QR Factorization Algorithms For Coarse-Grained Distributed Systems

Christian Heinrich Bischof
Ph. D. Thesis

TR 88-939
August 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

QR FACTORIZATION ALGORITHMS FOR COARSE-GRAINED DISTRIBUTED SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Christian Heinrich Bischof

August 1988

© Christian Heinrich Bischof 1988
ALL RIGHTS RESERVED

QR Factorization Algorithms for Coarse-Grained Distributed Systems

Christian Heinrich Bischof, Ph.D.
Cornell University 1988

We present the techniques of adaptive blocking and incremental condition estimation which we believe to be useful for the computation of common matrix decompositions in high-performance environments. We apply these new techniques to algorithms for computing the Householder QR factorization with and without pivoting on a coarse-grained distributed system. For reasons of portability, we use a pipelined scheme on a ring of processors as the basis of our algorithms.

To take advantage of possible floating point hardware on each node we develop a blocked version of the pipelined Householder QR algorithm that employs the compact WY representation for products of Householder matrices. While a strategy involving blocks of fixed width leads to increased floating point utilization per node, it also leads to increased load imbalance. To reconcile this tradeoff we introduce a variable width blocking strategy based on a model of the critical path of the algorithm. The resulting adaptive blocking strategy provides for good floating point performance per node while maintaining overall load balance. Experimental results on the Intel iPSC hypercube show that the adaptive blocking strategy performs indeed better than any fixed width blocking strategy.

In the second part of our thesis we develop methods for introducing pivoting into the distributed QR factorization algorithm. Incorporating the traditional column pivoting

strategy in a straightforward manner introduces a global synchronization constraint which results in increased communication overhead. A strictly local pivoting scheme avoids the resulting loss in efficiency, but has to be monitored for reliability. To this end, we introduce an incremental condition estimator which allows us to update the estimate of the smallest singular value of an upper triangular matrix R as new columns are added to R. The update requires only $O(n)$ flops and the storage of $O(n)$ words between successive steps. Experiments indicate that the incremental condition estimator is reliable despite its small computational cost. Using the incremental condition estimator we are then able to guard against the selection of troublesome pivot columns in our local pivoting scheme at little extra cost. Simulation results show that the resulting algorithm is about as reliable as the traditional QR factorization algorithm with column pivoting.

Biographical Sketch

The author was born on May 19, 1960 in Aschaffenburg, a city in the north-east of Bavaria. Due to "environmental circumstances" little Christian got accustomed to hearty brews at an early age. This experience was to be of considerable benefit later in life when his paths crossed with those of two Irishmen of apparently similar upbringing.

After graduating from Gymnasium in 1979, the author completed his mandatory stint in the army and then entered the Julius-Maximilans-Universität in Würzburg to study Mathematics. In the fall of 1983 he entered the graduate program in Computer Science at Cornell University. The first year at Cornell proved to be rather traumatic since he had to catch up with the English language, American culture and computer science all at the same time. Things improved continually thereafter and in October 1987 he married Kathy Worthingon. Although she is a speech language pathologist, Kathy has tried so far unsuccessfully to eradicate the remaining Germanisms in her husbands language.

After the completion of his degree, the author will join the Mathematics and Computer Science Division of Argonne National Laboratories as a Wilkinson Fellow in Computational Mathematics. Chicago suburbia will certainly take some getting used to after having enjoyed the beautiful surroundings of Ithaca for five years.

To my mother Anneliese and to my wife Kathy

Acknowledgements

I thank my advisor, Charles Van Loan, for his support and advice over the last three years. Charlie encouraged me to pursue my own ideas and left me considerable freedom in doing so yet I could always count on his knowledgeable advice when I needed it. His comments also considerably improved the readability of my papers and taught me a lot about writing. It was fun to work with Charlie.

I thank Tom Coleman and Mike Todd for serving on my committee and for many improvements they suggested. Besides it was always nice to have a little chat with them.

I would also like to thank Dr. Enrico Clementi at IBM Kingston for giving me the opportunity to gain practical computing experience on the LCAP system. My thanks also to Doug Logan and the other members of the LCAP staff for their friendly support.

My special thanks to Kathy for her love, understanding and encouragement throughout. Her company made life much nicer and when we were apart it gave me something to look forward to.

I thank my mother for diligently keeping me informed about what was happening back home and for her support throughout. Her impressive CARE packages filled with cake, coffee and chocolate made her a living legend with my housemates.

Thanks and a hearty "Prost" to my longtime housemates Kieran Herley and Pat

Stephenson. The evening chats at N. Tioga when we were sipping whisky or quaffing a beer together were fun and contributed greatly to maintaining my sanity when I was buried up to my ears in work. Kieran was also a dedicated member of the cross country skiing and hiking "bush whacking squad" together with Peter Kochevar and Brad Vander Zanden. Together we discovered many a beautiful spot in the forests around Ithaca. I must admit that to a great extent that was due to us getting lost quite frequently since we spent more time discussing the pubs than planning the trip.

Lastly I wish to acknowledge the financial support I received from the Fulbright Commission, from the U.S. Army Research Office through the Mathematical Sciences Institute at Cornell University, from the Office of Naval Research under contract N14-83-K-640 and from the National Science Foundation under contract DCR 86-2310. Computations for this thesis were in part performed at the Advanced Computing Facility of the National Supercomputer Center at Cornell which is supported in part by the National Science Foundation and IBM.

Table of Contents

1 Introduction 1
- 1.1 Related Work . . . 3
- 1.2 Blocking Techniques to Exploit Vector Hardware . . . 5
- 1.3 Controlled Local Pivoting . . . 7
- 1.4 Outline . . . 7

2 Householder QR Algorithms for Uniprocessors 9
- 2.1 The Classical Householder QR Factorization Algorithm . . . 9
- 2.2 The Traditional Column Pivoting Scheme . . . 12
- 2.3 The Rank-Revealing QR Factorization . . . 15
- 2.4 Jacobi SVD Algorithms and the QR Factorization . . . 19

3 The Pipelined Householder QR Algorithm with Adaptive Blocking 23
- 3.1 The Pipelined Householder QR Algorithm . . . 24
- 3.2 The Compact WY Representation . . . 28
- 3.3 Variable Blocking . . . 32
- 3.4 An Adaptive Blocking Strategy . . . 38

4 The Pipelined Householder QR Algorithm with Controlled Local Pivoting 46
- 4.1 Local Pivoting . . . 47
- 4.2 An Incremental Estimator for the Smallest Singular Value of a Triangular Matrix . . . 52
- 4.3 The QR Algorithm with Controlled Local Pivoting . . . 57

5 Conclusions 64

A Matlab Code for the Incremental Condition Estimator 71

B Matlab Code Simulating the Controlled Local Pivoting Algorithm 76

Bibliography 85

List of Tables

3.1 The Pipelined Householder QR Algorithm on the Intel iPSC 27
3.2 Execution Rates of *gencwy* and *appcwy* 33
3.3 The Pipelined Block QR Algorithm Using Blocks of Fixed Width . . . 34
3.4 The Pipelined Block QR Algorithm Using Blocks of Variable Width . . 44

4.1 max/avg values of $\hat{\sigma}_{min}(R)/\sigma_{min}(R)$ for Test 1 55
4.2 max/avg values of $\hat{\sigma}_{min}(R)/\sigma_{min}(R)$ for Test 2 without Pivoting 55
4.3 max/avg values of $\hat{\sigma}_{min}(R)/\sigma_{min}(R)$ for Test 2 with Pivoting 56
4.4 min/avg/max Values of the Condition Numbers of R using Local and Global Pivoting 60
4.5 Frequency of Accepting Columns for the Exponential Distribution . . . 62

List of Figures

1.1 The Gentleman–Kung Scheme 4

2.1 The Traditional Householder QR Factorization Algorithm 11
2.2 The QR Factorization Algorithm with Column Pivoting 13
2.3 Computing a Rank-Revealing QR Factorization 18

3.1 The Pipelined Householder QR Algorithm 25
3.2 The Block QR Factorization Algorithm 31
3.3 The Pipelined Block Householder QR Algorithm with Variable Blocking: Startup Phase 36
3.4 The Pipelined Block Householder QR Algorithm with Variable Blocking: Main Loop 37
3.5 Accumulating Q_1 in Reverse Order 39
3.6 Algorithm Determining Block Widths Using Critical Path Model . . . 42
3.7 Block Widths Determined by Critical Path Model 43

4.1 The Pipelined QR Algorithm with Local Pivoting: Startup Phase . . . 50
4.2 The Pipelined QR Algorithm with Local Pivoting: Main Loop 51
4.3 Condition Number Distribution of R for the *break 9* Distribution with $p = 32$ 61

5.1 The QR Factorization Algorithm with Restricted Column Pivoting . . 67

Chapter 1

Introduction

The QR factorization is one of the basic tools of numerical linear algebra. Given an $m \times n$ matrix A, we compute an $n \times n$ permutation matrix P, an $m \times m$ orthogonal matrix Q and an $n \times n$ upper triangular matrix R such that

$$AP = Q \begin{pmatrix} R \\ 0 \end{pmatrix} . \tag{1.1}$$

Since P reorders the columns of A, this decomposition is usually referred to as the QR *decomposition with column pivoting.* In many applications it is not necessary to reorder the columns of A, so $P = I$ and (1.1) simplifies to

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix} . \tag{1.2}$$

Q can be computed via a sequence of *Householder Transformations*

$$H = I - 2uu^T, \|u\|_2 = 1$$

or alternatively by a sequence of *Givens rotations*

$$G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}, c^2 + s^2 = 1 .$$

(see [9,52]). The Householder approach requires about half as many floating point operations and for that reason is usually preferred if A is dense. An algorithm using Givens rotations on the other hand requires fewer memory accesses and can be superior on machines where memory access is slow [35]. Givens rotations are also very useful for sparse problems [9,55,58] since they can be used to introduce zeroes in a very selective fashion.

The applications for the QR factorization are numerous. The most common is the solution of the *linear least squares problem*

$$\min \|Ax - b\|_2 . \tag{1.3}$$

Other applications are the computation of the range and null space of A. Since R has the same singular value distribution as A, the QR factorization can also be useful in reducing an $m \times n$ problem involving A to an $n \times n$ problem involving R. The savings can be substantial if $m \gg n$ and an example is the Chan SVD algorithm [14].

In this thesis we present algorithms for computing the QR factorization of a dense matrix on coarse-grained distributed systems. In this context we use the new techniques of adaptive blocking and incremental condition estimation. The architecture we have in mind consists of a moderate number (up to a hundred, say) of homogeneous processors. Each processors has its own clock, contains a certain amount of local memory and communicates with other processors only by exchanging messages. There is no shared memory or centralized control in the system. While this class of machines encompasses only a small portion of the parallel machines introduced over the last few years [39] it has received much attention due to the inherent scalability of the architecture and the resulting promise in computational power. Well known machines of this type are the hypercube machines [42,59,94].

1.1 Related Work

For the algorithm designer trying to take advantage of the potential of such a machine the distributed nature of the system poses a challenge. As a result there has been considerable interest in algorithms for computing the QR factorization. The Householder QR algorithm steps through the matrix A one column at a time and so a straightforward way to distribute the computation is to assign columns of A to processors in a round-robin fashion. This technique of staggering the computation has been widely used also for other factorizations [48,66,73,83]. The computation then proceeds in a pipelined fashion and is inherently load balanced. Also the data mapping is simple so the results of the QR factorization can readily be interfaced to other routines. We will refer to this algorithm as the *Pipelined Householder QR Algorithm.*

Other parallel approaches are based on the observation that two Givens rotations involving disjoint pairs of rows are completely independent and can be executed independently. In the context of systolic arrays this was exploited by Gentleman and Kung [49]. Figure 1.1 shows the time step at which an element can be eliminated by a Givens rotation involving adjacent rows using the Gentleman–Kung scheme. For example four Givens rotations are performed during the seventh time step.

Other systolic algorithms based on Givens rotations have been suggested by Heller and Ipsen [60], Modi and Clarke [82] and Luk [77]. For the reduction of a full matrix these algorithms typically rely on a triangular array of processors. Elden [44] adapted the Gentleman-Kung scheme to the coarse-grained setting and suggested how the original triangular array of processors could be efficiently simulated on a linear array and on a torus of processors.

Another approach that has been popular in the parallel setting is the divide-and-conquer approach. This is the basic idea underlying the algorithms of Chamberlain [13], Chu and George [22] and Pothen [85,86]. To illustrate, assume that we have a 16×4

$$\begin{pmatrix} x & x & x & x \\ 8 & x & x & x \\ 7 & 9 & x & x \\ 6 & 8 & 10 & x \\ 5 & 7 & 9 & 11 \\ 4 & 6 & 8 & 10 \\ 3 & 5 & 7 & 9 \\ 2 & 4 & 6 & 8 \\ 1 & 3 & 5 & 7 \end{pmatrix}$$

Figure 1.1: The Gentleman–Kung Scheme

matrix distributed by rows across four processors. To reduce the first column, each processor independently reduces (either via Givens or Householder reductions) its local first column. Using Chu and George's terminology, we refer to this stage of the computation as the *independent annihilation phase (IAP)*. Then P_0 and P_1 collaborate in reducing the (1,1) entry in P_1 while at the same time P_2 and P_3 collaborate in reducing the (1,1) entry in P_3. Finally P_0 and P_2 work together to reduce the (1,1) entry in P_2. This sequence of reductions is referred to as the *cooperative merging phase (CMP)*. The algorithm then proceeds to the next column. It can be seen from this description that CMP requires the embedding of a tree structure into the processor topology and for that reason the divide-and-conquer scheme has been very popular for hypercube architectures.

A somewhat different approach has been taken by Coleman and Plassman [27]. In their algorithm the matrix is distributed by rows but reduced via Householder reductions. Overlapping the generation of the Householder vector with its application results in a low communication overhead for their algorithm. The architectural assumptions

of this algorithm are also somewhat less stringent than for the divide-and-conquer approach in that a gather/broadcast primitive is required instead of an explicit tree embedding.

This point raises the important issue of code portability across different architectures. As an algorithm designer one often has to choose between an algorithm design that is reasonably efficient and portable across a variety of machines and one that is close to optimal for a given architecture but may be hard to port. This decision is especially important in the parallel setting where there exists a vast choice of architectures but not yet a commonly accepted programming model for numerical algorithms on parallel machines (some models are suggested in [20,36,87,97]). We decided to emphasize portability and to that end we believe the reliance on a simple machine model to be important. This is one of the reasons why we chose the pipelined Householder QR algorithm as the basis for the algorithms we present. It only requires processors to be configurable in a ring and to communicate via nearest-neighbour communication. As a consequence this algorithm could also easily be simulated on a shared memory machine by simulating the pipelined nature of the algorithm via a queue.

1.2 Blocking Techniques to Exploit Vector Hardware

An aspect that has not received much attention yet is the exploitation of vector processing hardware on each node. While the Intel VX and FPS T-Series hypercubes are currently the only machine offering such a feature, we believe it will become quite common in the future due to decreasing hardware costs. A methodology that has been sucessful on vector machines is the reliance on *block algorithms* [3,35,47,90]. By considering the matrix at the highest level as a collection of submatrices (the so-called *blocks*)

we can obtain algorithms which are rich in matrix-matrix operations like matrix-matrix multiplication or high-rank updates.

While of little importance on traditional scalar oriented machines, data movement has a major impact on the execution rate of vector processing machines. This fact is well explained in [32,34]. Compared to matrix-vector or especially vector-vector operations matrix-matrix operations allow a high degree of reuse of data stored in vector registers or cache. When computing a matrix-matrix operation with blocks of size k, the ratio of data movement to arithmetic operations is typically $O(1/k)$. This *surface-to-volume effect* [40] prevents degradation of performance by processors being idle waiting for data. Kernels common to numerical linear algebra codes have been identified in the *BLAS (Basic Linear Algebra Subprograms)* proposals. Matrix-matrix kernels are identified in the Blas 3 proposal [38], matrix-vector kernels in the BLAS 2 proposal [37] and vector-vector kernels in the Blas 1 description [70]. Block algorithms are also easily adapted to changing processor characteristics by adjusting the block size and introducing efficient implementations of the BLAS kernels. There is no need to change the overall algorithm.

These advantages of block algorithms extend to the parallel setting. For some examples see the papers by Bischof [5], Schreiber [89,90] and Van Loan [102]. Due to the surface-to-volume effect for a block oriented algorithm the relative cost of communication can potentially be lessened by increasing the block size. Transferring data in larger chunks also can decrease the impact of startup costs and message buffer fragmentation.

As we show we can extend the pipelined Householder QR algorithm to incorporate blocking without complicating the overall algorithmic structure. Using blocks of fixed width turns out to be not an optimal choice for the pipelined algorithm. While wide blocks insure good floating point processor utilization they also decrease the load balance of the algorithm and as a result increase processor idle time. To overcome this difficulty we devise an adaptive blocking technique that is based on a model of

the critical path of the algorithm. We believe this technique to be useful also in the context of developing block versions of other pipelined algorithms.

1.3 Controlled Local Pivoting

Another issue is how easily and at what cost pivoting can be incorporated into the QR algorithm. Of the work previously surveyed, only Coleman and Plassman [27] address this issue. In computing the QR factorization with pivoting, the columns of A are chosen one-by-one according to a certain criterion. Incorporating pivoting in a straightforward fashion into the pipelined Householder QR algorithm destroys pipelining since all processors have to synchronize to determine which column to choose next. We have developed a way to remove this global communication constraint and to make strictly local pivoting sufficient. This is due to a newly developed incremental condition estimation technique which allows us to update our estimate of the smallest singular value of the current R at every step with $O(n)$ work while saving only $O(n)$ words of information between sucessive steps. Using this tool we can control the local selection of pivot columns in a reliable fashion. We believe this condition estimator to be a useful tool in monitoring numerical stability whenever a triangular matrix is generated one column (or row) at a time such as in the LU and Cholesky factorizations, both in the dense and sparse settings. The incremental condition estimator allows us to restrict the choice of pivot columns without compromizing the reliability of the factorization.

1.4 Outline

The outline of our thesis is as follows: In chapter 2 we review the classical Householder QR algorithm and the traditional column pivoting strategy and outline Chan's rank-

revealing QR decomposition algorithm. We also briefly comment on the usefulness of the QR decomposition in the context of Jacobi SVD algorithms.

Chapter 3 describes the pipelined Householder QR algorithm on which all of our algorithms will be built. We introduce the WY representation which allows us to group a series of Householder updates into one convenient update. We motivate the load balancing problems resulting from the straightforward introduction of blocking into the pipelined Householder QR algorithm and show how a critical path analysis can be used to vary the blocksize such that high execution rates per node are maintained without compromising load balance.

In chapter 4 we motivate the problems that pivoting poses in our parallel algorithm and introduce our incremental condition estimator. Combining the condition estimator with the pipelined QR scheme results in an algorithm that uses only local pivoting and is only marginally more expensive than the QR algorithm without pivoting. Simulation results show that its numerical behavior is also very similar to the QR algorithm with full column pivoting.

Chapter 5 summarizes our contribution and suggest other applications of adaptive blocking and incremental condition estimation.

Chapter 2

Householder QR Algorithms for Uniprocessors

In this chapter we discuss the traditional QR factorization algorithms based on Householder transformations. The classical Householder QR algorithm is reviewed and the need for a pivoting strategy for the solution of rank-deficient least squares problems is motivated. We present the column pivoting strategy suggested by Businger and Golub [12] and briefly mention alternative pivoting strategies. We introduce the *rank-revealing QR factorization* [16] which combines the best features of QR and singular value decompositions. Lastly we comment on the importance of the QR factorization in the context of Jacobi methods for computing the singular value decomposition.

2.1 The Classical Householder QR Factorization Algorithm

A Householder matrix (or Householder transformation)

$$H = H(u) = I - 2u\,u^T, \|u\|_2 = 1 \tag{2.1}$$

is a symmetric and orthogonal transformation. When a vector x is multiplied by H, it is reflected in the hyperplane $\text{span}(u)^{\perp}$. We can use a Householder transformation to reduce a given nonzero vector x to a multiple of the canonical unit vector e_1 by setting

$$u = \frac{x + \text{sign}(x_1)\,\|x\|_2\,e_1}{\|x + \text{sign}(x_1)\,\|x\|_2\,e_1\|_2} \tag{2.2}$$

Then

$$H(u)\,x = -\text{sign}(x_1)\,\|x\|_2\,e_1.$$

If

$$Q = H_1 \cdots H_k$$

is a product of Householder matrices, then the computed matrix $\hat{Q}$ has the attractive property that it is orthogonal to working precision ϵ [105], i.e.

$$\|\hat{Q} - Q\|_2 = O(\epsilon)\,.$$

Application of a Householder matrix H to a given matrix A involves a matrix-vector multiplication

$$z \leftarrow A^T u \tag{2.3}$$

and a rank-one update

$$A \leftarrow A - 2uz^T\,. \tag{2.4}$$

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

$$u \leftarrow genhh(x)$$

returns u as defined by (2.2) and

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

returns $H(u)\,A$.

Figure 2.1 describes the traditional Householder QR algorithm for computing the QR decomposition (1.2) of an $m \times n$ matrix A $(m \geq n)$. 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**

$u_i \leftarrow genhh(a(i:m,i))$

$a(i:m,i:n) \leftarrow apphh(u_i\,, a(i:m,i:n))$

end for

Figure 2.1: The Traditional Householder QR Factorization Algorithm

If we define a *flop* to be an addition or a multiplication, the generation of the Householder vectors requires

$$flops_{genhh} = 3mn - \frac{3}{2}n^2 \tag{2.5}$$

flops while the application of the Householder matrices requires

$$flops_{apphh} = 2mn^2 - \frac{2}{3}n^3 - 2mn + 4\frac{1}{2}n^2 \tag{2.6}$$

flops for a total cost of

$$flops_{tradQR} = 2mn^2 - \frac{2}{3}n^3 + mn + 3n^2\,. \tag{2.7}$$

The orthogonal matrix

$$Q = H(u_1)\cdots H(u_n)$$

is usually not formed explicitly but stored in factored form by storing the Householder vectors u_i.

Once we have computed the QR factorization we can use it to solve the linear least squares problem for a matrix A of full rank. With

$$Q^T b = \begin{pmatrix} c \\ d \end{pmatrix} \begin{matrix} n \\ m-n \end{matrix}$$

we have

$$\|Ax - b\|_2^2 = \|Q^T Ax - Q^T b\|_2^2 = \|Rx - c\|_2^2 + \|d\|_2^2 .$$

So the solution x_{LS} of the linear least squares problem (1.3) can be found by solving the equation system

$$R\, x_{LS} = c \tag{2.8}$$

and the residual is $\|d\|_2^2$.

2.2 The Traditional Column Pivoting Scheme

To solve a least squares problem involving a rank deficient A, the previously described method fails since R will have at least one zero diagonal entry. Viewed another way, the first n columns of Q as produced by the Householder algorithm of Figure 2.1 are not a basis for the range space of A.

To solve that problem, the column pivoting strategy suggested by Businger and Golub [12] tries to compute the QR factorization of a subset of columns of A whose span is a good approximation to span(A). Viewed geometrically [52, p.168, P.6.4-5], the idea is to choose at each step that column of A that has the highest two-norm residual with respect to the subspace spanned by the columns that were selected before. Intuitively we should obtain a reasonably independent basis for $R(A)$ in this fashion since we always select the column that is "farthest away" from the span of the currently selected columns. This strategy can be integrated nicely into the Householder QR algorithm [52, p.164] and the resulting algorithm is shown in Figure 2.2.

foreach $i \in \{1, \ldots, n\}$ **do**

$perm_i = i$; $res_i = \|a(:,i)\|_2$

end foreach

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

Let $pvt \in \{i, \ldots, n\}$ be such that res_{pvt} is maximal

if $(res_{pvt} = 0)$ **then**

break { A has exact rank $i-1$ }

else { exchange columns pvt and i }

$perm_i \leftrightarrow perm_{pvt}$; $a(:,i) \leftarrow a(:,pvt)$; $res_{pvt} \leftarrow res_i$;

$u_i \leftarrow genhh(a(i{:}m\,,i))$;

$a(i{:}m\,,i{:}n) \leftarrow apphh(u_i\,, a(i{:}m\,,i{:}n))$;

foreach $j \in \{i+1, \ldots, n\}$ **do**

$res_j \leftarrow \sqrt{res_j^2 - a(i,j)^2}$;

end foreach

end if

end for

Figure 2.2: The QR Factorization Algorithm with Column Pivoting

The vector *perm* is used to store the permutation matrix P. If $perm(i) = k$, then the kth column of A has been permuted into the ith position. 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. 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 [31, p. 9.17](we suppressed this detail in Figure 2.2). In practice this is rarely the case and so the additional cost for incorporating column pivoting into the QR factorization is only $O(n^2)$ flops.

The hope is that the column pivoting results in a QR factorization where

$$R = \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix} \begin{matrix} n-r \\ r \end{matrix}$$

has a small lower right hand block R_{22}. If $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n$ are the singular values of A it can easily be shown [52, p. 19] that

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

So a small $r \times r$ trailing submatrix of R shows that A is close to being rank-r deficient. If we partition

$$\begin{matrix} Q = [& Q_1 & , & Q_2 &] \\ & n-r & & m-(n-r)+1 & \end{matrix}$$

then span(Q_1) is a good approximation to the stable part of the range of A. If we ignore the r small singular values (i.e. consider $R_{22} = 0$) the general solution to the least squares problem (1.3) is then given by

$$x_{LS} = P \begin{pmatrix} R_{11}^{-1}(c - R_{12}z) \\ z \end{pmatrix} \begin{matrix} n-r \\ r \end{matrix}$$

where $c = Q_1^T b$ and z is an arbitrary $(n-r)$ vector. The column pivoting strategy just described works very well in practice in that if A is close to a rank-deficient matrix it

will in general produce a small trailing submatrix R_{22}. It is widely used and for that reason we refer to it also as the "traditional" column pivoting strategy. One of the rare examples where it fails will be presented in the next section.

The traditional pivoting strategy is geared towards extracting a reasonably independent set of vectors from the columns of A (this is in general referred to as the "subset selection problem" [52, p. 414]). It does not take the right-hand side b into account at all. Another pivoting strategy that is geared towards decreasing the least squares residual

$$\|Ax_{LS} - b\|_2^2$$

as quickly as possible is suggested in [52, exercise P6.4-8]. Here the next column of A chosen is always the one that maximizes the decrease in the current residual. This strategy will not in general produce a well-conditioned set of columns from A as b will contain components in the direction of the left-singular vectors of A corresponding to the almost zero singular values. It will however produce a solution with a lower least squares residual. Depending on the application this might be preferable. It seems that deflation methods[15,17,100] would be useful in this context.

It should be noted that the decision on the numerical rank of A is a delicate one that depends very much on the problem area. It has been shown however [51] that a well-defined gap in the singular values of A is necessary to make a sensible decision on the numerical rank of A and we tacitly assume that.

2.3 The Rank-Revealing QR Factorization

As already mentioned, the traditional column pivoting strategy works very well in practice but one can construct matrices that are almost rank deficient where the traditional pivoting strategy will not produce a small trailing submatrix R_{22}. A well known example (originally suggested by Kahan) is

$$A_n = \mathrm{diag}(1, s, s^2, \ldots, s^{n-1}) \begin{pmatrix} 1 & -c & \cdots & \cdots & -c \\ 0 & 1 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 1 & -c \\ 0 & \cdots & \cdots & 0 & 1 \end{pmatrix} + \mathrm{diag}(n\epsilon, (n-1)\epsilon, \ldots, \epsilon) \quad (2.9)$$

where $c^2 + s^2 = 1$ 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.

The most reliable way to arrive at a decision about the rank of A (and solve the least squares problem if desired) is to compute the singular value decomposition

$$A = U\Sigma V^T \quad (2.10)$$

of A. Here U and V are orthogonal matrices whose columns are the left- and right-singular vectors of A, respectively. $\Sigma = \mathrm{diag}(\sigma_i)$ contains the singular values $\sigma_1 \geq \ldots \geq \sigma_n$ of A. However the SVD requires several times as many flops as the QR factorization to compute [52, p. 175] and for that reason a QR factorization is usually preferred.

So the general question is whether for a nearly rank deficient A we can find a permutation P such that the QR factorization (without pivoting) of AP exhibits a small R_{22}. Chan called such a factorization a *rank-revealing QR factorization.* In the case where A is rank-1 deficient, it turns out that this permutation is determined by the right singular vector of A corresponding to the smallest singular value. This was first pointed out in [51].

Theorem 2.1 *Let $A = U\Sigma V^T$ be the singular value decomposition of A, let $V = [v_1, \ldots, v_n]$ be the columns of V and let Π be the permutation the permutes the largest element (in modulus) of v_n into the n-th position, i.e. $\Pi v_n = y$ with $\mid y_n \mid = \|y\|_\infty$. Then if $A\Pi = QR$ is the QR factorization of $A\Pi$, then*

$$\mid r_{nn} \mid \leq \sqrt{n}\, \sigma_n.$$

This means that not the full SVD of A is needed to reveal the rank of A but only the singular vector v_n. Given any QR factorization of A, v_n can be computed via inverse iteration in $O(n^2)$ flops. If we then exchange columns n and pvt where

$$\mid (v_n)_{pvt} \mid = \|v_n\|_\infty$$

and update the QR factorization then according to the previous theorem $\mid r_{nn} \mid$ overestimates σ_n by at most a factor of $\sqrt{n}$ and hence can be used as a reliable indicator of whether A is rank-deficient or not.

To solve the problem for the case where A is nearly rank-r deficient with $r > 1$, this idea was extended by Chan [16] and Foster [46] to higher dimensions. The idea is first to compute any QR factorization of A and then "peel off" the small singular values of R one by one by computing an approximate singular vector at each step. Chan also showed how to compute upper and lower bounds $\hat{\sigma}_i$ and $\check{\sigma}_i$ of σ_i. This allows to implement the algorithm in an adaptive fashion: The algorithm can be terminated if the lower bounds indicate that all the small singular values have been revealed. Chan proves that if r is not too large then his algorithm will correctly identify the numerical rank of A. A simplified version of the rank-revealing QR algorithm is given in Figure 2.3. The extra work beyond the initial QR factorization is $O(r\, n^2)$.

Compute $AP = Q \begin{pmatrix} R \\ 0 \end{pmatrix}$ for some permutation matrix P.

$i \leftarrow n+1$;

repeat

$i \leftarrow i-1$;

$R_{11} \leftarrow R(1:i, 1:i)$; $R_{12} \leftarrow R(1:i, i+1:n)$; $R_{22} \leftarrow R(i+1:n, i+1:n)$;

Compute the right singular vector v of R_{11} corresponding to $\sigma_{min}(R_{11})$ via inverse iteration.

Determine the permutation Π such that $|(\Pi v)_i| = \|\Pi v\|_\infty$.

Let Π exchange entries k_i and i $(k_i < i)$.

Compute the QR factorization $R_{11}\Pi = \tilde{Q}_1 \bar{R}_{11}$ via a sequence of $2(i-k_i)$ Givens rotations.

$$P \leftarrow P \begin{pmatrix} \Pi & 0 \\ 0 & I_{n-i+1} \end{pmatrix}; \quad Q \leftarrow Q \begin{pmatrix} \tilde{Q}_1 & 0 \\ 0 & I_{n-i+1} \end{pmatrix}; \quad R \leftarrow \begin{pmatrix} \bar{R}_{11} & \tilde{Q}_1^T R_{12} \\ 0 & R_{22} \end{pmatrix};$$

Compute $\check{\sigma}_i$ and $\hat{\sigma}_i$ such that $\check{\sigma}_i \leq \sigma_i \leq \hat{\sigma}_i$.

until$(\check{\sigma}_i > \textit{threshold})$

Figure 2.3: Computing a Rank-Revealing QR Factorization

2.4 Jacobi SVD Algorithms and the QR Factorization

As already pointed out in the context of the rank-deficient least-squares problem, the SVD requires several times as many floating point operations as the QR decomposition. It is however a much more versatile decomposition since in addition to the singular values it can reveal the numerical range and null space of A and A^T exactly. Some examples of its use can be found in chapter 12 of [52]. The QR factorization is however an important part of most SVD algorithms and we already mentioned Chan's algorithm [14]. In this section we will comment on the use of the QR factorization in the context of parallel Jacobi SVD schemes.

To compute the SVD on parallel machines, Jacobi algorithms have received widespread attention [4,5,11,19,21,28,43,76,78,89]. Here a matrix is reduced to diagonal form by iteratively solving a series of subproblems of the form

$$P_{ij} \equiv \begin{pmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{pmatrix} . \tag{2.11}$$

Jacobi [67] originally designed a method for the solution of the symmetric eigenvalue problem (see [45,52,105]). We choose a Givens rotation

$$G_{ij} = \begin{pmatrix} c_{ij} & s_{ij} \\ -s_{ij} & c_{ij} \end{pmatrix}, c_{ij} = cos\phi_{ij}, s_{ij} = sin\phi_{ij}$$

such that for

$$\begin{pmatrix} \bar{a}_{ii} & \bar{a}_{ij} \\ \bar{a}_{ji} & \bar{a}_{jj} \end{pmatrix} \equiv G_{ij}^T P_{ij} G_{ij}$$

we have

$$\bar{a}_{ij}^2 + \bar{a}_{ji}^2 \leq \theta^2 (a_{ij}^2 + a_{ji}^2) \tag{2.12}$$

for some $0 \leq \theta < 1$. Let $\tilde{A}$ be the matrix we obtain by updating rows and columns i and j of A with G_{ij}^T and G_{ij} respectively and define

$$\text{off}(A) \equiv \sum_{i \neq j} a_{ij}^2.$$

Then it is not hard to see that

$$\text{off}(\tilde{A}) \leq \text{off}(A) - (1 - \theta^2)(a_{ij}^2 + a_{ji}^2).$$

Since A can be considered diagonal when

$$\text{off}(A) = O(\epsilon \| A \|_F)$$

where ϵ is the machine precision, we can reduce A to diagonal form by solving a judiciously chosen sequence of two-by-two subproblems. The most common sequence is the cyclic-by-row ordering

$$(1,2),(1,3),\ldots,(1,n),(2,3),(2,4)\ldots,(2,n),\ldots,(n-1,n).$$

If the rotation angles are restricted to

$$| \phi_{ij} | < \frac{\pi}{2} \tag{2.13}$$

then the Jacobi method will ultimately converge quadratically. Global convergence is guaranteed by the cyclic-by-row ordering or alternatively thresholding, i.e. skipping a subproblem when

$$a_{ij}^2 + a_{ji}^2 < \tau$$

for a chosen threshold τ.

Jacobi methods are attractive in the parallel setting as subproblems P_{ij} and P_{kl} can be solved concurrently if $i \neq k$ and $j \neq l$. The cyclic-by-row ordering is obviously not very amenable to concurrency and so for parallel machines other orderings have

been suggested (see [6,11,43,79] for some examples). Some of these orderings can be shown to be equivalent to the cyclic-by-row ordering [80,95] and so global convergence is guaranteed without thresholding.

Kogbetliantz [69] generalized Jacobi's method to compute the SVD of a square matrix A. If A is not square, the QR factorization is used to reduce it to a square matrix. By determining Givens rotations $G1_{ij}$ and $G2_{ij}$ such that (2.12) holds for

$$\begin{pmatrix} \tilde{a}_{ii} & \tilde{a}_{ij} \\ \tilde{a}_{ji} & \tilde{a}_{jj} \end{pmatrix} \equiv G1_{ij}^T P_{ij} G2_{ij}$$

a nonsymmetric A can be reduced to diagonal form. Unfortunately the rotation angles defining $G1_{ij}$ and $G2_{ij}$ are no longer guaranteed to fulfill (2.13) and as a result quadratic convergence of Kogbetliantz's method could not be proven for a long time although it was observed experimentally. Recently however quadratic convergence was shown in the case where the subproblems (2.11) are triangular and the norm-decreasing transformations $G1_{ij}$ and $G2_{ij}$ maintain triangularity [18,53,84]. The QR factorization is used here even for a square matrix to reduce it initially to upper triangular form.

A different approach for computing the SVD was taken by Hestenes [61]. By implicitly applying the Jacobi method previously described to $A^T A$, this algorithm computes an orthonormal $n \times n$ matrix V such that

$$Q \equiv AV = [q_1, \ldots, q_n]$$

is an $m \times n$ matrix with orthogonal columns q_i. The short form of the SVD of A is then obtained by setting

$$\sigma_i \equiv \|q_i\|_2, i = 1, \ldots, n$$

and scaling the columns of Q

$$U \equiv QD^{-1}, \text{where} D \equiv diag(\sigma_1, \ldots, \sigma_n).$$

The advantage of this method is that transformations have to be applied to A only from the right which can be advantageous in parallel or paged environments. On the other hand, care has to be taken in Hestenes' method in the presence of small singular values. If σ_i is small, then q_i will in all likelyhood not be orthogonal to the other columns of Q. This means that during the iteration process thresholding has to be applied to prevent q_i from corrupting other columns of Q [21,28]. Furthermore in the presence of small singular values Q need not be orthogonal if the method has converged, i.e. Q^TQ is diagonal to working precision. Hansen [21] suggests a reorthogonalization of U via a Gram-Schmidt method but a QR factorization could be used as well.

It also has been observed experimentally [21,53] that convergence of the described methods can be increased by preprocessing A using a QR factorization with column pivoting as described in section 2.2. The intuitive reason is that a QR factorization with column pivoting tends to grade the columns of R in order of decreasing column norm which is similar to the final SVD where $\sigma_1 \geq \cdots \geq \sigma_n$.

It can be seen from this brief overview that an efficient implementation of the QR factorization is a necessary condition for a good implementation of Kogbetliantz's or Hestenes' method for computing the SVD. Some implementations of Kogbetliantz's method are described in [7,5,19,53,78,89,93,101], Hestenes' method is used in [4,21,28, 43,76].

Chapter 3

The Pipelined Householder QR Algorithm with Adaptive Blocking

In this chapter we present an algorithm for computing the QR factorization without pivoting on a ring of vector processors. As was pointed out in chapter 1 we rely on this simple machine model to design an algorithm that is reasonably portable between parallel machines with different architectures.

The traditional Householder QR algorithm of Figure 2.1 processes the columns of A from the left to the right and so the distribution of the columns of A to processors in a round-robin fashion is quite a natural approach. This leads to the pipelined Householder QR algorithm which is described in detail in section 3.1.

Due to its reliance on matrix-vector multiplication and rank-one updates this algorithm is not optimal with regard to exploiting possible available vector processing hardware. An algorithm relying on matrix-matrix operations and high-rank updates would be much more suited to such an architecture. To that end we introduce in section 3.2 the *WY Representation* which allows us to bundle a sequence of Householder updates into one convenient update. Using the WY representation we can formulate

a block version of the Householder QR algorithm that is more appropriate for vector processing machines.

The WY representation can also readily be integrated into the pipelined Householder QR algorithm to increase the execution speed on each node of a parallel machine. Unfortunately a fixed blocking strategy is not the optimal choice for our parallel algorithm. Choosing large blocks results in good efficiency per node but poor load balancing while narrow blocks lead to a load balanced computation but fail to fully exploit the arithmetic capabilities of each node. We reconcile this tradeoff with an adaptive blocking strategy that is based on a model of the critical path of the algorithm. We also present numerical results gained on the Intel iPSC/1 hypercube.

3.1 The Pipelined Householder QR Algorithm

We parallelize the Householder QR algorithm of Figure 2.1 by distributing 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*[1]. Each processor executes the algorithm given in simplified form in Figure 3.1 (to save space the abbreviation "HH" is used 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 neighbour of $proc_k$, respectively.

The application and generation of Householder vectors is staggered across processors so that the computation proceeds in what can be described a "pipelined" fashion.

[1]Since we are pursuing a column-oriented scheme, we are assuming that n is reasonably larger than p. If that is not the case then a row-oriented scheme like that of [27] would be more appropriate.

processor $proc_k$

$lcnt \leftarrow 0$; {counter for HH vectors generated in $proc_k$}

$gcnt \leftarrow 0$; {counter for HH vectors generated globally}

if $(k = 0)$ **then** {generate initial HH vector}

$u \leftarrow genhh(c(:,1))$; send u to p_{right};

$c(:, 1:cols_0) \leftarrow apphh(u, c(:, 1:cols_0))$; $lcnt \leftarrow gcnt \leftarrow 1$;

end if

while $(lcnt < cols_k)$ **do** {main loop}

receive u from p_{left}; $gcnt \leftarrow gcnt + 1$;

if (u not generated by p_{right}) **then** send u to p_{right} **end if**

if $(k = gcnt \bmod p)$ **then** { my turn to generate HH vector }

$lcnt \leftarrow lcnt + 1$;

$c(gcnt:m, lcnt) \leftarrow apphh(u, c(gcnt:m, lcnt))$; {update first column}

$\hat{u} \leftarrow genhh(c(gcnt+1:m, lcnt))$; { generate new HH vector }

if $(gcnt + 1 < n)$ **then** send $\hat{u}$ to p_{right} **end if**

$c(gcnt:m, lcnt+1:cols_k) \leftarrow apphh(u, c(gcnt:m, lcnt+1:cols_k))$;

{ complete previous HH update }

$gcnt \leftarrow gcnt + 1$;

$c(gcnt:m, lcnt:cols_k) \leftarrow apphh(\hat{u}, c(gcnt:m, lcnt:cols_k))$;

{ apply new HH update }

else

$c(gcnt:m, lcnt+1:cols_k) \leftarrow apphh(u, c(gcnt:m, lcnt+1:cols_k))$;

{ simply apply HH update }

end if

end while

Figure 3.1: The Pipelined Householder QR Algorithm

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 interfacing the QR algorithm to other algorithms easy, for example to an equation solving routine. It should be noted here that the effecive solution of triangular linear equation systems on distributed architectures is not an easy problem and has received much attention lately [56,57,73,74]. 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 average, each processor receives and sends every Householder vector once resulting in n sends and receives and a total of $mn - \frac{n^2}{2}$ 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-neighbour 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's mailbox) then we can hope to further decrease communication overhead. For these reasons this pipelining technique has been widely used [48,66,73,83] also for other factorizations.

We performed experiments with the pipelined QR algorithm on a 16-node Intel iPSC/1 hypercube at the Cornell Theory Center (see [42] for a detailed description of this machine). A Gray code mapping [88] was used to embed the ring of processors. The code was written entirely in Fortran, 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. in single precision. We mention that the Intel iPSC implements asynchronous communication primitives. In implementing *genhh* and *apphh* we used the assembler Blas 1 routines provided by Intel to take full advantage of the Intel 80287 numerical coprocessor in each node. Experimental results for matrices of size 750×125 and 750×250 are shown in Table 3.1.

Table 3.1: The Pipelined Householder QR Algorithm on the Intel iPSC

Problem Size	p	t_{max}	Δt	pt_{genhh}	pt_{apphh}	pt_{ohead}
750×125	16	61	6.4	2.1	84.7	13.0
	8	110	1.6	2.3	91.7	6.0
	4	210	0.4	2.4	95.1	2.4
750×250	16	205	1.5	1.1	92.5	6.2
	8	393	0.4	1.2	95.9	2.9
	4	772	0.1	1.2	97.6	1.2

Here $t_{max}\,(t_{min}, t_{avg})$ are the maximal (minimal, average) processor execution times in seconds and

$$\Delta t = \frac{t_{max} - t_{min}}{t_{avg}} * 100 \tag{3.1}$$

is a measure of the load imbalance. pt_{genhh} and pt_{apphh} are the percentages of the total execution time that each node spends on average for the generation of Householder vectors and their application. pt_{ohead} was obtained by bracketing the send/receive calls and measures the average percentage of the execution time that a processor spends communicating and waiting idle.

It can be seen that the algorithm performs nicely. As long as each node has a reasonable number of columns to process, communication overhead is low, the computation is nicely load balanced and as a result the algorithm is very efficient. The

750×125 example on 16 processor also shows that the pipelining scheme deteriorates once a node does not have enough columns to process. In this case the startup and wind-down phase of the algorithm in which some processors are idle takes up a substantial portion of the overall execution time. This will be explained in more detail in section 3.3.

3.2 The Compact WY Representation

Looking again at Table 3.1 we notice by how much the application of Householder vectors (the *application step*) dominates the running time of the algorithm. This is despite the fact that benchmarks on one node show *genhh* (the *reduction step*) executing at an average rate of 18.6 KFlops which is slower than the average execution rate of 34.8 KFlops for *apphh*. The same qualitative observation was made when implementing the traditional Householder QR algorithm on vector processing machines. There it was also noted that matrix-matrix operations can execute much faster than matrix-vector or vector-vector operations due to the already mentioned surface-to-volume effect of data movement to arithmetic operations [38].

The rationale behind block algorithms is then the following: If we view the matrix as a collection of matrix blocks, then the update steps will in a natural fashion be matrix-matrix operations provided we can formulate the reduction of a block matrix as a single matrix operation instead of a series of "small" updates. This block reduction will in all likelyhood require some more floating point operations, but the increased speed of the update should more than make up for the extra cost. This is indeed the case [8,40,47,90] and as a result block algorithms will play a major role in the LAPACK project [29] which is currently underway to provide the functionality of the LINPACK and EISPACK subroutine packages [31,96] with algorithms more suitable for todays high performance computers.

To arrive at a block formulation of the Householder QR algorithm we must be able to express a series of Householder reductions in a convenient closed form. Bischof and Van Loan [8] expressed the product

$$Q = H_1 \cdots H_b$$

of a series of $m \times m$ Householder matrices

$$H = I - 2uu^T, \|u\|_2 = 1$$

in the so-called *WY Representation*

$$Q = I + WY^T \tag{3.2}$$

where W and Y are $m \times b$ matrices. Schreiber and Van Loan [92] and independently DuCroz [41] refined this representation by expressing $W = YT$ where T is a $b \times b$ upper triangular matrix. Schreiber and Van Loan called the resulting representation

$$Q = I + YTY^T \tag{3.3}$$

the *Compact WY Representation* since it requires only about half as much storage as the original WY representation (3.2) in the typical case where $m \gg b$. 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

$$\begin{aligned} \tilde{Y} &= \begin{pmatrix} Y & u \end{pmatrix} \\ z &= -2TY^Tu \\ \tilde{T} &= \begin{pmatrix} T & z \\ 0 & -2 \end{pmatrix}. \end{aligned} \tag{3.4}$$

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. The accumulation of T requires

$$flops_{accT} = mb^2 - mb - \frac{b^3}{3}$$

flops and $\frac{b^2}{2}$ extra words for storage. Since typically $m \gg b$ 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

$$Z \leftarrow A^T Y T \tag{3.5}$$

and a rank-b update

$$A \leftarrow A + Y Z^T \tag{3.6}$$

instead of a series of b matrix-vector multiplications (2.3) and rank-one updates (2.4).

We can now express the block Householder QR algorithm in terms of the primitives *gencwy* (generate compact WY factor) and *appcwy* (apply compact WY factor):

$$[Y, T] \leftarrow gencwy(A)$$

returns the compact WY factors T and Y such that

$$A = (I + YTY^T) \begin{pmatrix} R \\ 0 \end{pmatrix}.$$

gencwy first computes the QR factorization of A using the traditional Householder QR algorithm of Figure 2.1 and then accumulates T as described in (3.4).

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

performs the updates (3.5) and (3.6). Figure 3.2 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 $b_m \times b_n$, so $m = Mb_m$ and $n = Nb_n$. 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. Notice the similiarity between the block QR algorithm and its unblocked counterpart of Figure 2.1.

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

$[Y,T] \leftarrow gencwy(A(i:M,i))$

$A(i:M,i:N) \leftarrow appcwy\ (Y,T,A(i:M,i:N))$

end for

Figure 3.2: The Block QR Factorization Algorithm

The generation of the compact WY factors requires

$$flops_{gencwy} = 3mnb_n - \frac{3}{2}n^2b_n \tag{3.7}$$

flops while the application of the compact WY factors requires

$$flops_{appcwy} = 2mn^2 - \frac{2}{3}n^3 + \frac{n^2}{b_n}(\frac{m}{2} - \frac{n}{6}) + b_n(\frac{5}{2}n^2 + 2mn) \tag{3.8}$$

flops for a total cost of

$$flops_{blockQR} = 2mn^2 - \frac{2}{3}n^3 + \frac{n^2}{b_n}(\frac{m}{2} - \frac{n}{6}) + n^2b_n + 5mnp\,. \tag{3.9}$$

Comparing these flop counts with (2.5), (2.6) and (2.7) we see that the block algorithm is somewhat more expensive in terms of floating point operations. But when these floating point operations are on a vector processing machine executed faster than the ones for the unblocked algorithm, we still come out ahead. Bischof and Van Loan [8], Harrod [54] and Mayes [81] used the WY representation to compute the QR

factorization on the FPS-164/MAX, the Alliant FX/8 and the IBM 3090, respectively. Van Loan [102] used it in the context of a block QR scheme on a shared memory multiprocessor. We also mention that a slightly different block Householder formulation using the inverse of a triangular matrix instead of T was given by Walker [104] and that a more general mathematical framework for block orthogonal transformation was developed by Schreiber and Parlett [91].

3.3 Variable Blocking

Introducing blocking into the pipelined QR algorithm is as easy as for the uniprocessor algorithm. The top level structure stays the same; the algorithmic change is captured in the transition from scalar to block entries and in the transition from *genhh* to *gencwy* and *apphh* to *appcwy*. We also now send compact WY factors instead of Householder vectors. On a distributed machine with vector processing nodes this can be of considerable advantage since the block QR updates are better suited to exploiting the vector capabilities of each node.

A compact WY factor $[Y,T]$ where Y is $m \times b$ and T is $b \times b$ can be transmitted as one message of length mb by exploiting the fact that the upper triangle of Y is zero and the diagonal entries of T are all -2. The unblocked scheme on the other hand would transmit b Householder vectors of length $m, m-1, \ldots, m-b+1$ to effect the same update. So the number of words transmitted in the block pipelined scheme is somewhat higher but since typically $m \gg b$ this increase should be negligible. On the other hand the total number of messages sent is decreased in the block scheme which might be of advantage for systems with large startup costs.

We implemented the block pipelined scheme on the Intel iPSC/1 hypercube since it was the only usable distributed memory machine at Cornell. This machine is not an optimal choice in that the Intel 80287 numerical coprocessor in each node implements

scalar floating point arithmetic and so the possible gains from block algorithms will be slight at best. Intel also provides Blas 1 assembler kernels for their machine, but for the Blas 2 and 3 kernels we had to resort to the Fortran versions distributed by NETLIB [33]. As a result the *gencwy* and *appcwy* kernels using the Blas 2 and 3 kernels execute in fact slower than the *genhh* and *apphh* kernels using Blas 1 calls. To illustrate, the average KFlop rate for *apphh* on a single node was 34.8 KFlops whereas *appcwy* executed at 25.7 KFlops.

The block primitives *gencwy* and *appcwy* do however exhibit the general behavior that we expect from block reductions in that wider blocks result in higher execution rates. This is demonstrated in Table 3.2 which shows the execution rate in KFlops of *gencwy* for a $300 \times b$ block factor as well as the execution rate of *appcwy* for applying a $300 \times b$ factor to a 300×100 matrix for b in the range from 1 to 8. So although the Intel iPSC is not a machine that will take advantage of our block scheme we will be able to illustrate the important points of our block pipelined algorithm. It should be noted however that on a true vector processor the increase in execution rate with b would be more pronounced.

Table 3.2: Execution Rates of *gencwy* and *appcwy*

b	1	2	3	4	5	6	7	8
gencwy	14.45	18.89	20.79	21.83	22.48	22.92	23.24	23.47
appcwy	21.27	23.65	24.61	25.07	25.30	25.40	25.43	25.45

The performance of the block pipelined algorithm on a 500×300 and 500×150 problem using 16 processors and different block widths is shown in Table 3.3. Here b is the block width, t_{tot} is the maximal execution time, Δt as defined in (3.1) is a measure of the load imbalance, pt_{ohead} is the average percentage of the overall execution time spent communicating and waiting idle and kw_{tot} is the average observed kflop rate per

node ignoring overhead. As expected an increase in the block width does result in an increase of the execution rate per node. But we also notice that wider blocks result in higher communication overhead and increased load imbalance. This unwelcome effect counteracts the potential gain resulting from the faster execution rate per node.

Table 3.3: The Pipelined Block QR Algorithm Using Blocks of Fixed Width

Problem Size	b	t_{max}	Δt	pt_{ohead}	kw_{tot}
500×150	1	79.5	3.6	8.6	17.8
	2	75.4	11.7	15.9	21.3
	3	79.9	23.8	21.4	22.8
	4	88.7	37.0	27.8	23.6
	5	99.4	57.6	32.0	24.1
500×300	1	264	0.8	4.5	17.0
	2	233	2.5	8.3	20.7
	3	230	5.0	11.8	22.3
	4	236	8.4	15.3	23.2
	5	244	12.7	18.4	24.0

To gain some understanding of this phenomenon, notice that if $n = Nb$ then a block pipelined algorithm using blocks of width b behaves very much like the unblocked algorithm for an $m \times N$ problem except that messages sent are bigger and reduction and application steps take longer. Increasing the block width b or alternatively reducing the number of block columns N then increases the relative cost of the startup and wind-down phase of the algorithm. This was already observed in Table 3.1 for the unblocked algorithm on a 750×125 problem.

While processor $proc_0$ performs the first reduction step, all other processors are idle. Then the first factor has to be passed through the ring of processors so that all

processors can start work. The bigger the block width b the longer this first step will take. Towards the end of the computation on the other hand, wider blocks increase the likelihood that the next processor in line will complete its update before the current processor has generated a new compact WY factor and transmitted it. As a result, the processor after the one currently computing the next reduction step will be idle. This shows that wide block columns are not advantageous in the first and final steps of the block pipelined algorithm.

So we have a tradeoff: narrow blocks decrease the cost of the first and final phases of the algorithm but do not fully exploit floating point hardware. Wide blocks on the other hand exhibit good floating point performance at the cost of very expensive startup and wind-down phases. To reconcile these two trends we abandon fixed width blocks and allow block columns of variable widths. Variable blocking allows us to use narrow block column widths in the first and final stages of the algorithm while processing wide columns in between when a substantial amount of work still is to be done on each node. It is important to note that the structure of the pipelined block algorithm remains unchanged by the introduction of variable blocking. The only difference is that block columns are not all of the same width any more.

The pipelined Householder block QR algorithm with variable width blocks is shown in Figures 3.3 and 3.4. Here N is the number of block columns, the array C is local to each processor and contains the $COLS_k$ block columns assigned to $proc_k$ ($\sum_{i=0}^{p-1} COLS_k = N$). Again we assume that block columns have been dealt out in a round-robin fashion. The array *width* describes the width of the block columns such that $width(i), i = 1, \ldots, COLS_k$ is the number of columns in local block column number i. $ln \equiv \sum_{i=1}^{COLS_k} width(i)$ is the number of columns contained in $proc_k$ and $roff$ and $coff$ are such that $c(roff+1:m, coff+1:ln)$ is the matrix that is still left to process in $proc_k$. "CWY" is used as abbreviation for "compact WY".

In much the same manner we can accumulate Q if so desired. If $N(A)$ defines the

processor $proc_k$

$lcnt \leftarrow 0$; {counter for CWY factors generated in $proc_k$}

$gcnt \leftarrow 0$; {counter for CWY factors generated globally}

$roff \leftarrow 0$; {row offset for current subproblem}

$coff \leftarrow 0$; {column offset for current subproblem}

$ln \leftarrow \sum_{i=1}^{COLS_k} width(i)$; { number of local columns }

if $(k = 0)$ **then** {generate initial CWY factor}

$[Y, T] \leftarrow gencwy(c(:, 1:width(1)))$; send $[Y, T]$ to p_{right};

$c(:, 1:ln) \leftarrow appcwy(Y, T, c(:, 1:ln))$;

$lcnt \leftarrow gcnt \leftarrow 1$;

$roff \leftarrow coff \leftarrow width(1)$;

end if

Figure 3.3: The Pipelined Block Householder QR Algorithm with Variable Blocking: Startup Phase

processor $proc_k$

while $(lcnt < COLS_k)$ **do**

receive $[Y,T]$ from p_{left} ; $gcnt \leftarrow gcnt + 1$;

if ($[Y,T]$ not generated by p_{right}) **then** send $[Y,T]$ to p_{right} **end if**

$ncols \leftarrow$ number of columns in Y;

if $(k = gcnt \bmod p)$ **then** { my turn to generate CWY factor }

$lcnt \leftarrow lcnt + 1$; $tmp\_coff \leftarrow coff + width(lcnt)$;

$c(roff+1:m, coff+1:tmp\_coff)$ {update first block column}

$\leftarrow appcwy(Y,T,c(roff+1:m, coff+1:tmp\_coff))$;

$[\hat{Y},\hat{T}] \leftarrow gencwy\ (c(roff+ncols+1:m, coff+1:tmp\_coff))$;

{ generate new CWY factor }

if $(gcnt + 1 < N)$ **then** send $[\hat{Y},\hat{T}]$ to p_{right} **end if**

$c(roff+1:m, tmp\_coff+1:ln)$ { complete previous CWY update }

$\leftarrow appcwy(Y,T,c(roff+1:m, tmp\_coff+1:ln))$;

$roff \leftarrow roff + ncols$; $gcnt \leftarrow gcnt + 1$;

$c(roff+1:m, coff+1:ln) \leftarrow appcwy(\hat{Y},\hat{T},c(roff+1:m, coff+1:ln))$;

{ apply new CWY update }

$roff \leftarrow roff + width(lcnt)$; $coff \leftarrow tmp\_coff$;

else

$c(roff+1:m, coff+1:ln) \leftarrow appcwy(Y,T,c(roff+1:m, coff+1:ln))$;

{ simply apply CWY update }

$roff \leftarrow roff + ncols$;

end if

end while

Figure 3.4: The Pipelined Block Householder QR Algorithm with Variable Blocking: Main Loop

nullspace of A and $R(A)$ its nullspace, then with

$$\begin{array}{cccc} Q = [& Q_1 & , & Q_2 \quad] \\ & n & & m-n+1 \end{array}$$

we have

$$span\,(Q_1) = R(A) \text{ and } span\,(Q_2) = N(A^T).$$

It should be noted that variable blocking is not of concern when performing the accumulation of Q, Q_1 or Q_2 since there is no reduction step delaying the propagation of the compact WY factors. It is however natural to maintain the blocking used for A for Q_1 and to distribute Q_1 in the same wrap-around fashion as A. A simplified version of the algorithm to accumulate Q_1 is given in Figure 3.5. Q_1 is accumulated in reverse order to save floating point operations (compare [52, p.148]).

We assume here that Q_1 has been initialized to the identity, that $[Y_i, T_i]$ is the ith compact WY factor that was generated in $proc_k$ and that the primitive

$$A \leftarrow appcwy2(Y, T, A)$$

computes the update $A \leftarrow QA$. Lastly assume that $roff, coff$ are such that $c(roff+1:m, coff+1:ln)$ was the last subproblem reduced in $proc_k$ when the QR factorization of A was computed and that $proc_{last}$ was the processor that reduced the last block column of A. $q_{current}$ is used as an abbreviation for $q(roff+1:m\,, coff+1:ln)$.

Q_2 can be accumulated in much the same fashion. Since there is no natural blocking defined for Q_2 by the blocking of A, a fixed width blocking strategy would be adequate for Q_2.

3.4 An Adaptive Blocking Strategy

To determine a good blocking strategy for the QR factorization of A we employ a model of the critical path of the pipelined algorithm. As was already mentioned before

processor $proc_k$

$lcnt \leftarrow COLS_k$; $gcnt \leftarrow N - (last - k) - 1$;

if $(k > last)$ **then** $gcnt \leftarrow gcnt - p$ **end if**

{$gcnt$ counts how many CWY factors $proc_k$ still has to apply }

$remember \leftarrow gcnt \bmod p$; { initial position w.r.t. p_{last} }

if $(k \neq last)$ **then** send $[Y_{lcnt}, T_{lcnt}]$ to p_{right} **end if**

$q_{current} \leftarrow appcwy2(Y_{lcnt}, T_{lcnt}, q_{current})$;

while $(gcnt > 0)$ **do** {main loop}

receive $[Y, T]$ from p_{left} ; $ncols \leftarrow$ number of columns in Y;

$gcnt \leftarrow gcnt - 1$; $roff \leftarrow roff - ncols$;

if (([Y, T] not generated by p_{right}) **and** $(gcnt + 2 < N)$) **then**

send $[Y, T]$ to p_{right}

end if

if $(((gcnt - 1) \bmod p = remember)$ **and** $(gcnt > 0))$ **then**

$lcnt \leftarrow lcnt - 1$; $gcnt \leftarrow gcnt - 1$; send $[Y_{lcnt}, T_{lcnt}]$ to p_{right}

$q_{current} \leftarrow appcwy2(Y, T, q_{current})$; { apply previous CWY factor }

$roff \leftarrow roff - width(lcnt)$; $coff \leftarrow coff - width(lcnt)$;

$q_{current} \leftarrow appcwy2(Y_{lcnt}, T_{lcnt}, q_{current})$; { apply current CWY factor }

else

$q_{current} \leftarrow appcwy2(Y, T, q_{current})$; { simply apply CWY factor }

end if

end while

Figure 3.5: Accumulating Q_1 in Reverse Order

swift generation of new Housholder or compact WY factors is crucial in preventing the next processor in line from being idle. So the critical path is that *from the point that a compact WY factor has been forwarded, the generation and transmission of a new compact WY factor should not take longer than the time it takes the next processor to apply the just forwarded update.* Implicit here is the assumption that the next processor is just ready to receive the next update when it is being delivered. This is a reasonable conservative assumption in that it allows for the next processor still being busy completing a previous update but rules out that a processor is sitting idle.

To be more precise, assume that processor $proc_i$ still has to process an $\tilde{m} \times n_i$ matrix and that its right neighbour $proc_{i+1}$ still has a $\tilde{m} \times n_{i+1}$ matrix to deal with. Let $t_{gencwy}(m,b)$ be the time it takes to generate a compact WY factor for an $m \times b$ matrix, $t_{appcwy}(m,n,b)$ the time it takes to apply an $m \times b$ factor to an $m \times n$ matrix and $t_{send}(m)$ the time it takes to deliver m words to a neighbouring processor. $proc_i$ now receives a compact WY factor of width $\tilde{b}$ and forwards it to $proc_{i+1}$. If processor $proc_{i+1}$ now receives the compact WY factor of width $\tilde{b}$ immediately when it is delivered and processor $proc_i$ decides to generate a new compact WY factor of width b ($b \leq n_i$) then in order to fulfill our critical path requirement we have to make sure that

$$t_{appcwy}(\tilde{m}, b, \tilde{b}) + t_{gencwy}(\tilde{m}, b) + t_{send}(\tilde{m}b) \leq t_{appcwy}(\tilde{m}, n_{i+1}, \tilde{b}). \tag{3.10}$$

In other words, the time that $proc_i$ takes to apply the previous compact WY factor of width $\tilde{b}$ to the b columns defining the new update plus the time for computing the new compact WY factor and transmitting it should be no more than the time it takes $proc_{i+1}$ to apply the update of width $\tilde{b}$ to its $\tilde{m} \times n_{i+1}$ submatrix. We want to choose b as large as possible consistent with this restriction to maximize floating point utilization.

If we model t_{gencwy}, t_{appcwy} and t_{send} we can use (3.10) to model the critical path of the pipelined block QR algorithm. Assuming that we distribute columns equally

at first, we know that the first block should have width $b_1 = 1$ so that the very first compact WY factor can be generated as soon as possible by $proc_0$. Then we can determine the width b_2 of the next compact WY factor to be created in $proc_1$ given that $proc_2$ is applying a factor of width b_1 to its columns. Once b_2 has been created, we then determine b_3 and so on. The resulting algorithm for determining the widths of the block columns a priori (on the host or on the nodes) is given in Figure 3.6. For simplicity we assume that p divides n and that initially columns are distributed evenly among processors. $width(i)$ is the width of the ith block column, $cols\_left(i)$ is the number of columns currently left to process in processor $proc_i$. $stage$ is the number of the block column we are currently generating and $proc$ is the number of the processor who has to generate a new block reduction whose width has to be determined. The function $find\_bsize(\tilde{m}, n_i, \tilde{b}, n_{i+1})$ returns the largest $b \leq n_i$ satisfying (3.10) using a binary search for b on the interval $[1, min(\tilde{b}, n_i)]$.

It is worth pointing out that the widths of the block columns can also be determined dynamically. Each processor can run "its" part of the algorithm in Figure 3.6 whenever it is up to him to generate the next WY factor. The only difference is then that we cannot distribute block columns in a wrap-around fashion beforehand (the blocking is not known yet) and so the dynamic blocking will be equivalent to computing a QR factorization $AP = QR$ with P determined by the blocking strategy.

To model t_{gencwy} and t_{appcwy} we performed runs with $m = \{25, 50, \ldots, 500\}$, $n = \{20, 40, \ldots, 200\}$ and $b = \{1, 2, \ldots, 10\}$ on a single node of the cube and used a least squares fit to arrive at

$$\hat{t}_{gencwy}(m, b) = 1.1149\text{e-}4\, mb^2 + 1.7406\text{e-}3\, b^2 + 7.5502\text{e-}5\, mb + 4.522\text{e-}3 \qquad (3.11)$$

$$\hat{t}_{appcwy}(m, n, b) = 1.4260\text{e-}4\, mnb + 2.8599\text{e-}4\, nb^2 + 7.1034\text{e-}5\, mn + 0.6228 \qquad (3.12)$$

as models for t_{gencwy} and t_{appcwy}. The relative residual of this fit over all data points

```
cols_left(i) ← n/p, i ∈ {0, ..., p − 1}
width(1) ← 1; cols_left(0) ← cols_left(0) − 1;
last_block_size ← 1; total_processed ← 1;
stage ← 2; proc ← 0;
while (total_processed < n) do
    next_proc ← right_neighbour(proc, p);
    if (cols_left(next_proc) > 0) then
        { determine width of next block column }
      block_size ← find_bsize(m − total_processed, cols_left(proc),
                              last_block_size, cols_left(next_proc))
      width(stage) ← block_size;
      cols_left(proc) ← cols_left(proc) − block_size;
      total_processed ← total_processed + block_size;
      last_block_size ← block_size;
    else { clean up leftover columns }
      left_over ← n − total_processed;
      break
    end if
    proc ← next_proc;
    stage ← stage + 1;
end while
while (left_over > 0) do    { redistribute leftover columns }
    width(stage) ← 1; stage ← stage + 1; left_over ← left_over − 1;
end while
```

Figure 3.6: Algorithm Determining Block Widths Using Critical Path Model

was 1.25e-2 for $\hat{t}_{gencwy}$ and 1.50e-2 for $\hat{t}_{appcwy}$. For t_{send} we used the model

$$\hat{t}_{send}(m) = (891 + 7.2\,m)\,10\text{e-}6 \tag{3.13}$$

given by Dunigan [42]. All times are in seconds. The blockings obtained using these models for the 500×150 and 500×300 problem on 16 processors are shown in Figure 3.7. Since under the Intel iPSC operating system version we were using the length of any

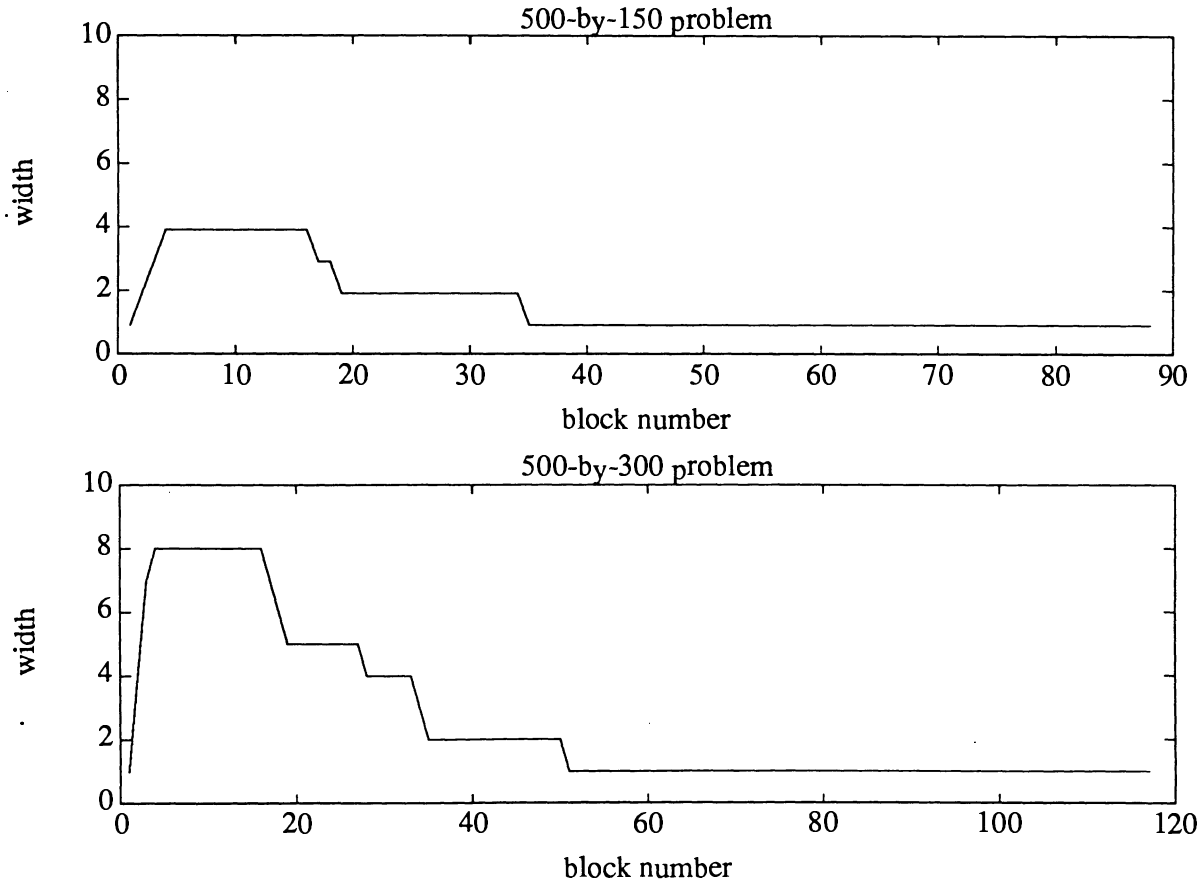

Figure 3.7: Block Widths Determined by Critical Path Model

single message cannot exceed 16K bytes, we restricted the maximal block width to be no more than eight. We see that the 500×150 problem was decomposed into 88 blocks with widths up to 4 and the 500×300 problem was grouped into 118 blocks with widths

up to 8. The block width assignments we obtained for the 500×300 problem on 16 processors are the same that we would have obtained for the 500×150 problem on 8 processors.

We notice the quick increase of the block width at the beginning of the computation and the gradual decrease towards the end. The area under the curve also gives a rough indication of the percentage of the total work already completed. For example, in the 500×300 problem we have performed about 90% of the total computational work after going around the ring twice since the first 32 block columns together make up 188 columns of A. The comparison between the two plots also shows how the variable blocking strategy adapts itself to the problem at hand. Using these variable block widths we then obtain the performance shown in Table 3.4. The variable blocking strategy performs indeed better than any fixed blocking strategy. Floating point utilization per node is high and load balance is good as shown by the low numbers for Δt.

Table 3.4: The Pipelined Block QR Algorithm Using Blocks of Variable Width

Problem Size	t_{max}	Δt	pt_{ohead}	kw_{tot}
500×150	72.1	4.0	16.2	21.6
500×300	229	0.9	16.9	23.7

So our model for the critical path of the block pipelined algorithm has resulted in an adaptive blocking strategy that is easy to compute and performs better than any fixed width blocking strategy. Variable block widths insure good processor utilization while keeping the cost of the first and final stages low. Another advantage is that the variable blocking strategy adapts itself readily to the problem parameters m, n and p and requires no knowledge of the intricacies of the parallel machine once models for t_{gencwy}, t_{appcwy} and t_{send} have been determined (this is an installation procedure that can be

automated). We should keep in mind however that this model is a simple approximation to the asynchronous behaviour of the block pipelined algorithm. For example the adaptive blocking strategy experiences a higher overhead pt_{ohead} than expected and it is not clear why this is the case. Also it seems that incorporating blocking techniques into Givens oriented schemes seems to be more complicated (especially if load balance is to be maintained). Van Loan [102] addresses some of these difficulties in the context of a block version of the Gentleman-Kung scheme for a shared memory machine.

Chapter 4

The Pipelined Householder QR Algorithm with Controlled Local Pivoting

In this chapter we show how pivoting can be incorporated into the pipelined Householder QR algorithm at little additional cost. In section 4.1 we motivate why on a parallel architecture the traditional column pivoting strategy and the rank-revealing QR factorization can be significantly more expensive to compute than the QR factorization without pivoting. As an alternative we suggest a local pivoting scheme that can be incorporated into the pipelined Householder QR algorithm at little extra cost. To make this strategy reliable, we introduce an incremental condition estimation technique which allows us to update our estimate of the smallest singular value of the current R at every step with $O(n)$ work while saving only $O(n)$ words of information between successive steps. In section 4.3 we present numerical experiments that show that the local pivoting strategy behaves about as well as the traditional global pivoting strategy. These experiments also show the advantages of incorporating the incremental

condition estimator into the traditional column pivoting strategy to guard against the known pathological cases.

4.1 Local Pivoting

On a single processor the Householder QR factorization without pivoting 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 [16]. 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. As was shown in section 3.1 the Householder QR algorithm without pivoting can be very efficiently parallelized simply by pipelining the computation. 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 all processors have to synchronize to select the next pivot column. Each processor can easily choose its local candidate pivot column by considering only the columns that are assigned to it. Choosing the global pivot column on the other hand requires that each processor either makes its local pivot information known to all other processors or that a designated processor collects all the local pivot information. So not only does global pivoting introduce considerable extra communication overhead, but it also forces the program into a lockstep mode which results in a serious loss of efficiency on machines that previously could profit from the pipelining.

In Chan's 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. On average this requires two iteration

steps [16] each involving the solution of an equation system of the form

$$R^T R x = y$$

and hence 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 [56,57,73,74] this problem can by no means be parallelized as efficiently as the inital QR factorization. In addition 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 efficiently [26,27]. 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 significant part of the overall computation time on a parallel machine.

The easiest way out of this dilemma is to forgo global pivoting altogether and develop a local pivoting strategy, i.e. each processor limits its choice of pivot columns to the ones it houses. As in the pipelined Householder QR algorithm without pivoting it is important to generate new Householder vectors as quickly as possible. In the pipelined algorithm without pivoting the column determining the next Householder vector is known 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 local 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 Figure 2.2, 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 next pivot column, complete the Householder update on this column and compute the next Householder vector. A simplified version of the resulting algorithm for a ring of processors is given in Figures 4.1 and 4.2. The notation we use is the same as in Figures 2.2 and 3.1 and for simplicity we assume that columns have been dealt out using the wrap mapping.

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 very ill-conditioned but will not necessarily have a small lower right hand block. So in order for the local pivoting strategy to be reliable in identifying a well-conditioned R, we have to *guard against choosing nearly dependent pivot columns.*

processor $proc_k$

$lcnt \leftarrow 0$; {counter for HH vectors generated in $proc_k$}

$gcnt \leftarrow 0$; {counter for HH vectors generated globally}

foreach $i \in \{1, \ldots, cols_k\}$ **do**

$perm_i \leftarrow (k+1) + (i-1)p$; { wrap mapping}

$res_i \leftarrow \|c(:,i)\|_2$;

end foreach

if $(k = 0)$ **then** {determine first pivot column}

$lcnt \leftarrow gcnt \leftarrow 1$;

Let $pvt \in \{1, \ldots, cols_0\}$ be such that res_{pvt} is maximal

$perm_1 \leftrightarrow perm_{pvt}$; $c(:,1) \leftrightarrow c(:,pvt)$; $res_{pvt} \leftarrow res_1$;

$u \leftarrow genhh(c(:,1))$; send u to p_{right} ;

$z \leftarrow C^T u$; $c(1,:) \leftarrow c(1,:) - 2\,u(1)\,z^T$;

foreach $j \in \{2, \ldots, cols_0\}$ **do** { update residuals }

$res_j \leftarrow \sqrt{res_j^2 - c(1,j)^2}$;

end foreach

$c(2\!:\!m,:) \leftarrow c(2\!:\!m,:) - 2\,u(2\!:\!m)\,z^T$; {complete matrix update}

end if

Figure 4.1: The Pipelined QR Algorithm with Local Pivoting: Startup Phase

processor $proc_k$

while $(lcnt < cols_k)$ **do** {main loop}

receive u from p_{left} ; $gcnt \leftarrow gcnt + 1$;

if (u not generated by p_{right}) **then** send u to p_{right} **end if**

if ($k = gcnt \bmod p$) **then** { my turn to generate next HH vector }

$lcnt \leftarrow lcnt + 1$;

{ complete enough of $H(u)$ update to determine next pivot column }

$z \leftarrow c(gcnt\!:\!m, lcnt\!:\!cols_k)^T u$;

$c(gcnt+1, lcnt\!:\!cols_k) \leftarrow c(gcnt+1, lcnt\!:\!cols_k) - 2\,u(1)\,z^T$;

$res_i \leftarrow \sqrt{res_i^2 - c(gcnt, i)^2}$, $i \in \{lcnt, \ldots, cols_k\}$

Let $pvt \in \{lcnt, \ldots, cols_k\}$ be such that res_{pvt} is maximal

$c(gcnt\!:\!m, pvt) \leftarrow c(gcnt\!:\!m, pvt) - 2\,u(2\!:\!m-gcnt+1)\,z(pvt-lcnt+1)$;

$\hat{u} \leftarrow genhh(c(gcnt+1\!:\!m, pvt))$; $gcnt \leftarrow gcnt + 1$;

if $(gcnt < n)$ **then** send $\hat{u}$ to p_{right} **end if**

$c(gcnt\!:\!m, lcnt\!:\!pvt-1)$ { complete $H(u)$ update }

$\leftarrow c(gcnt\!:\!m, lcnt\!:\!pvt-1) - 2\,u(2\!:\!m-gcnt+2)\,z(1\!:\!pvt-lcnt)^T$;

$c(gcnt\!:\!m, pvt+1\!:\!cols_k) \leftarrow c(gcnt\!:\!m, pvt+1\!:\!cols_k)$

$-2\,u(2\!:\!m-gcnt+2)\,z(pvt-gcnt+2\!:\!cols_k-lcnt+1)^T$;

$c(:, pvt) \leftrightarrow c(:, lcnt)$; $perm_{lcnt} \leftrightarrow perm_{pvt}$; $res_{pvt} \leftarrow res_{lcnt}$;

$c(gcnt\!:\!m, lcnt+1\!:\!cols_k)$ { complete $H(\hat{u})$ update }

$\leftarrow apphh(\hat{u}, c(gcnt, lcnt+1\!:\!cols_k))$;

else { apply $H(u)$ update }

$c(gcnt\!:\!m\,, lcnt+1\!:\!cols_k) \leftarrow apphh(u, c(gcnt\!:\!m\,, lcnt+1\!:\!cols_k))$;

end if

$res_i \leftarrow \sqrt{res_i^2 - c(gcnt, i)^2}$, $i \in \{lcnt+1, \ldots, cols_k\}$

end while

Figure 4.2: The Pipelined QR Algorithm with Local Pivoting: Main Loop

4.2 An Incremental Estimator for the Smallest Singular Value of a Triangular Matrix

To guard against choosing "bad" pivot columns, we have to monitor the smallest singular value $\sigma_{min}(R_i)$ where R_i is the leading $i \times i$ upper triangular matrix generated after applying i Householder transformations to A. The exact computation of $\sigma_{min}(R_i)$ (by inverse iteration for example) is too expensive, especially since a good order-of-magnitude estimate suffices for our purposes.

A common idea underlying condition estimators [23,24,63] is to exploit the implication

$$Rx = d \implies \frac{1}{\sigma_{min}(R)} = \|R^{-1}\|_2 \geq \frac{\|R^{-1}d\|_2}{\|d\|_2} = \frac{\|x\|_2}{\|d\|_2}$$

by generating a large norm solution x to a moderately sized right hand side d and then to use

$$\hat{\sigma}_{min}(R) := \frac{\|d\|_2}{\|x\|_2}$$

as an estimate for $\sigma_{min}(R)$. The hope is that x will be an approximate singular vector corresponding to the smallest singular value and that as a consequence $\hat{\sigma}_{min}(R)$ will not be too much of an over-estimate of $\sigma_{min}(R)$. Our choice of algorithms for an condition estimator is severely restricted by the fact that it is not feasible to access the previously generated R when we want to decide on the suitability of a new pivot column. To be more precise, given a good estimate $\hat{\sigma}_{min}(R)$ defined by a large norm solution x to $Rx = d$ and a new column $\begin{pmatrix} v \\ \gamma \end{pmatrix}$ of R, we want to obtain a large norm solution y to

$$R'y = \begin{pmatrix} R & v \\ 0 & \gamma \end{pmatrix} y = d'$$

without accessing R again. None of the condition estimators surveyed by Higham [64] has that property, but the two-norm condition estimator suggested by Cline, Conn and

Van Loan [23,103] can be modified to conform to those restrictions. The idea then is the following:

Given x such that $R^T x = d$ with $\|d\|_2 = 1$, find $s := \sin\varphi$ and $c := \cos\varphi$ such that $\|y\|_2$ is maximized where $y = \begin{pmatrix} z \\ \delta \end{pmatrix}$ solves

$$\begin{pmatrix} R^T & 0 \\ v^T & \gamma \end{pmatrix} y = \begin{pmatrix} sd \\ c \end{pmatrix}. \tag{4.1}$$

We here exploit the fact that R' and R'^T have identical singular values. An easy calculation shows that maximizing $\|y\|_2$ is equivalent to maximizing

$$\Phi(\varphi) = s^2\beta - 2\alpha sc \tag{4.2}$$

where

$$\alpha = v^T x \text{ and } \beta = \gamma^2 x^T x + \alpha^2 - 1. \tag{4.3}$$

Taking derivatives in (4.2) and setting $\eta = \beta/(2\alpha)$ we find two possible solutions:

$$s_{1,2} = \frac{1}{\sqrt{1+\mu_{1,2}}}$$

where

$$\mu_{1,2} = \eta \pm \sqrt{1+\eta^2}.$$

The corresponding cosine values are

$$c_{1,2} = s_{1,2}\,\mu_{1,2}.$$

For the special case $\alpha = 0$ we obtain the possibilities

$$c_1 = 1\,, s_1 = 0 \text{ and } c_2 = 0\,, s_2 = 1\,.$$

To choose between the two possibilites, we compute $\Phi(s_1)$ and $\Phi(s_2)$ and choose the sine/cosine pair that results in the greater value for Φ. The new approximate singular vector y as defined by (4.1) is then given by setting

$$z := sx \quad \text{and} \quad \delta := \frac{c - s\alpha}{\gamma} .$$

The resulting estimate for the smallest singular value $\sigma_{min}(R')$ of R' is

$$\hat{\sigma}_{min}(R') = \frac{1}{\|y\|_2} .$$

From this description it is clear that this condition estimator satisfies our algorithmic constraints. Given a current R_i we only need to save the current solution x and its norm $\|x\|_2$ to arrive at an estimate for $\sigma_{min}(R_{i+1})$. Furthermore the calculation is inexpensive. For a $k \times k$ matrix R_i we only need $2k$ flops to arrive at an estimate for $\sigma_{min}(R_{i+1})$. So altogether it costs only n^2 flops run this condition estimator alongside the generation of an $n \times n$ triangular matrix. The PRO-MATLAB routine implementing one step of the incremental condition estimator is given in Appendix A.

To assess the accuracy of our condition estimator, we performed the suite of tests suggested by Higham [64]. Three different types of test matrices are employed. In each test, upper triangular matrices R were generated by computing the QR factorization of various $n \times n$ matrices A for $n = 10, 25, 50$ both with and without column pivoting.

Test1 (see Table 4.1): The elements of A were chosen as random numbers from the uniform distribution on $[-1, 1]$. Fifty matrices were generated for each n. As observed by Higham, this type of matrix usually is well-conditioned. Over the whole test the minimum, maximum and average values of the two-norm condition number $\kappa_2(A) = \sigma_1/\sigma_n$ were $21, 1.4 \cdot 10^4$ and $2.0 \cdot 10^3$ respectively.

Test2 (see Tables 4.2 and 4.3) and *Test 3*: In these tests we used random matrices A with preassigned singular value distributions $\{\sigma_i\}$. Random orthogonal matrices U and V were generated using the method of Stewart [99] and then A was formed as in

Table 4.1: max/avg values of $\hat{\sigma}_{min}(R)/\sigma_{min}(R)$ for Test 1

pivoting	$n = 10$	25	50
no	2.2/1.4	6.6/2.4	7.0/3.1
yes	2.3/1.5	3.2/2.0	3.9/2.6

Table 4.2: max/avg values of $\hat{\sigma}_{min}(R)/\sigma_{min}(R)$ for Test 2 without Pivoting

κ_2	$n = 10$	25	50
10	1.8/1.3	1.7/1.4	1.6/1.4
10^3	3.0/1.9	2.5/2.0	3.2/2.2
10^6	8.1/1.9	6.3/2.6	4.2/2.8
10^9	6.1/2.2	5.9/3.0	5.2/3.2

(2.10). For each value of n and each singular value distribution, fifty matrices were generated by choosing different matrices U and V. For test 2 we chose the exponential distribution

$$\sigma_i = \alpha^i, \ 1 \leq i \leq n$$

where $\alpha < 1$ is determined by $\kappa_2(A)$. For test 3, we chose the sharp-break distribution

$$1 = \sigma_1 = \cdots = \sigma_{n-1} > \sigma_n = \frac{1}{\kappa_2(A)}.$$

The figures given in Tables 4.1–4.2 are the ratios

$$\hat{\sigma}_{min}(R)/\sigma_{min}(R) \geq 1$$

The first number in each pair is the maximum ratio over the fifty matrices and the second is the average ratio. All results were rounded to two significant digits. For Test 3 we observed a ratio of 1.0 (i.e. the estimate had at least two correct figures)

Table 4.3: max/avg values of $\hat{\sigma}_{min}(R)/\sigma_{min}(R)$ for Test 2 with Pivoting

κ_2	$n = 10$	25	50
10	1.6/1.3	1.6/1.4	1.7/1.4
10^3	2.2/1.5	2.3/1.8	2.5/2.0
10^6	2.8/1.5	3.4/2.1	3.4/2.5
10^9	2.4/1.6	3.3/2.2	4.3/2.7

in all cases. These results show that our conditon estimator is reliable in producing good estimates. We overestimate $\sigma_{min}(R)$ only by a small factor and the results vary only little with condition number, matrix size and singular value distribution. Pivoting increases the accuracy of the condition estimator and we can confidently expect similar accuracy when applying this estimator to matrices R generated by the local pivoting strategy.

We also mention that it would be preferable to have a *lower* bound for $\sigma_{min}(R)$ instead of the upper bound that the incremental condition estimator is computing. To that end we experimented with the lower bounds derived from comparison matrices [2, 62,68] which can also be updated in an incremental fashion. We found however that these bounds were in most cases underestimating the smallest singular value by several orders of magnitude (this is consistent with Higham's [64] results) and as a result were not suitable for controlling the local pivoting strategy.

4.3 The QR Algorithm with Controlled Local Pivoting

With the condition estimator we now have the tool to insure the reliability of the local pivoting strategy. Using the same notation as in Figures 4.1 and 4.2 processor k now can check whether $c(:,j)$ is a reasonable choice for the next pivot column before computing $\hat{u}$. 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 a new approximate singular vector y for R_{gcnt+1}.

With

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

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

$$\hat{\sigma}_{min}(R_{gcnt+1}) = \frac{1}{\eta_1 \|y\|_2} \tag{4.4}$$

as an estimate for the smallest singular value of R_{gcnt+1} and

$$\hat{\kappa}\,(R_{gcnt+1}) = \eta_2\, \omega \|y\|_2 \tag{4.5}$$

as an estimate for the true condition number of R_{gcnt+1}. The scaling factors η reflect the trust we have in the accuracy of our estimates. A large η will result in too pessimistic

estimates, a small η might lead to underestimation. Based on our numerical results we recommend $\eta_1 = 3$. Since

$$\omega \leq \|A\|_2 \leq \|A\|_F \leq \sqrt{n}\omega$$

Higham [65] suggests

$$\eta_2 = \sqrt[4]{n}.$$

The choice of η_2 reflects the fact that in general the norm of the largest column is a good estimator for the largest singular value of a matrix.

Comparing the estimates (4.4) or (4.5) against a chosen threshold we will then accept or reject a candidate pivot column. The exact threshold depends heavily on the application, in particular the accuracy of the initial data. If the data is accurate to machine precision ϵ, a candidate pivot will in general be rejected if

$$\hat{\sigma}_{min}(R_{gcnt+1}) = O(1/\epsilon).$$

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 $\hat{u}$, send $(\hat{u}, y, \|y\|_2)$ to its right neighbour and then proceed as in Figure 4.2. It should be noted that 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 neighbour), while $\hat{u}$ will eventually be known to all processors. So the propagation of the condition estimator results 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

$$AP = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R_{11} & R_{12} \\ 0 & \hat{A} \end{pmatrix} \tag{4.6}$$

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 further 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.

To assess the numerical behavior of the proposed local pivoting scheme, we simulated the parallel algorithm using PRO-MATLAB and compared it with the traditional QR factorization algorithm with global column pivoting. Various 100×100 matrices were generated and the local pivoting strategy simulated on 8 and 32 processors. The MATLAB code is listed in Appendix B.

For tests 1 to 3 we generated 50 random matrices for each singular value distribution $\{\sigma_i\}$. 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: $\sigma_1 = 1; \sigma_i = \alpha^{i-1} (i = 2, \ldots, 99)$; $\alpha = (10^{-9})^{\frac{1}{99}}$.

Setting the rejection threshold for the smallest singular value to 10^{-7} and discounting the estimate for the smallest singular value (4.4) by a factor of $\eta_1 = 3$ we reject a candidate pivot column in the parallel algorithm if

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

Table 4.4: 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

For the traditional QR factorization algorithm we use the last diagonal entry of R_{gcnt+1} as estimate and reject a candidate pivot column if

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

Table 4.4 shows the condition numbers of the upper triangular matrices R generated by controlled local pivoting and by traditional column pivoting on those matrices. Letting σ_{cutoff} be the smallest singular value greater than 10^{-7} then the optimal value we can 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 about 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 will deteriorate if the matrix is highly rank-deficient and the number of columns per processor is small. This is not very surprising in that the pivoting choices of each processor are very limited. Due to the high dimension of the numerical null 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 4.4 are 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, we obtain for the other 49 runs the condition number distribution shown in Figure 4.3. We see that in the vast majority of cases the condition numbers of R were less than 50. It should be noted that in the case where we only have 3 columns per processor the computational speed of the pipelined algorithm would in all likelihood not be good anyhow due to the already mentioned dominance of the startup and final phases of the algorithm.

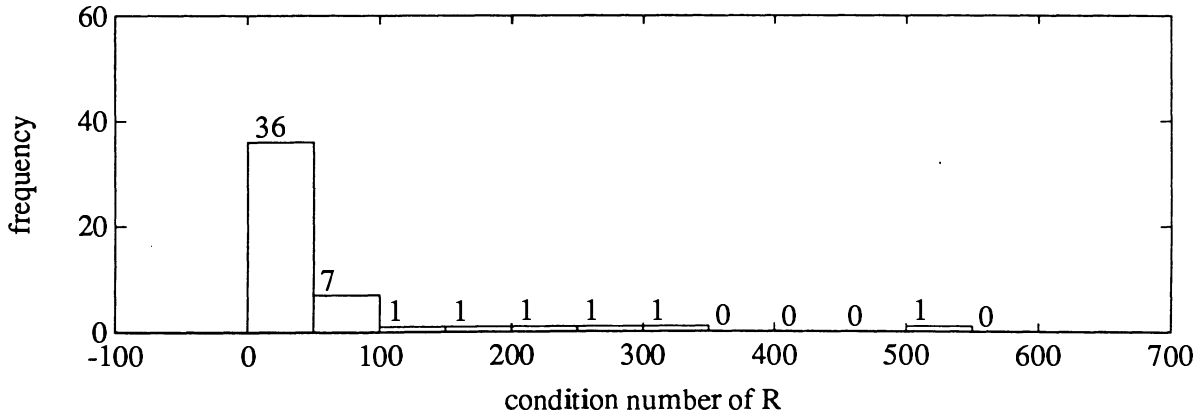

Figure 4.3: Condition Number Distribution of R for the *break 9* Distribution with $p = 32$

For the sharp break distributions there is a well-defined gap between the singular values before and after the acceptance threshold and both local and global column pivoting identify the numerical nullspace correctly in all cases. As already pointed out earlier, the determination of numerical rank becomes problematic if there is no well-defined gap between singular values that are considered "large" and "small". The

Table 4.5: 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

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:

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

The column pivoting strategy reflects this difficulty in accepting less than 77 columns and the observed results are displayed in Table 4.5. 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 very reliable in that it leans towards a small underestimate of the dimension of the numerical range space of A.

Our last example shows the advantage resulting from integrating our incremental condition estimator into the global pivoting scheme. We already mentioned earlier that the matrix A_n defined by (2.9) is a well-known example where the QR factorization with column pivoting fails since even in floating point arithmetic the matrix is its own QR factorization 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. However the condition estimator integrated into the parallel scheme detects the ill-conditioning of the leading principal submatrices A_k — it never overestimates the smallest singular value by a factor of more than 1.5. So while the column pivoting scheme fails the incremental condition estimator insures that failure will not go unno-

ticed. Given its negligible extra cost this suggest the usefulness of incorporating the incremental condition estimator into the global traditional column pivoting scheme.

The matrix A_n is also an example where the local pivoting scheme can perform better than the global one. 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 simulated on 8 processors on the other hand correctly identifies the numerical nullspace of $\tilde{A}_{50}$. While this is an exceptional occurence due to the special structure of A_n, it is nonetheless surprising since intuitively one would expect the global pivoting strategy to always perform better than the local one.

Chapter 5

Conclusions

In this thesis we presented the techniques of adaptive blocking and incremental condition estimation and applied them to compute the Householder QR factorization with and without column pivoting efficiently on a distributed architecture. A pipelined algorithm on a ring of processor determined the overall control structure of our algorithms to achieve a reasonable degree of portability.

The adaptive width blocking strategy was introduced into the pipelined Householder QR algorithm to facilitate better floating point utilization without incurring a substantial load imbalance. We motivated why a block QR factorization algorithm using blocks of fixed width will be a suboptimal choice in that the increased floating point speed will be counteracted by processors being idle at the beginning and final stages of the algorithm. We reconciled the trade-off between individual speed and global load balance by introducing blocks of variable width. Considering a window of two processors in the pipeline we developed a recurrence relation that determined the block widths that prevented processors from being idle. It is worth pointing out that the same reasoning would apply if we used a broadcast operation instead of the forwarding scheme we employed to propagate Householder vectors or compact WY

factors among processors.

Viewed in a larger context, we addressed the question how to distribute the workload in a situation where problem data had been assigned to processors in a static fashion. This severely limits the tasks that a given processor can execute and so makes it harder to prevent processors from being idle. This is why on shared memory machines dynamic scheduling techniques are so popular. Here a processor can work on totally different parts of a matrix at different stages of the algorithm. Processor idle time is prevented by switching a free processor at run time to any task that is ready to execute. We do not have this freedom when the problem data have been assigned to processors in a static fashion. The adaptive blocking strategy we presented tries to prevent processor idle time by using timing information about the program kernels to quantify the critical path of the algorithm. Obviously the pipelined algorithm we described can be simulated on a shared memory machine and it will be interesting to see how it compares with a dynamically scheduled version of the QR algorithm.

A different adaptive blocking strategy could be useful for a vector uniprocessor. To illustrate assume that we want to compute the QR factorization of an $m \times n$ matrix. To reduce the first 20 columns we could either employ two block transformations of width 10 or one of width 20. Using the notation of section 3.4 the first alternative requires time $t^1 \equiv t^1_{reduce} + t^1_{apply}$ where

$$t^1 reduce = t_{gencwy}(m, 10) + t_{gencwy}(m - 10, 10)$$

and

$$t^1_{apply} = t_{appcwy}(m, n - 10, 10) + t_{appcwy}(m - 10, n - 20, 10).$$

The second alternative requires $t^2 \equiv t^2_{reduce} + t^2_{apply}$ where

$$t^2_{reduce} = t_{gencwy}(m, 20)$$

and

$$t^2_{apply} = t_{appcwy}(m, n - 20, 20).$$

We know intuitively from the behaviour of *appcwy* and *gencwy* that $t^2_{reduce} > t^1_{reduce}$ and $t^2_{apply} < t^1_{apply}$. The relation between t^1 and t^2 on the other hand cannot be judged a priori without actually computing t^1 and t^2 since we do not know whether the additional work we invested in computing a wider WY factor was offset by the speed of the update. Then again it might have been more advantageous to divide the problem into three block columns of width 8, 7 and 5. So the question arises whether it is worthwhile and feasible to determine an "optimal" blocking strategy a priori for a given problem. It seems that dynamic programming [1, p.67] can be used advantageously here.

In the discussion so far relatively accurate timing models for the program kernels were taken for granted. On the Intel iPSC a rather ad-hoc least squares fit of the data was sufficient since the execution rate of the kernels did not change that much. For true vector processors the changes in execution speed will be much more pronounced. Research is needed to determine which timing models are best suited to capture this behaviour. In this context we will also explore whether nonlinear least squares techniques [30] and orthogonal distance regression methods [10,52] are more suited than straightforward least squares techniques to match the experimental data with the chosen model. It is important to keep in mind that the generation of timing models will have to be packaged in an installation routine that is robust in the sense that it provides accurate timing models across several systems. Thereafter adaptive blocking strategies can use this information to insure that a given problem executes efficiently on the machine at hand. This provides for a great degree of "performance portability" between machines which is important for libraries that are developed to run on several machines.

The second part of the thesis presented an incremental condition estimator that allowed us to update the estimate for the smallest singular value of the upper triangular matrix R as new columns were added to R. The update required only $O(n)$ flops and the

saving of $O(n)$ words between successive steps. Despite its small computational cost, experiments with a variety of matrices demonstrated the reliability of the condition estimation algorithm.

The incremental condition estimator makes it possible to restrict the number of pivoting choices in matrix algorithms without compromising the reliability of the overall factorization. We used this freedom to arrive at a local pivoting strategy for the QR factorization on a distributed memory machine. In the same fashion a local pivoting strategy for Gaussian elimination or Cholesky factorization could be devised. Given that pairwise pivoting [98] has essentially the same numerical properties as Gaussian elimination with partial pivoting our local pivoting strategy should behave accordingly.

On a uniprocessor, limiting the choice of pivot columns can be used to arrive at a block version of the QR factorization algorithm with column pivoting. A rough sketch of the right-looking [40] version of the resulting algorithm for an $m \times n$ matrix A is given in Figure 5.1. Obviously *block_size*, *window_size* and the acceptance thresholds

for $i = 1$ **to** n **step** *block_size* **do**

apply QR factorization with column pivoting guarded by incremental
condition estimator to determine *block_size* independent columns
in $a(i{:}m\,, i{:}i + \mathit{window\_size} - 1)$.
Accumulate T as defined by the *block_size* HH vectors Y
just determined.
$a(i{:}m\,, i + \mathit{window\_size}{:}m) \leftarrow \mathit{appcwy}\ (Y, T, a(i{:}m\,, i + \mathit{window\_size}{:}m)))$

end for

Figure 5.1: The QR Factorization Algorithm with Restricted Column Pivoting

for a candidate column need not be fixed and we will explore these issues.

We are also planning to investigate how incremental condition estimation can be used for sparse matrices. To illustrate the issues to be addressed, consider the matrix

$$A = \begin{pmatrix} B & 0 & D_B \\ 0 & C & D_C \\ 0 & 0 & D \end{pmatrix} \tag{5.1}$$

where B is $n_B \times n_B$, C is $n_C \times n_C$ and all matrices are dense. A matrix of this form arises for example from one level of nested dissection [50]. Assume that we compute the QR factorization of A and that we run the incremental condition estimator alongside the factorization. During the first n_B steps the incremental condition estimator computes an approximate singular vector x_B of B. Upon starting the decomposition of C, we will have $v = 0$ and $\gamma = c(1,1)$ in (4.1). So α in (4.3) will be zero and we will choose either $\begin{pmatrix} x_B \\ 0 \end{pmatrix}$ or $\begin{pmatrix} 0 \\ 1/\gamma \end{pmatrix}$ as an approximate singular vector for $\begin{pmatrix} B & 0 \\ 0 & c(1,1) \end{pmatrix}$. Notice also that α in (4.3) will be zero throughout the decomposition of C as long as

$$1/\|x_B\|_2 < |\ c(i,i)\ |$$

The solution here is to ignore B during the decomposition of C and to compute an approximate singular vector x_C for C. Then $1/max(\|x_B\|_2, \|x_C\|_2)$ will be a good estimate for $\sigma_{min}(\begin{pmatrix} B & 0 \\ 0 & C \end{pmatrix})$.

The next question then is how to merge x_B and x_C upon starting the decomposition of D. If we choose $\begin{pmatrix} x_B \\ 0 \end{pmatrix}$ as an approximate singular vector for $\begin{pmatrix} B & 0 \\ 0 & C \end{pmatrix}$ we are ignoring D_C's contribution in our condition estimator, if we choose $\begin{pmatrix} 0 \\ x_C \end{pmatrix}$ then we are ignoring D_B. Intuitively it looks as if $\begin{pmatrix} x_B \\ x_C \end{pmatrix}$ might be a good choice but this issue has to be explored.

If these concerns can be addressed in a satisfactory fashion we can employ the incremental condition estimator to compute a sparse QR factorization. Corresponding to any matrix A there is an elimination tree [71,72,75]. The nodes in the graph correspond to cliques in the filled graph of A and reordering columns in a clique will not change the sparsity structure of the factorization. Any postorder traversal [1] of the elimination tree then defines a perfect elimination ordering of A, i.e. we will not generate any additional fill. Together with John Lewis from Boeing Computing Services we are planning to investigate how this structure can be used efficiently in the context of a pivoting scheme that limits the choice of pivot columns to the cliques corresponding to the nodes in the elimination graph.

Another issue in the general sparse setting is that we must avoid rescaling entries of the approximate singular vector x unless they are really needed. For computing $z \leftarrow sx$ at every step will result in an $O(n^2)$ algorithm which may dominate the sparse factorization time. The key here is to hold off rescaling entries of x until they are truly needed. To illustrate, assume that we have computed x and that for the next k steps of the factorization the first l entries of x are not needed since the first l entries of the current v will always be zero. Then we can accumulate the sines $s_1, \ldots, s_k$ and apply $\prod_{i=1}^{k} s_i$ to $x_1, \ldots, x_l$ in one operation after the k steps not involving $x_1, \ldots, x_l$ have been completed. We will explore how this idea can be made precise using the graph-theoretic techniques for sparse matrices.

An application we are also considering is the application of incremental condition estimation techniques to the solution of sparse systems. Threshold pivoting [25, p. 13] is often used in the solution of sparse equation systems to maintain stability while trying to limit fill-in. Using our incremental condition estimator we can substitute threshold pivoting by a pivoting strategy that is solely geared towards minimizing fill-in but whose stability is monitored by the incremental condition estimator. It is likely that we have to abandon the sparsity-centered approach only in the later stages of the

decomposition. At this point the matrix still to be processed will be pretty much filled in and we incur no penalty when switching to the traditional pivoting schemes.

Lastly, we believe incremental condition estimation to be useful in general whenever a triangular matrix is generated a row or column at a time. By running the condition estimator hand in hand with the factorization we can cheaply monitor the condition number of the triangular matrix and take *immediate* action should numerical trouble arise. In contrast running an condition estimator after the factorization has been completed will give no indication as to where the numerical problems arose.

Appendix A

Matlab Code for the Incremental Condition Estimator

```
function [new_x,new_xnorm] = smin_ccvl_j(x,xnorm,aj,j)
%
%      Given that x is an approximate null vector of norm
%      "xnorm" of the lower triangular (j-1)x(j-1) matrix L,
%      smin_ccvl_j computes an approximate null vector new_x of
%      norm "new_xnorm" for the matrix       [  L  ]
%                                      L2 := [ aj' ]
%      in the sense that new_x is a large norm solution for
%           L2*x = b, where twonorm(b) = 1.

OVERFLOW = 1e60;
UNDERFLOW = 1e60;

%      Initialization (j == 1)
%      =======================

if ( j == 1)
       if ( abs(aj(1)) > UNDERFLOW )
            new_x(1) = 1/aj(1);
            new_xnorm = abs(new_x(1));
       else
            new_x(1) = 0;
            new_xnorm = 0;
       end
       return
```

```
end
%
%       Usual case (j > 1)
%       ==================

%    scale the objective function by xnorm if xnorm > 1
%    to avoid overflows

utx = x'*aj(1:j-1);

if (xnorm > 1)
        beta = aj(j)*aj(j)*xnorm+(utx/xnorm)*utx-ONE/xnorm;
        alpha = utx/xnorm;
else
        % check whether matrix already recognized as rank-deficient
        if (xnorm == ZERO)
             new_xnorm = ZERO;
             new_x(1:j) = zeros(j,1);
        end
        beta = aj(j)*aj(j)*(xnorm^2)+(utx^2)-ONE;
        alpha = utx;
end

%            alpha > 0 (usual case)
%            **********************

if (abs(alpha) > ZERO)

        rmax = max(TWO*abs(alpha),abs(beta));
        root = rmax*sqrt((TWO*alpha/rmax)^2+(beta/rmax)^2);

        signalpha = sign(alpha);
        mu1 = signalpha*beta+root;
        mu2 = signalpha*beta-root;

        rmax = max(TWO*abs(alpha),abs(mu1));
        denom1 = rmax*sqrt((TWO*alpha/rmax)^2+(mu1/rmax)^2);

        rmax = max(TWO*abs(alpha),abs(mu2));
        denom2 = rmax*sqrt((TWO*alpha/rmax)^2+(mu2/rmax)^2);
```

```
        s1 = TWO*abs(alpha)/denom1;
        c1 = mu1/denom1;
        s2 = TWO*abs(alpha)/denom2;
        c2 = mu2/denom2;
        phi1 = s1*(s1*beta - TWO*alpha*c1);
        phi2 = s2*(s2*beta - TWO*alpha*c2);
        if (phi2 > phi1)
               c1 = c2;
               s1 = s2;
               phi1 = phi2;
        end
else
         phi1 = ZERO;
end

%           small or zero alpha
%           ********************

if (alpha+ONE == ONE)

   if ( abs(aj(j)) > ONE/xnorm)
          s2 = ONE;
          c2 = ZERO;
          phi2 = (aj(j)*xnorm)^2;
   else
          s2 = ZERO;
          c2 = ONE;
          phi2 = ONE;
   end
   if (xnorm > 1), phi2 = phi2/xnorm; end

   if (phi2 > phi1)
          c1 = c2;
          s1 = s2;
   end
end

%           computation of new_x and new_xnorm with
%           special care not to produce overflows.
```

```
%           ***************************************

%     update x and the norm and the right hand side
%     be careful to avoid overflow when dividing by aj(j)

t1 = xnorm*s1;
t2 = c1-utx*s1;

if (abs(aj(j)) < ONE)

%      check for overflow when dividing by aj(j)

       if ((aj(j) == ZERO) | ..
            (abs(t2) > OVERFLOW*abs(aj(j))))

%           xnorm would be greater OVERFLOW

            new_xnorm = 0;
            new_x(1:j) = zeros(j,1);
            return
       end
end

t2 = t2/aj(j);
rmax = max(abs(t1),abs(t2));
root = sqrt((t1/rmax)^2+(t2/rmax)^2);

if (rmax > ONE)

%      check whether new xnorm would be greater OVERFLOW

       if (root > OVERFLOW/rmax)

%           xnorm would be greater than OVERFLOW

            new_xnorm = 0;
            new_x(1:j) = zeros(j,1);
               return
        end
end
```

```
new_x = [ s1*x(1:j-1) ; t2 ];  % to generate a column vector
new_xnorm = rmax*root;

%  last line of function smin_ccvl_j
```

Appendix B

Matlab Code Simulating the Controlled Local Pivoting Algorithm

```
function [A,qraux,jpvt,selected,sminRhat] = ..
  pipelined_qr(A,no_processors,threshold,no_local_columns,offset)
%
%  simulates the pipelined QR algoritm with controlled pivoting.
%

%  Householder vector format is such that the first entry of
%  the Householder vector is scaled to 1.
%
%  =========
%  on entry:
%  =========
%  A        matrix on which QR factorization is to be simulated
%           it is assumed that A has been redistributed according
%           to the column wrap scheme.
%  no_processors number of processors to be simulated
%  threshold     smallest singular value we are willing to accept
%  no_local_columns(1:no_processors)
%           number of columns in a processor
%  offset(1:no_processors)
%           offset that the columns assigned to this processor
%           have with regard to the start of the global matrix.
%  ========
```

```
%  on exit:
%  ========
%  A             The upper triangle of A including the diagonal
%           has been overwritten with the triangular factor R,
%           The lower triangle of A has been overwritten by
%           the Householder vectors.
%  qraux(1:no_columns(A))
%           if global columns i has been selected as a pivot
%           column, then the Householder update determined by
%           column i is
%                (I - qraux(i)*hh*hh')
%           where
%                hh = [1;A(selected(i):no_rows(A),i];
%           if column i was never selected as a pivot column,
%           then qraux(i) is the residual that column i has with
%           respect to the space spanned by the colums selcted.
%  jpvt(1:no_columns(A))
%           jpvt(i) is the number of the global column that
%           is occupying global position i at the end of the
%           computation.
%  selected if column i was selected as the jth Householder
%           column, then selected(i) = j. If column i was
%           considered as a Householder column but rejected
%           because of the threshold criterion,
%                selected(i) = - shat
%           where shat is the estimated smallest singular value
%           of the matrix one would have obtained by selecting
%           column i as next pivot column.
%           If column i was never even considered a pivot
%           column candidate, selected(i) = -Inf.
%  sminRhat est. smallest sing. value for the upper triangular
%           matrix R determined by the pivot columns in the
%           QR factorization.
%
%  ==================
%  subroutines called:
%  ==================
%  mymod     special modulo arithmetic since processors are
%           counted from one on.
%  smin_ccvl_j  incremental condition estimator
```

```
%
%  ===============
%  local variables:
%  ===============
%  no_rows, no_columns
%           number of rows and columns of A.
%  hh_step  The "serial number" of the Householder vector
%           that we are currently trying to compute.
%  processor     the number of the processor whose turn it is to
%           compute the next Householder vector
%  qraux(1:no_columns)
%           qraux(i) is the residual that column i has with
%           respect to the span of columns already selected.
%  exact(1:no_columns)
%           exact residuals of a previous pivoting step
%           used in determining when to recompute qraux.
%  jpvt(1:no_columns)
%           jpvt(i) is initialized to the
%           index of the column that is in the globally ith
%           position after the columns have been distributed
%           according to the wrap scheme. For example
%           jpvt(1) = 1, jpvt(2) = no_processors + 1, etc.
%           at the beginning,
%           During the computation jpvt(i) is updated to contain
%           the index of the column currently in the ith position.
%  status(1:no_processors)
%           ALIVE     if a processor still has potential pivot
%                column candidates
%           DEAD if a processor has exhausted it supply of
%                reasonable pivot columns and is now only
%                applying updates.
%           FINISHED  if a processor has finished its last
%                column.
%  no_dead, no_finished
%           number of processors with status DEAD or FINISHED.
%  eligible(1:no_processor)
%           which column in a processor is the first one
%           eligible to be chosen as a pivot column. So the
%           range of possible pivot columns is locally
%           eligible(p):no_local_columns(p), globally it is
```

```
%           offset(p)+eligible(p):offset(p)+no_local_columns(p)
%  low, high     abbreviations for
%           eligible(p)+offset(p) and
%           offset(p) + no_local_columns(p) respectively
%  pvt_index     global index of candidate pivot column.
%  success_flag to indicate whether candidate pivot column was
%           acceptable.
%  xnorm_smin    two-norm of x_smin


%      ******************
%      * Initialisation *
%      ******************

ALIVE = 10; FINISHED = 20; DEAD = 30;
[no_rows,no_columns] = size(A);

       % status, eligible
       % ================

for p = 1:no_processors
   status(p) = ALIVE;
   eligible(p) = 1;
end % for p

       % jpvt, qraux and exact, selected
       % ===============================

for p = 1:no_processors
   for i = 1:no_local_columns(p)
      temp = offset(p)+i;
      jpvt(temp) = (p-1)+(i-1)*no_processors+1;
      qraux(temp) = norm(A(:,temp));
      exact(temp) = qraux(temp);
      selected(temp) = -Inf;
   end
end

       % no_dead, no_finished, hh_step, processor
       % x_smin, xnorm_smin
```

```
        % ========================================

no_dead = 0; no_finished = 0;
processor = 1; hh_step = 1;
x_smin = []; xnorm_smin = 0;

%
%       ***************************************************
%       * MAIN LOOP: repeat until no more acceptable pivot *
%       * columns left in any processor.                  *
%       ***************************************************
%
while (no_dead + no_finished < no_processors)

  if (status(processor) == ALIVE)

            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
            % Try to find an acceptable pivot columns in    %
            % this processor.                    %
            %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      low = offset(processor) + eligible(processor);
      high = offset(processor) + no_local_columns(processor);

            % compute estimated smallest singular value
            % of matrix resulting from candidate pivot column
            % ==============================================

      [dummy,maxk] = max(qraux(low:high));
      pvt_index = low + maxk - 1;
      if (A(hh_step,pvt_index) == ZERO)
        gamma = -qraux(pvt_index);
      else
        gamma = -sign(A(hh_step,pvt_index))*qraux(pvt_index);
      end
      candidate_column = [A(1:hh_step-1,pvt_index);gamma];
      [new_x_smin,new_xnorm_smin] = smin_ccvl_j(x_smin,xnorm_smin,..
                              candidate_column,hh_step);

            % check whether acceptable
```

```
        % ======================

success_flag = FALSE;
if (new_xnorm_smin > ZERO)
    if (ONE/new_xnorm_smin > threshold)
       success_flag = TRUE;
       x_smin = new_x_smin;
       xnorm_smin = new_xnorm_smin;
    else
       selected(pvt_index) = -ONE/new_xnorm_smin;
    end
else
    selected(pvt_index) = ZERO;
end

if (success_flag == TRUE)

       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
       % candidate pivot column was acceptable.  %
       % Exchange columns and compute HH vector. %
       %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

       % exchange first eligible column and pivot column
       % ==============================================

   if (pvt_index ~= low)
     t= A(:,low);A(:,low)=A(:,pvt_index);A(:,pvt_index)=t;
     t=jpvt(low);jpvt(low)=jpvt(pvt_index);jpvt(pvt_index)=t;
     qraux(pvt_index) = qraux(low);
     exact(pvt_index) = exact(low);
  end

       % compute HH vector unless it would have length one
       % =================================================

   if (hh_step < no_rows)

       % scale first entry of HH vector to one.
     pvt_norm = norm(A(hh_step:no_rows,low));
     if (A(hh_step,low) == ZERO)
```

```
   beta = pvt_norm;
else
   beta = sign(A(hh_step,low))*pvt_norm;
end
qraux(low) = (A(hh_step,low)+beta)/beta;
hh_vector = [ONE;..
     A(hh_step+1:no_rows,low)/(A(hh_step,low)+beta)];
selected(low) = hh_step;
A(hh_step,low) = -beta;
A(hh_step+1:no_rows,low) = hh_vector(2:1+no_rows-hh_step);

  % check whether this processor has processed
  % all its columns
  % =======================================

if (eligible(processor) == no_local_columns(processor))
      status(processor) = FINISHED;
      no_finished = no_finished + 1;
else
      eligible(processor) = eligible(processor) + 1;
end

  %%%%
```

```
                % update pivot information
                % ========================

            for k = low:high
              if (qraux(k) > 0)
                temp = ONE - (abs(A(hh_step,k))/qraux(k))^2;
                temp = max(temp,ZERO);
                temp2 = ONE + 0.05*temp*(qraux(k)/exact(k))^2;
                if (temp2 == ONE) % recompute residual
                   qraux(k) = norm(A(hh_step+1:no_rows,k));
                   exact(k) = qraux(k);
                else
                   qraux(k) = qraux(k)*sqrt(temp);
                end
              end
            end % for k

          end % if processor alive

        end % for p

        hh_step = hh_step + 1;

      else %  hh_step == no_rows

        qraux(low) = -TWO;    % for consistency
        selected(low) = hh_step;
        status(processor) = FINISHED;
        no_finished = no_finished + 1;

      end % if hh_step

    else % if success_flag false

      status(processor) = DEAD;
      no_dead = no_dead + 1;

    end % if success_flag

  end % if processor alive
```

```
  processor = mymod(processor+1,no_processors);

end % while main loop

if (xnorm_smin > ZERO)
   sminRhat = 1/xnorm_smin;
else
   sminRhat = ZERO;
end

% last line of function pipelined_qr
```

Bibliography

[1] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Ned Anderson and Ilkka Karasalo. On computing bounds for the least singular value of a triangular matrix. *BIT*, 15:1–4, 1975.

[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 et al., editor, *Proceedings of CONPAR 86*, pages 25–39. Springer Verlag, New York, 1986.

[4] Michael Berry and Ahmed Sameh. Multiprocessor Jacobi algorithms for dense symmetric eigenvalue and singular value decompositions. In *Proceedings International Conference on Parallel Processing*, pages 433–440, 1986.

[5] 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.*

[6] Christian H. Bischof. The two-sided block Jacobi method on a hypercube architecture. In Michael T. Heath, editor, *Hypercube Multiprocessors*, pages 612–618. SIAM Press, 1987.

[7] Christian H. Bischof and Charles F. Van Loan. *Computing the Singular Value Decomposition on a Ring of Array Processors*, volume 127 of *Mathematics Studies Series*, pages 51–66. North Holland, Amsterdam, 1986.

[8] 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.

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

[10] Paul Boggs, Richard Byrd, and Robert Schnabel. A stable and efficient algorithm for nonlinear orthogonal distance regression. *SIAM Journal on Scientific and Statistical Computing*, 8(6):1052–1078, 1987.

[11] R. P. Brent, F. T. Luk, and C. F. Van Loan. Computation of the singular value decomposition using mesh-connected processors. *Journal VLSI Computer Systems*, 1:242–270, 1985.

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

[13] R. M. Chamberlain and M. J. D. Powell. QR factorization for linear least squares problems on the hypercube. Technical Report CCS 86/10, Chr. Michelsen Institute, 1986.

[14] Tony F. Chan. An improved algorithm for computing the singular value decomposition. *ACM Transactions on Mathematical Software*, 8:72–83, 1982.

[15] Tony F. Chan. Deflated decomposition of solutions of nearly singular systems. *SIAM Journal on Numerical Analysis*, 21(4):121–134, 1984.

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

[17] Tony F. Chan and Per Christian Hansen. Computing truncated SVD least squares solutions by rank revealing QR factorizations. *SIAM Journal on Scientific and Statistical Computing*, 1988. to appear.

[18] J. P. Charlier and P. Van Dooren. On Kogbetliantz's SVD algorithm in the presence of clusters. *Linear Algebra and its Applications*, 95:135–160, 1987.

[19] J. P. Charlier, M. Vanbegin, and P. Van Dooren. On efficient implementations of Kogbetliantz's algorithm for computing the singular value decomposition. *Numerische Mathematik*, 52:279–300, 1988.

[20] Marina Chen. A design methodology for synthesizing parallel algorithms and architectures. Technical Report YALEU/DCS/TR-457, Yale University, Department of Computer Science, June 1986.

[21] Per Christian Hansen. Reducing the number of sweeps in Hestenes method. In E. F. Deprettere, editor, *Singular Value Decomposition and Signal Processing*. North-Holland, 1988. *to appear*.

[22] Eleanor Chu and Alan George. QR factorization of a dense matrix on a hypercube multiprocessor. Technical Report TM-10691, Oak Ridge National Laboratory, Engineering Physics and Mathematics Division, February 1988.

[23] A. K. Cline, A. R. Conn, and C. F. Van Loan. *Generalizing the LINPACK Condition Estimator*, volume 909 of *Lecture Notes in Mathematics*, pages 73–83. Springer Verlag, Berlin, 1982.

[24] A. K. Cline, C. B. Moler, G. W. Stewart, and J. H. Wilkinson. An estimate for the condition number of a matrix. *SIAM Journal on Numerical Analysis*, 16:368–375, 1979.

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

[26] Thomas F. Coleman and Guangye Li. Solving systems of nonlinear equations on a message-passing multiprocessor. Technical Report CS-87-887, Cornell University, Department of Computer Science, November 1987.

[27] Thomas F. Coleman and Paul Plassman. The solution of nonlinear least-squares problems on a message-passing multiprocessor. Technical Report CS-TR-88-923, Cornell University, Department of Computer Science, June 1988.

[28] P. P. M de Rijk. A one-sided Jacobi algorithm for computing the singular value decomposition on a vector processing computer. Technical Report 86-21, Dept. of Mathematics, University of Amsterdam, 1986.

[29] 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.

[30] John Dennis and Robert Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

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

[32] 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.

[33] Jack Dongarra and Eric Grosse. Distribution of mathematical software by electronic mail. *Communications of the ACM*, 30(5):403–407, 1987.

[34] 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.

[35] Jack Dongarra, Ahmed Sameh, and Danny Sorensen. Implementation of some concurrent algorithms for matrix factorization. *Parallel Computing*, 3(1):25–34, 1986.

[36] Jack Dongarra and Danny Sorensen. A portable environment for developing parallel programs. *Parallel Computing*, 5(1&2):175–186, 1987.

[37] 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.

[38] Jack J. Dongarra, Jeremy DuCroz, Ian Duff, and Sven Hammarling. A proposal for a set of level 3 basic linear algebra subprograms. Technical Report ANL–MCS–TM88, Argonne National Laboratory, Mathematics and Computer Sciences Division, April 1987.

[39] 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.

[40] 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.

[41] Jeremy Du Croz, 1987. Private Communication.

[42] T. H. Dunigan. Hypercube performance. In Michael T. Heath, editor, *Hypercube Multiprocessors*, pages 178–192. SIAM Press, 1987.

[43] P. J. Eberlein. On using the Jacobi method on a hypercube. In Michael T. Heath, editor, *Hypercube Multiprocessors*, pages 605–611. SIAM Press, 1987.

[44] Eldén, Lars. A parallel QR decomposition algorithm. Technical Report LITH-MAT-R-1988-02, Linköping University, Department of Mathematics, 1988.

[45] G. E. Forsythe and P. Henrici. The cyclic Jacobi method for computing the principal values of a complex matrix. *Transactions American Mathematical Society*, 94:1–23, 1960.

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

[47] 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.

[48] George A. Geist and Michael T. Heath. Parallel Cholesky factorization on a hypercube multiprocessor. Technical Report ORNL–6190, Oak Ridge National Laboratory, Engineering Physics and Mathematics Division, 1985.

[49] W. M. Gentleman and H. T. Kung. *Matrix Triangularization by Systolic Arrays*, volume 298 of *Proceedings of the SPIE*, pages 19–26. 1982.

[50] Alan George and Joseph Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.

[51] 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.

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

[53] Vjeran Hari and Kresemir Veselic. On Jacobi methods for singular value decompositions. *SIAM Journal on Scientific and Statistical Computing*, 8(5):741–754, 1987.

[54] 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.

[55] Michael T. Heath. Numerical methods for large sparse linear least squares problems. *SIAM Journal on Scientific and Statistical Computing*, 5:497–513, 1984.

[56] Michael T. Heath and Charles H. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. Technical Report ORNL/TM–10384, Oak Ridge National Laboratory, Engineering Physics and Mathematics Division, 1987.

[57] Michael T. Heath and Charles H. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. Unpublished Manuscript, 1988.

[58] Michael T. Heath and Danny C. Sorensen. A pipelined Givens method for computing the QR decomposition of a sparse matrix. *Linear Algebra and its Applications*, 77:180–203, 1984.

[59] Micheal Heath, editor. *Hypercube Multiprocessors 1987*. SIAM press, 1987.

[60] Don E. Heller and Ilse F. Ipsen. Systolic networks for orthogonal decompositions. *SIAM Journal on Scientific and Statistical Computing*, 4(2):261–269, 1983.

[61] M. Hestenes. Inversion of a matrix by biorthogonalization and related results. *SIAM Journal on Applied Mathematics*, 6:51–90, 1958.

[62] Nicholas J. Higham. Upper bounds for the condition number of a triangular matrix. Technical Report Numerical Analysis Report No. 86, University of Manchester, England, 1983.

[63] Nicholas J. Higham. Efficient algorithms for computing the condition number of a tridiagonal matrix. *SIAM Journal on Scientific and Statistical Computing*, 7:150–165, 1986.

[64] Nicholas J. Higham. A survey of condition number estimation for triangular matrices. *SIAM Review*, 29(4):575–598, 1987.

[65] Nicholas J. Higham, 1988. Private Communication.

[66] Ilse Ipsen, Youcef Saad, and Martin Schultz. Dense linear systems on a ring of processors. *Linear Algebra and its Applications*, 77:205–239, 1986.

[67] C. G. J. Jacobi. Über ein leichtes Verfahren die in der Theorie der Säcular-Störungen vorkommenden Gleichungen numerisch aufzulösen. *Crelle's Journal*, 30:51–94, 1846.

[68] Ilkka Karasalo. A criterion for truncation of the QR-decomposition algorithm for the singular linear least squares problem. *BIT*, 14:156–166, 1974.

[69] E. Kogbetliantz. Diagonalization of general complex matrices as a new method for solution of linear equations. In *Proc. Intern. Congr. Math.*, pages 356–357, Amsterdam, 1954.

[70] 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.

[71] C. E. Leiserson and J. G. Lewis. Orderings for parallel sparse symmetric factorization. Technical Report ETA-TR-85, Boeing Computer Services, Engineering and Scientific Services Division, March 1988.

[72] J. G. Lewis and B. W. Peyton. A fast implementaion of the Jess and Keyes algorithm. Technical Report ETA-TR-90, Boeing Computer Services, Engineering and Scientific Services Division, May 1988.

[73] Guangye Li and Thomas F. Coleman. A parallel triangular solver for a hypercube multiprocessor. Technical Report 86-787, Cornell University, Department of Computer Science, 1986.

[74] Guangye Li and Thomas F. Coleman. A new method for solving triangular systems on distributed memory message-passing multiprocessors. Technical Report 87-812, Cornell University, Department of Computer Science, 1987. *to appear in SIAM J. Scientific and Statistical Comp.*

[75] Joseph Liu. The role of elimination trees in sparse factorization. Technical Report CS-87-12, Dept. of Computer Science, York University, October 1987.

[76] Franklin T. Luk. Computing the singular value decomposition on the ILLIAC IV. *ACM Transactions on Mathematical Software*, 6:524-539, 1980.

[77] Franklin T. Luk. A rotation method for computing the QR decomposition. *SIAM Journal on Scientific and Statistical Computing*, 7(2):452-459, 1986.

[78] Franklin T. Luk. A triangular processor array for computing the singular value decomposition. *Linear Algebra and its Applications*, 77:259-274, 1986.

[79] Franklin T. Luk and Haesun Park. On parallel Jacobi orderings. Technical Report EE-CEG-86-5, Cornell University, Department of Electrical Engineering, 1986.

[80] Franklin T. Luk and Haesun Park. A proof of convergence for two parallel Jacobi methods. Technical Report EE-CEG-86-12, Cornell University, Department of Electrical Engineering, 1986.

[81] P. J. D. Mayes, 1988. Private Communication.

[82] J. J. Modi and M. R. B. Clarke. An alternative Givens ordering. *Numerische Mathematik*, 43:83-90, 1986.

[83] Cleve Moler. Matrix computation on distributed memory multiprocessors. In Michael T. Heath, editor, *Hypercube Multiprocessors 1986*. SIAM Press, 1986.

[84] C. C. Paige and P. Van Dooren. On the quadratic convergence of Kogbetliantz's algorithm for computing the singular value decomposition. *Linear Algebra and its Applications*, 77:301–313, 1986.

[85] Alex Pothen, Somesh Jha, and Udaya Vemapulati. Orthogonal factorization on a distributed memory multiprocessor. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*. SIAM Press, 1987.

[86] Alex Pothen and Padma Raghavan. Distributed orthogonal factorization: Givens and Householder algorithms. Technical Report CS–87–24, The Pennsylvania State University, 1987.

[87] Terrence W. Pratt. PISCES: An environment for parallel scientific computation. Technical Report 85–12, NASA Langley Research Center, Institute for Computer Applications in Science and Engineering, 1985.

[88] Youcef Saad and Martin H. Schultz. Topological properties of hypercubes. Technical Report YALEU/DCS/RR-389, Yale University, Department of Computer Science, 1985.

[89] Robert Schreiber. Solving eigenvalue and singular value problems on an undersized systolic array. *SIAM Journal on Scientific and Statistical Computing*, 7(2):441–451, 1986.

[90] Robert Schreiber. Block algorithms for parallel machines. Technical Report 87–5, Rensellaer Polytechnic Institute, Department of Computer Science, 1987.

[91] Robert Schreiber and Beresford Parlett. Block reflectors: Computation and applications. Technical report, Rensellaer Polytechnic Institute, Department of Computer Science, 1987.

[92] Robert Schreiber and Charles Van Loan. A storage efficient WY representation for products of Householder transformations. Technical Report CS–87–864, Cornell University, 1987.

[93] David Scott, Michael Heath, and Robert Ward. Parallel block Jacobi eigenvalue algorithms using systolic arrays. *Linear Algebra and its Applications*, 77:345–355, 1986.

[94] Charles Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, 1985.

[95] Gautam Shroff and Robert Schreiber. Convergence of block Jacobi methods. Technical Report 87–25, Rensellaer Polytechnic Institute, Department of Computer Science, 1987.

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

[97] Lawrence Snyder. Parallel programming and the Poker environment. *IEEE Computer*, 17(7):27–36, 1984.

[98] Danny C. Sorensen. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Transactions on Computers*, C-34(3), 1985.

[99] G. W. Stewart. The efficient generation of random orthogonal matrices with an application to condition estimators. *SIAM Journal on Numerical Analysis*, 17:403–409, 1980.

[100] G. W. Stewart. On the implicit deflation of nearly singular systems of linear equations. *SIAM Journal on Scientific and Statistical Computing*, 2:136–140, 1981.

[101] Charles F. Van Loan. The block Jacobi method for computing the singular value decomposition. Technical Report CS-85-680, Cornell University, Department of Computer Science, 1985.

[102] Charles F. Van Loan. A block QR factorization scheme for loosely coupled systems of array processors. Technical Report CS-86-797, Cornell University, Department of Computer Science, 1986.

[103] Charles F. Van Loan. On estimating the condition of eigenvalues and eigenvectors. *Linear Algebra and its Applications*, 88/89:715–732, 1987.

[104] Homer F. Walker. Implementation of the GMRES method using Householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 9(1):152–163, 1988.

[105] James H. Wilkinson. *The Algebraic Eigenvalue Problem.* Clarendon Press, 1965.