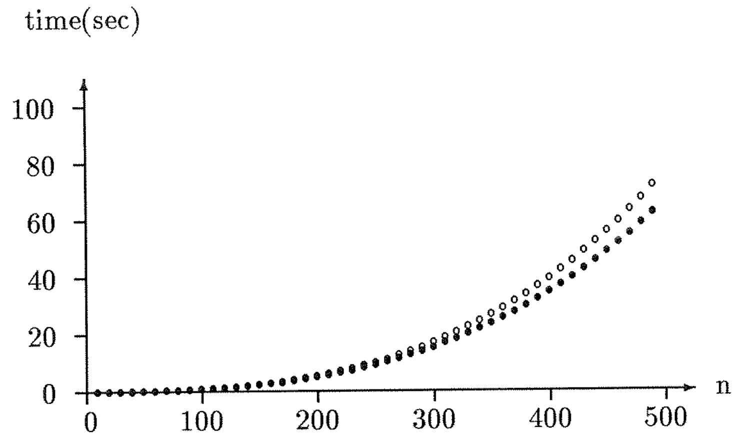

Figure 4.4: Observed and expected timings for $p = 16$ during Householder Orthogonalization (key:: •: observed, ∘: expected)

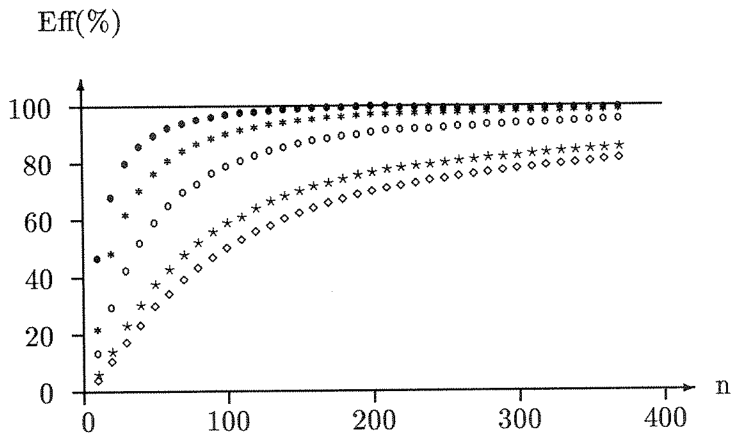

Figure 4.5: Observed efficiencies attained during Householder Orthogonalization (key:: •: $p = 2$, $*$: $p = 4$, ∘: $p = 8$, $\star$: $p = 16$, $\diamond$: $p = 24$)

once $n/p > 6$. The parallel Householder orthogonalization algorithm shows higher efficiencies for the same problem size than any of the algorithms presented in this paper. Comparing it to Gaussian elimination we see that they share the same communication expense but the computation required during each submatrix update is nearly doubled in the Q-R decomposition. This means that more useful computation is performed in parallel for the cost of a broadcast and hence the algorithm is more efficient.

4.2 Modified Gram-Schmidt Method

If we partition A and Q by columns and consider the elements of R we can express the Q-R factorization as,

$$(a_1, \ldots, a_n) = (q_1, \ldots, q_n) \begin{pmatrix} \rho_{11} & \rho_{12} & \cdots & \rho_{1n} \\ 0 & \rho_{22} & \cdots & \rho_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & \rho_{nn} \end{pmatrix}$$

and clearly,

$$\begin{aligned} a_k &= \sum_{i=1}^{k} \rho_{ik} q_i \\ &= \rho_{kk} q_k + \sum_{i=1}^{k-1} \rho_{ik} q_i. \end{aligned}$$

If A has linearly independent columns, then R must be nonsingular and therefore $\rho_{kk} \neq 0$ and we can solve for q_k,

$$q_k = \frac{1}{\rho_{kk}} \left(a_k - \sum_{i=1}^{k-1} \rho_{ik} q_i \right),$$

where the orthonormality of the q_i implies that

$$\rho_{ik} = q_i^T a_k \qquad \text{for } i = 1, \ldots, k.$$

The expression for q_k can be interpreted as the normalized vector resulting from subtracting from a_k all its components in the direction of the already formulated q_i. This ensures that q_k is orthogonal to span$(q_1, \ldots, q_{k-1})$.

These expressions for q_k and ρ_{ik} lead to the Classical Gram-Schmidt (CGS) method which generates the kth column of Q and R in the kth step of the algorithm, but unfortunately exhibits poor numerical behavior [2]. However a rearrangement of the computations yields the more stable Modified Gram-Schmidt (MGS) method.

4.2.1 Sequential Algorithm

Since Q has orthonormal columns, $Q^TQ = I$ and we can rewrite $A = QR$ as $Q^TA = R$ or,

$$\begin{pmatrix} q_1^T \\ q_2^T \\ \vdots \\ q_n^T \end{pmatrix} (a_1, \ldots, a_n) = \begin{pmatrix} \rho_{11} & \rho_{12} & \cdots & \rho_{1n} \\ 0 & \rho_{22} & \cdots & \rho_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & \rho_{nn} \end{pmatrix}.$$

From this we see that,

$$\rho_{kj} = q_k^T a_j \qquad \text{for } j = k, \ldots, n$$

and if we normalize q_k by setting $q_k = a_k/\|a_k\|_2$, then

$$\rho_{kk} = q_k^T a_k = \|a_k\|_2.$$

The condition that the q_i be orthogonal requires that we subtract from a_j its component in the direction of q_k, for each a_j not yet orthogonal to q_k (i.e., $j > k$). This can be done each time a new q_k is formed by,

$$a_j \leftarrow a_j - (q_k^T a_j) q_k \qquad \text{for } j > k.$$

The algorithm (see Figure 4.6) begins by setting $q_1 = a_1/\|a_1\|_2$ and then determining the first row of R and subtracting the q_1 component from each of $a_2, a_3, \ldots, a_n$. In the second iteration, q_2 is normalized the second row of R determined and the q_2 component subtracted from $a_3, \ldots, a_n$. At this point, $q_2 = 1/\rho_{22}(a_2 - (q_1^T a_2)q_1)$ and the orthogonality of q_1 and q_2 is easily verified since $q_1^T q_1 = 1$ and

$$q_1^T q_2 = \frac{1}{\rho_{22}}(q_1^T a_2 - (q_1^T a_2) q_1^T q_1) = 0.$$

The orthogonality of the q_i can be shown inductively.

In the kth iteration finding the norm and scaling a_k requires approximately $2m\gamma$ time and the execution of the inner loop takes approximately $2(n-k)m\gamma$ time. Summing over all iterations the total time complexity is given as,

$$\mathrm{T}_1(m, n) \approx \sum_{k=1}^{n} 2m(n - k + 1)\gamma \approx mn^2\gamma. \tag{4.3}$$

Algorithm 4.3 *This algorithm uses MGS to factor A into the product QR where A is overwritten by Q. In the kth iteration the kth column of Q and the kth row of R are determined.*

```
for $k = 1, \ldots, n$
    $\rho_{kk} \leftarrow \|a_k\|_2$
    $q_k = a_k \leftarrow a_k / \rho_{kk}$
    for $j = k+1, \ldots, n$
        $\rho_{kj} \leftarrow q_k^T a_j$
        $a_j \leftarrow a_j - \rho_{kj} q_k$
```

Figure 4.6: Modified Gram-Schmidt

4.2.2 Parallel Algorithm

The parallel MGS algorithm (see Figure 4.7) employs only the broadcast primative to factor a matrix A with linearly independent columns into the product of a matrix Q with orthonormal columns and an upper triangular matrix R. Again we distribute the matrix A to the ring of processors in a column wrapped fashion so that each processor has approximately (n/p) columns.

The algorithm loops through a set of computations for each of the n columns of the matrix A. In the kth iteration, the processor that owns a_k, $\mathbf{P}(k)$, generates q_k by normalizing a_k and then broadcasts this vector to all other processors. Once a processor receives q_k it can update its columns a_j $(j > k)$ by subtracting the q_k component of a_j from a_j.

In Figure 4.7, the expressions in brackets indicate the effective contribution to the time complexity of that step in each iteration. Determining ρ_{kk} and scaling a_k requires $2m\gamma$ time, and the effective time to perform the broadcast is equivalent to the time to perform two node-to-neighbor communications with a message of m double precision floating point numbers. Communication time in each iteration is $2(\alpha + m\beta)$. The $(k+1)$th iteration begins as $\mathbf{P}(k+1)$ completes updating its columns of A. Processor $\mathbf{P}(k+1)$ owns $\lceil (n-k)/p \rceil$ columns that must be updated. The update of each column requires a dot product and saxpy operation each of order m. Using the approximation $\lceil (n-k)/p \rceil \approx (n-k)/p + 1$, the time for $\mathbf{P}(k+1)$ to update its columns is then, $2m(\frac{n-k}{p} + 1)\gamma$. Summing these three expressions over all

Algorithm 4.4 *This algorithm drives $\mathbf{P}_i$, a node in a ring of processors to form the orthonormal basis which spans the column space of A. A is overwritten by Q of the product $A = QR$, where Q has orthonormal columns and R is an upper triangular matrix with positive diagonal elements. The matrix is distributed among the processors in a column wrapped fashion.*

```
for $k = 1, \ldots, n$
    if $k \in \mathbf{P}_i$
        $\rho_{kk} \leftarrow \|a_k\|_2$                         $[m\gamma]$
        $q_k = a_k \leftarrow a_k / \rho_{kk}$                   $[m\gamma]$
        broadcast $q_k$                                          $[2(\alpha + m\beta)]$
    else
        receive $q_k$
    for $j \in \mathbf{P}_i$ and $j > k$
        $\rho_{kj} \leftarrow q_k^T a_j$
        $a_j \leftarrow a_j - \rho_{kj} q_k$                     $[2m\lceil \frac{n-k}{p} \rceil \gamma]$
```

Figure 4.7: Parallel Modified Gram-Schmidt

iterations gives the total time complexity of the algorithm,

$$\begin{aligned} \mathrm{T}_p(n) &\approx \sum_{k=1}^{n} \left[2m\gamma \left(\frac{n-k}{p} + 2 \right) + 2(\alpha + m\beta) \right] \\ &\approx \left(\frac{n^2 m}{p} + 4nm \right) \gamma + 2n(\alpha + m\beta). \end{aligned} \tag{4.4}$$

This algorithm also approaches 100% efficiency for large n and fixed p. Comparing (4.3) and (4.4) we see that,

$$\lim_{n \to \infty} \frac{\mathrm{T}_1(n)}{\mathrm{T}_p(n)} = p.$$

However, as p approaches n the two terms $(2nm)\beta$ and $(4nm)\gamma$ gain significance and efficiency drops. The term $(4nm)\gamma$ reflects the cost of the sequential portion of each iteration, where one processor determines ρ_{kk} and normalizes q_k.

4.2.3 Implementation and Numerical Experiments

The routine PMGS (parallel modified Gram-Schmidt) drives each node in a ring of processors to form the orthonormal basis which spans the column space

PMGS(M,N,A,R,INFO)

- On Entry

  M number of rows in the matrix

  N number of columns in the matrix

  A is a pointer to a vector of type FLT which contains columns of the matrix to be factored.

- On Return

  A a pointer to a vector containing columns of the matrix A which has been overwritten by Q a matrix with orthonormal columns.

  R a pointer to a vector containing columns of the $n \times n$ upper triangular matrix R, distributed among the processors in a column wrapped fashion.

  INFO is a pointer to an integer
  $= 0$ normal value
  $= k$ if the kth column was linearly dependent

Figure 4.8: PMGS

of A, a rectangular matrix with linearly independent columns. A is factored into the product QR, where Q has orthonormal columns and R is an upper triangular matrix with positive diagonal elements. A is overwritten by Q. The matrix is distributed among the processors in a column wrapped fashion. The calling sequence and argument descriptions are given in Figure 4.8.

Figure 4.9 plots the observed timings obtained by PMGS with various ring sizes on the Symult S2010 for problems ranging up to $n = 490$. For these timings m was set to $n + 2$. Because of memory restrictions, the maximum problem size was limited below 490 for certain ring sizes. Figure 4.10 exhibits the agreement between observed and theoretical run times.

Figure 4.11 plots the observed efficiency attained by the parallel algorithm for various ring sizes and problem sizes. Reasonable efficiencies can be obtained once $n/p > 6$. The parallel modified Gram-Schmidt algorithm shows similar efficiencies to the parallel Householder orthogonalization algorithm. Each communication is more expensive yet more computation is performed for each broadcast.

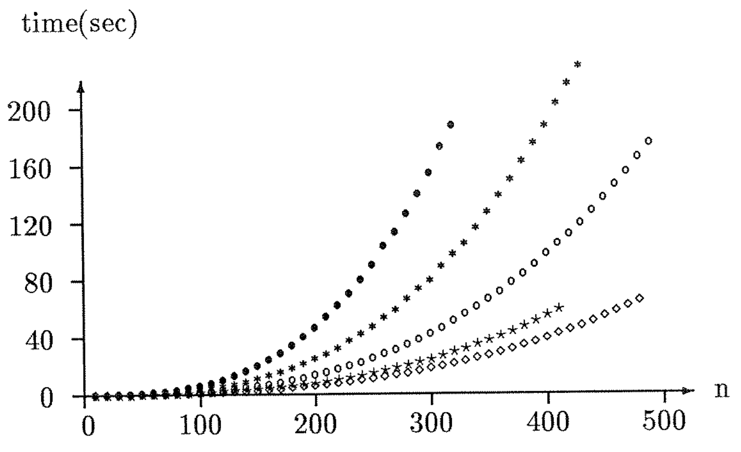

Figure 4.9: Observed timings attained during Modified Gram-Schmidt (key:: •: $p = 2$, $*$: $p = 4$, $\circ$: $p = 8$, $\star$: $p = 16$, $\diamond$: $p = 24$)

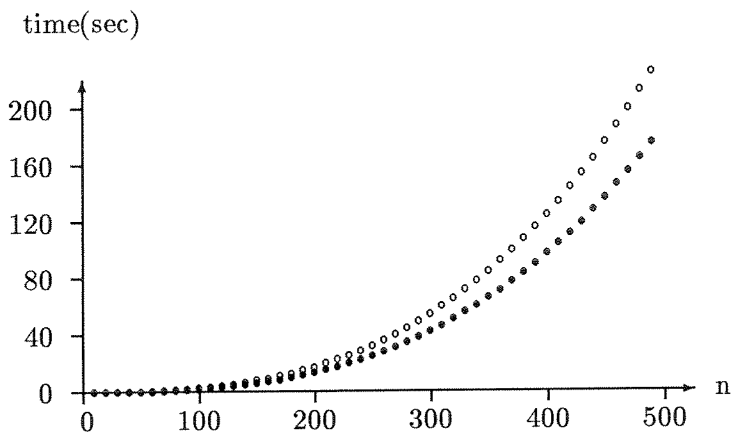

Figure 4.10: Observed and expected timings for $p = 8$ during Modified Gram-Schmidt (key:: •: observed, $\circ$: expected)

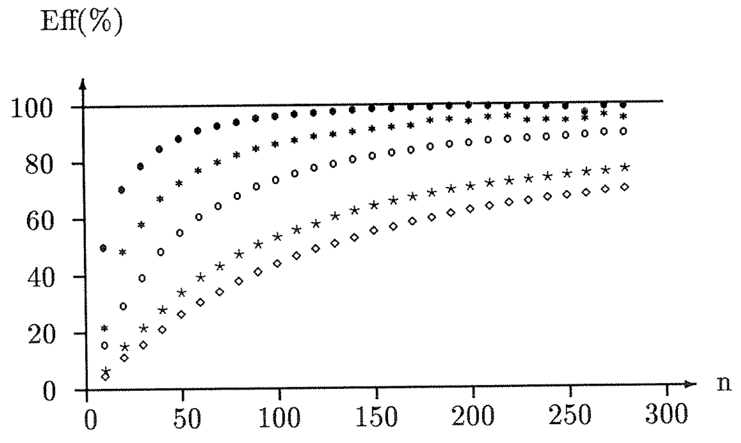

Figure 4.11: Observed efficiency attained during Modified Gram-Schmidt (key:: •: $p = 2$, $*$: $p = 4$, $\circ$: $p = 8$, $\star$: $p = 16$, $\diamond$: $p = 24$)

Chapter 5

Householder Reduction to Hessenberg Form

In this chapter we will describe the reduction of a matrix to upper Hessenberg form by orthogonal similarity transformations. If the matrix is symmetric the result is a tridiagonal matrix, however the algorithm that will be presented does not expect or take advantage of symmetry.

A matrix, A, is in upper Hessenberg form if $\alpha_{ij} = 0$ for $i > j+1$. A 5×5 upper Hessenberg matrix has the form,

$$\begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \end{pmatrix}.$$

Reducing a matrix to upper Hessenberg form, before applying the QR algorithm to compute the Schur decomposition, lowers the cost of an iteration by an order of magnitude from $O(n^3)$ to $O(n^2)$.

5.1 Sequential Algorithm

The object of the algorithm is to compute the product,

$$U^T A U = H$$

where H is upper Hessenberg and U is unitary and is the product of Householder matrices, $H_1, \ldots, H_{n-2}$.

The algorithm (see Figure 5.1) begins by determining a Householder transformation, $\tilde{H}_1$ to zero the first column of $A = A^{(1)}$ below the first subdiagonal. A is then multiplied by

$$H_1 = \begin{pmatrix} I_1 & 0 \\ 0 & \tilde{H}_1 \end{pmatrix}$$

from the left and the right. Post multiplication of $H_1 A^{(1)}$ by H_1 does not affect the first column so we have,

$$A^{(2)} = H_1 A^{(1)} H_1 = \begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \end{pmatrix}.$$

In the next step $H_2 = \operatorname{diag}(I_2, \tilde{H}_2)$ is determined so that,

$$A^{(3)} = H_2 A^{(2)} H_2 = \begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & \times & \times & \times \end{pmatrix}.$$

Again the application of H_2 from the right does not fill in the zero elements in the first or second columns.

More generally, let $A^{(1)} = A \in \mathbf{R}^{n\times n}$ and after the $(k-1)$th step

$$(H_1, \ldots, H_{k-1})^T A (H_1, \ldots, H_{k-1}) = A^{(k)} = \begin{array}{c} \left(\begin{array}{ccc} \multicolumn{2}{c}{A_{11}^{(k)}} & A_{12}^{(k)} \\ 0 & a_{22}^{(k)} & A_{23}^{(k)} \end{array}\right) \\ \begin{array}{ccc} \scriptstyle k-1 & \scriptstyle 1 & \scriptstyle n-k \end{array} \end{array} \begin{array}{c} \scriptstyle k \\ \scriptstyle n-k \\ \end{array}$$

where $A_{11}^{(k)} \in \mathbf{R}^{k\times k}$ is upper Hessenberg. In the kth step we determine $\tilde{H}_k \in \mathbf{R}^{(n-k)\times(n-k)}$ such that $\tilde{H}_k a_{22}^{(k)}$ is a multiple of $e_1 \in \mathbf{R}^{n-k}$ (cf. Section 4.1.1) and compute $H_k A^{(k)} H_k$, where $H_k = \operatorname{diag}(I_k, \tilde{H}_k)$. The premultiplication in the kth iteration takes the form,

$$\begin{aligned} H_k A^{(k)} &= \begin{pmatrix} I_k & 0 \\ 0 & \tilde{H}_k \end{pmatrix} \begin{pmatrix} \multicolumn{2}{c}{A_{11}^{(k)}} & A_{12}^{(k)} \\ 0 & a_{22}^{(k)} & A_{23}^{(k)} \end{pmatrix} \\ &= \begin{pmatrix} \multicolumn{2}{c}{A_{11}^{(k)}} & A_{12}^{(k)} \\ 0 & \eta e_1 & \tilde{H}_k A_{23}^{(k)} \end{pmatrix} \end{aligned}$$

Post multiplication by H_k produces,

$$\begin{aligned} A^{(k+1)} = H_k A^{(k)} H_k &= \begin{pmatrix} A_{11}^{(k+1)} & A_{12}^{(k)} \\ 0 & \tilde{A}_{23}^{(k)} \end{pmatrix} \begin{pmatrix} I_k & 0 \\ 0 & \tilde{H}_k \end{pmatrix} \\ &= \begin{pmatrix} A_{11}^{(k+1)} & A_{12}^{(k)} \tilde{H}_k \\ 0 & \tilde{A}_{23}^{(k)} \tilde{H}_k \end{pmatrix}. \end{aligned}$$

Algorithm 5.1 *This algorithm overwrites $A \in \mathbf{R}^{n\times n}$ with the upper Hessenberg matrix U^TAU where $U = H_1 \dots H_{n-2}$ is the product of Householder matrices. The vector v and scalar $\beta = 2/v^Tv$ which determine $\tilde{H}_k$ are stored below the first subdiagonal of A.*

```
$A^{(1)} = A$
for $k = 1, \dots, n-2$
    determine $\tilde{H}_k$ such that $\tilde{H}_k(\alpha_{k+1,k}, \dots, \alpha_{nk})^T = (\eta, 0, \dots, 0)^T$
    $\tilde{H}_k = $ diag$(I_k, \tilde{H}_k)$
    $A^{(k+1)} \leftarrow H_k A^{(k)} H_k$
```

Figure 5.1: Householder Reduction to Hessenberg Form

The cost of determining $\tilde{H}_k$ is $(n-k)\gamma$, and the premultiplication which consists of computing $\tilde{A}_{23}^{(k)} = \tilde{H}_k A_{23}^{(k)}$ requires $2(n-k)^2\gamma$ time, as explained in Section 4.1.1. The post multiplication consists of computing both $A_{12}^{(k)}\tilde{H}_k$ and $\tilde{A}_{23}^{(k)}\tilde{H}_k$, and requires $2n(n-k)\gamma$ time. To see this let us examine how $A_{12}^{(k)}\tilde{H}_k$ can be formed. Let $A_{12}^{(k)} = (a_{k+1}, \dots, a_n)$ and $\tilde{H}_k = I - \beta vv^T$, and then

$$A_{12}^{(k)}\tilde{H}_k = A_{12}^{(k)} - \beta(A_{12}^{(k)}v)v^T.$$

First we compute

$$A_{12}^{(k)}v = \sum_{j=k+1}^{n} a_j \nu_j = y$$

which takes approximately $(n-k)k\gamma$ time. Then each of the $(n-k)$ columns of $A_{12}^{(k)}\tilde{H}_k$ can be formed in $(n-k)k\gamma$ time by

$$a_j \leftarrow a_j - \beta\nu_j y.$$

Forming $A_{12}^{(k)}\tilde{H}_k$ requires $2k(n-k)\gamma$ time and by similar computations $\tilde{A}_{23}^{(k)}\tilde{H}_k$ can be formed in $2(n-k)^2\gamma$ time. Performing both of these operations completes the post multiplication. This makes the total time complexity,

$$T_1(n) = \sum_{k=1}^{n-2} \Big[(n-k) + 2(n-k)^2 + 2n(n-k)\Big]\gamma \approx \frac{5n^3}{3}. \tag{5.1}$$

5.2 Parallel Algorithm

A parallel Householder reduction to upper Hessenberg form can be implemented on a ring of processors using the broadcast, vector sum, and total exchange communications primitives. Again the matrix A is distributed to the ring of processors in a column wrapped fashion so that each processor has approximately (n/p) columns.

The algorithm loops through a sequence of operations for each of the first $(n-2)$ columns of A. During the kth iteration, a Householder matrix, $\tilde{H}_k$, of order $(n-k)$ is computed, $A^{(k)}$ is multiplied by $\mathrm{diag}(I_k, \tilde{H}_k) = H_k$ from the left and then H_k multiplies this result from the right. Up through the formation of $H_k A^{(k)}$ the algorithm is similar to the Q-R factorization by Householder transformations (cf. Section 4.1.2). The Householder transformation, $\tilde{H}_k$, can be completely determined by processor $\mathbf{P}(k)$. Once computed, $\mathbf{P}(k)$ can broadcast v and β and all processors can simultaneously participate in the premultiplication.

Post multiplication requires additional communication. The operation consists of overwritting the last $(n-k)$ columns of $A^{(k)}$ with,

$$\begin{pmatrix} A_{12}^{(k)} \\ \tilde{A}_{23}^{(k)} \end{pmatrix} \tilde{H}_k = \begin{pmatrix} A_{12}^{(k)} \\ \tilde{A}_{23}^{(k)} \end{pmatrix} - \beta \begin{pmatrix} A_{12}^{(k)} \\ \tilde{A}_{23}^{(k)} \end{pmatrix} vv^T.$$

The first step is to compute

$$y = \begin{pmatrix} A_{12}^{(k)} \\ \tilde{A}_{23}^{(k)} \end{pmatrix} v = (a_{k+1}, \ldots, a_n)v = \sum_{j=k+1}^{n} a_j \nu_j$$

which requires each processor, $\mathbf{P}_i$, to form the partial sum,

$$y^{(i)} = \sum_{a_j \in \mathbf{P}_i} a_j \nu_j$$

and then for all processors to synchronize to form and receive the vector,

$$y = \sum_i y^{(i)}.$$

This is done through the vector sum and total exchange primitives (cf. Section 2.1.4 and 2.1.6).

Once y is formed and known to each processor, all processors can complete the update of their portion of the last $(n-k)$ columns by computing,

$$a_j \leftarrow a_j - \beta \nu_j y.$$

Algorithm 5.2 *This algorithm drives processor $\mathbf{P}_i$, a node in a ring of processors, to reduce a general matrix, A, to upper Hessenberg form by applying Householder transformations. The matrix is distributed among the processors in a column wrapped fashion.*

```
for $k = 1, \ldots, n-2$
  if $k \in \mathbf{P}_i$
    determine $\tilde{H}_k : \tilde{H}_k(\alpha_{k+1,k}, \ldots, \alpha_{nk})^T = \eta e_1$   $[(n-k)\gamma]$
    broadcast $v$ and $\beta$                                  $[(p-1)(\alpha + (n-k)\beta)]$
  else
    receive $v$ and $\beta$
  update $a_j \leftarrow a_j - \beta(v^T a_j)v, \;\; j \in \mathbf{P}_i > k$          $[(2\lceil \frac{n-k}{p} \rceil (n-k))\gamma]$
  compute $y^{(i)} = \sum \nu_j a_j, \;\; j \in \mathbf{P}_i > k$               $[\lceil \frac{n-k}{p} \rceil n\gamma]$
  vector sum $y = \sum y(i)$ leaving $y_i$ on $\mathbf{P}_i$         $[(p-1)(\alpha + \lceil \frac{n}{p} \rceil (\beta + \gamma)]$
  total exchange $y_i$                                      $[(p-1)(\alpha + \lceil \frac{n}{p} \rceil \beta]$
  update $a_j \leftarrow a_j - \beta \nu_j y, \;\; j \in \mathbf{P}_i > k$               $[\lceil \frac{n-k}{p} \rceil n\gamma]$
```

Figure 5.2: Parallel Householder Reduction to Hessenberg Form

Figure 5.2 lists the pseudo code which drives each node; the expressions in brackets indicate the effective contribution to the time complexity of that step in each iteration. As in the sequential algorithm, computing the Householder transformation requires $(n-k)\gamma$ time. Because of the subsequent synchronization required by a vector sum operation, the effective cost of the broadcast is the time for the message to completely traverse the ring. The message is a vector of $(n-k)$ double precision floating point numbers so the time to communicate in each iteration is $(p-1)(\alpha + (n-k)\beta)$. The update associated with the premultiplication requires a dot product and saxpy operation each of order $(n-k)$. Since each processor has $\lceil (n-k)/p \rceil$ columns, the time to perform this update is $2\lceil (n-k)/p \rceil (n-k)\gamma$. Computing the partial sum, $y^{(i)}$ entails scaling and summing $\lceil (n-k)/p \rceil$ vectors of length n. This costs $\lceil (n-k)/p \rceil n\gamma$ time. The vector sum with vectors of length n costs $(p-1)(\alpha + \lceil n/p \rceil \beta + \lceil n/p \rceil \gamma)$ and the total exchange of vectors of size $\lceil n/p \rceil$ costs $(p-1)(\alpha + \lceil n/p \rceil \beta)$ (cf. Section 2.1.4 and 2.1.6).

Using the approximation $\lceil x \rceil \approx x + 1$, and summing these expressions over all iterations gives the total time complexity of the algorithm,

$$\begin{aligned}
\mathrm{T}_p(n) &\approx \sum_{k=1}^{n-2} \left[2\left(\frac{n-k}{p}+1\right)(2n-k)+n-k \right]\gamma \\
&\quad +(p-1)\left(2\alpha + \left(n-k+\frac{n}{p}\right)\beta\right) \\
&\approx \left(\frac{5n^3}{3p}+\frac{n^2}{2}\right)\gamma + 3np\alpha + \frac{n^2p}{2}\beta. \qquad (5.2)
\end{aligned}$$

The term $(n^2/2)\gamma$ accounts for the cost of a single processor determining $\tilde{H}_k$ every iteration and becomes significant as p approaches n. For $n \gg p$ however, (5.2) compares favorably with the sequential time complexity (5.1) and if p is fixed and the problem size increases, 100% utilization of the nodes will be approached.

Note: Because of the synchronization required by the distributed vector sum, this algorithm cannot pipeline the computations of several iterations and mask a portion of the broadcast with computation as done in the QR decomposition. For this reason, we expect that this algorithm will perform more efficiently when implemented using the pipelined broadcast, as planned in future work.

5.3 Implementation and Numerical Experiments

The routine PGEHR (parallel general matrix Hessenberg reduction) drives each node in a ring of processors to reduce a general matrix, A, by applying Householder transformations, so that $H = U^TAU$ is upper Hessenberg and U is unitary. The matrix is distributed among the processors in a column wrapped fashion. The calling sequence and argument descriptions are given in Figure 5.3.

Figure 5.4 plots the observed timings obtained by the parallel algorithm with various ring sizes on the Symult S2010 for problems ranging up to $n = 490$. Figure 5.5 plots the expected and observed timings obtained by the parallel algorithm with a ring of sixteen processors. The almost 10% discrepancy is largely caused by the overestimation of the cost of an inner product.

Figure 5.6 plots the observed efficiency attained by the parallel algorithm for various ring sizes and problem sizes. Reasonable efficiencies can be obtained once $n/p > 10$.

PGEHR(N,A,INFO)

- On Entry

  N order of the matrix

  A is a pointer to a vector of type FLT which contains columns of the matrix to be reduced. Each column of A has been extended to a length of $n+2$ to store further information required to recover the Householder transformation used at each step.

- On Return

  A a pointer to a vector containing columns of the matrix A which has been overwritten by H above the second subdiagonal. The vectors v which define the transformations $H_k = I - \beta vv^T$ are stored below the first subdiagonal. The first element of v and β are stored in $A[n+2][k]$ and $A[n+1][k]$ respectively. The unitary matrix $U = H_1 H_2 \ldots H_n$ can be formed from these vectors.

  INFO is a pointer to an integer
  $= 0$ normal value
  $= k$ if the kth column was already reduced

Figure 5.3: PGEHR

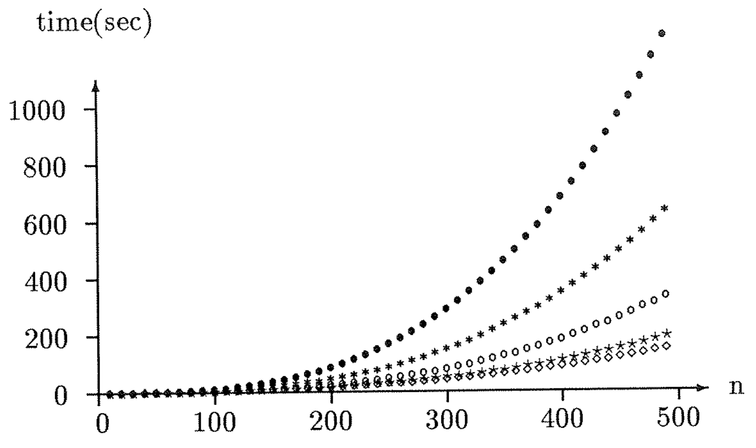

Figure 5.4: Observed timings attained during Reduction to Upper Hessenberg Form (key:: •: $p = 2$, $*$: $p = 4$, ∘: $p = 8$, $\star$: $p = 16$, $\diamond$: $p = 24$)

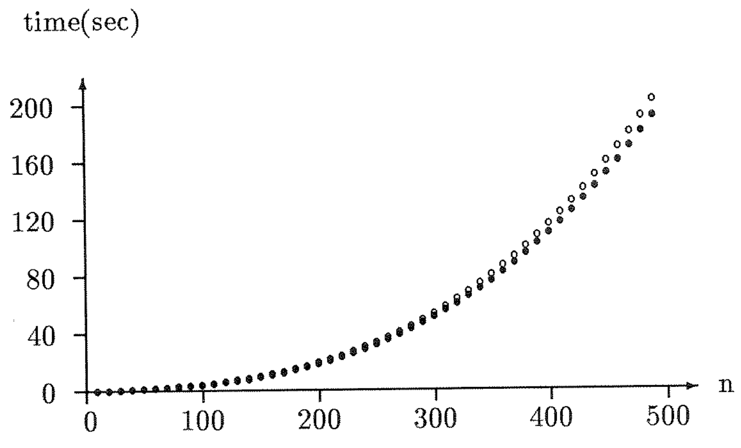

Figure 5.5: Observed and expected timings for $p = 16$ during Householder Reduction to Hessenberg Form (key:: •: observed, ∘: expected)

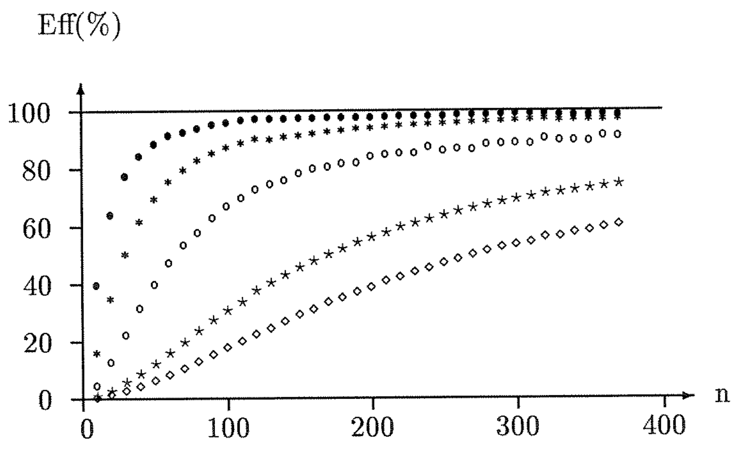

Figure 5.6: Observed efficiencies attained during Reduction to Upper Hessenberg Form (key:: •: $p = 2$, $*$: $p = 4$, $\circ$: $p = 8$, $\star$: $p = 16$, $\diamond$: $p = 24$)

Chapter 6

Conclusion

In this chapter, we will summarize the findings of Chapters 3 through 5, and discuss other applications and the future direction of this work.

6.1 Summary

In this thesis, we have seen several matrix algorithms and their parallel implementations. Using the column wrapped storage scheme and some of the communication primatives, which were described in Chapter 2, it was shown that reasonable efficiencies can be attained in all cases for large enough problem sizes.

The communication needs, time complexity and performance for each of these algorithms are listed in Table 6.1. The column headed by n/p indicates the ratio of the problem size to the number of processors at which 70% efficiency is achieved. This ratio gives an indication of how efficiently the algorithm utilizes its communications. The smaller this ratio is, the more quickly the algorithm achieves high efficiencies as n grows.

Four of the algorithms, PGEFA, PPOFA,PQRDC and PMGS, have similar communication needs and exploit the non-pipelined implementation of the broadcast primitive by masking communication with computation (Section 3.1.2). They differ mainly in the amount of parallel and sequential computations they require. They are all comparable (within a constant factor) with respect to how quickly they achieve 70% efficiency, since for each broadcast they all perform $O(mn)$ or $O(n^2)$ flops in parallel. The algorithm PTRSL, however, only performs $O(n)$ flops per communication and reaches higher efficiencies much more slowly.

The algorithm PGEHR, has more extensive communication needs. These include a distributed vector sum and total exchange which prohibit the pipelining of computations allowed by the broadcast primitive. It is expected that this algorithm

Algorithm	Communications	Time Complexity	n/p
PGEFA	broadcast	$\left(\frac{n^3}{3p} + n^2\right)\gamma + 2n\alpha + n^2\beta$	19
PPOFA	broadcast	$\left(\frac{n^3}{6p} + \frac{3n^2}{4}\right)\gamma + 2n\alpha + n^2\beta$	29
PTRSL	node-to-neighbor	$\left(\frac{n^2}{2p}\right)\gamma + (n-1)(\alpha + p\beta)$	100
PQRDC	broadcast	$\left(n^2m - \frac{n^3}{3}\right)\frac{\gamma}{p} + 3n\left(m - \frac{n}{2}\right)\gamma$ $+ 2n\left(\alpha + \left(m - \frac{n}{2}\right)\beta\right)$	10
PMGS	broadcast	$\left(\frac{n^2m}{p} + 4nm\right)\gamma + 2n(\alpha + m\beta)$	11
PGEHR	broadcast vector sum total exchange	$\left(\frac{5n^3}{3p} + \frac{n^2}{2}\right)\gamma + 3np\alpha + \frac{n^2p}{2}\beta$	14

Table 6.1: Algorithm Scorecard:
n/p = columns per processor at which 70% efficiency was achieved.

will perform better when implemented using the pipelined broadcast, as planned in future work. However, the algorithm still achieved high efficiencies without overlapping communication and computation.

It is important to note that there are limitations to the number of processors that can be used while still achieving high efficiencies for these algorithms. First, there is the practical consideration of the computation time exceeding what is considered reasonable. For example, with $p = 1000$, the expected completion time of PQRDC is on the order of 10^4 seconds for $n/p = 10$. Secondly, for each algorithm, as p approaches n the term resulting from sequential computation and the β term in the time complexity both become more significant. Until finally, when $p = n$, communication costs are on the same order as computation costs and efficiency suffers. Note, that for the column wrapped storage scheme, p cannot exceed n.

6.2 Other Applications and Future Work

The algorithms implemented in this thesis have used only some of the communication primitives of Chapter 2. Below are some other algorithms that employ these and other primitives.

The nonsymmetric QR algorithm with Hankel-wrapped storage uses the broadcast and one-way-shift operations [13, 16]. The two-sided Jacobi method for symmetric matrices, using the blocked Hankel-wrapped storage scheme, uses the total exchange and one-way-shift operations [4, 12, 13]. The one-sided Jacobi method for finding the singular value decomposition of a matrix is parallelized using the column blocked storage scheme and the one-way-shift operations [4].

The broadcast and global compare operations are required to implement the QR decomposition with column pivoting in parallel when using column wrapped storage. Conjugate gradient methods for sparse linear systems can be parallelized using the one-way-shift and inner product operations [14]. Finally, a parallel implementation of the power method to find eigenvalues requires the vector sum and inner product.

In general, the broadcast primitive is appropriate when performing an update (e.g., scaling or applying a transformation from the left) on the columns of a matrix, which are distributed over all processors. The node-to-neighbor communication is most likely used in situations with limited parallel potential or in handling exceptions. As we have seen in algorithm 5.2, the vector sum is required when performing a matrix vector multiplication, and it must be followed by the total exchange if the result is to be distributed to all processors. The one-way-shift is useful when processors must share border data. The global compare is needed when finding pivot elements or columns. The need for the inner product is obvious and we are still searching for an application of the data transpose.

It is our goal to increase the scope of these implementations to include the standard routines of packages like LINPACK and EISPACK and to extend the primitives to parallel computers with larger numbers (> 1000) of processors.

Appendix A

Parallel Matrix Subroutines in C

A.1 PGEFA

```
#include <math.h>
extern int np;
extern int myprocno;

/*
 *   PGEFA - drives each node in a ring of p processors to
 *   factor a square general matrix, A, with type FLT(double)
 *   entries by Gaussian elimination. The matrix is distributed
 *   among the processors in a column wrapped fashion.
 *
 *   ON ENTRY
 *     a      *FLT -  pointer to vector containing columns of
 *            the matrix A to be factored.
 *     n      int - order of the matrix
 *   ON RETURN
 *     a      columns of the LU decomposition where U is upper
 *            triangular and L is unit lower triangular whose
 *            elements are the multipliers used to obtain U.
 *     ipvt   *int - pointer to vector of pivot indices
 *     info   *int - = 0 normal value
 *                   = k if the (k)th pivot element was zero
 *
 *   JULY 20,1989 - James W. Juszczak,
 *                  University of Texas at Austin
 *
 *   SUBROUTINES AND FUNCTIONS
 *     blas - saxpy, sswap, isamax, scopy
 *     amcom - broadcast, broad_rec
 */
```

```
pgefa(n,a,ipvt,info)
FLT *a;
int n, *ipvt, *info;
{
int pk,                         /* processor that owns col k */
    kcols,      /* # cols of kth submatrix held by processor */
    h,tmp;
register int k, i, j;
register FLT *cp, *wp, temp;
FLT *work, *col, *akk, *akj;

*info = 0;
h = (myprocno < n%np) ? n/np+1: n/np;
work = a + h*n;
for (k = 0; k < n-1; k++) {
    wp = work;
    pk = k%np;
    kcols = h-((myprocno<pk)?k/np+1:k/np);
    if (myprocno == pk) {
        col = a + (k/np)*n ;                /* top of kth col */
        akk = col + k;
        ipvt[k] = k + isamax(n-k,akk,1);    /* find pivot row */
        cp = col + ipvt[k];
        temp = *cp;                                    /* swap */
        *cp = *akk;
        *akk = temp;
        if (temp == (FLT)0.0e0) {
            *work = -1.0;
            *info = k+1;
            broadcast(work,(n-k)*sizeof(FLT),1,0,1,0);
            break;
            }
        cp = akk + 1;
        *wp++ = (FLT)ipvt[k];
        for(i=k+1;i<n;i++)              /* compute multipliers */
            *wp++ = *cp++/temp;
        broadcast(work,(n-k)*sizeof(FLT),1,0,1,0);
        scopy(n-k-1,work+1,1,akk+1,1);
        sswap(kcols-1,akk+n,n,col+ipvt[k]+n,n);
        }
    else {
        broad_rec(work);
        if (*work == -1.0) {
            *info = k+1;
            ipvt[k] = k;
            }

        else {
            ipvt[k] = (int)(*wp++);
            if ((tmp = myprocno-pk)<0) tmp += np;
            col = (a + ((k+tmp)/np)*n);       /* 1st col > k */
            sswap(kcols,col+k,n,col+ipvt[k],n);
            }
        }
    if (*work == -1.0) break;      /* rest of column is zero */

    for (j = k+1; j < n; j++)       /* update next submatrix */
        if (myprocno == j%np) {
            col = (a+(j/np)*n);            /* top of jth col */
            akj = col + k;
            saxpy(n-k-1,-(*akj),work+1,1,akj+1,1);
            }
    }
}
```

A.2 PPOFA

```
#include <math.h>
#define ZERO (FLT)0.0e0
#define SQRT(x) (FLT)sqrt((FLT)(x))
extern int np, myprocno;
/*
 *   PPOFA - drives each node in a ring of p processors to
 *   factor a symmetric positive definite matrix, A, with
 *   type FLT(double) entries using the Cholesky algorithm.
 *   The matrix is distributed among the processors in a
 *   column wrapped fashion.
 *
 *   ON ENTRY
 *     a      *FLT -  pointer to vector containing columns
 *            of the matrix A to be factored. Only the lower
 *            triangle, including the diagonal is used.
 *     n      int - order of the matrix
 *   ON RETURN
 *     a      columns of L, the Cholesky triangle, in the
 *            lower triangle of the matrix A; the upper
 *            triangle of A remains unchanged. The
 *            factorization is not complete if info is not zero.
 *     info   *int - = 0 normal value
 *                   = k if the leading principle submatrix
 *                     of order k is found not to be positive
 *                     definite.
 *
 *   JULY 28,1989 - James W. Juszczak,
 *                  University of Texas at Austin
 *
 *   SUBROUTINES AND FUNCTIONS
 *     blas - saxpy, scopy
 *     amcom - broadcast, broad_rec
 */


ppofa(n,a,info)
FLT *a;
int n, *info;
```

```
{
int h, pk,                    /* pk: processor that owns col k */
    kcols;      /* # cols of kth submatrix held by processor */
register int k, i, j;
register FLT *cp, *wp, temp;
FLT *work, *col, *akk, *ajj;

*info = 0;
h = (myprocno < n%np) ? n/np+1: n/np;
work = a + h*n;
for (k = 0; k < n; k++) {
    wp = work;
    pk = k%np;
    kcols = h-((myprocno<pk)?k/np+1:k/np);
    if (myprocno == pk) {    /* processor that owns column k */
        col = a + (k/np)*n ;               /* top of kth col */
        akk = col + k;
        if ((temp = *akk) <= ZERO) {
            *work = (FLT)-1.0e0;
            *info = k+1;
            broadcast(work,(n-k)*sizeof(FLT),1,0,1,0);
            break;
            }
        *akk = SQRT(temp);
        temp = *akk;
        cp = akk + 1;
        *wp++ = *akk;
        for(i=k+1;i<n;i++)                      /* scale col k */
            *wp++ = *cp++/temp;
        broadcast(work,(n-k)*sizeof(FLT),1,0,1,0);
        scopy(n-k,work,1,akk,1);
        }
    else
        broad_rec(work);

    if (*work == (FLT)-1.0e0) {
        *info = k+1;
        return;
        }
    for (j = k+1; j < n; j++)             /* update submatrix */
        if (myprocno == j%np) {
            col = (a+(j/np)*n);             /* top of jth col */
            ajj = col + j;
            wp = work + (j-k);
            saxpy(n-j,-(*wp),wp,1,ajj,1);  /* update jth col */
            }
    }
}
```

A.3 PTRSL

```
#include <math.h>
#define ZERO (FLT)0.0e0
extern int np, myprocno;
/*
 *   PTRSL - drives each node in a ring of p processors
 *   to solve the upper triangular system of equations
 *   Ax=b by back substitution. The matrix is distributed
 *   among the processors in a column wrapped fashion.
 *
 *   ON ENTRY
 *     a      *FLT - is a pointer to a vector of type FLT
 *            which contains columns of the upper triangular
 *            matrix.
 *     n      int - order of the matrix
 *     t      *FLT - is a pointer to a vector of type FLT
 *            which contains the first (n-p+1) elements of
 *            b on  P(n) and 0 on all other processors.
 *     s      *FLT - is a pointer to a vector of type FLT
 *            which contains the last (p-1) elements of b
 *            on P(n) and 0 on all other processors.
 *   ON RETURN
 *     a      *FLT - is a pointer to a vector of type FLT
 *            which contains columns of the unchanged upper
 *            triangular matrix.
 *     x      *FLT - is a pointer to a vector of type FLT
 *            which contains the solution. The full solution
 *            vector is distributed among the processors such
 *            that x[j] is on the processor that owns column j
 *            of A, P(j).
 *     info   *int   = 0 normal value, A is non-singular
 *                   = k if the (k)th diagonal element of A
 *                     is zero
 *   NOVEMBER 2,1989 - James W. Juszczak,
 *                     University of Texas at Austin
 *   SUBROUTINES AND FUNCTIONS
 *     blas - saxpy
 *     amcom - send_left, rec_right
 */
```

```
ptrsl(n,a,x,t,s,info)
FLT *a,*x,*t,*s;
int n, *info;
{
int pj,                              /* processor that owns col j */
    tmp, size;
register int k, i, j;
register FLT xj, temp;
FLT *col, *ajj;

*info = 0;
size = np*sizeof(FLT);
k = (myprocno < n%np) ? n/np: n/np-1;
for (j = n-1; j >= 0; j--) {
    pj = j%np;
    if (myprocno == pj) {    /* processor that owns column j */
        col = a + (j/np)*n;                /* top of jth col */
        ajj = col + j;
        if (j != n-1) {
            rec_right(s);
            if (s[np-1] != 0.0) {  /* singularity detected ? */
                *info = (int)s[np-1];
                if (((myprocno != 0)?myprocno-1:np-1)!=(*info%np))
                    send_left(s,size);
                return;
                }
            }
        if ((xj = *ajj) != ZERO)
            temp = (j <= n-np) ? t[j] : 0.0;
        else {                         /* matrix is singular */
            *info = j;
            s[np-1] = (FLT)j;
            send_left(s,size);
            return;
            }
        xj = (s[0] + temp)/xj;
```

```
                                                    /* update s vector */
        if (j > 0) {
            for (i=0;i<np-2;i++) {
                if ((tmp = j-i-1) < 0) break;
                temp = (tmp < n-np+1) ? t[tmp] : 0.0;
                s[i] = s[i+1] - *(ajj-i-1) * xj + temp;
                }
            if (j >= (np-1))
                s[np-2] = - *(ajj-np+1)*xj + t[j-np+1];
            else
                s[np-2] = 0.0;
            }
        if (j != 0) send_left(s,size);      /* send s vector */
        saxpy(j-np+1,-xj,col,1,t,1);       /* update t vector */
        x[k--] = xj;
        }
    }
}
```

A.4 PQRDC

```
#include <math.h>
#define MIN(x,y) ((x) < (y) ? (x) : (y))
extern int np, myprocno;
/*
 *   PQRDC - drives each node in a ring of processors to form
 *   the orthogonal QR  decomposition of a general rectangular
 *   matrix, A, by applying Householder transformations.
 *   The matrix is distributed among the processors
 *   in a column wrapped fashion.
 *
 *   ON ENTRY
 *     a      *FLT -  pointer to a vector containing columns of
 *            the matrix A. Each column of A has been extended
 *            to a length of m+2 to store further information
 *            required to recover the Householder transformation
 *            at each step.
 *     m      int - number of rows in the matrix
 *     n      int - number of columns in the matrix
 *   ON RETURN
 *     a      *FLT -  pointer to a vector containing columns of
 *            the matrix A which has been overwritten by the
 *            triangular matrix R, on and above the diagonal.
 *            The vectors u which define the transformations,
 *            Uk = I -beta*uuT, are stored below the first
 *            subdiagonal. Beta and the first element of u are
 *            stored in A[m+1][k] and A[m+2][k] respectively.
 *            The orthogonal matrix Q = U1*U2* ... *Ur can be
 *            formed from these vectors.
 *     info   *int   = 0 normal value
 *                   = k if the (k)th column was already reduced
 *
 *   NOVEMBER 7,1989 - James W. Juszczak,
 *                     University of Texas at Austin
 *
 *   SUBROUTINES AND FUNCTIONS
 *     blas - saxpy, snrm2, sdot
 *     amcom - broadcast, broad_rec
 */
```

```
pqrdc(a,m,n,info)
FLT *a;
int m, n, *info;
{
register FLT beta, *col, *u, *work, n2, *akk;
FLT snrm2(), sdot();
register int j, k, r;

if (m<=0 || n<=0) return;
r = MIN(m-1,n);
*info = 0;
work = (FLT *) malloc((m+1)*sizeof(FLT));

for (k = 0; k < r; k++) {
               /* DETERMINE THE TRANSFORMATION, U=I-uuT*beta */
   if (myprocno == k%np) {
      *work = 0.0;
      col = a + (k/np)*(m+2);              /* top of kth col */
      akk = col+k;
      n2 = snrm2(m-k,akk,1);
      if (n2 == (FLT)0.0) {
         *work = (FLT)-1.0;
         *info = k;
         broadcast(work,(m-k+1)*sizeof(FLT),1,0,1,0);
         continue;
         }
      if (*akk < 0.0) n2 = -n2;
      *akk += n2;
      *(col+m) = beta = 1.0/(n2*(*akk));
      u = akk;
      broadcast(u,(m-k+1)*sizeof(FLT),1,0,1,0);
      *(col+m+1) = *u;
      }
   else {
      broad_rec(work);
      if (*work == (FLT)-1.0) {    /* rest of column is zero */
         *info = k;
         continue;
         }
      else {
         u = work;
         beta = *(u+m-k);
         }
      }

            /* PREMULTIPLY, A <- UA  or aj <- aj - u(uT)aj*beta */
   for (j = k+1; j < n; j++) { register FLT *akj, temp;
      if (myprocno == j%np) {
         col = (a+(j/np)*(m+2));              /* top of jth col */
         akj = col+k;
         temp = sdot(m-k,akj,1,u,1)*beta; /* temp = (uT)aj*beta */
         saxpy(m-k,-temp,u,1,akj,1);       /* aj <- aj - temp*u */
         }
      }
   if (myprocno == k%np) *akk = -n2;
   }
free(work);
}
```

A.5 PMGS

```
#include <math.h>
extern int np, myprocno;


/*
 *   PMGS - drives each node in a ring of processors to form
 *   an orthonormal basis which spans the column space of A,
 *   a general rectangular matrix with linearly independent
 *   columns. A is factored into the product QR, where Q has
 *   orthonormal columns and R is an upper triangular matrix
 *   with positive diagonal elements. A is overwritten by Q.
 *   The matrix is distributed among the processors in a column
 *   wrapped fashion.
 *
 *   ON ENTRY
 *     a       *FLT -  pointer to a vector containing columns
 *             of the matrix A.
 *     m       int - number of rows in the matrix
 *     n       int - number of columns in the matrix
 *   ON RETURN
 *     a       *FLT -  pointer to a vector containing columns of
 *             the matrix A which has been overwritten by Q, a
 *             matrix with orthonormal columns.
 *     r       *FLT -  pointer to vector containing the n x n upper
 *             triangular matrix R distributed to the processors in
 *             column wrapped fashion.
 *     info    *int   = 0 normal value
 *                    = k if the (k)th column was linearly dependent
 *
 *   NOVEMBER 12,1989 - James W. Juszczak,
 *                      University of Texas at Austin
 *
 *   SUBROUTINES AND FUNCTIONS
 *     blas - sscal, saxpy, snrm2, sdot
 *     amcom - broadcast, broad_rec
 */
```

```
pmgs(m,n,a,r,info)
FLT *a, *r;
register int m, n;
int *info;
{
register FLT *aj, *ak, *qk, temp, *rkj, *work;
FLT snrm2(), sdot();
register int j, k;

if (m<=0 || n<=0) return;
*info = 0;
work = (FLT *) malloc(m*sizeof(FLT));
ak = a;                           /* points to top of kth col of A */

for (k = 0; k < n; k++) {
   if (myprocno == k%np) {
      *work = (FLT)0.0;
      *(r+k) = temp = snrm2(m,ak,1);
      if (temp == (FLT)0.0) {
         *work = (FLT)-1.0;
         broadcast(work,sizeof(FLT),1,0,1,0);
         *info = k;
         return;
         }
      sscal(m,(FLT)1.0/temp,ak,1);
      qk = ak;
      broadcast(qk,m*sizeof(FLT),1,0,1,0);
      r += n;
      ak += m;
      }
   else {
      broad_rec(work);
      if (*work == (FLT)-1.0) {
         *info = k;
         return;
         }
      else
         qk = work;
      }
   rkj = r+k;
   for (j = k+1; j < n; j++) {
      if (myprocno == j%np) {
         aj = a + (j/np)*m;
         *rkj = temp = sdot(m,aj,1,qk,1);
         saxpy(m,-temp,qk,1,aj,1);
         rkj += n;
         }
      }
   }
free(work);
}
```

A.6 PGEHR

```
#include <math.h>
extern int np, myprocno;

/*
 *   PGEHR - drives each node in a ring of p processors to
 *   reduce a square general matrix, A, to upper Hessenberg
 *   form by determining and applying Householder unitary
 *   similarity transformations. The matrix is distributed
 *   among the processors in a column wrapped fashion.
 *
 *   ON ENTRY
 *     a      *FLT -  pointer to vector containing columns of
 *            the matrix A. Each column of A has been extended
 *            to a length of n+2 to store further information
 *            required to recover the Householder transformation
 *            at each step.
 *     n      int - order of the matrix
 *   ON RETURN
 *     a      *FLT -  pointer to vector containing
 *            columns of the matrix A which has been overwritten
 *            by the upper Hessenberg matrix above the second
 *            sub-diagonal. The vectors u which define the
 *            transformations, Uk = I -beta*uuT, are stored
 *            below the first subdiagonal. Beta and the first
 *            element of u are stored in A[n+1][k] and A[n+2][k]
 *            respectively.
 *     info   *int   = 0 normal value
 *                   = k if the (k)th column was already reduced
 *
 *   NOVEMBER 2,1989 - James W. Juszczak,
 *                     University of Texas at Austin
 *
 *   SUBROUTINES AND FUNCTIONS
 *     blas - saxpy, snrm2, sdot, scopy
 *     amcom - broadcast, broad_rec, dvec_add, exchange
 */
```

```
pgehr(a,n,info)
FLT *a;
int n, *info;
{
register FLT beta, *col, *u, *work, *x, *y, n2, *ak1k;
FLT snrm2(), sdot(), ZERO[1];
register int j, k, tmp;

*ZERO=(FLT)0.0;
*info = 0;
work = (FLT *) malloc((3*n+np)*sizeof(FLT));
x = work + n;
y = x + n;
for (k = 0; k < n-2; k++) {
                        /* DETERMINE THE TRANSFORMATION, U=I-uuT*beta */
    if (myprocno == k%np) {
    register FLT temp;
       col = a + (k/np)*(n+2);                     /* top of kth col */
       ak1k = col+k+1;
       n2 = snrm2(n-k-1,ak1k,1);
       if (n2 == (FLT)0.0) {
          *work = (FLT)-1.0;
          *info = k;
          broadcast(work,(n-k)*sizeof(FLT),1,0,1,0);
          continue;
          }
       if (*ak1k < (FLT)0.0) n2 = -n2;
       *ak1k += n2;
       *(col+n) = beta = 1.0/(n2*(*ak1k));
       u = ak1k;
       broadcast(u,(n-k)*sizeof(FLT),1,0,1,0);
       *(col+n+1) = *u;
       }
    else {
       broad_rec(work);
       if (*work == (FLT)-1.0) {          /* rest of column is zero */
          *info = k;
          continue;
          }
```

```
      else {
         u = work;
         beta = (*(u+n-k-1));
         }
      }
                /* PREMULTIPLY, A <- UA  or aj <- aj - u(uT)aj*beta */
   for (j = k+1; j < n; j++) { register FLT *ak1j, temp;
       if (myprocno == j%np) {
         col = (a+(j/np)*(n+2));                  /* top of jth col */
         ak1j = col+k+1;
         temp = sdot(n-k-1,ak1j,1,u,1)*beta;/* temp = (uT)aj*beta */
         saxpy(n-k-1,-temp,u,1,ak1j,1);      /* aj <- aj - temp*u */
         }
      }
              /* POSTMULTIPLY, A <- AU or aj <- aj - aj*u(ut)*beta) */
   scopy(n,ZERO,0,x,1);
   for (j = k+1; j < n; j++) { register FLT ujk;
      if (myprocno == j%np) {
         col = (a+(j/np)*(n+2));                  /* top of jth col */
         ujk = *(u+j-k-1);
         saxpy(n,ujk,col,1,x,1); /* x<-x+aj*u[j] for each aj on P */
         }
      }
   scopy(n,ZERO,0,y,1);                 /* y <- sum x over all P's */
   tmp = n/np;
   if (n%np != 0) tmp++;
   dvec_add(x,n,tmp*myprocno+y);
   exchange(y,tmp*sizeof(FLT));
                                      /* complete submatrix update  */
   for (j = k+1; j < n; j++) { register FLT temp;
      if (myprocno == j%np) {
         col = (a+(j/np)*(n+2));                  /* top of jth col */
         temp = *(u+j-k-1) * beta;
         saxpy(n,-temp,y,1,col,1);          /* aj <- aj - u[j]*y */
         }
      }
   if (myprocno == k%np) *ak1k = -n2;
   }                                   /* end outermost FOR loop */
free(work);
}
```

BIBLIOGRAPHY

[1] Dongarra, J.J., Moler, C.B., Bunch, J.R., and Stewart, G.W., *LINPACK User's Guide*, SIAM, Philadelphia, 1979

[2] Golub, G.H. and Van Loan, C.F., *Matrix Computations*, Johns Hopkins Press, 1983

[3] Ipsen, I.C.F., Saad, Y., and Schultz, M. H., "Complexity of Dense-Linear-System Solution on a Multiprocessor Ring," *Linear Algebra and its Applications*, 77:205-239, 1986

[4] Juszczak, J.W. and van de Geijn, R.A., "An Experiment in Coding Portable Parallel Matrix Algorithms," *Proceedings of the Fourth Annual Hypercube Conference*, to appear, 1989

[5] Li, G. and Coleman, T.F., "A Parallel Triangular Solver for a Distributed-Memory Multiprocessor," *Siam J. Sci. Stat. Comput.*, Vol. 9, No. 3, May 1988

[6] Li, G. and Coleman, T.F., "A New Method for Solving Triangular Systems on Distributed-Memory Message-Passing Multiprocessors," *Siam J. Sci. Stat. Comput.*, Vol. 10, No. 2, March 1989

[7] McBryan, O.A. and Van de Velde, E.F., "Hypercube Algorithms and Implementations," *SIAM J. Sci. Stat. Comp.*, Vol. 8, No. 2, March 1987

[8] O'Leary, D.P. and Stewart, G.W., "Data-Flow Algorithms for Parallel Matrix Computations," *Communications of the ACM*, Vol. 28, No. 8, August 1985

[9] Saad, Y. and Schultz, M.H., "Data Communication in Parallel Architectures," Yale University, Research Report YALEU/DCS/RR-461, 1986

[10] Seitz, C.L., Seizovic, J., and Su, W., "The C Programmer's Abbreviated Guide to Multiprocessor Programming," Caltech Computer Science Technical Report Caltech-CS-TR-88-1, 1988.

[11] Stewart, G.W., *Introduction to Matrix Computations*, Academic Press, 1973

[12] Van de Geijn, R. A., "A Novel Storage Scheme for Parallel Jacobi Methods," The University of Texas at Austin, Dept. of Computer Sciences, TR-88-26

[13] Van de Geijn, R. A., "Storage Schemes for Parallel Eigenvalue Algorithms," to appear in *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms,* G. Golub and P. Van Dooren (Ed.), NATO ASI Series, Springer Verlag, 1989

[14] Van de Geijn, R. A., "Machine Independent Parallel Numerical Algorithms," in *Parallel Supercomputing: Methods, Algorithms and Applications*, Graham F. Carey (Ed.), Wiley, 1989

[15] Van de Geijn, R. A., "Performance Evaluation of the AMETEK 2010: A Preliminary Report," unpublished manuscript

[16] Van de Geijn, R. A. and D.G. Hudson III, "Efficient Parallel Implementation of the Nonsymmetric QR Algorithm," in *Proceedings of the Fourth Annual Hypercube Conference*, to appear, 1989