A Parallel Implementation of the Nonsymmetric QR Algorithm for Distributed Memory Architectures

Greg Henry* David Watkins† Jack Dongarra‡

March 11, 1997

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 made 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, and the Intel ASCI Option Red Supercomputer are reported.

Key Words: Parallel computing, eigenvalue, Schur decomposition, QR algorithm
AMS (MOS) Subject Classification: 65F15, 15A18

1 Introduction

Over the years many methods for solving the parallel unsymmetric eigenvalue problem have been suggested. Most of these methods have serious drawbacks, either in terms of stability, accuracy, scalability, or requiring extra work. This paper describes a version of the QR algorithm [21] that has significantly better scaling properties than earlier versions, as well as being 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 [24, 44]. The original version due to J. G. F. Francis [21], 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 top to bottom of the matrix to complete the iteration.

*Computer Science, 111 Ayres Hall, University of TN, 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. Supported by the National Science Foundation under grant DMS-9403569

‡Computer Science, 111 Ayres Hall, University of TN, Knoxville, TN 37996-1301, dongarra@cs.utk.edu

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 equally well get 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 two and chase them one after the other down the subdiagonal in parallel. In principle this procedure should give the same result as a multishift iteration, but in practice (in the face of roundoff errors), our procedure performs much better [45].

Of the various parallel algorithms that have been proposed, the ones that have received 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 [36] 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, 10, 5, 7].)

Dongarra and Sidani [17] introduced tearing methods based on doing rank one updates to an unsymmetric Hessenberg matrix, resulting in two smaller problems, which are solved independently and then glued back together with a Newton iteration. This tends to suffer from stability problems since the two smaller problems might have arbitrarily worse condition than the parent problem [33].

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 [21, 1]. Matrix multiply methods tend to require many more flops, as well as sometimes encountering accuracy problems [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 [33]. Jacobi methods [24] have notoriously high flop counts. There are also methods by Dongarra, Geist, and Romine based on initial reductions to tridiagonal form [14, 48]. These might require fewer flops but they are plagued by instability. Against this competition, blocked versions of the implicit double shift QR algorithm [28, 31, 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: it lacks the vendor supported performance tuning of the BLAS (basic linear algebra subroutines [13, 34]), and it has data re-use similar to level-1 operations (it does O(n) flops on O(n) data [24].) This imposes an upper limit to how fast it can run on the high performance computers with a memory hierarchy. One attempt to rectify this problem was 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 for a GEMM-based (level-3 BLAS) algorithm to be used [3]. Unfortunately, this requires too many more flops and the GEMM itself has two of the three required dimensions very small [31]. However, even if a multishift QR algorithm is used without the additional matrix multiply (as was 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 [19] and Watkins [45, 46]. Because of this, a multishift size of $M = 6$ was implemented in LAPACK. It is not clear that this is faster than the double implicit shift QR when blocked [31].

Because of the difficulties in chasing large bulges, we restrict our analysis in this paper to bulges of degree two. 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 six).

The first attempts at parallelizing the implicit double shift QR algorithm were unsuccessful. See Boley et. al. [11], Geist et. al. [22, 23], Eberlein [20], and Stewart [38]. More successful methods came from vector implementations [16]. Usually, the key problem is to distribute the work evenly given its sequential nature.

A major step forward in work distribution was made by van de Geijn [40] in 1988. There, and in van de Geijn and Hudson [42], a wrap Hankel mapping was used to distribute the work evenly. A simple case of this, anti-diagonal mappings, was exploited in the paper by Henry and van de Geijn [32]. 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 indicing overheads and, in some cases, 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 [25]. That is, you could get 100 percent 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 [32] it was also proved that the standard double implicit shift QR algorithm (not just the one with anti-diagonal mappings) cannot be scalable. The same work also showed that if M shifts are employed to chase $M/2$ bulges of degree two, then the algorithm might be scalable as long as M was 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 [26, 38, 39, 41, 35]. However, it was long thought that this practice would require the use of out-of-date shifts, resulting in degradation of convergence [41]. What is new [45] (having seen [19]) 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 §3 it is necessary to review the difficulties in parallelizing the algorithm. This is done in §2.2. In §2.3 we make some modifications to overcome these difficulties. Finally, in §2.4 we give some experimental results to indicate the impact of the

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_0x is a multiple of e_1.
$H \leftarrow P_0HP_0$
for $i = 1, \ldots, n-2$
Compute P_i so that
P_iH has zero $(i+2, i)$ and
$(i+3, i)$ entries.
Update $H \leftarrow P_iHP_i$
Update $Q \leftarrow QP_i$
endfor

Figure 1: Sequential Single Bulge Francis HQR Step

changes.

2.1 Single Bulge

The double implicit Francis step [21] enables an iterative algorithm that goes from H (upper Hessenberg) to $H = QTQ^T$ (Schur decomposition [24, 48]). 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, 15, 18].

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 [24], 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 two is chased from top to bottom. Here, the Householder matrices are symmetric orthogonal transforms of the form:

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

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 [1] to the rows and columns of H corresponding to the submatrix; 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 after 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 Xs represent elements of the matrix, and the +s represent some bulge 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 be then 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 the maximum speed-up obtainable would be 6 [23]. Tricks to delay the application of the rows and/or column reflections appear to

[1] We use the term "HQR" to mean a practical Hessenberg QR iteration, for example, EISPACK's HQR code [37] or LAPACK's _HSEQR or _LAHQR code [1].

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 [27, 43]. To maximize the distribution of this computation, we could wrap our 2D 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 their product is 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 two times the minimum of R and C.

If one used an anti-diagonal mapping of the data [32], 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 [42]:

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

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 anti-diagonal distribution is ideal. Block diagonal mappings have been suggested [49], but these suffer from load imbalances that are avoided in the anti-diagonal 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 only involving 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 [48], 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 [48, 47]. 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, we must use out-of-date shifts. This practice results in subquadratic (although still superlinear) convergence [39, 41].

If we wish to chase many bulges at once without sacrificing quadratic convergence, we must change the shifting strategy. One of the strategies proposed in [3] was a generalization of the Wilkinson shift. Instead of choosing the two 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 two (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 [47] and has been observed in practice. We refer to each cycle of computing M shifts and chasing $M/2$ bulges as a *super-iteration*.

The question of how to determine the number of bulges $S = M/2$ per super-iteration is important. If one chooses $S = 1$, we have the standard double shift QR algorithm—but this has the scalability problems. If we choose S large enough, and the bulges are spaced appropriately, and we address these issues in the next section, then there are sufficient bulges to distribute the workload evenly. In fact, we later (§ 4.2.5) discuss 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 $O(S^3)$).

The general algorithm proceeds as described in Figure 2.

In Figure 2, the i index refers to the same i index as 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 started yet (when $i < 0$ or $i > n - 2$).

Here, we are spacing the bulges 4 columns apart, however it is clear that this spacing can be anything 4 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 super-iteration. This saves time because the information is in cache, but, more importantly, it reduces the number of iterations and total work. An entry that has become negligible early in a super-iteration might no longer meet the deflation criterion at the end of the super-iteration.

Another critical observation is that consecutive small subdiagonal elements may negatively impact convergence by washing out the effects of some of the shifts. Robust implementations of QR usually search for a pair of small subdiagonal elements along the unreduced submatrix

Multiple Bulge HQR Super-iteration
$e = \mathrm{eig}(H(n-m+1:n, n-m+1:n))$
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 $1 \leq i \leq n-2$
            Compute P_i so that
                $P_i H$ has zero $(i+2, i)$ and
                $(i+3, i)$ entries.
        if $i > n-2$ **then** $P_i = I$
        $H \leftarrow P_i H P_i$, $Q \leftarrow Q P_i$
    endfor
endfor

Figure 2: Sequential Multiple Bulge HQR Super-iteration

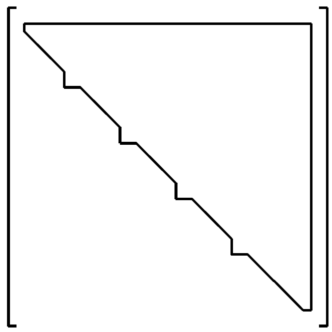

Figure 3: Pipelined QR steps

and attempt to start the bulge chasing from there. In the multishift and multi-bulge case, 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, and it could easily happen that 10 bulges were computed, but after the second bulge went through, the third bulge was unable to given that the subdiagonals became too large. We suggest maximizing the number of shifts that can go through, so that if the third bulge is unable to, we suggest skipping that one, and trying the fourth.

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 Serial 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 when something deflates off.

We are now ready to ask what is a reasonable way to compare the two algorithms in terms of work load.[2] 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

[2]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.

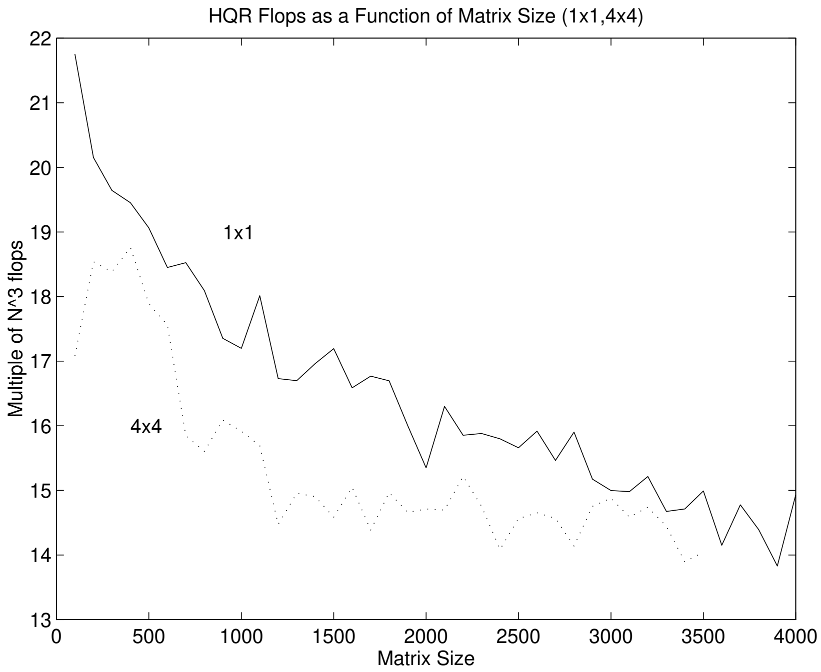

Figure 4: The Decreasing Average Flops for a 1x1 and 4x4 Grid

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 algorithm run on a 4×4 processor grid. The way the code is written, the number of bulges per super-iteration 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 [24] report the figure $25N^3$.

All of the numbers in Figure 4 are less than $25N^3$. More importantly there is a clear tendency, especially in the 1×1 case, for 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.[3] Thus one reckons 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.[4] This figure is derived as follows. The bulge chase is effected by application of $k-1$ Householder transformations of size 3×3. Each Householder transformation is applied to $N + 4$ rows/columns of the Hessenberg matrix H and

[3]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 pick should be viewed only as a rough average. In practice there is a great deal of variation.

[4]This is for the computation of the complete Schur form. If only the eigenvalues are wanted, the cost is $10k^2$ flops.

N columns of the transforming matrix Q. The cost of applying a Householder transformation to a single row/column is 10 flops. 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 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 gets 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 [48] 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 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 gives further support for 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 four (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 [48] that more iterations per eigenvalue are required in the earlier iterations (large matrices) 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.

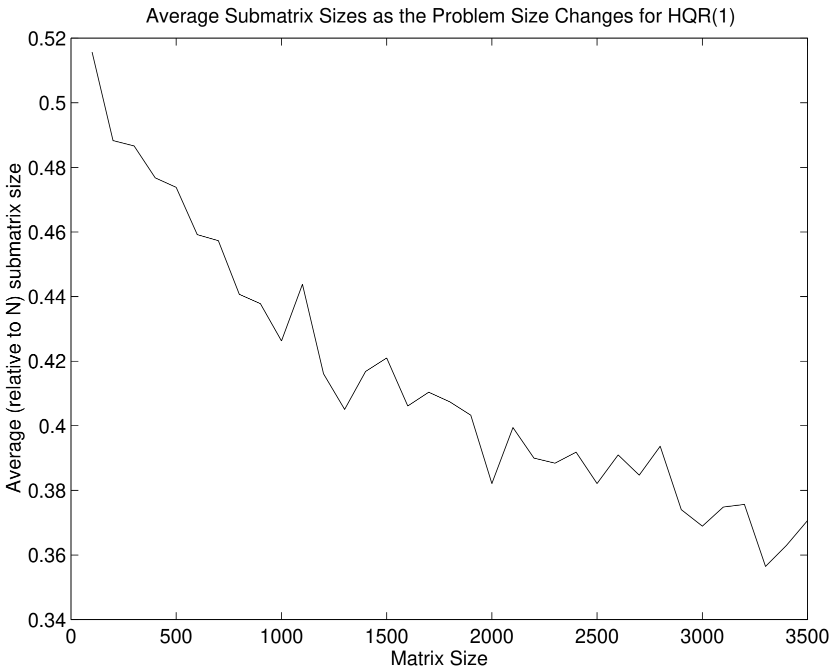

Figure 5: The Decreasing Average Submatrix Size

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 $P = R \times C$ 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 chopped into $H \times H$ blocks, which are parceled out to the processors by a torus wrap mapping. The bulges 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 must be large, since 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 super-iteration 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.

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 $H \times H$ blocks of the matrix, we must have $B \leq H$. 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 the 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.

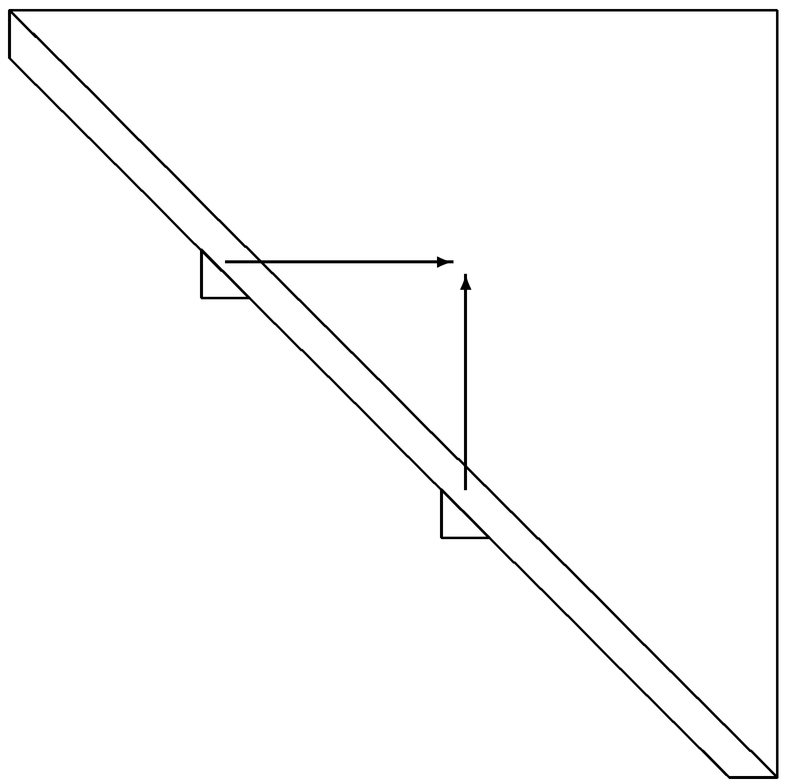

Figure 6: Multiple Bulge Broadcasts Colliding

3.1 Block Householder Transforms

Since Householder information arrives in bundles, we might as well apply the transformation in blocks to reduce 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 re-use fraction [31] is at best 10 flops per data element accessed in the limit. If B is one, which is the worst case, then only 3 flops are done per data element accessed. So, roughly one can improve the data re-use by a factor of 3 by applying these transforms simultaneously.

Due to diminishing returns, the data re-use is not significantly better when going from ten applications at a time to twenty. In fact, things are worse because one accesses almost twice the data for only a marginal improvement in data reuse. Because data 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 a 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 up all the bulge information to send it out, only to later re-load up all the information to receive it back in, might not justify the effort to parallelize the border communications.

This implies an entirely different approach to border communication should be used when there are a small number of rows (or columns) compared to a large number. The method currently implemented in our code is a hybrid approach. For a small number of rows or columns (three or less), 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 large 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 super-iteration 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 have the same amount of work. Thus it suffices to

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, $P = R \times C$
B	Number of Householder transforms per bundle
H	Blocking size of the matrix data on the 2D block torus wrap mapping.
V	Time to compute a single Householder transform of size three.
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 $N \gg W \geq 2S = M$

Table 1: Model Definitions

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 super-iteration 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 size H is small enough, compared to N, 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 super-iteration 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 is 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 four bulges ($S = 4$) and two rows ($R = 2$), one might except 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

$$\left(\frac{5N^2}{C}\lceil\frac{S}{R}\rceil + \frac{15N^2}{R}\lceil\frac{S}{C}\rceil\right)\gamma \tag{1}$$

For the special case of $S = R = C$, Equation 1 reduces to $(20N^2S/P)\gamma$ where the factor P in the denominator indicates perfect speedup. However, this expression and Equation 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

$$\frac{N-1}{B}\log(C)(\alpha + 3B\beta), \tag{2}$$

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

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

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/H border encounters, whose total overhead is thus

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

If $S > 1$ bulges are chased, up to C border crossings can be accomplished in parallel. If $S > C$, the most burdened processors will have to handle $\left\lceil \frac{S}{C} \right\rceil$ border crossings at a time. Each batch of $\left\lceil \frac{S}{C} \right\rceil$ 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

$$\frac{2N}{H}\left[\alpha + \frac{2N}{R}\left\lceil \frac{S}{C} \right\rceil \beta\right]. \tag{4}$$

The analysis of the border communication in the Hessenberg matrix H is more complicated, because both rows and columns need to 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

$$\frac{2N}{H}\left[\alpha + N/C\left(\lceil\frac{S}{R}\rceil\right)\beta\right] + \frac{2N}{H}\left[\alpha + N/R\left(\lceil\frac{S}{C}\rceil\right)\beta\right]. \tag{5}$$

We can remove the entire latency term in (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 processor's row and column can all 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, $S = P = R * C$) that all the nodes can be doing this step in parallel.

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

$$\lceil\frac{S}{\mathtt{lcm}(R,C)}\rceil N\left[V + 10B\right]\gamma. \tag{6}$$

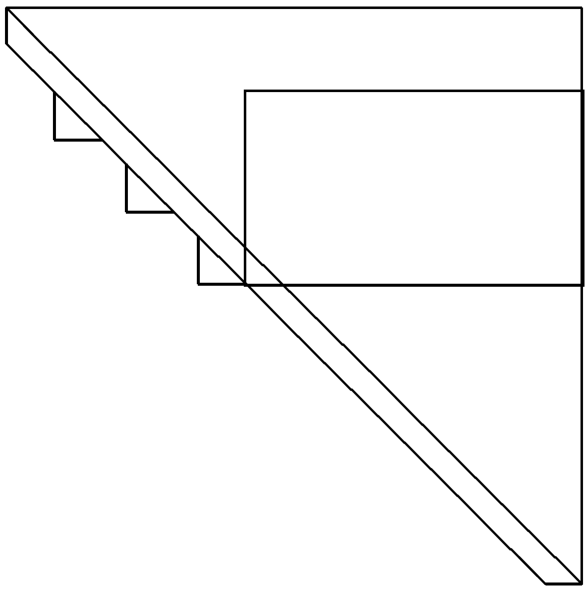

Figure 7: Delaying Transforms to Reduce the Pipeline Fill

4.2.5 Pipeline Start-up and Wind Down

In Figure 7, 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 like doing a parallel LU decomposition [9], 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 7 is that the boxed area does not need to be updated until all three bulges are going. This blocking can be used to reduce drastically the pipeline start-up and wind-down.[5]

Another observation to make is that if one uses more bulges S than are required to keep everyone busy, the pipeline start-ups become less important. Furthermore, any blocking done as suggested by Figure 7 reduces pipeline start-up but *not* wind down.

Since we have already modeled the total horizontal and vertical contribution time in §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.

[5]Our current implementation does not do this, so for the remainder of this section we will assume this is not done.

There are N/H block rows of the matrix. Assuming N is much larger than H, the number of horizontal flops for the first R transforms will be roughly $10NH/C$. 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 $10NH/C\gamma$. The ideal formula suggests the time should have been $10NH/(RC)\gamma$. We must therefore consider the wasted time given by the real time minus the ideal time. This is

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

When the first bulge reaches the second row, the second bulge can start. The two bulges require $10NH$ and $10N(N-H)$ flops to do the horizontal work. The ideal time for this would be $(20NH - 10H^2)/(RC)\gamma$. 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 $10NH/C\gamma$. The wasted time is then

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

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

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

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

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

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

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

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

$$\frac{10HR}{C}\left(N - \frac{3}{4} - \frac{1}{6}R\right)\gamma. \tag{7}$$

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 7, except that timings are approximately $(20NH/R)\gamma$. We then express the final vertical equivalent of Equation 7 as

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

The total overhead is for this section is therefore

$$\frac{10H}{RC}\left(R^2N - \frac{3}{4}R^2 - \frac{1}{6}R^3 + 2C^2N - \frac{3}{2}C^2 - \frac{1}{3}C^3\right)\gamma. \tag{8}$$

4.3 Overall Model

Because the algorithm is iterative, analysis of its overall performance is difficult. We can only guess at how many total iterations will be needed; faster convergence will be achieved for some matrices than for 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 §2.4 suggest that this may be an overestimate. Let us assume that these bulge chases are arranged into $2N/S$ super-iterations of S bulges each. We assume further that each super-iteration acts on the whole matrix. That is, we ignore deflations. This extremely pessimistic assumption assures that we will not overstate the performance of the algorithm. It also excuses us from considering the 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 super-iteration is obtained by summing (1), (3), (4), (5), (6), and (8). In terms of load balance, there are some clear advantages to taking $C \neq R$ (e.g. $\texttt{lcm}(R, C) = 1$). 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 of the expressions, we obtain the time for one superiteration,

$$\frac{20N^2k\gamma}{R} + \frac{2N}{B}\log(R)(\alpha + 3Bk\beta) + \frac{2N}{H}\left(3\alpha + \frac{4Nk\beta}{R}\right) + kN(V + 10B)\gamma + 30NH\gamma. \tag{9}$$

These terms correspond to flop count, broadcast overhead, border communication, bundling overhead, and pipeline startup/wind-down, respectively. We have simplified the last term by ignoring two small negative terms in (8).

The expression (9) reflects the tradeoffs that we have already noted. B needs to be big enough that broadcast communication is not dominated by latency but not so big that it causes serious bundling overhead. H needs to be big enough that border communication is not too expensive, but not so big that the pipeline startup 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 we have $\rho = 1/1800$.)

As before, let $S = kR$ be the number of double steps per super-iteration. We have $S = \sigma N$, where $\sigma = k\rho$. Assuming $2N$ iterations suffice, the number of super-iterations will be about $2N/(\sigma N) = 2/\sigma = O(1)$. Thus we shall assume that the total number of super-iterations 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 super-iterations required and assume that it is independent of N.

Let $\tau(N)$ denote the time to do one super-iteration, assuming that one can get the shifts for free. This is about the same as (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 H are large enough 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

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

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 (below) 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 super-iteration 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

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

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), 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), 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 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 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

$Nodes$	N	H	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 2: PDLAHQR Schur Decomposition Performance on the Intel Paragon Supercomputer

$DLAHQR$ in this case was 8.5 Mflops (compiled with -Knoieee -O4 -Mnoperfmon on the R5.0 compilers). Note that these results are better than those previously released in ScaLAPACK version 1.2 ALPHA. 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 this was the arbitrary cut off for doing the border communications in parallel (see §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 [32]. 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 the analogous results to Table 2, but running just an eigenvalue only version of the code. In this case, serial performance on the Intel Paragon system was around 8.2 Mflops (for roughly half the flops.) These positive results may lead to even better methods in future, since combining using `HQR` for finding eigenvalues and new GEMM-based inverse iteration methods for finding eigenvectors [30] might lead to completing the spectrum significantly faster than 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.

$Nodes$	N	H	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

Table 3: PDLAHQR Eigenvalue Only Performance on the Intel Paragon Supercomputer

$Nodes$	N	H	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 4: PDLAHQR Schur Decomposition Performance on the IBM SP2 Supercomputer

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 simply 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 ran 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 believe part of the problem was lack of dedicated time on the machine since these numbers were generated an interactive pool with others running other programs at the same time. These were done on thin nodes.

In Table 5 we ran 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 corresponds to actual flops computed. All problems except the problem run on 1 node used the exact same amount of memory per node. We also did a $N = 18000$ node run on 96 nodes (in an 8×12 configuration) that completed, with the right answer, in 34299 seconds. For all the parallel runs, H 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,

$Nodes$	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

Table 5: PDLAHQR Schur Decomposition on the Intel ASCI Option Red Supercomputer

the time to solution scales better than the parallel efficiency suggests.

6 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 even though it is an enhanced version of what appeared in ScaLAPACK version 1.2 ALPHA and will probably be in a future release. [6]

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 and the IBM SP2 at Cornell University and the IBM SP2 at Oak Ridge National Laboratory. We thank Intel, IBM, ORNL, and Cornell University. The work of David S. Watkins was supported by the National Science Foundation under grand DMS-9403569.

[6] To obtain a copy of the code please send e-mail to Greg Henry at ghenry@cs.utk.edu.

References

[1] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorenson, D., LAPACK Users' Guide, SIAM Publications, Philadelphia, PA, 1992

[2] Auslander, L., Tsao, A., *On Parallelizable Eigensolvers*, Advanced Appl. Math., Vol. 13, pp. 253–261, 1992

[3] Bai, Z., Demmel, D., *On a Block Implementation of Hessenberg Multishift QR Iteration* Argonne National Laboratory Technical Report ANL-MCS-TM-127, 1989, and International Journal of High Speed Computing, Vol. 1, 1989, p. 97–112

[4] Bai, Z., Demmel, J., *Design of a Parallel Nonsymmetric Eigenroutine Toolbox, Part I*, Parallel Processing for Scientific Computing, Editors R. Sincovec, D. Keyes, M. Leuze, L. Petzold, and D. Reed, pp. 391–398, SIAM Publications, Philadelphia, PA, 1993

[5] Bai, Z., Demmel J., *Design of a Parallel Nonsymmetric Eigenroutine Toolbox, Part II*, University of California at Berkeley Technical Report in Progress: 1/96

[6] Bai, Z., Demmel, J., Dongarra, J., Petitet, A., Robinson, H., Stanley, K., *The Spectral Decomposition of Nonsymmetric Matrices on Distributed Memory Parallel Computers*, LAPACK working note 91, University of Tennessee at Knoxville, Jan. 1995 To appear in SIAM Scientific Computing.

[7] Bai, Z., Demmel, J., Gu, M., *An Inverse Free Parallel Spectral Divide and Conquer Algorithm for Nonsymmetric Eigenproblems*, Draft appearing somewhere.

[8] Berry, M. W., Dongarra, J.J., Kim, Y., *A Highly Parallel Algorithm for the Reduction of a Nonsymmetric Matrix to Block Upper-Hessenberg Form*, Parallel Computing, Vol 21, No.8, August, 1995, pp 1189-1212.

[9] Blackford, S., Choi, J., Cleary, A., Demmel, J., Dhillon, I., Dongarra, J., Henry, G., Hammarling, S., Petitet, A., Stanley, K., Walker, D., Whaley, R.C., *ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance*, University of Tennessee at Knoxville Technical Report CS-95-283, LAPACK Working Note 95, 1995, Also: *Proceedings of Supercomputing '96*, 1996, ISBN 0-89791-851-1

[10] Byers, R., *Numerical Stability and Instability in Matrix Sign Function Based Algorithms*, Computational and Combinatorial Methods in Systems Theory, C. Byrnes and A. Lindquist, editors, pp. 185–200, North-Holland, 1986.

[11] Boley., D., Maier, R., *A Parallel QR Algorithm for the Nonsymmetric Eigenvalue Problem*, Univ. of Minn. at Minneapolis, Dept. of Computer Science, Technical Report TR-88-12, 1988

[12] Chakrabarti, S., Demmel, J., Yelick, K., *Modeling the Benefits of Mixed Data and Task Parallelism*, Seventh Annual ACM Symposium on Parallel Algorithms and Architectures, July 17-19, 1995, UC Santa Barbara, CA.

[13] Dongarra, J. J., Du Croz, J., Hammarling, and Duff, I. S., 1988, *A set of Level 3 basic linear algebra subprograms*, ACM TOMS, 16(1):1–17, March 1990.

[14] Dongarra, J. J., Geist, G.A., Romine, C.H., *Fortran Subroutines for Computing the Eigenvalues and Eigenvectors of a General Matrix By Reduction to General Tridiagonal Form*, ACM Trans. Math. Software, Vol. 18, 1992, p. 392-400

[15] Dongarra, J. J., Hammarling, S., J., Sorensen, D. C., *Block Reduction of Matrices to Condensed Forms for Eigenvalue Computations*, Journal of Computational and Applied Mathematics, 27:215–227, 1989.

[16] Dongarra, J. J., Kaufman, L., Hammarling, S., *Squeezing the Most out of Eigenvalue Solvers on High-Performance Computers* Linear Algebra and Its Applications, Vol. 77, 1986, pp. 113-136

[17] Dongarra, J. J., and Sidani, M., *A Parallel Algorithm for the Nonsymmetric Eigenvalue Problem*, SIAM J. Sci. Stat. Comput., Vol. 14, No. 3, pp. 542–569, May 1993. Also: Technical Report Number ORNL/TM-12003, ORNL, Oak Ridge Tennessee, 1991

[18] Dongarra, J. J., van de Geijn, R. A., *Reduction to Condensed Form on Distributed Memory Architectures,* Parallel Computing, 18, pp. 973–982, 1992.

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

[20] Eberlein, P. J., *On the Schur Decomposition of a Matrix for Parallel Computation*, IEEE Transactions on Computers, Vol. C-36, pp. 167–174, 1987

[21] Francis, J. G. F., *The QR Transformation: A unitary analogue to the LR Transformation, Parts 1 and 2*, Comp. J. 4, 1961, pp. 265-272, 332-45

[22] Geist, G.A., Davis, G.J., *Finding Eigenvalues and Eigenvectors of Unsymmetric matrices using a Distributed Memory Multiprocessor*, Parallel Computing, Vol 13, No. 2, pp. 199–209, 1990.

[23] Geist, G.A., Ward, R.C., Davis, G.J., Funderlic, R.E., *Finding Eigenvalues and Eigenvectors of Unsymmetric Matrices using a Hypercube Multiprocessor*, Monterey, CA, Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Editor G. Fox, pp. 1577–1582, 1988

[24] Golub, G., Van Loan, C., Matrix Computations, 2nd Ed., 1989, *The John Hopkins University Press.*

[25] Gupta, A., Kumar, V., *On the Scalability of FFT on Parallel Computers*, Proceedings of the Frontiers 90 Conference on Massively Parallel Computation, IEEE Computer Society Press, 1990

[26] D. E. Heller and I. C. F. Ipsen, *Systolic networks for orthogonal equivalence transformations and their applications,* Conference on Advanced Research in VLSI, M.I.T., 1982.

[27] Hendrickson, B.A., Womble, D.E., *The torus-wrap mapping for dense matrix calculations on massively parallel computers.*, SIAM J. Sci. Stat. Comput., Vol. 15, No. 5, pp.1201–1226, Sept. 1994

[28] Henry, G., *Improving the Unsymmetric Parallel QR Algorithm on Vector Machines*, 6th SIAM Parallel Conference Proceedings, March 1993

[29] Henry, G., *The Shifted Hessenberg System Solve Computation*, Cornell Theory Center Technical Report, CTC94TR163, 1/94

[30] Henry, G., *A Parallel Unsymmetric Inverse Iteration Solver*, 7th SIAM Parallel Conference Proceedings, 1994

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

[32] Henry, G., van de Geijn, R., *Parallelizing the QR Algorithm for the Unsymmetric Algebraic Eigenvalue Problem: Myths and Reality* Accepted for publication in SIAM Journal of Scientific Computing, Date unknown.

[33] Jessup, E.R., *A Case Against a Divide and Conquer Approach to the Nonsymmetric Eigenvalue Problem*, ORNL Technical Report ORNL/TM-11903, Oak Ridge Tennessee, 1991

[34] Kågström, B., Van Loan, C.F., *GEMM-Based Level-3 BLAS*, Cornell Theory Center Technical Report, CTC91TR47, 1/91.

[35] L. KAUFMAN, *A parallel QR algorithm for the symmetric eigenvalue problem,* J. Parallel and Distributed Computing, to appear.

[36] Lederman, S., Tsao, A., Turnbull, T., *A parallelizable eigensolver for real diagonalizable matrices with real eig envalues*, Technical Report TR-91-042, Supercomputing Research Center, 1991

[37] Smith, B., Boyle, J., Dongarra, J., Garbow, B., Ikebe, Y., Klema, V., Moler, C. *Matrix Eigensystem Routines - EISPACK guide, 2nd Ed.* Lecture Notes in Computer Science 6 Springer-Verlag, 1976

[38] Stewart, G. W., *A Parallel Implementation of the QR Algorithm*, Parallel Computing 5, pp. 187–196, 1987

[39] van de Geijn, R., A., *Implementing the QR-Algorithm on an Array of Processors*, Ph.D. Thesis, Department of Computer Science, Univ. of MD, TR-1897, 1987

[40] van de Geijn, R., A., *Storage Schemes for Parallel Eigenvalue Algorithms*, Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms, pp. 639–648, Editors G. Golub and P. Van Dooren, Springer Verlag, 1988

[41] R. A. VAN DE GEIJN, *Deferred shifting schemes for parallel QR methods,* SIAM J. Matrix Anal. Appl., 14 (1993) pp. 180–194.

[42] van de Geijn, R.,A., Hudson, D.G., *An Efficient Parallel Implementation of the Nonsymmetric QR Algorithm*, Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications, pp. 697–700, 1989

[43] van de Geijn, R. A., Walker, D., *Look at Scalable Dense Linear Algebra Libraries*, Scalable High-Performance Computing Conference, April 1992, IEEE Press.

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

[45] Watkins, D.S., *Shifting Strategies for the Parallel QR Algorithm*, SIAM J. Sci. Comput., 15 (1994), pp. 953–958.

[46] Watkins, D.S., *The Transmission of Shifts and Shift Blurring in the QR Algorithm*, Linear Algebra Appl., 241–243 (1996), pp. 877–896.

[47] Watkins, D.S., Elsner L., *Convergence of Algorithms of Decomposition Type for the Eigenvalue Problem*, Linear Algebra Appl. 143, 1991, p. 19–47.

[48] Wilkinson, J.H., The Algebraic Eigenvalue Problem, Oxford University Press, Oxford, 1965

[49] Wu, L., Chu, E., *New Distributed-memory Parallel Algorithms for Solving Nonsymmetric Eigenvalue Problems*, Proceedings of the 7th SIAM Parallel Conference on Parallel Processing for Scientific Computing, Ed. Bailey, et. al., SIAM Publications, Feb. 1995