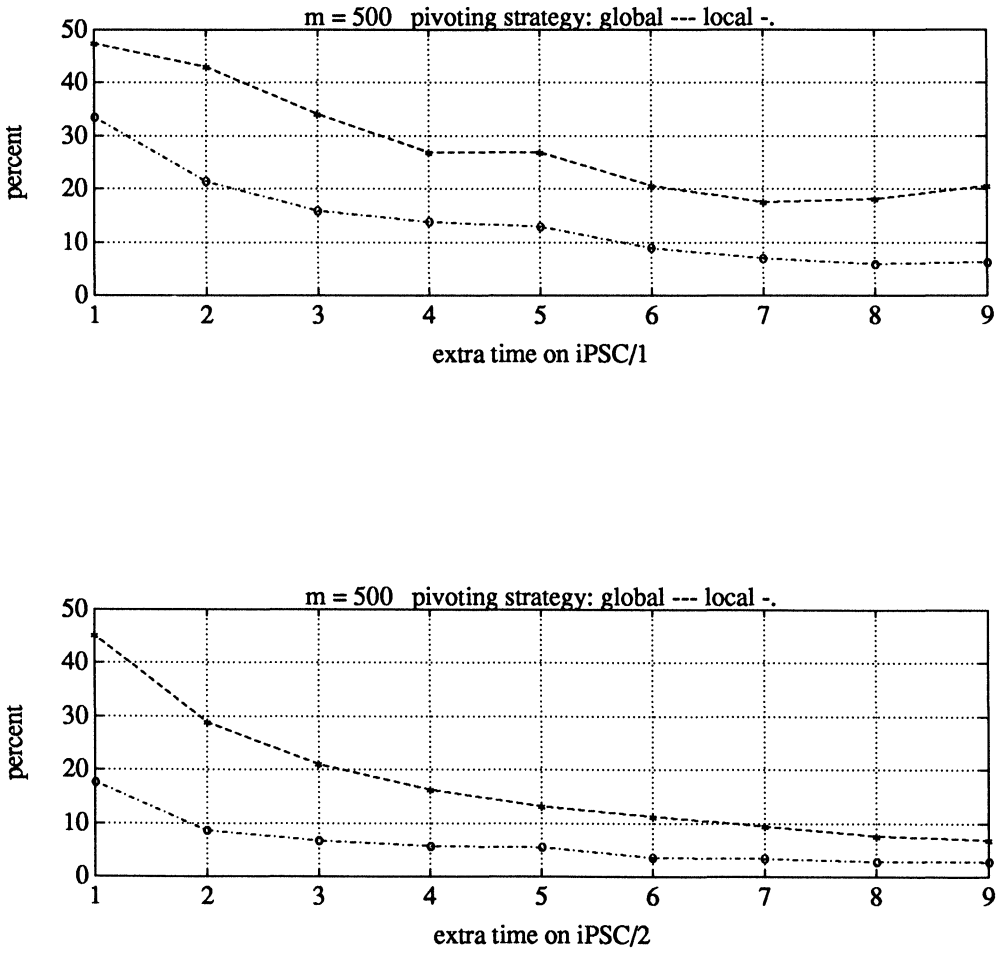

FIG. 8. *Extra time required for pivoting with respect to* QR *factorization without pivoting.*

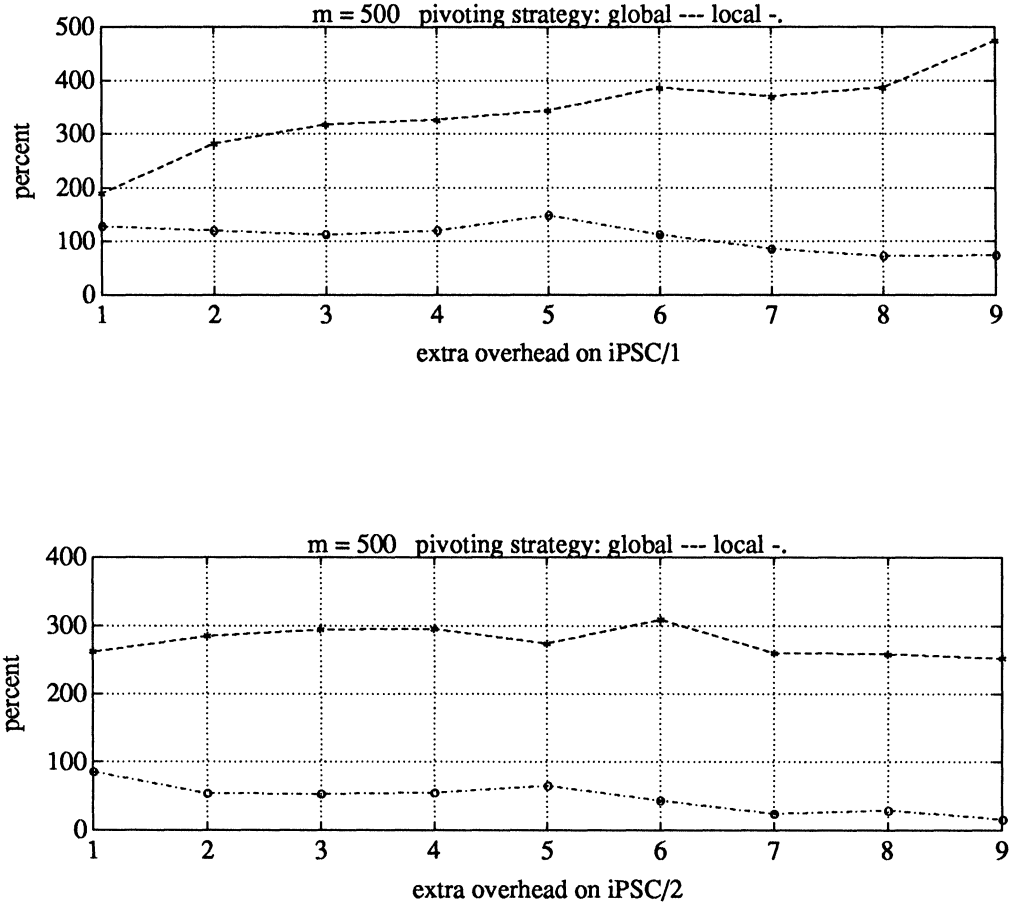

FIG. 9. *Extra communication overhead for pivoting with respect to* QR *factorization without pivoting.*

not surprising since—as shown in Figure 9—local pivoting increases the communication overhead only by about a factor of two, whereas the traditional column pivoting strategy can increase communication overhead almost fivefold. In addition, the extra communication overhead for the local pivoting strategy essentially stays flat with n whereas it grows with n for the global pivoting regime on the iPSC/1. So the local pivoting strategy is indeed superior to the traditional global pivoting strategy on MIMD machines which can profit from the pipelining scheme and for which global broadcast/gather operations are expensive.

6. Conclusions. We presented a new version of the Householder algorithm with column pivoting for computing a QR factorization which identifies rank and range space of a given matrix. To arrive at an algorithm that is better suited for parallel computation than the traditional regime, we employed a restricted pivoting scheme that restricted the choice of pivot columns to ones in local memory. The resulting algorithm is much better suited for a parallel machine, in particular, a MIMD machine with distributed memory. This is borne out by the implementation results on the Intel iPSC/1 and iPSC/2 hypercubes where the local pivoting scheme about halves the extra time required for pivoting compared to the traditional column pivoting scheme. Simulations also showed that the numerical properties of the suggested local pivoting scheme are comparable to those of the global pivoting strategy.

The key idea that made this algorithm work was the use of incremental condition estimation to restrict the choice of pivot column without giving up numerical reliability. Incremental condition estimation allowed us to "back up" when we were about to make a bad choice. Restricting pivot choices then resulted in an algorithm that displayed a much higher degree of locality of reference. We showed the resulting benefits for a MIMD machine, but obviously locality of reference is also crucial in achieving good performance on parallel machines with shared memory or one-processor machines employing a memory hierarchy. As a result, we believe restricted pivoting strategies to be advantageous in those environments as well. We are currently exploring this issue in the context of a block algorithm [2], [12], [17], [30] for computing the QR factorization with column pivoting on a shared memory machine with a memory hierarchy.

Acknowledgments. I am very grateful to Paul Plassmann from Cornell and Ed Kushner from Intel for their undaunted perseverance in porting the code from the iPSC/1 to the iPSC/2. I would also like to thank Intel Corporation for the opportunity to use one of their iPSC/2 cubes for my benchmarks.

REFERENCES

[1] R. ARLAUSKAS, *iPSC/2 system: a second generation hypercube*, in The Third Conference on Hypercube Concurrent Computers and Applications, G. Fox, ed., ACM Press, New York, 1988, pp. 38–42.

[2] M. BERRY, K. GALLIVAN, W. HARROD, W. JALBY, S.-S. LO, U. MEIER, B. PHILIPPE, AND A. SAMEH, *Parallel algorithms on the Cedar system*, in Proc. CONPAR 86, W. Händler, ed., Springer-Verlag, New York, 1986, pp. 25–39.

[3] C. H. BISCHOF, *A pipelined block QR decomposition algorithm*, in Parallel Processing for Scientific Computing, G. Rodrigue, ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1989, pp. 3–7.

[4] ———, *Incremental condition estimation*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 312–322.

[5] C. H. BISCHOF AND C. F. VAN LOAN, *The WY representation for products of Householder matrices*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. s2–s13.

[6] Å. BJÖRCK, *Difference Methods—Solutions of Equations in R^n*, Vol. II of Handbook of Numerical Analysis, Elsevier Publishers, 1989, chap. Least Squares Methods.
[7] P. A. BUSINGER AND G. H. GOLUB, *Linear least squares solution by Householder transformation*, Numer. Math., 7 (1965), pp. 269–276.
[8] T. F. CHAN, *Rank revealing* QR *factorizations*, Linear Algebra Appl., 88/89 (1987), pp. 67–82.
[9] T. F. COLEMAN, *Large Sparse Numerical Optimization*, Lecture Notes in Computer Science 165, Springer-Verlag, New York, 1984.
[10] T. F. COLEMAN AND G. LI, *Solving systems of nonlinear equations on a message-passing multiprocessor*, Tech. Report CS–87–887, Department of Computer Science, Cornell University, Ithaca, NY, November 1987.
[11] T. F. COLEMAN AND P. PLASSMAN, *The solution of nonlinear least-squares problems on a message-passing multiprocessor*, Tech. Report CS–TR–88–923, Department of Computer Science, Cornell University, Ithaca, NY, June 1988.
[12] J. DONGARRA, A. SAMEH, AND D. SORENSEN, *Implementation of some concurrent algorithms for matrix factorization*, Parallel Comput., 3 (1986), pp. 25–34.
[13] J. J. DONGARRA, J. R. BUNCH, C. B. MOLER, AND G. W. STEWART, LINPACK *Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1979.
[14] T. H. DUNIGAN, *Hypercube performance*, in Hypercube Multiprocessors, M. T. Heath, ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987, pp. 178–192.
[15] S. EISENSTAT, M. HEATH, C. HENKEL, AND C. ROMINE, *Modified cyclic algorithms for solving triangular systems on distributed-memory multiprocessors*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 589–600.
[16] L. V. FOSTER, *Rank and null space calculations using matrix decomposition without column interchanges*, Linear Algebra Appl., 74 (1986), pp. 47–71.
[17] K. GALLIVAN, W. JALBY, U. MEIER, AND A. SAMEH, *The impact of hierarchical memory systems on linear algebra algorithm design*, Tech. Report 625, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, September 1987.
[18] G. A. GEIST AND M. T. HEATH, *Parallel Cholesky factorization on a hypercube multiprocessor*, Tech. Report ORNL–6190, Engineering Physics and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, 1985.
[19] G. H. GOLUB, V. KLEMA, AND G. W. STEWART, *Rank degeneracy and least squares problems*, Tech. Report TR–456, Department of Computer Science, University of Maryland, 1976.
[20] G. H. GOLUB AND C. F. VAN LOAN, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, 1983.
[21] M. T. HEATH AND C. H. ROMINE, *Parallel solution of triangular systems on distributed-memory multiprocessors*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 558–588.
[22] I. IPSEN, Y. SAAD, AND M. SCHULTZ, *Dense linear systems on a ring of processors*, Linear Algebra Appl., 77 (1986), pp. 205–239.
[23] C. L. LAWSON AND R. J. HANSON, *Solving Least Squares Problems*, Prentice-Hall, Englewood Cliffs, NJ, 1974.
[24] G. LI AND T. F. COLEMAN, *A new method for solving triangular systems on distributed memory message-passing multiprocessors*, SIAM J. Sci. Statist. Comput., 10 (1987), pp. 382–398.
[25] ———, *A parallel triangular solver for a hypercube multiprocessor*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 485–502.
[26] C. MOLER, *Matrix computation on distributed memory multiprocessors*, in Hypercube Multiprocessors 1986, M. T. Heath, ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1986.
[27] C. MOLER, J. LITTLE, AND S. BANGERT, PRO-MATLAB *User's Guide*, The Mathworks, Sherborn, MA, 1987.
[28] A. POTHEN, S. JHA, AND U. VEMAPULATI, *Orthogonal factorization on a distributed memory multiprocessor*, in Hypercube Multiprocessors 1987, M. T. Heath, ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987.
[29] Y. SAAD AND M. H. SCHULTZ, *Topological properties of hypercubes*, Tech. Report YALEU/DCS/RR-389, Department of Computer Science, Yale University, New Haven, CT, 1985.
[30] R. SCHREIBER, *Block Algorithms for Parallel Machines*, IMA Volumes in Mathematics and its Applications 13, Springer-Verlag, Berlin, 1988, pp. 197–207.
[31] R. SCHREIBER AND C. VAN LOAN, *A storage efficient WY representation for products of Householder transformations*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 53–57.
[32] G. W. STEWART, *The efficient generation of random orthogonal matrices with an application to condition estimators*, SIAM J. Numer. Anal., 17 (1980), pp. 403–409.