SIAM J. Sci. Comput.
Vol. 24, No. 1, pp. 284–311

A PARALLEL IMPLEMENTATION OF THE NONSYMMETRIC QR ALGORITHM FOR DISTRIBUTED MEMORY ARCHITECTURES[*]

GREG HENRY[†], DAVID WATKINS[‡], and JACK DONGARRA[§]

Abstract. One approach to solving the nonsymmetric eigenvalue problem in parallel is to parallelize the QR algorithm. Not long ago, this was widely considered to be a hopeless task. Recent efforts have led to significant advances, although the methods proposed up to now have suffered from scalability problems. This paper discusses an approach to parallelizing the QR algorithm that greatly improves scalability. A theoretical analysis indicates that the algorithm is ultimately not scalable, but the nonscalability does not become evident until the matrix dimension is enormous. Experiments on the Intel Paragon system, the IBM SP2 supercomputer, the SGI Origin 2000, and the Intel ASCI Option Red supercomputer are reported.

Key words. parallel computing, eigenvalue, Schur decomposition, QR algorithm

AMS subject classifications. 65F15, 15A18

PII. S1064827597325165

1. Introduction.

Over the years many methods for solving the nonsymmetric eigenvalue problem in parallel have been suggested. Most of these methods have serious drawbacks, either in terms of stability, accuracy, or scalability, or they require extra work. This paper describes a version of the QR algorithm [24] that has significantly better scaling properties than earlier versions and that is stable, accurate, and efficient in terms of flop count or iteration count.

Most implementations of the QR algorithm perform QR iterations implicitly by chasing bulges down the subdiagonal of an upper Hessenberg matrix [28, 50]. The original version due to Francis [24], which has long been the standard serial algorithm, is of this type. It begins each iteration by choosing two shifts (for convergence acceleration) and using them to form a bulge of degree 2. This bulge is then chased from the top to the bottom of the matrix to complete the iteration. The shifts are normally taken to be the eigenvalues of the 2×2 submatrix in the lower right-hand corner of the matrix. Since two shifts are used, we call this a *double step.* The algorithm is the *implicit, double-shift QR algorithm.* One can get equally well some larger number, say M, of shifts by computing the eigenvalues of the lower right-hand submatrix of order M and using those shifts to form a larger bulge, a bulge of degree M. This leads to the *multishift* QR algorithm, which will be discussed below. The approach taken in this paper is to get M shifts, where M is a fairly large even number (say 40), and use them to form $S = M/2$ bulges of degree 2 and chase them one after another down the subdiagonal in parallel. In principle, this procedure should give the same result as

[*]Received by the editors July 29, 1997; accepted for publication (in revised form) September 21, 1999; published electronically June 18, 2002.
http://www.siam.org/journals/sisc/24-1/32516.html

[†]Computer Science Department, 111 Ayres Hall, The University of Tennessee, Knoxville, TN 37996-1301 (ghenry@cs.utk.edu).

[‡]Pure and Applied Mathematics, Washington State University, WA 99164-3113 (watkins@wsu.edu). Mailing address: 6835 24th Ave. NE, Seattle, WA 98115-7037. The work of this author was supported by the National Science Foundation under grant DMS-9403569.

[§]Innovative Computing Lab, Computer Science Department, 1122 Volunteer Blvd., The University of Tennessee, Knoxville, TN 37996-3450 (dongarra@cs.utk.edu) and Oak Ridge National Laboratory, P.O. Box 2008, Oak Ridge, TN 37831. The work of this author was supported in part by the Office of Scientific Computing, U.S. Department of Energy, under contract DE-AC05-96OR22464.

a multishift iteration, but in practice (in the face of roundoff errors), our procedure performs much better [51].

Of the various parallel algorithms that have been proposed, the ones that have received the most attention recently have been based on matrix multiplication. The reason is clear: large matrix multiplication is highly parallel. Auslander and Tsao [2] and Lederman, Tsao, and Turnbull [42] use multiply based parallel algorithms based on matrix polynomials to split the spectrum. Bai and Demmel [4] use similar matrix multiply techniques using the matrix sign function to split the spectrum (see also [6, 11, 5, 7]). These methods are similar to what we propose in that their focus is on scalability. On the other hand, they have higher flop counts and suffer drawbacks in accuracy.

Dongarra and Sidani [18] introduced tearing methods based on doing rank one updates to an unsymmetric Hessenberg matrix, resulting in two smaller problems that are solved independently and then glued back together with a Newton iteration. This tends to suffer from stability problems since the two smaller problems can have an arbitrarily worse condition than the parent problem [39].

In situations where more than just a few of the eigenvalues (and perhaps eigenvectors as well) are needed, the most competitive serial algorithm is the QR algorithm [24, 1]. Matrix multiply methods tend to require many more flops and sometimes encounter accuracy problems as well [4]. Although matrix tearing methods may have lower flops counts, they require finding all the eigenvectors and hence are only useful when all the eigenvectors are required. Furthermore, there are instances where they simply fail [39]. Jacobi methods [28] have notoriously high flop counts. There are also methods by Dongarra, Geist, and Romine [15] based on initial reductions to tridiagonal form (see also [55]). These might require fewer flops but they are plagued by instability. Against this competition, blocked versions of the implicit double shift QR algorithm [33, 36, 1] appear promising.

One serious drawback of the double implicit shift QR algorithm is that its core computation is based on Householder reflections of size 3. This is a drawback for several reasons: the algorithm lacks the vendor supported performance tuning of the BLAS (basic linear algebra subroutines [14, 40]), and it has data reuse similar to level-1 operations (it does $O(n)$ flops on $O(n)$ data [28]). This imposes an upper limit on how fast it can run on high performance computers with a memory hierarchy. One attempt to rectify this problem involved using the multishift QR algorithm of Bai and Demmel [3], which we mentioned earlier. The idea was to generate a large number M of shifts and use them to chase a large bulge. This allowed the use of a GEMM-based (a level-3 BLAS: GEneral Matrix-matrix Multiply) algorithm [3]. Unfortunately, this requires too many more flops, and in the GEMM itself two of the three required dimensions are very small [36]. The lesson is that adding an additional matrix multiply to an algorithm only helps when there is enough work taken out of the algorithm and replaced by the multiply. However, even if a multishift QR algorithm is used without the additional matrix multiply (which is the better strategy and is how it is implemented in LAPACK [1]), the algorithm has convergence problems caused by roundoff errors if the value of M is too large. This was discussed by Dubrulle [22] and Watkins [51, 52]. Because of this, a multishift size of $M = 6$ was implemented in LAPACK. It is not clear whether this is faster than the double implicit shift QR when the latter is blocked [36].

Because of the difficulties in chasing large bulges, we restrict our analysis in this paper to bulges of degree 2. Most of the results we present, with a few minor modifications to the modeling, would also hold true for slightly larger bulges (e.g., degree 6).

The first attempts at parallelizing the implicit double shift QR algorithm were unsuccessful. See Boley and Maier [12], Geist et al. [26, 27], Eberlein [23], and Stewart [45]. More successful methods came from vector implementations [17]. Usually, the key problem is how to distribute the work evenly given its sequential nature.

A major step forward in work distribution was made by van de Geijn [47] in 1988. There, and in van de Geijn and Hudson [49], a wrap Hankel mapping was used to distribute the work evenly. A simple case of this, antidiagonal mappings, was exploited by Henry and van de Geijn [37]. One difficulty these algorithms faced is that they all used non-Cartesian mappings. In these mappings, it is impossible to go across both a row and a column with a single fixed offset. Losing this important feature forces an implementation to incur greater indexing overheads and, in some cases, to have shorter loops. However, in Cartesian mappings, both rows and columns of the global matrix correspond to rows and columns of a local submatrix. If a node owns the relevant pieces, it can access both $A(i+1, j)$ and $A(i, j+1)$ as some fixed offset from $A(i, j)$ (usually 1 and the local leading dimension, respectively). This is impossible for Hankel mappings on multiple nodes. In addition to this problem, the algorithms resulting from all these works were only iso-efficient [30]. That is, you could get 100% efficiency, but only if the problem size was allowed to scale faster than memory does. Nevertheless, these were the first algorithms ever to achieve theoretically perfect speed-up.

In [37] it was also proved that the standard double implicit shift QR algorithm (not just the one with antidiagonal mappings) cannot be scalable. This same work also showed that if M shifts are employed to chase $M/2$ bulges of degree 2, then the algorithm might be scalable as long as M is at least $O(\sqrt{p})$, where p is the number of processors.

Here we pursue the idea of using M shifts to form and chase $M/2$ bulges in parallel. The idea of chasing multiple bulges in parallel is not new [31, 45, 46, 48, 41]. However, it was long thought that this practice would require the use of out-of-date shifts, resulting in degradation of convergence [48]. What is new [51] (having seen [22]) is the idea of generating many shifts at once rather than two at a time, thereby allowing all bulges to carry up-to-date shifts. The details of the algorithm will be given in the next section, but the important insight is that one can then use standard Cartesian mappings and still involve all the processors if the bulge chasing is divided evenly among them.

2. Serial QR algorithm.

We start this section with a brief overview of the sequential double implicit shift QR algorithm. Before we detail the parallel algorithm in section 3 it is necessary to review the difficulties in parallelizing the algorithm. This is done in section 2.2. In section 2.3 we make some modifications to overcome these difficulties. Finally, in section 2.4 we give some experimental results to indicate the impact of the changes.

2.1. Single bulge.

The double implicit Francis step [24] enables an iterative algorithm that goes from H (real upper Hessenberg) to $H = QTQ^T$ (real Schur decomposition [28, 55]). Here, Q is the orthogonal matrix of Schur vectors, and T is an upper quasi-triangular matrix (1×1 and 2×2 blocks along the main diagonal). We can assume that H is upper Hessenberg because the reduction to Hessenberg form is well understood and can be parallelized [8, 16, 19].

```
**Francis HQR Step**
$e = \text{eig}(H(N-1:N, N-1:N))$
Let $x = (H - e(1)I_N) * (H - e(2)I_N)e_1$
Let $P_0 \in \Re^{N \times N}$ be a Householder matrix such that
    $P_0 x$ is a multiple of $e_1$.
$H \leftarrow P_0 H P_0$
**for** $i = 1, \ldots, N-2$
        Compute $P_i$ so that
            $P_i H$ has zero $(i+2, i)$ and
                $(i+3, i)$ (for $i < N-2$) entries.
        Update $H \leftarrow P_i H P_i$
        Update $Q \leftarrow Q P_i$
**endfor**
```

FIG. 1. *Sequential single bulge Francis HQR step.*

The implicit Francis iteration assumes that H is unreduced (subdiagonals nonzero). As the iterates progress, wise choices of shifts allow the subdiagonals to converge to zero. As in [28], at some stage of the algorithm H might be in the following form:

$$H = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ & H_{22} & H_{23} \\ & & H_{33} \end{bmatrix}.$$

We assume that H_{22} is the largest unreduced Hessenberg matrix above H_{33} (which has converged) in the current iteration. The algorithm proceeds on the rows and columns defined by H_{22}.

One step of the Francis double shift QR algorithm is given in Figure 1. A single bulge of degree 2 is chased from top to bottom. Here, the Householder matrices are symmetric orthogonal transforms of the form

$$P_i = I - 2\frac{vv^T}{v^T v},$$

where $v \in \Re^N$ and

$$v_j = \left\{ \begin{array}{ll} 0 & \text{if } j < i+1 \text{ or } j > i+3 \\ 1 & \text{if } j = i+1 \end{array} \right\}.$$

Suppose the largest unreduced submatrix of H (H_{22} above) is in $H(k:l, k:l)$. We then apply the Francis HQR step to the rows and columns of H corresponding to the submatrix. (We use the term HQR to mean a practical Hessenberg QR iteration, for example, EISPACK's HQR code [43] or LAPACK's *_HSEQR* or *_LAHQR* code [1].) That is,

$$H(k:l,:) \leftarrow P_i H(k:l,:),$$

$$H(:,k:l) \leftarrow H(:,k:l) P_i.$$

Naturally, if we are not seeking a complete Schur decomposition but instead only desire the eigenvalues, then the updating of H_{23} and H_{12} above can be skipped. The two shifts are chosen to be

$$e = \text{eig}(H(l-1:l, l-1:l)).$$

In practice, after every few iterations, some of the subdiagonals of H will become numerically zero, and at this point the problem deflates into smaller problems.

2.2. Difficulties in parallelizing the algorithm. Consider the following upper Hessenberg matrix with a bulge in columns 5 and 6:

$$H = \begin{bmatrix} X & X & X & X & X & X & X & X & X & X \\ X & X & X & X & X & X & X & X & X & X \\ & X & X & X & X & X & X & X & X & X \\ & & X & X & X & X & X & X & X & X \\ & & & X & X & X & X & X & X & X \\ & & & & X & X & X & X & X & X \\ & & & & +_{7,5} & X & X & X & X & X \\ & & & & +_{8,5} & +_{8,6} & X & X & X & X \\ & & & & & & & X & X & X \\ & & & & & & & & X & X \end{bmatrix}.$$

Here, the X's represent elements of the matrix, and the $+$'s represent bulges created by the bulge-chasing step in Figure 1. A Householder reflection must be applied to rows 6, 7, and 8 to zero out $H(7:8,5)$. To maintain a similarity transformation, the same reflection must then be applied to columns 6, 7, and 8, thus creating fill-in in $H(9,6)$ and $H(9,7)$. In this way, the bulge moves one step down and the algorithm in Figure 1 proceeds.

Suppose one used a one-dimensional column wrapped mapping of the data. Then the application of the reflection to rows 6, 7, and 8 would be perfectly distributed amongst all the processors. Unfortunately, this would be unacceptable because applying all the column reflections, half the total work, would involve at most 3 processors, thus implying that the maximum speed-up obtainable would be 6 [27]. Tricks to delay the application of the row and/or column reflections appear to only delay the inevitable load imbalance. The same argument holds for using a one-dimensional row wrapped mapping of the data, in which the row transforms are unevenly distributed.

For scalability reasons, distributed memory linear algebra computations often require a two-dimensional block wrap torus mapping [32, 20]. To maximize the distribution of this computation, we could wrap our two-dimensional block wrap mapping as tightly as possible with a block size of one (this would create other problems which we will ignore for now). Let us assume the two-dimensional logical grid is $R \times C$, where $R \times C = P$ (the total number of processors). Then any row must be distributed amongst C processors and the row reflections can be distributed amongst no more than $3C$ processors. Similarly, column reflections can use at most $3R$ processors. The maximum speed-up obtainable is then $3(R + C)$, where in practice one might expect no more than 2 times the minimum of R and C.

If one uses an antidiagonal mapping of the data [37], then element $H(i, j)$ or (if one uses a block mapping as one should) submatrix H_{ij} is assigned to processor

$$(i + j - 2)\mathbf{mod}\ P,$$

where P is the number of processors. That is, the distribution amongst the processors is as follows [49]:

$$\begin{bmatrix} H_{1,1}^{(0)} & H_{1,2}^{(1)} & H_{1,3}^{(2)} & \cdots & H_{1,p-1}^{(p-2)} & H_{1,p}^{(p-1)} \\ H_{2,1}^{(1)} & H_{2,2}^{(2)} & H_{2,3}^{(3)} & \cdots & H_{2,p-1}^{(p-1)} & H_{2,p}^{(0)} \\ H_{3,1}^{(2)} & H_{3,2}^{(3)} & H_{3,3}^{(4)} & \cdots & H_{3,p-1}^{(0)} & H_{3,p}^{(1)} \\ \vdots & & \ddots & & \vdots & \vdots \\ H_{p,1}^{(p-1)} & & \cdots & & H_{p,p-1}^{(p-3)} & H_{p,p}^{(p-2)} \end{bmatrix},$$

where the superscript indicates the processor assignment. Clearly, any mapping where (if the matrix is large enough) any row and any column is distributed roughly evenly among all the processors would suffice. Unfortunately, no Cartesian mappings satisfy this criterion. There are reasons to believe that the antidiagonal distribution is ideal. Block diagonal mappings have been suggested [56], but these suffer from load imbalances that are avoided in the antidiagonal case.

By making a slight modification to the algorithm, one could chase several bulges at once and continue to use a two-dimensional Cartesian mapping. That is, if our grid is $R \times R$ and we chase R bulges, separated appropriately, then instead of involving only 3 rows and 3 columns, we would involve $3R$ rows and columns. This allows the work to be distributed evenly. Furthermore, we maintain this even distribution when we use any multiple of R bulges—a mechanism useful for decreasing the significance of pipeline start-up and wind-down.

2.3. Multiple bulges. The usual strategy for computing shifts is the Wilkinson strategy [55], in which the shifts are taken to be the eigenvalues of the lower 2×2 submatrix. This is inexpensive and works well as long as only one bulge at a time is being chased. The convergence rate is usually quadratic [55, 53]. However, this strategy has the following shortcoming for parallel computing: The correct shifts for the next iteration cannot be calculated until the bulge for the current iteration has been chased all the way to the bottom of the matrix. This means that if we want to chase several bulges at once and use the Wilkinson strategy, then we must use out-of-date shifts. This practice results in subquadratic (although still superlinear) convergence [46, 48].

If we wish to chase many bulges at once without sacrificing quadratic convergence, then we must change the shifting strategy. One of the strategies proposed in [3] was a generalization of the Wilkinson shift. Instead of choosing the 2 shifts to be the eigenvalues of the lower 2×2 matrix, one calculates the eigenvalues of the lower $M \times M$ matrix, where M is an even number that is significantly greater than 2 (e.g., $M = 32$). Then one has enough shifts to chase $M/2$ bulges in either serial or parallel fashion before having to go back for more shifts. This strategy also results (usually) in quadratic convergence, as was proved in [53] and has been observed in practice. We refer to each cycle of computing M shifts and chasing $M/2$ bulges as a *superiteration.*

The question of how to determine the number of bulges $S = M/2$ per superiteration is important. If we choose $S = 1$, we have the standard double-shift QR algorithm—but this has scalability problems. If we choose S large enough, and the bulges are spaced appropriately (we address these issues in the next section), then there are sufficient bulges to distribute the workload evenly. In fact, we later discuss in section 4.2.5 the motivations for choosing S larger than the minimum number required for achieving an even distribution of work. Of course, choosing S too large might result in greater flops overall or other general imbalances (since the computation of the shifts is usually serial and grows as $O(S^3)$).

```
Multiple Bulge HQR Superiteration
$e = \mathrm{eig}(H(N-M+1:N, N-M+1:N))$
for $k = 0, \ldots, N-6+2*M$
    for $j = M, M-2, M-4, \ldots, 2$
        $i = k - 2j + 4$
        if $i < 0$ then $P_i = I$
        if $i = 0$
            Let $x = (H - e(j-1)I_N) * (H - e(j)I_N)e_1$
            Let $P_i \in \Re^{N \times N}$ be a Householder matrix
                such that $P_i x$ is a multiple of $e_1$.
        if $1 \le i < N-2$
            Compute $P_i$ so that
                    $P_i H$ has zero $(i+2, i)$ and
                    $(i+3, i)$ entries
        if $i = N-2$
            Compute $P_i$ so that
                    $P_i H$ is zero at $(i+2, i)$
        if $i > N-2$ then $P_i = I$
        $H \leftarrow P_i H P_i$, $Q \leftarrow Q P_i$
    endfor
endfor
```

FIG. 2. *Sequential multiple bulge HQR superiteration.*

The general algorithm proceeds as described in Figure 2.

In Figure 2, the i index refers to the same i index as in the previous algorithm in Figure 1. Because there are multiple bulges, $M/2$ of them, there is a certain start-up and wind-down, in which case some of the bulges might have already completed or not yet started (when $i < 0$ or $i > n-2$).

Here, we are spacing the bulges four columns apart; however, it is clear that this spacing can be anything four or larger, and for the parallel algorithm we will give a rationale for choosing this spacing very carefully. In Figure 3, we see a Hessenberg matrix with four bulges going at once.

The shifts are the eigenvalues of a trailing submatrix. Notice that the bottom shifts are applied first ($j = M, M-2, \ldots, 2$). These are the ones that emerged first in the shift computation, and these are the shifts that are closest to the eigenvalues that are due to deflate next. Applying them first enhances the deflation process.

Complex shifts are applied in conjugate pairs. One fine point that has been left out of Figure 2 is that whenever a lone real shift appears in the list, it must be paired with another real shift. This is done by going up the list, finding the next real shift (and there will certainly be one), and launching a bulge with the two real shifts.

One critical observation is that whenever a subdiagonal element becomes effectively zero, it should be set to zero immediately, rather than at the end of the superiteration. This saves time because the information is in cache, but, more important, it reduces the number of iterations and total work. An entry that has become negligible early in a superiteration might no longer meet the deflation criterion at the end of the superiteration.

Another critical observation is that consecutive small subdiagonal elements may negatively impact convergence by washing out the effects of some of the shifts. Robust

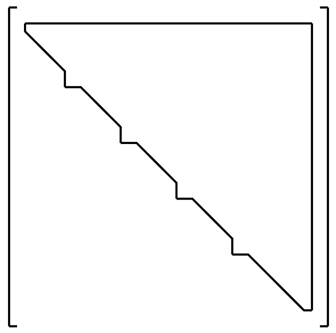

FIG. 3. *Pipelined QR steps.*

implementations of QR usually search for a pair of small subdiagonal elements along the unreduced submatrix and attempt to start the bulge chasing from there. In the multishift and multibulge cases, if any shifts could be started in the middle of the submatrix at some small subdiagonal elements, it might save flops to do so. However, the subdiagonals change with each bulge; it could easily happen that 10 bulges were precomputed, and after the second bulge went through, it created two consecutive small subdiagonal elements somewhere, and the third bulge then encountered accuracy difficulties and made no additional progress towards convergence. We found it necessary to maximize the number of shifts that can go through. Sometimes, using the same type of tests as in the single double-shift QR, we found that if the third double shift is unable to sweep the length of the submatrix without encountering this "washing out" effect, then it is possible that the fourth double shift might still work instead.

Finally, when we need M shifts, there is no reason to require that these be the eigenvalues of only the lower $M \times M$ submatrix. For example, we could take $W > M$ and choose M eigenvalues from the lower $W \times W$ submatrix. This strategy tends to give better shifts, but it is more expensive.

2.4. Experimental comparisons. We now have two different HQR algorithms: the standard one based on the iteration given in Figure 1, and the multiple bulge algorithm based on the iteration given in Figure 2. We treat convergence criteria the same and use the same outsides of the code to generate the largest unreduced submatrix and to determine deflation.

We are now ready to ask what is a reasonable way to compare the two algorithms in terms of workload. At this stage, we are not interested in execution time, because that is dependent on other factors such as the logical mapping and blocking sizes. The clearest method is a flop count. That is, run the two algorithms to completion in the exact same way, monitoring the flops as they proceed (including extra flops required in generating the shifts).

Since both algorithms normally converge quadratically, it is not unreasonable to expect them to have similar flop counts; if M is not made too large, the extra cost of the shift computation will be negligible.

Our practical experience has been that the flop count for the multiple bulge algorithm is usually somewhat less than for the standard algorithm. For example, in Figure 4 the flop counts for two versions of QR applied to random upper Hessenberg matrices with entries between -2 and 2 are given. The curve labeled 1×1 is the standard algorithm. The curve labeled 4×4 gives flop counts for a multiple bulge

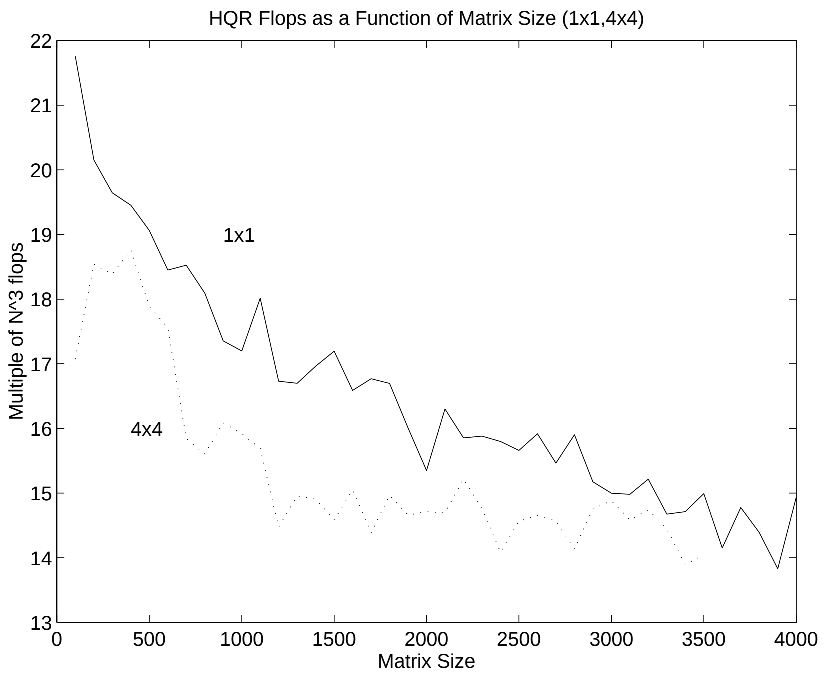

FIG. 4. *The decreasing average flops for a 1×1 and a 4×4 grid.*

algorithm run on a 4×4 processor grid. The code is written in a way so that the number of bulges per superiteration varies in the course of a given computation, but in these cases it was typically 4, i.e., $M = 8$.

The flop count of the Hessenberg QR algorithm is normally reckoned to be $O(N^3)$, based on the following considerations: Each iteration requires $O(N^2)$ work, at least one iteration will be needed for each eigenvalue or pair of eigenvalues, and there are N eigenvalues. Golub and Van Loan [28] report the figure $25N^3$.

All the numbers in Figure 4 are less than $25N^3$. More important, there is a clear tendency, especially in the 1×1 case, of the multiple of N^3 to decrease as N increases. Let us take a closer look at the flop counts.

Consider first the standard algorithm. Experience with matrices of modest size suggests that approximately four iterations (double steps) suffice to deflate a pair of eigenvalues. This number should not be taken too seriously; it depends on the class of matrices under consideration. If we want to be conservative, we might say five or six instead of four. Whatever number we choose should be viewed only as a rough average. In practice there is a great deal of variation. Thus we reckon that it takes about two iterations to calculate each eigenvalue. For each eigenvalue we deflate, we reduce the size of the active submatrix by one. A double QR iteration applied to a $k \times k$ submatrix of an $N \times N$ matrix costs about $20Nk$ flops. This is for the computation of the complete Schur form. If only the eigenvalues are wanted, then the cost is $10k^2$ flops. This figure is derived as follows. The bulge chase is affected by application of $k - 1$ Householder transformations of size 3×3. Suppose we apply a Householder transform to the rows and columns of H at step $i = 10 \ll N$ of Figure 2 (prior to the transform, there is a bulge that starts in column 10 with nonzeros in $H(12, 10)$ and $H(13, 10)$). The columns impacted are 10 through N and the rows impacted are 1 through 14 (the bulge slips down to $H(14, 11)$). In general, each Householder transformation is applied to $N + 4$ rows/columns of the Hessenberg matrix H and N columns of the transforming matrix Q. The cost of applying a Householder transformation of size 3 to a single row/column vector of size 3×1 or 1×3 is 10 flops. We must multiply this by the number of columns and rows involved ($N + 4$ for the Hessenberg matrix and N for Q). Thus the total is

$$10 \times (2N + 4) \times (k - 1) \approx 20Nk.$$

If we assume there are two iterations for each submatrix size k, we get a total flop count of approximately

$$2 \times 20N \sum_{k=1}^{N} k \approx 20N^3.$$

We could have obtained the count $25N^3$ reported by Golub and Van Loan [28] by assuming five iterations per pair instead of four. The figure $20N^3$ is closer to what we see in Figure 4. It is a particularly good estimate when N is small.

This count applies to the standard single bulge algorithm. The multiple bulge algorithm of Figure 2, which goes after the eigenvalues M at a time rather than two at a time, has very different deflation patterns. We can arrive at the figure $20N^3$ for this algorithm by assuming that $M/4$ eigenvalues are deflated per superiteration (with M shifts carried by $M/2$ bulges). This is approximately what is seen in practice, although there is a great deal of variation.

As N becomes large, the figure $20N^3$ looks more and more like an overestimate. The discrepancy can be explained as follows. Our flop count takes deflations into account, but it ignores the fact that the matrix can split apart in the middle due to some $H_{i+1,i}$ $(1 \ll i \ll N)$ becoming effectively zero. Many of the subdiagonal entries $H_{i+1,i}$ drift linearly toward zero [55] in the course of the computation, so it is to be expected that such splittings will sometimes occur. It is reasonable to expect splittings to occur more frequently in large problems rather than in small ones. Whenever such a split occurs, the flop count is decreased. As an extreme example, suppose that on an early iteration we get $H_{i+1,i} \approx 0$, where $i \approx N/2$. Then all subsequent operations are applied to submatrices of order $\approx N/2$ or less. Even assuming no subsequent splittings, the total flop count is about

$$2 \times 2 \times 20N \sum_{k=1}^{N/2} k \approx 10N^3,$$

which is half what it would have been without the split.

Figure 5 further supports the view that splittings are significant for large N. Because of deflations and splittings, all but the first few iterations are applied to submatrices of size $k < N$. The size of the submatrices decreases as the computation progresses. In Figure 5 we are looking at each iteration, computing the size of the submatrix we are working on divided by the original problem size, and then averaging these fractions of the course of the problem. Several different problems were done with random Hessenberg matrices consisting of elements from -2 to 2, and the average fraction is given in the figure. If a pair of eigenvalues is deflated every 4 (or whatever number of) iterations, as in our model, the average submatrix size will be $.5N$. In fact one might expect a somewhat larger average, based on the observation [55] that

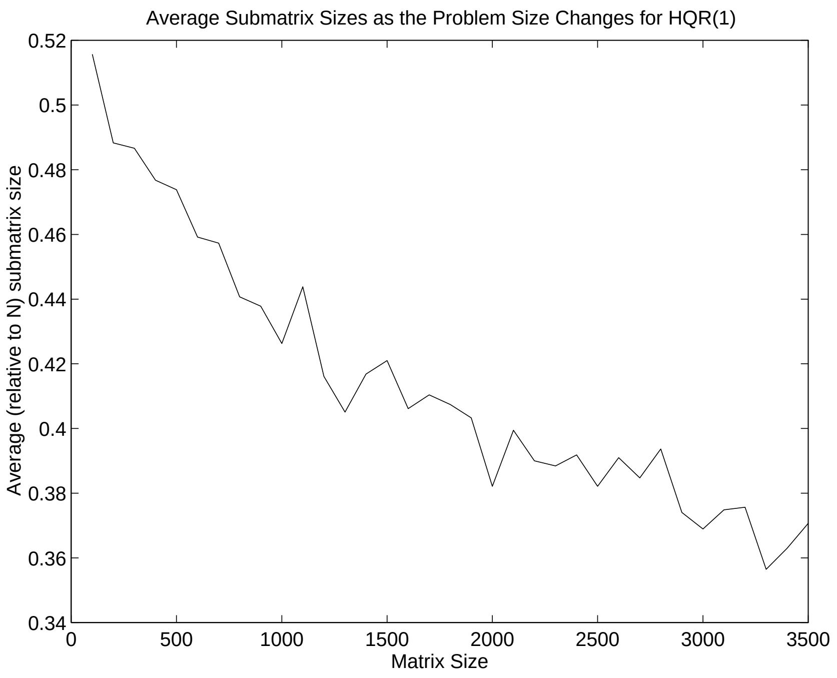

FIG. 5. *The decreasing average submatrix size.*

more iterations per eigenvalue are required in the earlier iterations (large matrices) rather than in the later iterations (small matrices). On the other hand, splittings will have the effect of decreasing the average. The fact that the average size is in fact less than $.5N$ and decreases as N is increased is evidence that splittings eventually have a significant effect.

3. Parallel QR algorithm.

The most critical difference between serial and parallel implementations of HQR is that the number of bulges must be chosen to keep the processors busy. Assume that the processors are arranged logically as a grid of R rows and C columns. Thus there are $P = R * C$ processors. Clearly, the number of bulges will optimally be a multiple of the least common multiple of R and C; that way all nodes will have equal work. As we shall see, there are tradeoffs involved in using more bulges than necessary. The matrix is decomposed into $L \times L$ blocks, which are parceled out to the processors by a two-dimensional (block cyclic) torus wrap mapping [9, 10]. We refer to the block size of the cyclic mapping as L. The bulges in our algorithm must be separated by at least a block, and remain synchronized, to ensure that each row/column of processors remains busy. Usually the block size L must be large; otherwise there will be too much border communication.

We try to keep the overall logic as similar to the well-tested standard QR algorithm as possible. For this reason each superiteration is completed entirely before new shifts are determined and another superiteration is begun. Information about the "current" unreduced submatrix must remain global to all nodes.

We now briefly describe the communication patterns necessary in the parallel implementation of a single superiteration of Figure 2. As just mentioned, the information on the size and start and stop of the largest unreduced Hessenberg matrix used for the superiteration must be made global to all nodes. This can be done either by broadcasting the diagonal information to all nodes and letting each node reach the same result, or by having one or more of the nodes determine the result and broadcast (via a tree broadcast) the results to everyone. We prefer the latter strategy. There are three forms of communication necessary in this algorithm. The first is the global communication just described at the start and end of each superiteration. The second comprises the broadcasts of the Householder transforms. The broadcasts of the Householder transforms are only for the rows and columns to which each transform is to be applied. Therefore, usually another tree broadcast from the subset of nodes consisting of a single row or column of nodes suffices. Finally, the last form of communication is a nearest neighbor communication. When a bulge moves from one block to another, there may be a neighbor communication of the border between the blocks that the two neighbors share; a broadcast is not necessary.

The Householder transforms are of size 3, which means they are specified by sending 3 data items. The latency associated with sending such small messages would be ruinous, so we bundle the information from several (e.g., 30) Householder transformations in each message. Let B denote the number of Householder transforms in each bundle. Since the processors own $L \times L$ blocks of the matrix, we must have $B \leq L$. Another factor that limits the size of B is that processors must sometimes sit idle while waiting for the Householder information. In order to minimize this effect, the processors that are generating the information should do so as quickly as possible. This means that while pushing the bulge ahead B positions, they should operate only on the $(B+2) \times (B+2)$ subblock through which the bulge is currently being pushed. The Householder transforms can be applied to the rest of the block after they have been broadcast to the processors that are waiting.

If many bulges are being chased simultaneously, there may be several bulges per row or column of processors. In that case, we can reduce latency further by combining the information from all bulges in a given row or column into a single message.

The broadcasts must be handled with care. Consider the situation depicted in Figure 6. Here, we have two bulges. Suppose, while the bottom bulge is doing a vertical broadcast, the top bulge starts a horizontal broadcast. This results in a collision that prevents these two broadcasts from happening in parallel. Our solution is to do all the vertical broadcasts at once, followed by all the horizontal broadcasts.

With this in mind, along the lines of Figure 2 we briefly give a pseudocode for the parallel algorithm. At this point, many details are lost to the pseudocode and, in particular, much of the details of the remainder of this section cannot be captured.

The general parallel algorithm proceeds as described in Figure 7.

3.1. Block Householder transforms. Since Householder information arrives in bundles, we might as well apply the transformation in blocks to increase data reuse. (Unfortunately, there is no BLAS for the application of a series of Householder transforms.) Normally, B Householder transforms of size 3 are received at once. If we apply them to $B+2$ columns of size N, we perform $10NB$ flops, and we must access $(B+2)N$ data. The data reuse fraction [36] is at best 10 flops per data element accessed in the limit. If B is 1, which is the worst case, then only 3 flops are done per data element accessed. So, one can roughly improve the data reuse by a factor of 3 by applying these transforms simultaneously.

Due to diminishing returns, the data reuse is not significantly better when going from 10 applications at a time to 20. In fact, things are worse because one accesses almost twice the data for only a marginal improvement in data reuse. Because data

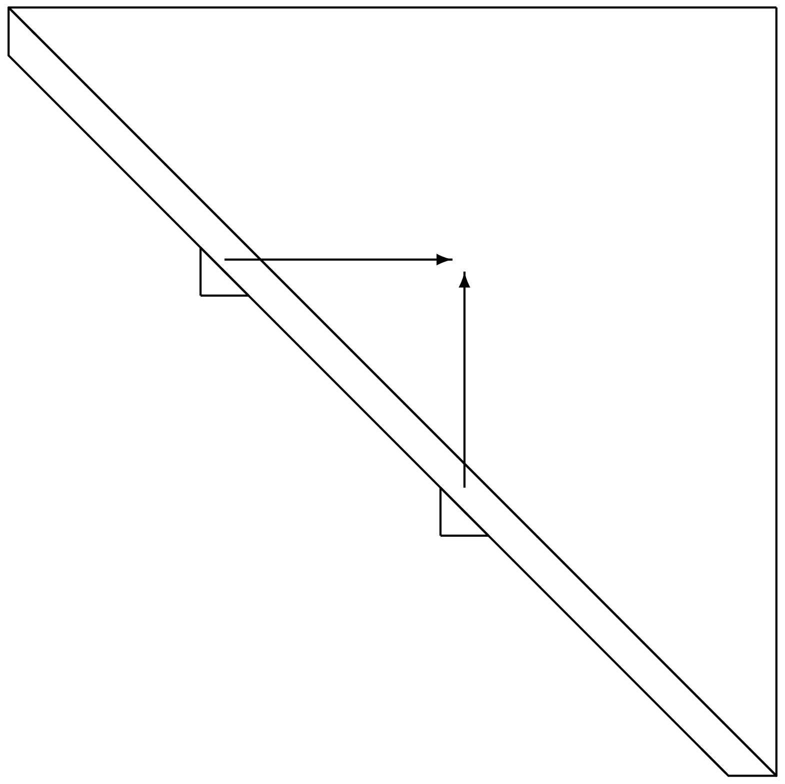
Fig. 6. *Multiple bulge broadcasts colliding.*

will be pushed out of cache, it is clear that one needs to find a compromise between data reuse and data volume.

Fortunately, the number of transforms applied at once is independent of anything we later determine for B. The only reasonable restriction we suggest is that the number of transforms applied at once in block fashion be no greater than B. In practice, one might apply these transforms in sets of two, three, or four.

3.2. Efficient border communications. When a bulge reaches a border between two processors or columns, some communication is necessary for the bulge to proceed. These "border communications" should be done in parallel if possible. That is, if we have 36 nodes logically mapped into 6 processor rows and 6 processor columns, and we have 6 bulges spaced a block apart, then we have 6 border communications that should happen at the same time. A border communication typically consists of a node sending data to another node and then waiting for its return while the other node updates the data. If they are handled one at a time, then this sequentializes a good portion of the computation. Nevertheless, it is clear that if there are only 2 rows and/or columns, the overhead costs of loading all the bulge information to send it out, only to later reload all the information to be returned, might not justify the effort to parallelize the border communications.

This implies that an entirely different approach to border communication should be used when there is a small number of rows or columns (we give our findings below for our implementation) compared to a larger number of rows or columns. The

```
Parallel Multiple Bulge HQR Superiteration
(All broadcasts are to a single row or column subset of the nodes unless at a
border, in which case it is to two rows or columns, or unless it is stated otherwise.)
($B$ is the bundle Householder blocking size)
Determine $M$ based on node grid and submatrix size
$e = \mathrm{eig}(H(N-M+1:N, N-M+1:N))$
$e$ is globally broadcast to all nodes.
Sort $e$ so that they are in pairs (real pairs together)
$k = 0$
while $k < N - 6 + 2 * M$
    for $j = M, M-2, M-4, \ldots, 2$
        $i_0 = k - 2j + 4$
        if $\mathrm{MOD}(k - 2j + 4, L) < L - 2$ (the middle of a block)
            Do the min amount of work nesc. to compute each $P_i$:
            for $i = i_0$, $\mathrm{MIN}(i_0 + B - 1, N - 2)$
                Compute $P_i$ as in Figure 2
            endfor
            Broadcast $P_i$ vertically.
            Broadcast $P_i$ horizontally.
        else (at a border)
            Communicate border info nesc. to compute $P_i$ above.
            Compute $P_i$ as above, treating $B$ as 1.
            Broadcast the results.
        endif
    endfor
    for $j = M, M-2, M-4, \ldots, 2$
        $i_0 = k - 2j + 4$
        if $\mathrm{MOD}(k - 2j + 4, L) < L - 2$ (the middle of a block)
            for $i = i_0$, $\mathrm{MIN}(i_0 + B - 1, N - 2)$
                $H \leftarrow HP_i$, $Q \leftarrow QP_i$
            endfor
            for $i = i_0$, $\mathrm{MIN}(i_0 + B - 1, N - 2)$
                $H \leftarrow P_iH$
            endfor
        else (at a border)
            Communicate border info nesc. to apply the transforms.
            Apply $P_i$ as described above, treating $B$ as 1.
        endif
    endfor
    Increment $k$ by the number of $B$ bundles just computed.
endwhile
```

FIG. 7. *Parallel multiple bulge HQR superiteration.*

method currently implemented in our code is a hybrid approach attempting to perform optimally for both a small and a large number of rows or columns. For a small number of rows or columns (three or less in our implementation), each bulge has its border communication resolved at once. That is, a node sends the data out and does nothing until that data has been returned and a new bulge can be worked on. For a larger

Table 1
Model definitions.

Variable	Definition
α	Message latency.
β	Time to send one double precision element. Sending a message of length n takes $\alpha + n\beta$ time units.
γ	Time to implement a single floating point operation used in a Householder application.
N	The matrix order/size.
R	Number of rows of processors.
C	Number of columns of processors.
P	Number of nodes, $P = R * C$.
B	Number of Householder transforms per bundle.
L	Blocking size of the matrix data on the 2D block torus wrap mapping.
V	Time to compute a single Householder transform of size 3.
M	Number of shifts.
S	Number of bulges. $S = M/2$.
W	Size of the generalized Wilkinson shift determination submatrix at the bottom. Note that $N \gg W \geq 2S = M$.

number of rows or columns (four or more), each bulge has its border communication resolved in parallel. All the rows (or columns) try to send the information out, all the recipients try to update the information at once and send it back, and then all the original senders try to receive the data.

4. Modeling.

The variable names used in this section are summarized in Table 1.

4.1. Serial cost analysis.

Consider first the serial cost of one superiteration of Figure 2. The shift computation costs $O(W^3)$, where W is the size of the shift determination matrix. Once the shifts have been determined, all bulge chases are independent and have the same amount of work. Thus it suffices to compute the cost of one iteration of Figure 1 and multiply it by S, the number of bulges. Within the loop indexed by i in Figure 1, each Householder transform of size 3 is applied to $N - i + 1$ triplets of rows and $i + 3$ triplets of columns. Since each transform applied to a row or column of size j requires $10j$ flops, there are approximately $10(N + 4)$ flops required on the Hessenberg matrix and $10N$ flops required on the Schur matrix Q. In addition, it takes time V to generate each Householder transform. There are $N - 1$ Householder transforms per bulge chase, so the cost of one bulge chase is $(10(2N + 4)\gamma + V)(N - 1)$. Thus the serial cost of one superiteration of Figure 2 is

$$(10(2N + 4)\gamma + V)(N - 1)S + O(W^3) = 20N^2S\gamma + O(N)$$

if $W \ll N$.

4.2. Parallel cost analysis.

The processors are arranged logically in a grid of R rows and C columns. We assume that the block distribution size L is small enough, compared to N, so that each row (column) of processors has about as much work as any other row (column). We divide the work into two categories: horizontal and vertical.

4.2.1. Computational cost.

The amount of computational work associated with each superiteration is roughly $10N^2S$ flops for the Hessenberg matrix and $10N^2S$ flops for the Schur vectors. The work on the Hessenberg matrix is initially half row transforms and half column transforms. The work on the Schur vectors consists of all column transforms. Thus the amount of horizontal work is $5N^2S$ flops. Each row is distributed evenly over C processors, so the execution time is about $(5N^2/C)\gamma$ per bulge. If there are 4 bulges ($S = 4$) and two rows ($R = 2$), then one might expect this time to double; hence we multiply by the ceiling of S/R to obtain $(5N^2/C)\lceil(S/R)\rceil\gamma$ for the horizontal work. Similarly, the time to do the vertical work is $((15N^2)/R)\lceil(S/C)\rceil\gamma$. Thus the total time for horizontal plus vertical computational work is

$$\left(\frac{5N^2}{C}\left\lceil\frac{S}{R}\right\rceil + \frac{15N^2}{R}\left\lceil\frac{S}{C}\right\rceil\right)\gamma. \tag{4.1}$$

For the special case of $S = R = C$, (4.1) reduces to $(20N^2S/P)\gamma$, where the factor P in the denominator indicates perfect speed-up. However, this expression and (4.1) ignore many overheads, all of which will be considered in the following subsections.

4.2.2. Broadcast communication. For each bulge chase, there are $N - 1$ Householder transforms. They are bundled together in groups of B, so there will be about $(N-1)/B$ bundles, each of which needs to be broadcast both horizontally and vertically. Since each bundle contains $3B$ data items, the communication overhead associated with each bundle is $\alpha + 3B\beta$. For horizontal messages, each message must be broadcast to a logical row of processors. Using a minimum spanning tree broadcast, this requires $\log(C)$ messages. When there is only one bulge ($S = 1$), the total horizontal broadcast overhead is therefore

$$\frac{N-1}{B}\log(C)(\alpha + 3B\beta), \tag{4.2}$$

and the total vertical broadcast overhead is

$$\frac{N-1}{B}\log(R)(\alpha + 3B\beta).$$

When there are $S > 1$ bulges, the amount of horizontal data to be broadcast gets multiplied by S. If there are no more bulges than rows ($S \leq R$), the horizontal broadcast overhead will be the same as if there were only one bulge, because the messages are broadcast in parallel. If $S > R$, we assume the broadcasts are combined so that there are at most R messages or a single message per row. The amount of horizontal data that must be broadcast on the most burdened row is $3B\lceil S/R\rceil$. Similar remarks apply to the vertical broadcasts. Therefore, the total time for horizontal and vertical transform broadcasts is

$$\frac{N-1}{B}\left[\log(C)\left(\alpha + 3B\left\lceil\frac{S}{R}\right\rceil\beta\right) + \log(R)\left(\alpha + 3B\left\lceil\frac{S}{C}\right\rceil\beta\right)\right]. \tag{4.3}$$

4.2.3. Border communication. In addition to Householder broadcasts, there is border communication whenever a bulge hits the boundary between two rows or columns of processors. At this point, it is necessary to send boundary data back and forth in order to push the bulge past this point.

Let us begin by considering border communication in the Schur matrix Q. This is easier to discuss than border communication in H because it involves only columns. At each border encounter, two columns of the matrix have to be passed from one processor to the next. This is a total of $2N$ numbers, which are split over R processor rows. Each message thus contains $2N/R$ numbers, and the time to pass R such messages in parallel is $\alpha + \frac{2N}{R}\beta$. It takes the same time to pass $2N$ numbers back. During a single bulge chase, there are about N/L border encounters, whose total overhead is thus

$$\frac{2N}{L}\left[\alpha + \frac{2N}{R}\beta\right].$$

If $S > 1$ bulges are chased, then up to C border crossings can be accomplished in parallel. If $S > C$, then the most burdened processors will have to handle $\lceil \frac{S}{C} \rceil$ border crossings at a time. Each batch of $\lceil \frac{S}{C} \rceil$ messages can be combined into a single large message to reduce latency. Thus the total overhead for border communication in the Schur matrix Q is

$$\frac{2N}{L}\left[\alpha + \frac{2N}{R}\left\lceil \frac{S}{C} \right\rceil \beta\right]. \tag{4.4}$$

The analysis of the border communication in the Hessenberg matrix H is more complicated, because both rows and columns need to be passed, and they are not all the same length. However, the total amount of data that has to be passed is the same as for the Schur matrix, and it is distributed roughly half and half between column and row communication. Because of the split between row and column communication, the average message is only half as long, so the latency is doubled. Thus the overhead for border communication in H is

$$\frac{2N}{L}\left[\alpha + N/C\left(\left\lceil \frac{S}{R} \right\rceil\right)\beta\right] + \frac{2N}{L}\left[\alpha + N/R\left(\left\lceil \frac{S}{C} \right\rceil\right)\beta\right]. \tag{4.5}$$

We can remove the entire latency term in (4.4) by combining the vertical communication of Q with the vertical communication of H.

4.2.4. Bundling and other overheads. Before a bundled horizontal transform broadcast can take place, one processor must compute the next B Householder transforms. Once these are computed, they are broadcast horizontally and vertically so that a processor's row and column can participate in the subsequent computation. This means that the computation of these transforms is on the critical path.

Each bulge must be advanced B steps, which requires doing the entire Francis iteration on a $(B+2) \times (B+2)$ submatrix. This requires time approximately

$$\left[BV + 10B^2\right]\gamma.$$

This is what forces B to remain small since it is done by only one node. It happens N/B times per bulge chase. Notice, however, that if R and C are relatively prime, and S is their least common multiple (that is, $S = P = R * C$), then all the nodes can do this step in parallel.

Let $\texttt{lcm}(R,C)$ denote the least common multiple of R and C. Since diagonal blocks will repeat every $\texttt{lcm}(R,C)$, we see that the overall overhead must look something like

$$\left\lceil \frac{S}{\texttt{lcm}(R,C)} \right\rceil N\left[V + 10B\right]\gamma. \tag{4.6}$$

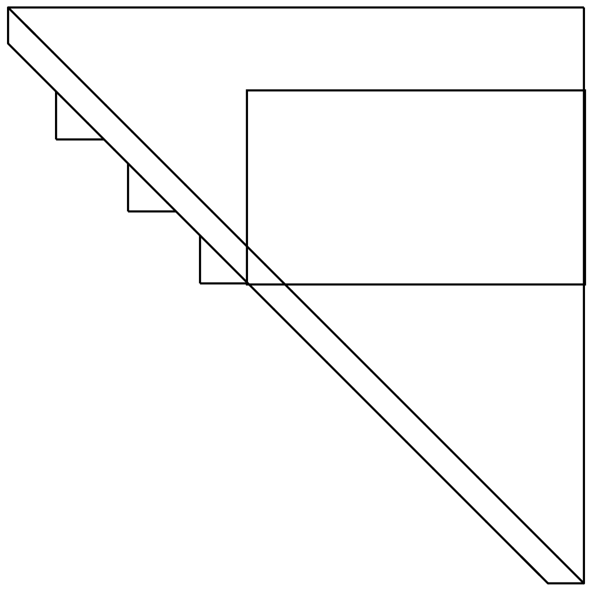

FIG. 8. *Delaying transforms to reduce the pipeline fill.*

4.2.5. Pipeline start-up and wind-down. In Figure 8, we suppose there are three processor rows and columns ($R = C = 3$) and three bulges ($S = 3$). Normally, it is not until the third bulge starts that all nine processors are occupied. Until then, there are pipeline start-up costs. Similarly, at the end of the iteration, there will be wind-down costs.

To one familiar with parallel linear algebra, the first instinct is to dismiss this overhead or to include a small "fudge factor" in some modeling. In cases such as a parallel LU decomposition [9, 10], for example, there is a pipeline start-up when doing the horizontal broadcast of the multipliers around a ring. The key difference, however, is that in that case all the nodes are busy while the pipe is beginning to grow. In this case, there are nodes completely idle until enough bulges have been created.

The key observation to make in Figure 8 is that the boxed area does not need to be updated until all three bulges are going. This blocking can be used to drastically reduce the pipeline start-up and wind-down. Since our current implementation does not do this, for the remainder of this section we will assume this is not done.

Another observation to make is that if one uses more bulges S than are required to keep everyone busy, then the pipeline start-ups become less important. Furthermore, any blocking done as suggested by Figure 8 reduces pipeline start-up but *not* wind-down.

Since we have already modeled the total horizontal and vertical contribution time in section 4.2.2, we would like to examine now the wasted time of nodes going idle if the code is unblocked. We start by examining the horizontal impact of pipeline start-up.

There are N/L block rows of the matrix. Assuming N is much larger than L, the number of horizontal flops for the first R transforms will be roughly $10NL/C$. As one marches down the submatrix, the horizontal work decreases, but it is initially at its largest. Although it is not required in the equations below, we assume for simplicity of introducing them that $S = R = C$. The total horizontal work has already been shown to be $5N^2S$ flops. The total horizontal time was assumed previously to be $(5N^2/C)\gamma$ when $S = R$. The total speed-up then is $R \times C$, which is perfect.

It is clear that the first bulge cannot have this ideal speed-up, and hence we now introduce additional time terms to reflect wasted time. The first bulge, for example, working on the first row, can only be done by one row of processors. The time spent is $10NL/C\gamma$. The ideal formula suggests the time should have been $10NL/(RC)\gamma$. We must therefore consider the wasted time given by the real time minus the ideal time. This is

$$\frac{10L}{RC}(NR - N)\gamma.$$

When the first bulge reaches the second row, the second bulge can start. The two bulges require $10NL$ and $10N(N - L)$ flops to do the horizontal work. The ideal time for this would be $(20NL - 10L^2)/(RC)\gamma$. The real time it takes, however, is the time it takes to do the most work on one row of processors (namely, the first row again). This time is again $10NL/C\gamma$. The wasted time is then

$$\frac{10L}{RC}(NR - 2N - L)\gamma.$$

When we continue this analysis for three bulges, we see that the flops required are $10NL$, $10N(N - L)$, and $10N(N - 2L)$. The ideal time would be $(30NL - 30L^2)/(RC)\gamma$. The wasted time is then

$$\frac{10L}{RC}(NR - 3N - 3L)\gamma.$$

Continuing as such we see the wasted time for when there are four bulges to be

$$\frac{10L}{RC}(NR - 4N - 6L)\gamma.$$

This continues for $(R - 1)$ steps until there are enough bulges to keep all the nodes busy. In general then, the wasted horizontal time for starting bulge j is

$$\frac{10L}{RC}\left(NR - \frac{3}{2}j + \frac{j^2}{2}\right)\gamma.$$

The total horizontal start-up wasted time over all the above equations is

$$\frac{10LR}{C}\left(N - \frac{3}{4} - \frac{1}{6}R\right)\gamma. \tag{4.7}$$

For simplicity, we ignore horizontal wind-down time as well as vertical start-up time. The only remaining item is the vertical wind-down time. This looks a lot like the horizontal equation (4.7), except that timings are approximately $(20NL/R)\gamma$. We then express the final vertical equivalent of (4.7) as

$$\frac{20LC}{R}\left(N - \frac{3}{4} - \frac{1}{6}C\right)\gamma.$$

The total overhead for this section is therefore

$$\frac{10L}{RC}\left(R^2N - \frac{3}{4}R^2 - \frac{1}{6}R^3 + 2C^2N - \frac{3}{2}C^2 - \frac{1}{3}C^3\right)\gamma. \tag{4.8}$$

4.3. Overall model. Because the algorithm is iterative, analysis of its overall performance is difficult. We can only guess how many total iterations will be needed; faster convergence will be achieved for some matrices but not others. Furthermore, different matrices have different deflation patterns, making it hard to model the reduction in size of the active matrix as the algorithm proceeds. For these reasons, we take a very crude approach to modeling overall performance. We shall assume that it takes about four double iterations (bulge chases) to deflate a pair of eigenvalues. At this rate, the entire job will take $2N$ bulge chases. Results in section 2.4 suggest that this may be an overestimation. Let us assume that these bulge chases are arranged into $2N/S$ superiterations of S bulges each. We assume further that each superiteration acts on the whole matrix. That is, we ignore deflations. This extremely pessimistic assumption ensures that we will not overstate the performance of the algorithm. It also excuses us from considering load imbalances that arise as the size of the active matrix is decreased by deflations. Each deflation causes a portion of the arrays holding H and Q to become inactive. As large portions of the arrays become inactive, processors begin to fall idle. Our model compensates for this effect by pretending that the computations are all carried out on the entire matrix; i.e., no portion of the matrix ever becomes inactive.

This approach gives us no information about the efficiency of the algorithm in terms of processor use relative to the actual flop count, but we believe it gives a reasonable estimate of the execution time of the algorithm.

The total time to execute one superiteration is obtained by summing (4.1), (4.3), (4.4), (4.5), (4.6), and (4.8). In terms of load balance, there are some clear advantages to taking $C \neq R$ (e.g., $\mathtt{lcm}(R, C) = 1$). However, in order to make the expressions tractable, we will now make the assumption $C = R$. We will also assume that $S = kR$, where k is an integer. This is a good choice for efficient operation. Summing all the expressions, we obtain the time for one superiteration,

(4.9)
$$\frac{20N^2k\gamma}{R} + \frac{2N}{B}\log(R)(\alpha + 3Bk\beta) + \frac{2N}{L}\left(3\alpha + \frac{4Nk\beta}{R}\right) + kN(V + 10B)\gamma + 30NL\gamma.$$

These terms correspond to flop count, broadcast overhead, border communication, bundling overhead, and pipeline start-up/wind-down, respectively. We have simplified the last term by ignoring two small negative terms in (4.8).

The expression (4.9) reflects the tradeoffs that we have already noted. B needs to be big enough so that broadcast communication is not dominated by latency, but not so big that it causes serious bundling overhead. L needs to be big enough so that border communication is not too expensive, but not so big that the pipeline start-up costs become excessive.

4.4. Scalability. Let us investigate how well the algorithm scales as $N \to \infty$. For simplicity we consider here the task of computing eigenvalues only. Similar (but better) results hold for the task of computing the complete Schur form. As we shall see, the algorithm is ultimately not scalable, but it is nearly scalable for practical values of N.

Since the amount of data is $O(N^2)$, the number of processors must be $O(N^2)$. Assuming the run time on a single processor is $O(N^3)$, the parallel run time should ideally be $O(N)$.

The processors are logically organized into R rows and C columns. We shall continue to assume that $R = C$. Thus we must have $R = O(N)$. Let us say $R = \rho N$, where ρ is some fixed constant satisfying $0 < \rho \ll 1$. (For example, in the runs shown in Table 2 in section 6 we have $\rho = 1/1800$.)

As before, let $S = kR$ be the number of double steps per superiteration. We have $S = \sigma N$, where $\sigma = k\rho$. Assuming $2N$ iterations suffice, the number of superiterations will be about $2N/(\sigma N) = 2/\sigma = O(1)$. Thus we shall assume that the total number of superiterations is $O(1)$. Since the figure $2N$ is only a rough estimate of the number of iterations, let us not commit to the figure $2/\sigma$ quite yet. For now we shall let q denote the number of superiterations required and assume that it is independent of N.

Let $\tau(N)$ denote the time to do one superiteration, assuming that one can get the shifts for free. This is about the same as (4.9), except that now we are just considering the time to calculate the eigenvalues. Thus we cut the flop count and the border communication in half. Let us assume that B and L are large enough so that we can ignore the latency terms. Then

$$\tau(N) = K_1 N \log \rho N + K_2 N + O(1),$$

where $K_1 = 6k\beta$ and

$$K_2 = \left[\frac{k}{\rho} + 3L + 10B\right] 10\gamma + \frac{4k}{L\rho}\beta.$$

Since ρ is tiny, we normally have $K_1 \ll K_2$. Thus the $N \log N$ term does not dominate $\tau(N)$ until N is enormous. The assumption that the shifts are free is also reasonable unless N is enormous. For example, the largest run listed in Table 2 (section 6) required computation of the eigenvalues of a 32×32 submatrix as shifts. This is a relatively trivial subtask when considered independently of the overall problem, considering that the dimension of the matrix is $N = 14400$.

We conclude that even for quite large N the execution time will be well approximated by $q\tau(N)$ and will appear to scale like $O(N)$. That is, the algorithm will appear to be scalable.

Only when N becomes really huge must the cost of computing shifts be taken into account. Eventually the submatrix whose eigenvalues are needed as shifts will be large enough that its eigenvalues should also be computed in parallel. Let us assume that the algorithm performs this computation by calling itself. Let $T(N)$ denote the time to compute the eigenvalues of a matrix of order N, including the cost of computing shifts. The shift computation for each superiteration consists of computing the eigenvalues of a matrix of order $W = 2\sigma N$, so it takes time $T(2\sigma N)$. Thus, making the simplification $\tau(N) = K_3 N \log N$ ($K_3 = K_1 + K_2$), we see that each superiteration takes time $K_3 N \log N + T(2\sigma N)$, so

$$T(N) = qK_3 N \log N + qT(2\sigma N).$$

We can calculate $T(N)$ by unrolling this recurrence. We have

$$T(N) \leq qK_3 N \log N (1 + 2\sigma q + (2\sigma q)^2 + \cdots + (2\sigma q)^j) = qK_3 N \log N \left(\frac{(2\sigma q)^{j+1} - 1}{2\sigma q - 1}\right),$$

where j is the depth of the recursion. Notice that if we had $2\sigma q < 1$, we could say that the geometric progression is bounded by

$$\frac{1}{1 - 2\sigma q}.$$

This would make $T(N) = O(N \log N)$, and the algorithm would be scalable, except for the insignificant factor $\log N$. Unfortunately $2\sigma q$ seems to be greater than 1, so this argument is not valid. If we assume $q = 2\sigma$, as suggested above, we have $2\sigma q = 4$. We are saved by the fact that the recursion is not very deep. An upper bound on j is given by $(2\sigma)^j N \geq 1$ or $j \leq -\frac{\log N}{\log 2\sigma}$. Making this substitution for j, we find that

$$T(N) \approx \frac{qK_3}{2\sigma q - 1} N^{1+\delta} \log N,$$

where

$$\delta = -\frac{\log 2\sigma q}{\log 2\sigma} > 0.$$

This shows that the algorithm is ultimately not scalable, but it is not a bad result if σ is small. If we assume $2\sigma q = 4$ and take $\sigma = 1/900$ (as in Table 2), then we have $T(N) = O(N^{1.23} \log N)$, which is not too much worse than $O(N)$. The assumption $2\sigma q = 4$ is actually a bit pessimistic. If we assume that the total number of iterations to convergence is $4N/3$, as suggested by (4), then we have $q = 4/(3\sigma)$, and $2\sigma q = 8/3$. Then we get $T(N) = O(N^{1.16}) \log N$. As long as we assume that $2\sigma q$ is constant, the power of N approaches 1 as $\sigma \to 0$. In this sense the algorithm is nearly scalable if σ is kept small.

5. Accuracy results. Our implementation of the parallel multishift QR algorithm can be found in the current version of the general purpose parallel linear algebra package ScaLAPACK [9, 10]. ScaLAPACK is built on top of a portable communications layer called the BLACS [54, 21]. BLACS have been implemented under PVM [25] and MPI [44], among many other communication standards.

As far as accuracy is concerned, this method has the same advantages as standard QR implementations. Each iteration involves the application of an orthogonal similarity transformation. Thus the algorithm is normwise backward stable.

Figure 9 provides computational affirmation of stability. ScaLAPACK runs were done on various matrices of size 50 to 1800. In all cases, the matrix was filled in by the random number generator PDMATGEN found in ScaLAPACK and then set to upper Hessenberg form. Two residual quantities are computed:

$$\|H - QSQ^T\| / (\|H\| * eps * N)$$

and

$$\|I - Q^T Q\| / (eps * N).$$

Here S is the quasi-triangular Schur matrix, H is the original Hessenberg matrix, and Q is the orthogonal transform matrix computed by PDLAHQR in ScaLAPACK. The infinity matrix norm is used. Results in Figure 9 include single and parallel runs from 1 to 8 nodes, in all combinations of row and column distributions (R and C), with block sizes L varying from 1 to 50. The accuracy results did not vary with L, R, C and, if anything, the results became more desirable for larger N.

6. Performance results. For most of the results in this section, only a single superiteration was performed. We have found that doing a single iteration tends to represent overall performance when we have run problems to completion. The number of bulges was set at twice the least common multiple of R and C (`lcm`(R, C)). For a

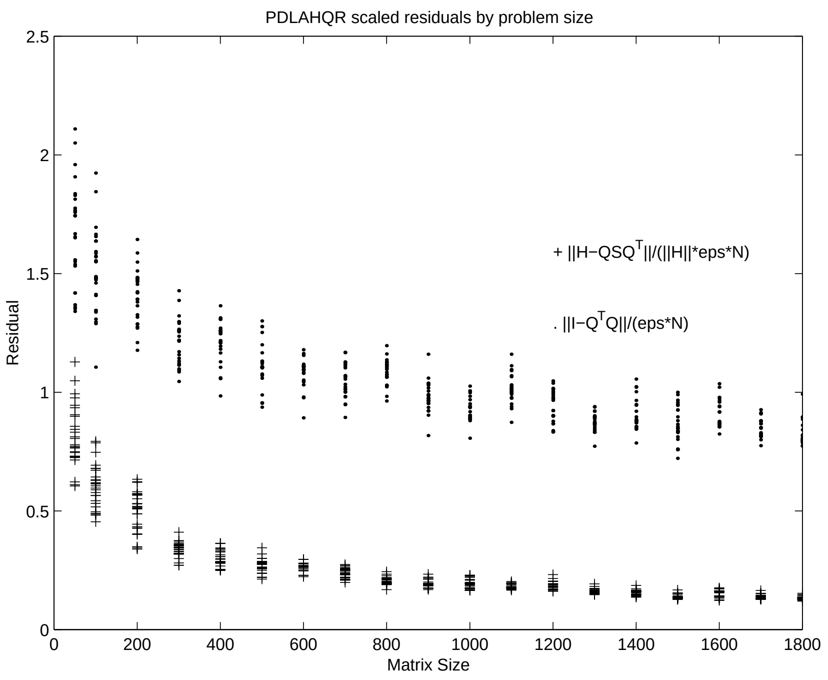

FIG. 9. *Accuracy plots for sample problem sizes.*

square number of nodes, $R = C$. In all cases, the same amount of memory per node was used (including the temporary scratch space). We provide the problem size and the efficiency as compared to $\_LAHQR$ from LAPACK [1].

In Table 2, we see the results for doing the first superiteration of a complete Schur decomposition on an Intel Paragon supercomputer running OSF R1.4. The serial performance of DLAHQR in this case was 8.5 Mflops (compiled with −Knoieee $-O4$ −Mnoperfmon on the R5.0 compilers). Note that these results, currently found in ScaLAPACK version 1.6, are better than those previously released in ScaLAPACK version 1.2 ALPHA (which corresponded to the technical note [38]). The serial performance of this code is around 10 Mflops, and efficiency results are reported against this and not DLAHQR. Had they been reported against DLAHQR, the 64-node job would have shown a speed-up of 66.6. The reason for the enhanced performance is the block application of Householder vectors in groups of 2 or 3. The reason for the blip in performance between 9 nodes and 16 nodes was that this was the arbitrary cutoff for doing the border communications in parallel (see section 3.2).

We consider the results in Table 2 encouraging. Efficiencies remained basically the same throughout all the runs, and the overall performance was in excess of the serial code it was modeled after.

We briefly compare this algorithm to the first successful parallel QR algorithm in [37]. That algorithm achieved maximum performance when using 96 nodes, after which the nonscalability caused performance degradation. The new algorithm achieves faster performance on 49 nodes and appears to scale on the Intel Paragon supercomputer.

In Table 3, we see results analogous to those in Table 2 but running just an eigenvalue-only version of the code. In this case, serial performance on the Intel

Table 2
PDLAHQR Schur decomposition performance on the Intel Paragon supercomputer.

Nodes	N	L	B	Efficiency
1	1800	100	30	1.00
4	3600	100	30	0.92
9	5400	100	30	0.88
16	7200	100	30	0.90
25	9000	100	30	0.89
36	10800	100	30	0.89
49	12600	100	30	0.89
64	14400	100	30	0.88

Table 3
PDLAHQR eigenvalue-only performance on the Intel Paragon supercomputer.

Nodes	N	L	B	Efficiency
4	3600	100	30	0.96
9	5400	100	30	0.90
16	7200	100	30	0.93
25	9000	100	30	0.92
36	10800	100	30	0.91
49	12600	100	30	0.91
64	14400	100	30	0.90

Paragon system was around 8.2 Mflops (for roughly half the flops). These positive results may lead to even better methods in the future, since combining the use of HQR for finding eigenvalues with new GEMM-based inverse iteration methods for finding eigenvectors [35] might lead to completing the spectrum significantly faster than the results in Table 2.

Furthermore, there are better load balancing properties to the eigenvalue-only code on a Cartesian mapping. Some runs taken to completion on this version of the code have better efficiencies on the overall problem than any of their analogous timings given in the rest of these tables. The only reason why we do not include these timings here is that we currently have no means of testing the accuracy of the solution, whereas all the other runs in the rest of the tables are tested by applying the computed Schur vectors QTQ^T on the Schur matrix T and ensuring that the result is close to the original Hessenberg matrix H.

The code also works with comparable efficiency for a wide range of choices of $R \neq C$. We do not wish it to be misunderstood that just because we have simplified many equations with $R = C$ that the code only works, or even only works well, under the condition that the number of nodes is square. In fact, the code performs within the same ranges and efficiencies for any number of nodes less than 64 (with the possible minor—around 10%—performance hits to odd-balls like 17, 19, etc.).

In Table 4 we see results of runs on a portion of Cornell's IBM SP2 supercomputer. On this machine, efficiencies did tend to drop as we increased the number of nodes. We found some of the timings erratic, and we believe part of the problem was lack of dedicated time on the machine since these numbers were generated in an interactive pool with others running other programs at the same time. These were done on thin nodes.

In Table 5 we see results of runs on a portion of Intel's new ASCI Option Red Teraflops technology supercomputer. We ran problems to completion on this machine. Despite running problems to completion, the Mflops reported correspond to actual

Table 4
PDLAHQR Schur decomposition performance on the IBM SP2 supercomputer.

Nodes	N	L	B	Mflops	Efficiency
1	1000	1000	500	47	1.00
2	1600	200	100	73	0.78
4	2000	250	100	147	0.78
6	3600	300	150	176	0.62
9	3600	200	100	283	0.67
12	3600	300	150	345	0.61

Table 5
PDLAHQR Schur decomposition on the Intel ASCI Option Red supercomputer.

Nodes	N	Mflops	Seconds	Efficiency
1	600	56		1.00
4	2200	156	1098	0.69
9	3300	316	1679	0.63
16	4400	542	2109	0.60
25	5500	815	2726	0.58
36	6600	1104	3381	0.55
49	7700	1392	3983	0.51
64	8800	1714	4886	0.48
81	9900	2052	5966	0.45
100	11000	2390	6954	0.43
121	12100	2757	7884	0.41
144	13200	3137	9290	0.39
169	14300	3464	10858	0.37
196	15400	3865	11894	0.35
225	16500	4180	13698	0.33

flops computed. All problems except the problem run on 1 node used the exact same amount of memory per node. We also did an $N = 18000$ node run on 96 nodes (in an 8×12 configuration) that completed, with the right answer, in 34299 seconds. For all the parallel runs, L was 100, and B was 25. The number of flops in these runs tended to drop from $16N^3$ or so to around $12N^3$, where it leveled off. Because of dropping flops, the time to solution scales better than the parallel efficiency suggests.

In Table 6 we see results of runs on an SGI Origin 2000 supercomputer using 195 Mhz IP27 processors with an instruction and data cache of 32Kbytes. We also ran problems to completion, and we still report the true Mflops for the actual work done. In addition, we have added a column to represent the number of N^3 flops each problem required, as well as a column reporting speed-up. We varied L slightly, so we also reported that. B was around 30 to 50, depending on L. The performance of the code on this machine was impressive even though we made no effort for optimization, and no compiler optimization flags were specified.

7. Conclusions.

In this paper, we present new results for the parallel nonsymmetric QR eigenvalue problem. These new results demonstrate that this method is competitive and has a reasonable efficiency.

This code is available in ScaLAPACK version 1.6 as pdlahqr.f (double precision) or pslahqr.f (single precision).

Acknowledgments. This work was initiated when Greg Henry was visiting the University of Tennessee at Knoxville, working with the ScaLAPACK project. We thank everyone involved in that project. Parallel runs were made on an Intel Paragon XP/S Model 140 supercomputer, the IBM SP2 at Cornell University and the IBM

TABLE 6
PDLAHQR Schur decomposition on the SGI Origin 2000.

Nodes	N	L	Mflops	N^3	Seconds	Efficiency	Speed-up
1	1000	100	91	16.7	182	1.00	1.0
4	2000	125	277	14.9	431	0.77	3.0
9	3000	100	556	14.0	681	0.68	6.1
16	4000	125	841	13.6	1037	0.58	9.2
25	5000	100	1252	13.5	1352	0.55	13.8

SP2 at Oak Ridge National Laboratory, and the SGI Origin 2000 at Los Alamos National Lab. The SGI runs were made with the wonderful assistance of Patrick Fay. We thank Intel, IBM, SGI, LANL, ORNL, and Cornell University.

REFERENCES

[1] E. Anderson, Z. Bai, S. Blackford, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorenson, *LAPACK Users' Guide*, 3rd ed., SIAM, Philadelphia, PA, 1999.

[2] L. Auslander and A. Tsao, *On parallelizable eigensolvers*, Adv. Appl. Math., 13 (1992), pp. 253–261.

[3] Z. Bai and J. Demmel, *On a block implementation of Hessenberg multishift QR iteration,* Internat. J. High Speed Comput., 1 (1989), pp. 97–112.

[4] Z. Bai and J. Demmel, *Design of a parallel nonsymmetric eigenroutine toolbox, Part* I, in Parallel Processing for Scientific Computing, R. Sincovec, D. Keyes, M. Leuze, L. Petzold, and D. Reed, eds., SIAM, Philadelphia, 1993, pp. 391–398.

[5] Z. Bai and J. Demmel, *Design of a Parallel Nonsymmetric Eigenroutine Toolbox, Part* II, Research Report 95-11, Department of Mathematics, University of Kentucky, Lexington, KY, 1995.

[6] Z. Bai, J. Demmel, J. Dongarra, A. Petitet, H. Robinson, and K. Stanley, *The spectral decomposition of nonsymmetric matrices on distributed memory parallel computers*, SIAM J. Sci. Comput., 18 (1997), pp. 1446–1461.

[7] Z. Bai, J. Demmel, and M. Gu, *An inverse free parallel spectral divide and conquer algorithm for nonsymmetric eigenproblems*, Numer. Math., 76 (1997), pp. 279–308.

[8] M. W. Berry, J. J. Dongarra, and Y. Kim, *A highly parallel algorithm for the reduction of a nonsymmetric matrix to block upper-Hessenberg form*, Parallel Comput., 21 (1995), pp. 1189–1212.

[9] S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance*, Technical Report CS-95-283, LAPACK Working Note 95, University of Tennessee at Knoxville, 1995; in Proceedings of Supercomputing '96, IEEE Computer Society Press, Los Alamitos, CA, 1996, CD-ROM.

[10] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, SIAM, Philadelphia, 1997.

[11] R. Byers, *Numerical stability and instability in matrix sign function based algorithms*, in Computational and Combinatorial Methods in Systems Theory, C. Byrnes and A. Lindquist, eds., North-Holland, Amsterdam, 1986, pp. 185–200.

[12] D. Boley and R. Maier, *A Parallel QR Algorithm for the Nonsymmetric Eigenvalue Problem*, Technical Report TR-88-12, Department of Computer Science, University of Minnesota at Minneapolis, 1988.

[13] S. Chakrabarti, J. Demmel, and K. Yelick, *Modeling the benefits of mixed data and task parallelism*, in Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, New York, 1995.

[14] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, *A set of level* 3 *basic linear algebra subprograms*, ACM Trans. Math. Software, 16 (1998), pp. 1–17.

[15] J. J. Dongarra, G. A. Geist, and C. H. Romine, *Fortran subroutines for computing the eigenvalues and eigenvectors of a general matrix by reduction to general tridiagonal form*, ACM Trans. Math. Software, 18 (1992), pp. 392–400.

[16] J. J. Dongarra, S. Hammarling, and D. C. Sorensen, *Block reduction of matrices to condensed forms for eigenvalue computations*, J. Comput. Appl. Math., 27 (1989), pp. 215–227.

[17] J. J. Dongarra, L. Kaufman, and S. Hammarling, *Squeezing the most out of eigenvalue solvers on high-performance computers,* Linear Algebra Appl., 77 (1986), pp. 113–136.

[18] J. J. Dongarra and M. Sidani, *A parallel algorithm for the nonsymmetric eigenvalue problem*, SIAM J. Sci. Comput., 14 (1993), pp. 542–569.

[19] J. J. Dongarra and R. A. van de Geijn, *Reduction to condensed form on distributed memory architectures,* Parallel Comput., 18 (1992), pp. 973–982.

[20] J. J. Dongarra, R. A. van de Geijn, and D. Walker, *A Look at scalable dense linear algebra libraries*, in Scalable High-Performance Computing Conference, IEEE Press, Piscataway, NJ, 1992, pp. 372–379.

[21] J. J. Dongarra and R. Whaley, *BLACS User's Guide* V1.0, Technical Report CS-95-281, University of Tennessee, Knoxville, TN, 1995.

[22] A. Dubrulle, *The Multishift QR Algorithm– Is it Worth the Trouble?*, IBM Scientific Center Technical Report Draft, Palo Alto, CA, 1992.

[23] P. J. Eberlein, *On the Schur decomposition of a matrix for parallel computation*, IEEE Trans. Comput., C-36 (1987), pp. 167–174.

[24] J. G. F. Francis, *The QR transformation: A unitary analogue to the LR transformation, Parts* 1 *and* 2, Comput. J., 4 (1961), pp. 265–272; 332–345.

[25] G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA, 1994.

[26] G. A. Geist and G. J. Davis, *Finding eigenvalues and eigenvectors of unsymmetric matrices using a distributed memory multiprocessor*, Parallel Comput., 13 (1990), pp. 199–209.

[27] G. A. Geist, R. C. Ward, G. J. Davis, and R. E. Funderlic, *Finding eigenvalues and eigenvectors of unsymmetric matrices using a hypercube multiprocessor*, in Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, G. Fox, ed., ACM, New York, 1988, pp. 1577–1582.

[28] G. Golub and C. Van Loan, *Matrix Computations,* 3rd ed., The Johns Hopkins University Press, Baltimore, MD, 1996.

[29] A. Greenbaum and J. Dongarra, *LAPACK Working Note* 17*: Experiments with QR/QL Methods for the Symmetric Tridiagonal Eigenvalue Problem*, Technical Report CS-89-92, Computer Science Department, The University of Tennessee, Knoxville, TN, 1989.

[30] A. Gupta and V. Kumar, *On the scalability of FFT on parallel computers*, in Proceedings of the Frontiers 90 Conference on Massively Parallel Computation, IEEE Computer Society Press, Piscataway, NJ, 1990.

[31] D. E. Heller and I. C. F. Ipsen, *Systolic networks for orthogonal equivalence transformations and their applications,* in Proceedings of the Conference on Advanced Research in VLSI, MIT, Cambridge, MA, 1982, pp. 113–122.

[32] B. A. Hendrickson and D. E. Womble, *The torus-wrap mapping for dense matrix calculations on massively parallel computers*, SIAM J. Sci. Comput., 15 (1994), pp. 1201–1226.

[33] G. Henry, *Improving the unsymmetric parallel QR algorithm on vector machines*, in Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing, R. F. Sincover et al., eds., SIAM, Philadelphia, 1993, pp. 353–357.

[34] G. Henry, *The Shifted Hessenberg System Solve Computation*, Technical Report CTC94TR163, Cornell Theory Center, Ithaca, NY, 1994.

[35] G. Henry, *A parallel unsymmetric inverse iteration solver*, in Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing, D. H. Bailey et al., eds., SIAM, Philadelphia, 1994, pp. 546-551.

[36] G. Henry, *Improving Data Re-Use in Eigenvalue-Related Computations*, Ph.D. Thesis, Cornell University, Ithaca, NY, 1994.

[37] G. Henry and R. van de Geijn, *Parallelizing the QR algorithm for the unsymmetric algebraic eigenvalue problem: Myths and reality,* SIAM J. Sci. Comput., 17 (1996), pp. 870–883.

[38] G. Henry, D. Watkins, and J. Dongarra, *A Parallel Implementation of the Nonsymmetric QR Algorithm for Distributed Memory Architectures*, Technical Report CS-97-352, Computer Science Department, University of Tennessee, Knoxville, TN, 1997.

[39] E. R. Jessup, *A case against a divide and conquer approach to the nonsymmetric eigenvalue problem*, J. Appl. Numer. Math., 12 (1993), pp. 403–420.

[40] B. Kågström and C. F. Van Loan, *GEMM-Based Level-3 BLAS*, Technical Report CTC91TR47, Cornell Theory Center, Ithaca, NY, 1991.

[41] L. Kaufman, *A parallel QR algorithm for the symmetric tridiagonal eigenvalue problem,* J. Parallel Distrib. Comput., 23 (1994), pp. 429–434.

[42] S. Huss-Lederman, A. Tsao, and T. Turnbull, *A parallelizable eigensolver for real diagonalizable matrices with real eigenvalues*, SIAM J. Sci. Comput., 18 (1997), pp. 869–885.

[43] B. Smith, J. Boyle, J. Dongarra, B. Garbow, Y. Ikebe, V. Klema, and C. Moler, *Matrix Eigensystem Routines - EISPACK Guide,* 2nd Ed., Lecture Notes in Comput. Sci. 6, Springer-Verlag, New York, 1976.

[44] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*, MIT Press, Boston, 1996.

[45] G. W. Stewart, *A parallel implementation of the QR algorithm*, Parallel Comput., 5 (1987), pp. 187–196.

[46] R. A. van de Geijn, *Implementing the QR-Algorithm on an Array of Processors*, Ph.D. Thesis, Department of Computer Science, University of Maryland, Baltimore, MD, 1987.

[47] R. A. van de Geijn, *Storage schemes for parallel eigenvalue algorithms*, in Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms, G. Golub and P. Van Dooren, eds., Springer Verlag, New York, 1988, pp. 639–648.

[48] R. A. van de Geijn, *Deferred shifting schemes for parallel QR methods,* SIAM J. Matrix Anal. Appl., 14 (1993), pp. 180–194.

[49] R. A. van de Geijn and D. G. Hudson, *An efficient parallel implementation of the nonsymmetric QR algorithm*, in Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications, ACM, New York, 1989, pp. 697–700.

[50] D. S. Watkins, *Fundamentals of Matrix Computations*, John Wiley and Sons, New York, 1991.

[51] D. S. Watkins, *Shifting strategies for the parallel QR algorithm*, SIAM J. Sci. Comput., 15 (1994), pp. 953–958.

[52] D. S. Watkins, *The transmission of shifts and shift blurring in the QR algorithm*, Linear Algebra Appl., 241/243 (1996), pp. 877–896.

[53] D. S. Watkins and L. Elsner, *Convergence of algorithms of decomposition type for the eigenvalue problem*, Linear Algebra Appl., 143 (1991), pp. 19–47.

[54] R. C. Whaley, *Basic Linear Algebra Communication Subpprograms: Analysis and Implementation across Multiple Parallel Architectures*, Technical Report CS-94-234, Computer Science Department, University of Tennessee, Knoxville, TN, 1994.

[55] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford University Press, Oxford, UK, 1965.

[56] L. Wu and E. Chu, *New distributed-memory parallel algorithms for solving nonsymmetric eigenvalue problems,* in Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing, D. H. Bailey et al., eds., SIAM, Philadelphia, 1995, pp. 540–545.