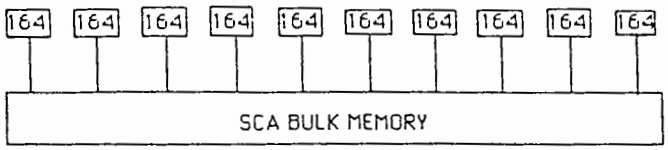
A BLOCK QR FACTORIZATION SCHEME FOR LOOSELY COUPLED SYSTEMS OF ARRAY PROCESSORS

CHARLES VAN LOAN†

1. Introduction. Computing the QR factorization of a matrix $A \in R^{m \times n}$ involves finding an orthogonal matrix $Q \in R^{m \times m}$ and an upper tringular matrix $R \in R^{m \times n}$ such that $A = QR$. This factorization has a prominent role to play in numerical linear algebra especially because of its bearing on the least square problem. A detailed description of the QR factorization and the various ways that it can be computed may be found in Golub and Van Loan (1983).

Parallel methods for computing the QR factorization have received considerable attention recently. For systolic arrays attention has focused on methods that rely on Givens rotations. See Gentleman and Kung (1981) or Heller and Ipsen (1983). Dongarra, Sameh, and Sorenson (1986) have implemented both parallel Givens and parallel Householder procedures on the Denelcor Hep.

In this paper we discuss a block version of the Gentlemen-Kung method that we have implemented on the IBM Kingston LCAP-1. This system consists of ten FPS-164 array processors (APs) that can communicate through several shared bulk memories. An overview of LCAP-1 is offered in Clementi and Logan (1985). The features of LCAP-1 that figure in the current work are depicted in the following diagram:



There are actually two levels of parallelism here because the APs are each capable of performing twenty parallel dot products. Indeed, the FPS-164/MAX's at Kingston each come equipped with two "MAX boards". The MAX Board enhancement enables each AP to perform matrix–matrix multiplication at a peak rate of 55 Mflops *if* the matrices involved are sufficiently large. Full exploitation of the FPS–164/ MAX requires having an algorithm that is rich in matrix multiplication. This is why we have chosen to develop a parallel *block* procedure. The blocking of the matrix A is largely a function of the 164/MAX architecture. For example, it turns out to be efficient to have block columns that are a multiple of twenty simply because the LCAP–1 APs can *each* perform twenty parallel dot products. Further details concerning the FPS–164/MAX architecture may be found in Charlesworth and Gustafson (1986).

The matrix A is stored in a 64 Mword bulk memory unit manufactured by Scientific Computing Associates (SCA). Thus, a dense problem of size 16K–by–4K could potentially be solved. The APs have approximately 600 Kwords of usable memory. This is enough to house, for example, a 1000–by–500 submatrix.

Data between the APs and the bulk memory flows at a rate of 44 Mbytes/sec. However, high latency associated with each transferred message demands that data be moved in fairly good–sized chunks in order to be efficient, e.g., 1000 words.

†Department of Computer Science, Cornell University, Ithaca, New York 14853

Additional nuances of the LCAP–1 system as they apply to our QR implementation are detailed later.

This paper is the first of several reports in which we explore the issues associated with parallel matrix computations on the LCAP–1. The parallel block QR factorization scheme that we encoded is derived in §2 and §3. Implementation details are covered in §4 and results in §5. Our current QR code can be improved in several ways as we often opted for the "easy way out" when confronted with an algorithmic dilemma. Despite this we feel that our LCAP–1 experience offers general perspective on large scale distributed matrix computations.

2. Parallel Givens QR. We say that $G \in R^{m\times m}$ is an adjacent Givens rotation in planes i–1 and i if G is the identity with the following 2–by–2 exception:

$$\begin{bmatrix} g_{i-1,i-1} & g_{i-1,i} \\ g_{i,i-1} & g_{ii} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \qquad 2 \le i \le m$$

Notice that G is orthogonal and that premultiplication by G affects just rows i–1 and i. If $x \in R^m$ then it is not hard to determine $(\cos(\theta), \sin(\theta))$ so that $y_i = 0$ if $y = Gx$. These and other Givens rotations issues are discussed in Golub and Van Loan (1983, pp. 43–47).

Adjacent rotations are important because they only combine adjacent rows or columns when applied to a matrix. Moreover, they can be used to compute the QR factorization of a matrix. Assuming $A \in R^{m\times n}$ $(m \ge n)$ we have:

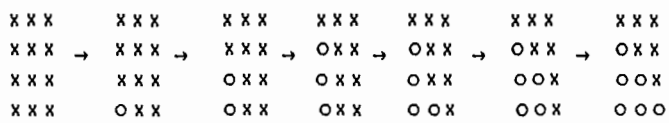
Algorithm 2.1.

```
For j = 1:n
    For i = m : -1 : j+1
        Determine an adjacent Givens rotation G_ij such
            that if y = G_ij^T A(:, j:j) then y_i = 0, i.e., zero a_ij .
        A := G_ij^T A
    end i
end j
```

Upon completion A is overwritten by R and

$$Q = (G_{m,1} \ \ldots \ G_{21}) \ldots (G_{m,n} \ldots G_{n+1,n})$$

Notice that the algorithm computes R column–by–column and that the zeroing within a column proceeds from the bottom up to the subdiagonal. Here is a depiction of the 4–by–3 case:



To indicate the inherent parallelism in this procedure we resort to a slightly larger example and number the a_{ij} in the order that they are zeroed:

x	x	x	x	
8	x	x	x	
7	15	x	x	
6	14	21	x	
5	13	20	26	m = 9 , n = 4
4	12	19	25	
3	11	18	24	
2	10	17	23	
1	9	16	22	

Recognize that the computation and application of G_{ij} can begin as soon as $G_{i-1,j-1}$ is applied to A. To illustrate this we tabulate the earliest "time step" that $a_{ij}(i > j)$ can be zeroed:

x	x	x	x	
8	x	x	x	
7	9	x	x	
6	8	10	x	
5	7	9	11	m = 9 , n = 4
4	6	8	10	
3	5	7	9	
2	4	6	8	
1	3	5	7	

With this notation we see in the example that four Givens updates can be performed during the seventh time step: G_{31}, G_{52}, G_{73}, and G_{94}. If we had 4 processors then they could each be assigned one of these tasks.

The parallelism that we have exposed in the above example can be formalized by rearranging the loop indexing in Algorithm 2.1 and noting that $m + n - 2$ timesteps are required.

Algorithm 2.2.

```
For k = 1: m+n-2
    For All j = 1:n
        i = m-k+1+2(j-1)
        if ( i ≤ m & i ≥ j+1)
            Determine G_ij to zero a_ij
            A := G_ij^T A
        end
    end j
end k
```

The "For All" statement reminds us that all of the updates $A := G_{ij}^T A$ associated with a given time step k are independent and can be performed in parallel.

We point out that G_{ij} can actually be computed "earlier" than we have indicated. For example, in the $(m,n) = (9,4)$ case above, we have assumed that G_{92} is computed as soon as G_{81} has been applied all the way across the matrix. In fact, G_{92} can be computed as soon as G_{81} has been applied to just the second column. For reasons that we given in §4, we have not implemented the "soon as possible" generation of G_{ij}.

Algorithm 2.2 and its natural variants can be mapped nicely onto systolic networks. See Heller and Ipsen (1983).

3. A Parallel Block QR Factorization Method. Some notation is required before a block version of algorithm 2.2 can be specified. Partition $A \in R^{m \times n}$ as follows:

$$A = \begin{bmatrix} A_{11} & \cdots & A_{1q} \\ \vdots & & \vdots \\ A_{p1} & \cdots & A_{pq} \end{bmatrix} \begin{matrix} m_1 \\ \\ m_p \end{matrix} \qquad (3.1)$$

$$\begin{matrix} n_1 & & n_q \end{matrix}$$

Here, A_{ij} is $m_i - by - n_j$ and we assume that $m_i \geq n_j$ for all i and j. If Q is an orthogonal matrix of dimension $m_{i-1} + m_i$ then we refer to

$$G_I(Q) = \text{ diag } (I_{m_1}, \ldots, I_{m_{i-2}}, \ Q, \ I_{m_{i+1}}, \ldots, \ I_{m_p})$$

as an adjacent "block Given" rotation in block planes $i-1$ and i.

Algorithm 3.1 (Block Givens QR Factorization).

For k = 1: p+q-2
  For All j = 1:q
    i = p-k+1+2(j-1)
    if (i ≤ p & i ≥ j+1)
      Determine orthogonal Q_{ij} such that

$$Q_{ij}^T \begin{bmatrix} A_{i-1,j} \\ A_{ij} \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix} \quad \text{(R upper triangular)}$$

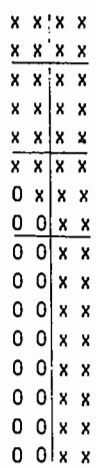
      Set $G_{ij} = G_i(Q_{ij})$ and update $A := G_{ij}^T A$
    end
  end j
end k

This procedure is identical to Algorithm 2.2 except that blocks are zeroed instead of scalars. Upon completion A is overwritten with a block upper triangular matrix R. Unless all the A_{ij} are square, then R will not be upper triangular as a scalar matrix. For example, if the partitioning in (3.1) is defined by $(m_1, m_2) = (3, 3)$ and $(n_1, n_2) = (2, 2)$ then Algorithm 3.1 overwrites A with

$$R = \begin{array}{cc|cc} x & x & x & x \\ 0 & x & x & x \\ 0 & 0 & x & x \\ \hline 0 & 0 & x & x \\ 0 & 0 & 0 & x \\ 0 & 0 & 0 & 0 \end{array}$$

Of course, it is possible to upper triangularize this matrix with further Givens operations, but that is an annoying but necessary follow-up computation.

However, there is a more serious problem associated with rectangular blocks. Consider the example $(m_1, m_2, m_3, m_4) = (2, 3, 3, 8)$, $(n_1, n_2) = (2, 2)$. At the beginning of the second time step A looks like



At this stage, Algorithm 3.1 specifies that we only upper triangularize the submatrix $A(3{:}8,1{:}2)$, i.e., the subproblem defined by blocks A_{21} and A_{31}. However, we see from the figure that a significant amount of zeroing in the second block column can take place concurrently. In particular, we could upper triangularize both $A(3{:}8,1{:}2)$ *and* $A(9{:}16,3{:}4)$.

In general, because the "bottom" submatrix A_{ij} in each subproblem is upper triangular, "taller" submatrices can be upper triangularized throughout Algorithm 3.1. In order to rearrange this algorithm so that "maximally tall" subproblems are solved at each stage, we need to drop the fixed row blocking in (3.1). We continue to assume that A has q block columns with widths $n_1 \ldots n_q$. However, instead of imposing a fixed blocking of A's rows we have chosen to determine the "height" of the subproblems through an integer parameter m_0 that satisfies $m_0 \geq n_1$. In our scheme, the subproblems in the first block column involve at most m_0 rows. Maximally tall subproblems are then solved in subsequent block columns at each step. To illustrate, consider the case $m = 100$, $m_0 = 20$, and $(n_1, n_2, n_3, n_4) = (2, 3, 5, 5)$:

Subproblem Row Ranges

Time Step	Column Ranges			
	1:2	3:5	6:10	11:15
1	81:100	--	-	-
2	63:82	83:100	-	-
3	45:64	65:85	86:100	-
4	27:46	47:67	68:90	91:100
5	9:28	29:49	50:72	73:95
6	1:10	11:31	32:54	55:77
7	-	3:13	14:36	37:59
8	-	-	6:18	19:41
9	-	-	-	11:23

In general four integers rowsrt(t,j), rowend(t,j), colsrt(j), and colend(j) are necessary to describe subproblem (t,j), e.g., 29, 49, 3, and 5 for subproblem (5,2). These index arrays and the total number of time steps t_f required can be computed as follows:

Algorithm 3.2. Let m, n, m_0, q and the column partitioning $(n_1, \ldots, n_q)$ be given with $m \geq n$ and $m_0 > n_1$. This algorithm determines t_f and the index arrays colsrt$(1 : q)$, colend$(1 : q)$, rowsrt$(1 : t_f$, 1:q), and rowend$(1 : t_f\ , 1 : q)$.

```
t_f = ceiling( max(0,m-m_0) / (m_0-n_1) ) + q
For t = 1 : t_f
   if t = 1
      For j = 1:q
         if  j = 1
               colsrt(1) = 1
               colend(1) = n_1
               rowsrt(t,j) = max( 1 , m-m_0+1)
         else
               colsrt(j) = colend(j-1) + 1
               colend(j) = colend(j-1) + n_j
               rowsrt(t,j) = m
         end
         rowend(t,j) = m
      end j
   else
      For j = 1:q
         if j = 1
               rowend(t,1) = rowsrt(t-1,1) + n_1 - 1
               rowsrt(t,1) = max(1, rowend(t,1)-m_0+1 )
         else
               rowend(t,j) = min( rowsrt(t-1,j) + n_j - 1 , m )
               rowsrt(t,j) = max( colsrt(j), min( rowend(t,j-1) +1, m))
         end
      end j
   end
end t
```

A couple of comments are in order. In block column 1, the subproblems "climb" at the "rate" $m_0 - n_1$ and so 1+ ceiling$(\max(0, m-m_0)/(m_0-n_1))$ steps are required to complete the processing of block column 1. Thereafter one block column per time step is completed. This explains the formula for t_f and why we must have $m_0 > n_1$.

In block column j, "serious" computation does not begin so long as rowsrt$(t,j) =$ rowend$(t,j) = m$. After block column j is fully triangularized, rowsrt$(t,j) =$ colsrt(j) and rowend$(t,j) =$ colend(j), conditions that normally signal that there is "nothing to do" in block column j. (An exception occurs when rowsrt$(t,j) =$ colsrt(j) and rowend$(t,j) =$ colend$(j) = m$.)

With subproblems specified by Algorithm 3.2 we can now describe the overall factorization procedure.

Algorithm 3.3 (Maximally Tall Block Givens QR Factorization).

Given m, n, m_0, q, the column partitioning $(n_1, \ldots, n_q)$ with $m \geq n$ and $m_0 > n_1$, the following algorithm overwrites $A \in R^{m \times n}$ with upper triangular $R = Q^T A$ where Q is orthogonal.

```
Compute t_f , rowsrt(1:t_f ,1:q)  , rowend(1:t_f ,1:q),
    colsrt(1:q), and colend(1:q) using Algorithm 3.2
For t = 1 : t_f
    For j = 1:q
        i_1 = rowsrt(t,j)
        i_2 = rowend(t,j)
        j_1 = colsrt(j)
        j_2 = colend(j)
        if ( i_1 = i_2 = m  or  ( i_1 = j_1 & i_2 = j_2 &  j_2 ≠ m ) )
            "Nothing to do."
        else
            Compute:  A(i_1:i_2,j_1:j_2)  = QR .
            Apply: A(i_1:i_2,j_1:n)  :=  Q^T A(i_1:i_2,j_1:n)
        end
     end j
end t
```

4. Implementation. In this section we discuss three issues associated with the implementation of Algorithm 3.3 on the LCAP-1 system: how A is arranged in shared memory, how the subproblems are solved, and how block column tasks are mapped onto processors.

The Storage of A. At time step t, the relevant row and column delimiters for the $j-th$ subproblem are $i_1 = \text{rowsrt}(t,j)$, $i_2 = \text{rowend}(t,j)$, $j_1 = \text{colsrt}(j)$, and $j_2 = \text{colend}(j)$. Here is what the array processor in charge of this subproblem must accomplish:

(1) Read $A(i_1 : i_2, j_1 : j_2)$ from shared memory.

(2) Compute an orthogonal Q such that $Q^T A(i_1 : i_2, j_1 : j_2) = R$ is upper triangular.

(3) Write the updated $A(i_j : i_2, j_1 : j_2)$ back into shared memory.

(4) Read $A(i_1 : i_2, j_1 + 1 : n)$ from shared memory.

(5) Apply Q^T to $A(i_j : j_2, j_1 + 1 : n)$.

(6) Write the updated $A(i_1 : i_2, j_2 + 1 : n)$ back into shared memory.

We assume that $A(i_1 : i_2, , j_1 : j_2)$ can fit into local memory but that because of its size, the processing of $A(i_1 : i_2, j_2 + 1 : n)$ may have to proceed in "chunks". That is, steps 4–5–6 may have to be repeated with a manageable segment of columns from $A(i_j : i_2, j_2 + 1 : n)$ each time. Note that Q stays in the AP during this process. Because one AP is responsible for applying a given Q, there is no need to pass Q on to another AP.

There is an overhead associated with traffic to and from shared memory. Reads and writes to shared memory are accomplished with a "**move**" command and can only involve continuous portions of memory. Using **move** to transfer n floating point words takes

$$T(n) = (100 + 8n/44) \quad \mu\text{sec}$$

Note that the 100 μsec startup degrades the 44mb/sec peak transfer rate. Thus, a vector of length 1000 takes 281 μsec to move for an effective data transfer rate of 28 mb/sec.

From the standpoint of processing the subproblem at hand, it would be ideal if $A(i_1 : i_2, j_1 : n)$ was continuous in shared memory for then a minimum number of **moves** would be required to carry out steps 1,3,4, and 6 above. For example, to read a contiguous 1000-by-500 submatrix from shared memory would require $T(500,000) = .09$ sec ($\equiv$ 44mb/sec). Unfortunately, storing by blocks in Algorithm 3.3 would impose significant buffer requirements and some tedious data manipulation within each AP. The buffer issue is fairly important because the AP's we used have limited local memory ($\approx$ 600 Kwords).

Because we didn't want addditional buffer requirements to limit further the size of "working" memory we chose to store A in column major order. This implies that r **moves** are required to move a submatrix with r columns. Thus, to read a 1000–by–500 submatrix requires $500 \cdot T(1000) = .14$ sec ($\equiv$ 28 mb/sec). This is actually a typical size for a submatrix move in our algorithm. When the overall implementation is considered, we can easily live with a 28 mb/sec data transfer rate.

Subproblem Solution. The basic computation in Algorithm 3.3 consists of computing a QR factorization and then applying the resulting orthogonal matrix to the "rest of A". The normal "Linpack" way to compute a QR factorization of a matrix $C \in R^{m_0 \times n_0}$ is to use Householder matrices. A Householder matrix is an orthogonal transformation of the form

$$P = I - 2vv^T \qquad v \in R^{m_0},\ \|v\|_2 = 1.$$

In the Linpack QR procedure Householders $P_1, \ldots, P_{n_0}$ are generated so that $P_{n_0} \cdots P_1 C = R$ is upper triangular. Note that $Q = P_1 \cdots P_{n_0}$.

We now consider the computation $Q^T B$ where B is some matrix. If Q is represented as a product of Householders, then the resulting algorithm is "rich" in matrix–vector multiplications. This is fine for many architectures. However, to exploit fully the FPS–164/MAX architecture, we need an update algorithm that is rich in matrix–matrix multiplication. We could accomplish this by explicitly forming the product $Q = P_1 \cdots P_{n_0}$ before applying it to B. But this would be very costly since $m_0 >> n_0$ usually. An unacceptably large $m_0 - by - m_0$ buffer would also be required by this approach.

Instead, we have chosen to use the "WY" representation for products of Householder matrices that is developed in Bischof and Van Loan (1985). In this scheme $m_0 - by - n_0$ matrices W and Y are generated such that

$$Q = P_1 \cdots P_{n_0} = 1 + WY^T$$

The ensuing update $B := Q_B^T = (1 + WY^T)^T B = B + Y(W^T B)$ is then obtained by a pair of matrix–matrix multiplications:

(i) $$Z = W^T B$$

(ii) $$B = B + YZ$$

For (i) we used the "MAX" routine **pdot** that can compute twenty parallel dot products. To initiate the parallel dot product the relevant twenty vectors must be placed in the MAX registers using another MAX routine called **ploadd**. We examine this in some detail so that an appreciate of MAX board computing can be obtained. Assume that W and Y are $m_0 - by - n_0$ and that n_0 (for simplicity) is a multiple of twenty. If B is $m_0 - by - k$ then here is how the matrix $Z = W^T B$ is formed:

```
For j = 1:20:n0
    Load W(1:m0 , j:j+19)... in to the max registers using
        pload.
    For i = 1:k
        Compute Z(j:j+19,i:i)  = W(1:m0 , j:j+19)  TB(1:m0,i:i)
        using pdot.
    end i
end j
```

The times required for each **pload** and **pdot** are approximately

$$\begin{aligned} \textbf{ploadd}: &\quad L(m_0) = 23 + 58.2^* m_0 \qquad (\mu\text{sec}) \\ \textbf{pdot}: &\quad D(m_0) = 29.7 + .738^* m_0 \qquad (\mu\text{sec}) \end{aligned}$$

Thus, $Z = W^T B$ is obtained in $(n_0/20)(L(m_0) + k \cdot D(m_0))\mu$sec. Since $Z =$ requires $2m_0 n_0 k$ flops, a calculation shows that the effective performance in megaflops is approximately given by

$$\text{Mflop}(W^T B) = \frac{55}{1 + 40/m_0 + 79/k + 31/m_0 k}$$

This expression reveals the penalty for short vectors (small m_0) and for low re–use (small k). Here is a table of some representative Mflop($W^T B$) values:

	k = 100	k = 500	k = 1000	k = 5000
m_0 = 100	25	35	37	39
m_0 = 500	29	44	47	50
m_0 = 1000	30	46	49	52
m_0 = 2000	30	47	50	53

Table 4.1

We mention that because the MAX registers can handle vectors up to length 2047, the subproblem height parameter m_0 should be chosen so that rowend(t, j) - rowsrt$(t, j) \leq 2047$ for all t and j.

We now turn our attention to the rank–n_0 update $B \leftarrow B + YZ$ that makes up the second half of the $B \leftarrow (1 + WY^T)^T B$ computation. For this calculation the FPS–164/MAX has a parallel saxpy capability that appears well suited. With two MAX boards it is possible to perform nine saxpys of the form $c_i \leftarrow c_i + s_i y$ in parallel. Note that this is a rank-one update: $C \leftarrow C + ys^T$. Here is how the update of B would proceed using the parallel saxpy routine **pvsma** and the attending load/unload routines **ploadv** and **punldv**. For simplicity, assume that k is a multiple of 9.

```
For j = 1:9:k
      Use ploadv to load B(1:m0,j:j+8)   into the max registers.
      For i = 1:n0
          Use pvsma to perform the update
             B(1:m0,j:j+8)  ← B(1:m0,j:j+8)  + Y(1:m0,i:i)Z(i:i,j:j+8)
      end i
      Use punldv to write the updated B(1:m0,j:j+8)   back to memory.
end j
```

Reasoning as we did to determine $\text{Mflop}(W^TB)$, it can be shown that

$$\text{Mflop}(B+YZ) = \frac{24}{1+34/m_0+70/n_0+62/m_0n_0}$$

Note that the re–use factor is now n_0 rather than k. This is unfortunate since in our application we typically have $k > m_0 \gg n_0$. If we look at some typical values of $\text{Mflop}(B+YZ)$, then this is what we find:

	n_0 = 20	n_0 = 40	n_0 = 60	n_0 = 80
m_0 = 100	4.9	7.7	9.5	10.8
m_0 = 500	5.2	8.5	10.7	12.3
m_0 = 1000	5.3	8.6	10.9	12.6
m_0 = 2000	5.3	8.6	11.0	12.7

Table 4.2

Thus, **pvsma** is ill–suited for the $B \leftarrow B+YZ$ update when compared to the 23–53 Mflop rates sustained by the **pdot** computation of $Z = W^TB$. For this reason we chose to use a new FPS parallel matrix multiply routine called **pmmul** that can perform the update $B \leftarrow B+YZ$ at rates more consistent with the values in Table 4.1

Two final comments about subproblem solution. The first concerns the recording of the orthogonal matrix Q. This matrix is the product of Householder matrices. Of course, these Householders are clustered and applied in WY form during Algorithm 3.3. But we can save all the Householder vectors by overwriting each zeroed subcolumn of A by the corresponding Householder vector. In particular, whenever a subcolumn $v \in R^d$ of A is zeroed by a Householder matrix $(1+2uu^T/u^Tu)$, we store $u(2:d)$ in $v(2:d)$ with the convention $u(1) = 1$. It is then possible to retrieve Q from the final array A so long as the index arrays rowsrt, rowend, colsrt, and colend are available.

Lastly, we mention that the subproblem QR factorizations in Algorithm 3.3 are typically of matrices that have a band structure. Indeed, it is usually the case that $A(\text{rowsrt}(t,j):\text{rowend}(t,j),\ \text{colsrt}(j):\text{colend}(j))$ has lower bandwidth $\text{rowsrt}(t-1,j)-\text{rowsrt}(t,j)$. This fact is exploited when the QR factorization is computed and the resulting WY factors found.

Load Balancing and Scheduling. Suppose Algorithm 3.3 is to be implemented on array processors $AP_1, \ldots, AP_p$. At time step t in Algorithm 3.3 there are q independent tasks to perform.

Task (t, j) involves

$$\begin{aligned} \text{Factoring:} \quad & A(\text{rowsrt}(t,j) : \text{rowend}(t,j), \text{colsrt}(j) : \text{colend}(j)) = QR \\ \text{Computing:} \quad & Q^T A(\text{rowsrt}(t,j) : \text{rowend}(t,j), \text{colsrt}(j) : n)) \end{aligned}$$

Here, t and j satisfy $1 \leq t \leq t_f$ and $1 \leq j \leq q$. If $p = q$ then an immediate load balancing problem arises if each block column has the same width because task (t, j) generally has more matrix to update than task $(t, j+1)$. One way around this difficulty is to make each block column wider than its predecessor. We illustrate this for the case $q = 2$ with block column widths n_1 and n_2. Assuming a subproblem height of m_0 then approximately $2m_0n_1^2 + 2n_1m_0n_2$ flops are required for task $(t, 1)$. On the other hand, $2m_0n_2^2$ flops are required for task $(t, 2)$ if we again assume a subproblem height of m_0. These two flop counts are approximately equal if $(n_1/n_2) \approx .62$.

For general q it is possible to work out quotients n_j/n_{j+1} for $j = 1 : q-1$ so that approximate load balancing results for the column partitioning $n_1, \ldots, n_q$. Of course, in practice it would make more sense to base column partitioning guidelines upon benchmarks rather than upon flop counts. We have not pursued this.

Instead we make the block column widths narrow enough so that the number of independent tasks q is significantly larger than the number p of assigned APs. Approximate load balancing is then achieved by assigning AP_k to block columns $j = k : p : q$. For example, if $p = 3$ and $q = 12$, then AP_1 works on block columns 1, 4, 7 and 10, AP_2 is assigned to block columns 2,5,8, and 11, while AP_3 is applied to block columns 3,6,9, and 12. In a typical time step, each AP will work on 4 subproblems with a greater balance of work than if $q = 3$. This style of distributing tasks has been widely used in parallel matrix factorization work, see George, Heath, and Liu (1985). A fringe benefit of this approach is that we can choose block column widths to be a multiple of twenty. This allows for efficient exploitation of the 164/MAX architecture that permits twenty parallel dot products. In our examples we used uniform block column widths of twenty and thus $q \approx n/20$.

To actually execute algorithm 3.3 in parallel on LCAP–1 we implemented a lock– step synchronization scheme using "barriers". The blocking arrays rowrst, rowend, colsrt, and colend are determined by the host and then downloaded into the p array processors assigned to the computation. The matrix A is also downloaded into the shared memory through the APs. The AP_k then executes the following program:

Algorithm 4.1 (Processor k's Share of Algorithm 3.3).

```
For t = 1:t_f
   For j = k:p:q
      Compute  A(rowsrt(t,j):rowend(t,j),colsrt(j):colend(j))  = QR
      Update   A(rowsrt(t,j):rowend(t,j),colsrt(j):n)
   end j
   Barrier
end t
```

When the barrier is encountered, execution is suspended until all the other AP programs reach their barrier. After this is accomplished the processing of the next time step begins.

Further details about the LCAP–1 system software required by our implementation may be found in Chin and Lorenzo (1986).

Some Results and Conclusions. In testing our implementation we ran our codes on random matrices $A \in R^{m\times n}$ with the property that $A(1:m, 1:n-1)e = A(1:m, n:n)$ where e is the vector of all ones. The correctness of R was then confirmed by checking the equations $R(1:n-1, 1:n-1)e = R(1:n-1, n:n)$ and $R(n,n) = 0$.

We report on two of the several examples that we solved using the parallel QR code. We do not pretend that our results are conclusive. They merely confirm some natural suspicions and point the way to future research.

The first example indicates that we can get away with our lock step, coarse grained approach if A is large enough and suitably blocked. Here is what we found by using one, two, and three APs to solve an $(m,n) = (5000, 1000)$ problem with $m_0 = 1000, q = 50$, and $n_1 = \cdots = n_{50} = 20$.

Number of Processors	Time (seconds)	Speed-Up	Effective Mflop
1	606	1.00	17
2	310	1.95	33
3	211	2.87	48

Table 5.1

About 25% of the elapsed time is spent on transmitting submatrices to and from the shared memory. To see roughly where this percent comes from consider the update $B \leftarrow (1 + WY^T)^T B$ of a $1000 - by - 500$ submatrix B in shared memory where $W, Y \in R^{1000\times 20}$. If this update is performed at a rate of 30 Mflops then approximately 1.3 seconds must be devoted to computation. To transfer B to or from shared memory requires about .14 seconds. Thus, the fraction of time spent on communication is approximately $.18 \approx .28/1.58$.

We next discuss an example where the load balancing is not quite so nice, resulting in a degradation of performance. In the example $m = 5040, m_0 = 1040$, $n = 500$, $q = 13$, and $n_1 = \cdots = n_{12} = 40, n_{13} = 20$. Three APs were used and thus block column tasks are assigned as follows:

$$\mathrm{AP}_1 \leftarrow (1,4,7,10,13) \qquad \mathrm{AP}_2 \leftarrow (2,5,8,11) \qquad \mathrm{AP}_3 \leftarrow (3,6,9,12)$$

Because only five steps are required to process each block column, there are never more than five "active" tasks at any one time step. This makes load balancing a little problematical. The following table indicates the time (in seconds) that each AP spends computing at each timestep.

Time Step	AP_1	AP_2	AP_3
1	3.52	0.00	0.00
2	3.64	3.15	0.00
3	3.64	3.39	2.82
4	3.64	3.42	3.16
5	5.85	3.39	3.16
6	3.10	5.31	4.83
7	2.70	2.82	4.83
8	2.70	2.46	2.58
9	3.88	2.46	2.24
10	2.05	3.42	2.24
11	1.76	1.79	3.01
12	1.76	1.52	1.54
13	1.99	1.52	1.30
14	.62	1.52	1.30
15	.43	1.61	1.30
16	.43	0.00	0.13
17	.43	0.00	0.00

Table 5.2

The time required for the entire computation is 51.2 seconds, the sum of the maximum times in each row of the table. If computation was equally shared at each time step then approximately 38.1 seconds would be required for the complete computation.

The somewhat inefficient use of the APs highlighted by the second example could be rectified in several ways:

1 Choose a smaller m_0. This would have the effect of increasing the number of tasks to be shared at each time step.

2 Vary the block column widths so as to even out the update work.

3 Instead of letting the AP that generates a Q be entirely responsible for its application, share the update.

We have not fully explored these possibilities. Note that the first and third suggestions imply smaller matrix multiplications and thereby reduced 164/MAX performance.

A more promising way to address the load balancing issue would be to incorporate a dynamic scheduling of tasks as is discussed, for example, in George, Heath, and Liu (1985) and Dongarra, Sorenson, and Sameh (1986). One way to do this is to order the tasks (t, j) defined in §4 as follows:

$$(1,1),(1,2),\ldots,(1,q),(2,1),(2,2),\ldots,(2,q),\ldots,(t_f,1),\ldots,(t_f,q)$$

After completing a task each AP would go to this list and "grab" the next available task subject to rules that preserve the integrity of the overall procedure. We will report on this elsewhere.

Acknowledgements. The implementation of the algorithm described in this paper would not have been possible without the expert assistance of Dr. Doug Logan at IBM Kingston and my Ph.D. student Chris Bischof at Cornell.

This work has been supported by ONR contract N00014–83–K–0640, the Mathematical Sciences Institute at Cornell which is sponsored by the Army Research Office, and by the IBM Corporation. Computations were performed at IBM Kingston and on the Production Supercomputer Facility at Cornell which is supported in part by the National Science Foundation and IBM.

REFERENCES

C. Bischof and C. Van Loan, *The WY Representation for products of Householder matrices*, (1987), to appear in Siam J. Scientific and Statistical Computing.

A.E. Charlesworth and J.L. Gustafson, *Introducing replicated VLSI to supercomputing: the FPS–164/MAX scientific computer*, (1986), IEEE Computer, March, 10–23.

S. Chin and D. Lorenzo, *Parallel Computation on the Loosely coupled array of processors: tools and guidelines*, (1986), Report KGN–25, IBM Kingston, Dept. 48B, Bldg 963, Kingston, NY 12401.

E. Clementi and D. Logan, *Parallel processing with the loosely coupled array processor system*, (1985), Report KGN–43, IBM Kingston, Dept. 48B, Bldg 963, Kingston, NY 12401.

J. Dongarra, A.H. Sameh and D.C. Sorenson, *Implementation of some concurrent algorithms for matrix factorization*, (1986), Parallel Computing, 3, pp. 25–34.

W.M. Gentleman and H.T. Kung, *Matrix triangularization by systolic arrays*, (1982), Proc. SPIE Vol. 298, Real Time Signal Processing IV, 19–26.

A. George, M.T. Heath, and J. Liu, *Parallel Cholesky factorization on a multiprocessor*, (1985), Report ORNL 6124, Math. Sci. Div., Oak Ridge National Laboratory, Oak Ridge, TN.

G.H. Golub and C. Van Loan, *Matrix Computations*, (1983), The Johns Hopkins University Press, Baltimore, Md.

D. Heller and I.C.F. Ipsen, *Systolic networks for orthogonal decompositions*, (1983), Siam J. Scientific and Stat. Computing, 4, 261–269.