[9] T.H. Dunigan. Performance of the Intel iPSC/860 and NCUBE 6400 hypercubes. Technical Report ORNL/TM-11790, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1991.

[10] G.A. Geist and G.J. Davis. Finding eigenvalues and eigenvectors of unsymmetric matrices using a distributed-memory multiprocessor. *Parallel Computing*, 13:199–209, 1990.

[11] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, second edition, 1989.

[12] I.C.F. Ipsen, Y. Saad, and M.H. Schultz. Complexity of dense-linear-system solution on a multiprocessor ring. *Lin. Alg. Appl.*, 77:205–239, 1986.

[13] G.W. Juszczak. Efficient portable parallel matrix computations. Master's thesis, University of Texas at Austin, 1989. Technical Report TR-89-38.

[14] J.W. Juszczak and R.A. van de Geijn. An experiment in coding portable parallel matrix algorithms. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, 1989.

[15] S.L. Lillevik. The Touchstone 30 Gigaflop DELTA Prototype. In *Sixth Distributed Memory Computing Conference Proceedings*, pages 671–677. IEEE Computer Society Press, 1991.

[16] Y. Saad and M.H. Schultz. Data communciation in parallel architectures. Research Report YALEU/DCS/RR-461, Yale University, 1986.

[17] R.A. van de Geijn. Machine independent parallel numerical algorithms. In G.F. Carey, editor, *Parallel Supercomputing: Methods, Algorithms and Applications*, chapter 3, pages 33–44. Wiley, 1989.

[18] R.A. van de Geijn. Efficient global combine operations. In *Sixth Distributed Memory Computing Conference Proceedings*, pages 291–294. IEEE Computer Society Press, 1991.

[19] R.A. van de Geijn. Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems. Computer Science report TR-91-28, Univ. of Texas, 1991.

[20] R.A. van de Geijn. On global combine operations. LAPACK Working Note 29, Technical Report CS-91-129, University of Tennessee, 1991.

We have started to investigate different methods for mapping matrices to nodes. In [19], we show that wrapping onto logical tori greatly improves the performance of the LU factorization on the Intel Touchstone Delta. Future work will include the investigation of using this storage method for the reduction algorithms.

Acknowledgements

We would like to thank Al Geist for commenting on an early draft of this paper. Special thanks are given to the members of the Concurrent Supercomputing Consortium for their cooperation and contribution of time and access to the Touchstone Delta System.

References

[1] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. Basic Linear Algebra Communication Subprograms. In *Sixth Distributed Memory Computing Conference Proceedings*, pages 287–290. IEEE Computer Society Press, 1991.

[2] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. LAPACK for distributed memory architectures: Progress report. In *Proceedings Fifth SIAM Conference on Parallel Processing for Scientific Computing*, Philadelphia, 1991. SIAM.

[3] H.Y. Chang, S. Utku, M. Salama, and D. Rapp. A parallel Householder tridiagonalization strategem using scattered row decomposition. *Intl. J. Num. Meth. Eng.*, 26:857–873, 1987.

[4] H.Y. Chang, S. Utku, M. Salama, and D. Rapp. A parallel Householder tridiagonalization strategem using scattered square decomposition. *Parallel Computing*, 6:297–312, 1988.

[5] Jack Dongarra and Susan Ostrouchov. LAPACK block factorization algorithms on the Intel iPSC/860. LAPACK Working Note 24, Technical Report CS-90-115, University of Tennessee, Oct. 1990.

[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[7] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[8] Jack J. Dongarra, Sven J. Hammarling, and Danny C. Sorensen. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27, 1989.

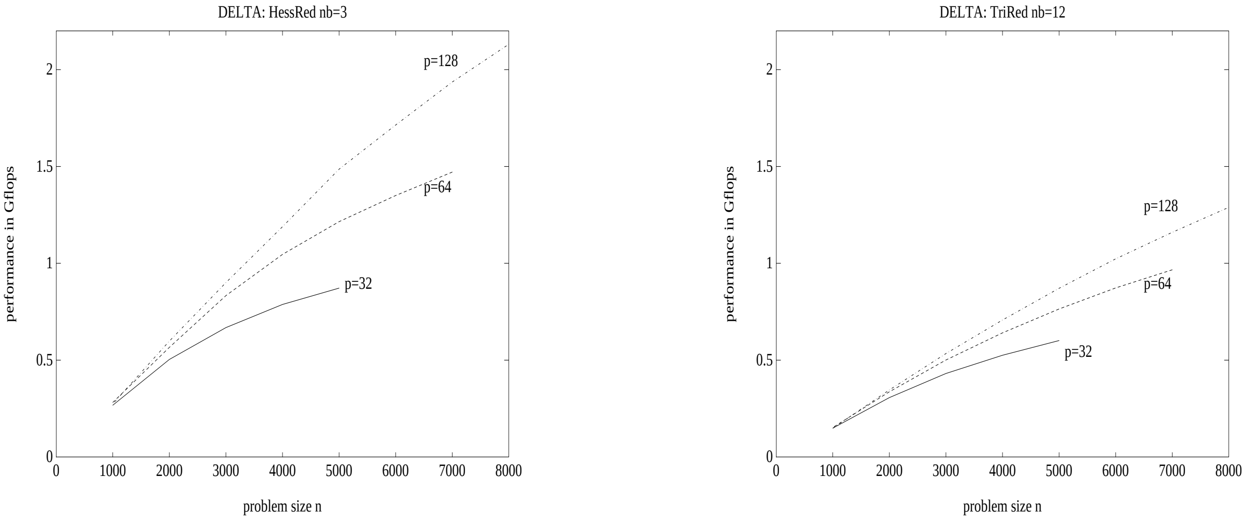

Figure 3: GFLOPS atained for various numbers of nodes when the problem size is varied. For the Hessenberg reduction, $nb = 3$, for the tridiagonal reduction, $nb = 12$.

5.2 Reduction to Tridiagonal Form

Figure 1 also shows the execution time for the parallel reduction to tridiagonal form. From this graph, it can be concluded that large block sizes yield better performance. This is due to the fact that during the update given by Equation 2 the submatrix must be updated one panel at a time, since the lower triangular part of the matrix A is wrapped onto the processors. For the same reason, the performance of the matrix-vector product (BLAS2) is affected.

The overall performance of the reduction to tridiagonal form is worse than that of the reduction to Hessenberg form (Figure 3). This can be explained as follows: The number of floating point operations is reduced by a factor 2.5 as compared to the reduction to Hessenberg form. The time spent in the broadcast is unchanged. The time spent in the global summation is approximately halved. As a result, the ratio of communication to computation is higher than for the reduction to Hessenberg form.

6 Conclusion

We have demonstrated that the LAPACK code for reducing a matrix to Hessenberg or tridiagonal form can be rewritten for current generation MIMD distributed memory computers in a relatively straight forward manner.

On the Intel Touchstone Delta, efficiency is hampered to a large degree by the cost of communication and the synchronous nature of the algorithm. If larger problems are solved, this becomes less significant. Although the Intel Touchstone Delta system has sufficient memory to store matrices of order 25000, we limited ourselves to problems that required less than 30 minutes to complete.

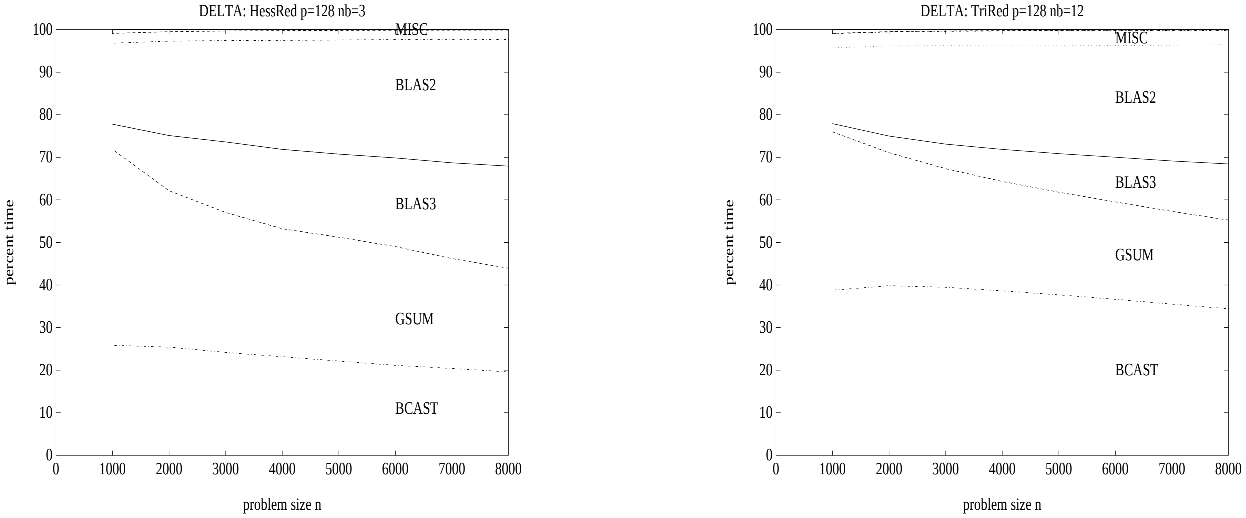

Figure 2: Allocation of execution time when $p = 128$, $nb = 3$ and the problem size n is varied. Again, the space between two curves equals the time spent in the indicated operation.

Gigaflops double precision, ≈ 40 Gigaflops single precision, and an aggregate system memory of ≈ 8 Gigabytes. The interconnection network employs a Mesh Routing Chip (MRC), developed at the California Institute of Technology, at each system node. Each MRC provides five channels, one for its associated node and four for its adjacent neighbors in the two-dimensional mesh. The channels are comprised of two, unidirectional buses: one for data flow into the MRC, one for data flow out of the MRC. The peak interprocessor communications bandwidth is ≈ 30 MBytes/s in each direction. The system supports explicit message-passing, with a latency of ≈ 75 microseconds via worm-hole routing using a packet-based protocol. Interconnect blocking is minimized by interleaving packets associated with distinct messages which need to traverse the same interconnect path.

5.1 Reduction to Hessenberg Form

Figure 1 shows the performance of the parallel reduction to Hessenberg form as a function of the problem size `n` and the block size `nb` for $p = 128$. Performance is most influenced by the performance of the Level 2 and 3 BLAS. From this graph, it can be concluded that $nb = 3$ yields reasonable performance. We will use this block size in subsequent discussions.

Communication overhead is the main contributor to the reduction in performance, as can be seen from Figures 1 and 2. In particular, the global summation and broadcast operations are major contributors to the total execution time. This is not supprising, considering a broadcast of a vector of length $O(n)$ and global summation of vectors of length n is required for each column of W that is formed (in addition to the summation of at least one smaller vector).

The performance attained as a function of problem size is clear from Figure 3. In this graph, $nb = 3$ and performance is given for various numbers of nodes. The overall performance is somewhat disappointing: The LAPACK reduction routine on a single processor attains about 45 MFLOPS.

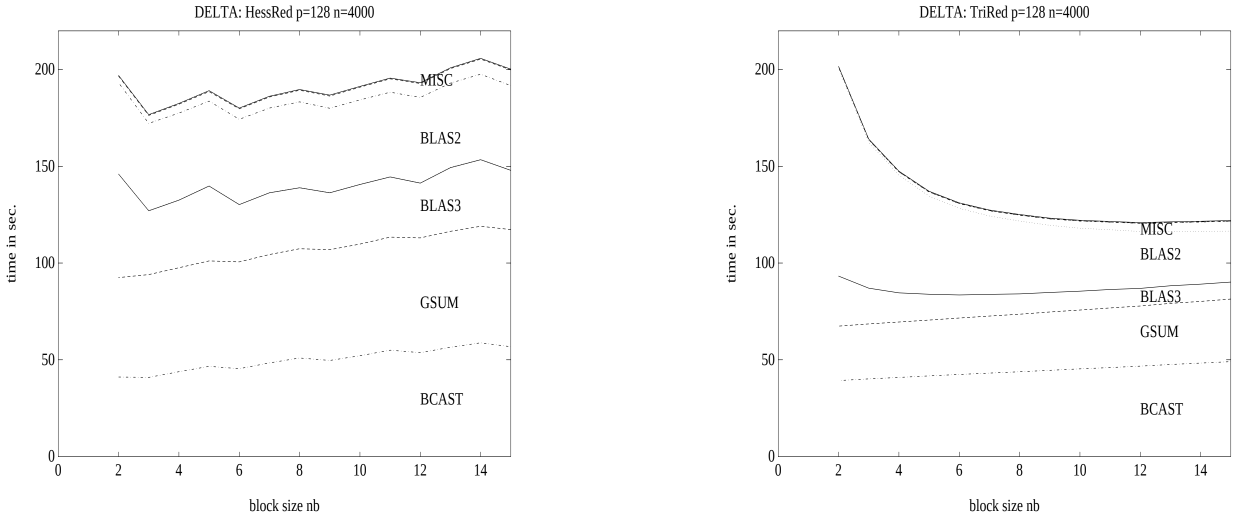

Figure 1: Total computation time for 128 nodes when $n = 4000$ and the block size n_b is varied. The space between two curves equals the time spent in the indicated operation. The times for the global sum (GSUM) and broadcast (BCAST) include some idle time that is due to load imbalance.

4.4 Parallel Implementation: Blocked Tridiagonal Reduction

The parallel implementation of the reduction to tridiagonal form for symmetric A proceeds similarly. Consider the steps given in Section 4.3: In Step 1, $[W]_{*,1:j} = [V]_{*,1:j}$; In Step 3, $z = x$, which can be either formed separately on all nodes or distributed among the nodes, which requires a global summation; Step 4-6 are merged, where $[W]_{*,j+1} = [V]_{*,j+1}$ is computed by

$$\begin{aligned} y &= \beta(A^{(k)}[U]_{*,j+1} - [V]_{*,1:j}x - [U]_{*,1:j}x) \\ [V]_{*,j+1} &= y - 1/2\beta([U]^{\mathrm{T}}_{*,j+1}y)[U]_{*,j+1} \end{aligned}$$

where $\beta A^{(k)}[U]_{*,j+1}$ is computed using the same trick as in Section 3.4.

5 Experiments

In this section, we report the performance of the parallel reduction algorithms on the Intel Touchstone Delta system using the Portland Group compiler and assembly coded *single precision* BLAS routines by Kuck and Associates.

The Intel Touchstone Delta system is a distributed-memory, message-passing multicomputer of the Multiple Instruction Multiple Data (MIMD) class developed jointly by the Defense Advanced Research Projects Agency (DARPA) and the Intel Corporation [15]. It is comprised of 520 i860-based nodes, each having 16 Megabytes (MBytes) of memory, interconnected via a communications network having the topology of a two-dimensional rectangular grid. (Scaling is not restricted to a power-of-two increment typical of hypercube topologies.) It has a peak performance of $\approx$ 32

Finally, we examine how the computation of U, V, and W can be distributed among the nodes. Assume the computation has progressed to where panel s is being reduced, i.e., $k = (s-1)m+1$. Assume the first j columns of U, V, and W have been computed and are distributed as desired. The computation of the $(j+1)$st column of these matrices proceeds as follows:

1. On node $\mathbf{P}_{(s-1)\mathrm{mod} p}$, form the $(j+1)$st column of the current panel of $A^{(k+j)}$:

$$[A_s^{(k+j)}]_{*,j+1} = [A]_{*,k+j}^{(k+j)} = [A]_{*,k+j}^{k} - [U]_{*,1:j}[V]_{k+j,1:j}^{\mathrm{T}} - [W]_{*,1:j}[U]_{k+j,1:j}^{\mathrm{T}}$$

   Since

$$[V]_{k+j,1:j} = [V_s]_{j+1,1:j}$$

   all information for this operation is available on this node.

2. On $\mathbf{P}_{(s-1)\mathrm{mod} p}$, compute $([U]_{*,j+1}, \beta)$ and distribute to all nodes.

3. Next, we must form three intermediate results,n

$$\begin{array}{rcl} x & = & [V]_{*,1:j}^{\mathrm{T}}[U]_{*,j+1} \\ y & = & [U]_{*,1:j}^{\mathrm{T}}[U]_{*,j+1} \\ z & = & [W]_{*,1:j}^{\mathrm{T}}[U]_{*,j+1} \end{array}$$

   The first requires partial sums of vectors to be accumulated on each processor, followed by a global summation of the results, leaving the results on all processors. The latter two can either be computed in the same way or they can be computed separately on each processor, leading to redundant computation, but less communication overhead.

4. Assuming x, y, and z have been computed,

$$[V]_{*,j+1} = A^{(k)\mathrm{T}}[U]_{*,j+1} - [V]_{*,1:j}x - [U]_{*,1:j}z$$

   can be computed, leaving the resulting column distributed among the nodes.

5. Computing $W_{*,j+1}$ requires

$$w = A^{(k)}[U]_{*,j+1} - [U]_{*,1:j}x - [W]_{*,1:j}y$$

   to be computed. Just like the computation of w in Algorithm 3, this proceeds in two stages: columns of $A^{(k)}$ on each of the processors are summed after being multiplied by appropriate elements of $[U]_{*,j+1}$. Next, each of the vectors $[U]_{*,1:j}x$ and $[W]_{*,1:j}y$ is partitioned into p approximately equal subvectors and computation of each subvector is assigned to a node. After each node computes its section of these two vectors, and subtracts them from the partial sum of columns, a global summation computes the desired w, leaving the result on all nodes.

6. Finally,

$$[W]_{*,j+1} = w - \beta(w^{\mathrm{T}}[U]_{*,j+1})[U]_{*,j+1}$$

   is formed on all nodes.

where

$$
\begin{aligned}
([U]_{*,j+1}, \beta) &= H^{(k+j)}([A^{(k+j)}]_{*,k+j}) \\
[V]_{*,j+1} &= A^{(k+j)\mathrm{T}}[U]_{*,j+1} = (A^{(k)} - [U]_{*,1:j}[V]^{\mathrm{T}}_{*,1:j} - [W]_{*,1:j}[U]^{\mathrm{T}}_{*,1:j})^{\mathrm{T}}[U]_{*,j+1} \\
w &= A^{(k+j)}[U]_{*,j+1} = (A^{(k)} - [U]_{*,1:j}[V]^{\mathrm{T}}_{*,1:j} - [W]_{*,1:j}[U]^{\mathrm{T}}_{*,1:j})[U]_{*,j+1} \\
[W]_{*,j+1} &= w - \beta(w^{\mathrm{T}}[U]_{*,j+1})[U]_{*,j+1}
\end{aligned}
$$

The general strategy for reorganizing Algorithm 2 now becomes:

1. Partition the matrix into panels of width m.
2. For $k = 1$, compute matrices U, V, and W by computing the successive Householder transformations. (Notice that for given j, in order to compute u, only the $(k + j)$th column of $A^{(k+j)}$ needs to be formed.)
3. Update $A^{(k+m)} = A^{(k)} - UV^{\mathrm{T}} - WU^{\mathrm{T}}$. (Note: only columns $k+m, \ldots, n$ need to be updated, since columns $k, \ldots, k + m - 1$ were updated during the computation of U, V, and W.)
4. Repeat for $k = m + 1, 2m + 1, \ldots$.

Notice that the third step can now be written as two matrix-matrix operations. The bulk of the formation of the matrices requires m matrix-vector operations.

4.2 Sequential Implementation: Blocked Tridiagonal Reduction

The blocked algorithm for the reduction to tridiagonal form for the symmetric problem is reorganized similarly, except that in this case $W = V$, so Equation 2 becomes

$$H^{(k+m)} \cdots H^{(k)} A^{(k)} H^{(k)} \cdots H^{(k+m)} = A^{(k)} - UV^{\mathrm{T}} - VU^{\mathrm{T}}$$

and only the lower triangular portion of A is updated.

4.3 Parallel Implementation: Blocked Hessenberg Reduction

We now describe the parallel implementation of the blocked reduction to Hessenberg form. We will use panel-wrapped storage, where the panel width corresponds to m, the width of the panel used for the sequential blocked algorithm.

Understanding how to perform the computation in parallel is closely related to how matrices U, V, and W must be distributed in order to be able to perform the update in Equation 2. Partition V^{T} like $A^{(k)}$:

$$V^{\mathrm{T}} = \left(V_1^{\mathrm{T}} \; V_2^{\mathrm{T}} \; \cdots \; V_r^{\mathrm{T}} \right)$$

If we update $A_j^{(k)}$ on node $\mathbf{P}_{(j-1)\mathrm{mod}p}$, then U, W and V_j must be known to this node. Hence we must compute these matrices in such a way that U and W eventually reside on all nodes, while V^{T} is panel-wrapped distributed among the nodes.

$$
\begin{array}{ll}
\texttt{do } j = k, n, & \textit{(16)} \\
\quad \texttt{if } j \in \mathbf{P}_i \texttt{ then update } [A]_{*,j} = [A]_{*,j} - \beta\nu_j u - \beta\upsilon_j w & \textit{(17)} \\
\texttt{enddo} \quad \textit{(18)} \texttt{ enddo} & \textit{(19)}
\end{array}
$$

Statement *(14)* indicates that y is the result the global summation of vectors y_i. A minor redundancy exists since all processors compute w once y has been computed. This can be overcome by replacing statements *(14)* and *(15)* by

$$
\begin{array}{lll}
y_i = y_i - \beta(u^{\mathrm{T}}y)u & \text{(part of length} \approx (n-j)/p) & \textit{(14)} \\
\texttt{gsum } w = y_i & & \textit{(15)}
\end{array}
$$

so that all processors participate in subtracting $\beta(u^{\mathrm{T}}y)u$ before the global summation.

3.4 Parallel Implementation: Tridiagonal Reduction

Parallel implementation of the reduction to tridiagonal form for a symmetric A proceeds similarly, with one major difference: Since only the lower triangular part of matrix A contains useful information, we compute y as follows: Let $A = L + R$, where L and R equal the lower triangular and strictly upper triangular parts of A, respectively. Notice that R^{T} equals the strictly lower triangular portion of L, and hence both are assigned to nodes in column-wrapped fashion. Now $y = Au = Lu + Ru$ can be computed by:

$$
\begin{array}{l}
y_i = 0 \\
\texttt{do } j = k, n \\
\quad \texttt{if } j \in \mathbf{P}_i \texttt{ then} \\
\qquad \eta_j = \eta_j + u^{\mathrm{T}}[L]_{*,k} \\
\qquad y_i = y_i + \upsilon_j [R]_{j,*}^{\mathrm{T}} (= y_i + \upsilon_j [L]_{*,j}) \\
\texttt{enddo} \\
y_i = y_i - \beta(u^{\mathrm{T}}y)u \qquad \text{(part of length} \approx (n-j)/p) \\
\texttt{gsum } y = \sum y_i
\end{array}
$$

4 Blocked Algorithms

In [8] it is shown how reorganizing portions of the above algorithms in terms of Level 3 BLAS yields algorithms that perform considerably better on computers with vector processors and/or hierarchical memories. In this section we discuss sequential blocked algorithms for reduction to Hessenberg and tridiagonal form as well as their parallel implementation.

4.1 Sequential Implementation: Blocked Hessenberg Reduction

We first consider how the application of m Householder transformations can be combined:

$$
H^{(k+m)} \cdots H^{(k)} A^{(k)} H^{(k)} \cdots H^{(k+m)} = A^{(k)} - UV^{\mathrm{T}} - WU^{\mathrm{T}} \tag{2}
$$

3.3 Parallel Implementation: Hessenberg Reduction

Given p processing nodes $\mathbf{P}_0, \ldots, \mathbf{P}_{p-1}$, our parallel implementation will assume that the columns of A have been assigned to the nodes in column-wrapped fashion.

This choice of assignment allows us to parallelize Algorithm 2 as follows:

1. For all k, updating of column j of matrix A is performed by node $\mathbf{P}_{(j-1)\mathrm{mod} p}$.

2. During the kth iteration, the computation of (u, β) is performed by $\mathbf{P}_i$ such that $k \in \mathbf{P}_i$, , i.e., $\mathbf{P}_{(k-1)\mathrm{mod} p}$, after which it is distributed to all nodes.

3. Subtracting the jth column of $\beta u v^{\mathrm{T}}$ from column j requires only jth element of v, ν_j, to be known to the node that owns column j. This is convenient, since $\nu_j = u^{\mathrm{T}}[A]_{*,j}$, which can be formed by this node once u has been received. This means v can be computed in parallel, leaving the different elements of v on the nodes that computed them.

4. Subtracting the jth column of $\beta w u^{\mathrm{T}}$ from column j requires both v_j and $w = Au$ to be known to node $\mathbf{P}_{(j-1)\mathrm{mod} p}$. Vector $w \in \mathbf{R}^n$ is computed as follows: Let B_i equal the columns of A that are assigned to node $\mathbf{P}_i$. If the corresponding elements of u are appropriately packed into a vector u_i^*, then $Au = \sum_{\mathrm{all nodes}} y_i$, where $y_i = B_i u_i^*$. Hence Au can be formed by first computing partial results y_i in parallel on all nodes, followed by a global summation of the partial results, leaving Au on all nodes. Next, $u^{\mathrm{T}} A u = u^{\mathrm{T}} y$ and w can be formed. Notice that there is some (insignificant) redundant computation in this last step, since all nodes perform the same computation.

The resulting parallel implementation of Algorithm 2 is given by the following pseudo-code that drives each node $\mathbf{P}_i$:

Algorithm 3 *Parallel Hessenberg Reduction*

$i =$ `index of node`
`do` $k = 1, \ldots, n-2$
    `if` $k \in \mathbf{P}_i$ `then`
        `compute` $(u, \beta) = H^{(k)}([A]_{*,k})$
        `broadcast` (u, β) `to all nodes`
    `else`
        `receive` (u, β)
    $y_i = 0$
    `do` $j = k, n$
        `if` $j \in \mathbf{P}_i$
            $\nu_j = u^{\mathrm{T}} A$
            $y_i = y_i + v_j [A]_{*,j}$
    `enddo`
    `gsum` $y = \sum y_i$
    $w = y - \beta (u^{\mathrm{T}} y) u$

previous use of superscripts since in the reduction algorithms the Householder transformation computed during the kth iteration has this property. We will also use the pair (u, β) to denote the transformation, i.e., $(u, \beta) = H^{(k)}(x)$ will denote the vector u and scalar β s.t. $H^{(k)}(x) = (I - \beta u u^{\mathrm{T}})$. Since u and β are not uniquely defined, we will always take u to be normalized so that it has a unit kth element.

3 Unblocked Algorithms

In this section, we explain how simple algorithms for the reductions to Hessenberg and tridiagonal forms for the eigenvalue computation can be implemented on sequential and parallel architectures.

3.1 Sequential Implementation: Hessenberg Reduction

The reduction of matrix $A^{(1)} = A$ to Hessenberg form can be written as $A^{(n-1)}$, where

$$A^{(k+1)} = H^{(k)} A^{(k)} H^{(k)} = H^{(k)} H^{(k-1)} \cdots H^{(1)} A^{(1)} H^{(1)} \cdots H^{(k-1)} H^{(k)}$$

Here $H^{(k)} = H^{(k)}([A^{(k)}]_{*,k})$. Letting $(u, \beta) = H^{(k)}$,

$$A^{(k+1)} = H^{(k)} A^{(k)} H^{(k)} = A^{(k)} - \beta u v^{\mathrm{T}} - \beta w u^{\mathrm{T}}$$

where

$$v^{\mathrm{T}} = u^{\mathrm{T}} A^{(k)} \qquad \text{and} \qquad w = A^{(k)} u - \beta (u^{\mathrm{T}} A u) u \tag{1}$$

This yields the following algorithm for reducing a matrix to Hessenberg form:

Algorithm 2 *Hessenberg Reduction*

do $k = 1, \ldots, n-2$
  compute $(u, \beta) = H^{(k)}([A]_{*,k})$
  $v^{\mathrm{T}} = u^{\mathrm{T}} A$
  $w = Au - \beta(u^{\mathrm{T}} A u) u$
  update $A = A - \beta u v^{\mathrm{T}} - \beta w u^{\mathrm{T}}$
enddo

3.2 Sequential Implementation: Tridiagonal Reduction

If A is symmetric, then Equations (1) can be replaced by $y = \beta A u$ and $v = w = y - 1/2 \beta u^{\mathrm{T}} y u$, and the matrix is being reduced to tridiagonal form. In this case, it is only necessary to update the lower triangular part of matrix A at each iteration.

use of Level 1, 2, and 3 BLAS. Communication is through a proposed communication library, the Basic Linear Algebra Communication Subprograms (BLACS) [1], which makes the code portable.

The paper is organized as follows: Assumptions and notation are given in Section 2. As an introduction to the parallel implementation of blocked algorithms, unblocked algorithms and their parallel implementation are given in Section 3. Blocked versions are discussed in Section 4. Results from experiments on the Intel Touchstone Delta system can be found in Section 5. Concluding remarks are given in the final section.

2 Assumptions and Notation

We will assume that our multicomputer consists of p nodes, labeled $\mathbf{P}_0, \ldots, \mathbf{P}_{p-1}$ which are connected by some communication network that allows broadcasting of messages and combining of global data (in the form of global summation).

For our formulae, we adopt the following notation: Scalars, vectors, and matrices are denoted by lower case Greek, lower case, and upper case arabic letters, respectively. The ith element of a vector is denoted by a corresponding greek letter with subscript i ($\chi_i, \eta_i, \upsilon_i$, and ν_i for vectors x, y, u, and v, respectively). Given a vector x, the vector consisting of its elements $i, \ldots, j$ is denoted by $x_{i:j}$. Given matrix A, the submatrix consisting of elements of rows $i, \ldots, j$ and columns $k, \ldots, l$ is denoted by $[A]_{i:j,k:l}$. If all rows are involved, the notation $[A]_{*,k:l}$ will be used. Superscripts are generally reserved for iteration indices.

We will use the following mapping of matrices to nodes: Given $A \in \mathbf{R}^{n\times n}$ and panel width $m \geq 1$, assume for simplicity that $n = r * m$ and partition

$$A^{(k)} = \left(A_1^{(k)} \; A_2^{(k)} \; \cdots \; A_r^{(k)} \right)$$

where $A_j^{(k)} \in \mathbf{R}^{n\times m}$ is a panel of width m. The *panel-wrapped* storage scheme assigns $A_j^{(k)}$ to node $\mathbf{P}_{(j-1)\mathrm{mod} p}$. I.e., $A_{i+1}, A_{i+p+1}, \ldots$ are assigned to $\mathbf{P}_i$. If $m = 1$, the result is the familar *column-wrapped* storage scheme [12]. For notational convenience, we define $j \in \mathbf{P}_i$ to be true if and only if column j of the matrix is assigned to node $\mathbf{P}_i$.

The basic operations utilized by the reduction algorithms are the computation and application of Householder transformations:

Theorem 1 *Given a vector $x \in \mathbf{R}^n$, one can find a vector $u \in \mathbf{R}^n$ and scalar β s.t.*

$$(I - \beta u u^{\mathrm{T}})x = (\chi_1, \ldots, \chi_k, \pm\eta, 0, \ldots, 0)^{\mathrm{T}}$$

where $\eta = ||x_{k+1:n}||_2$.

Indeed, $u = (0, \ldots, 0, \chi_{k+1} \mp \eta, \chi_{k+2}, \ldots, \chi_n)^{\mathrm{T}}$ and $\beta = 2/u^{\mathrm{T}}u$ will give the desired result. The sign is chosen to correspond to the sign of χ_{k+1}, thereby minimizing roundoff error in the computation of u.

The transformation $I - \beta u u^{\mathrm{T}}$ will subsequently be denoted by $H^{(k)}(x)$, where here the superscript indicates that elements $\chi_1, \ldots, \chi_k$ are not affected. This notation is consistent with the

Reduction to Condensed Form for the Eigenvalue Problem on Distributed Memory Architectures *

Jack J. Dongarra † and Robert A. van de Geijn ‡

October 28, 1991

Abstract

In this paper, we describe a parallel implementation for the reduction of general and symmetric matrices to Hessenberg and tridiagonal form, respectively. The methods are based on LAPACK sequential codes and use a panel-wrapped mapping of matrices to nodes. Results from experiments on the Intel Touchstone Delta are given.

1 Introduction

In this paper, we are concerned with the parallel implementation on distributed memory MIMD parallel computers of the LAPACK routines for performing the reduction to Hessenberg form and the reduction to tridiagonal form. These reductions are an important first step in the computation of the eigenvalues of matrices.

The LAPACK project is an effort to update the classical linear algebra software packages LINPACK and EISPACK to allow more efficient use of shared memory or traditional supercomputers. Efficiency is attained by writing these routines as much as possible in Level 2 and 3 BLAS [6, 7], reducing the ratio of memory accesses to floating point operations executed and allowing for encapsulation of parallel operations on shared memory architectures.

While parallel implementations of algorithms for solving linear systems have been widely studied [5, 12], the reduction to condensed form has not enjoyed the same attention. A parallel unblocked Hessenberg reduction algorithm based on column wrapped storage is given in [13, 14]. In [10], a reduction based on Gaussian transformations is reported. The reduction of symmetric matrices assuming row wrapped and grid wrapped storage is addressed in [3, 4]. Our approach is different in that we start with highly efficient sequential code [8]. Efficiency on each node is attained by

*This work was supported in part by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615 and by the Applied Mathematical Science Research Program, Office of Energy Research, U.S. Department of Energy, under Contract DE-AC05-84OR21400.

†Dept. of Computer Sciences, Univ. of TN, Knoxville, TN 37996, and Mathematical Sciences Section, ORNL, Oak Ridge, TN 37831, dongarra@cs.utk.edu

‡Dept. of Computer Sciences, Univ. of TX, Austin, TX 78712, rvdg@cs.utexas.edu. Most of this work was performed while this author was on leave at the Univ. of TN.