PARALLELIZING THE QR ALGORITHM FOR THE UNSYMMETRIC ALGEBRAIC EIGENVALUE PROBLEM: MYTHS AND REALITY

GREG HENRY* AND ROBERT VAN DE GEIJN†

Abstract. *Over the last few years, it has been suggested that the popular QR algorithm for the unsymmetric eigenvalue problem does not parallelize. In this paper, we present both positive and negative results on this subject: In theory, asymptotically perfect speedup can be obtained. In practice, reasonable speedup can be obtained on a MIMD distributed memory computer, for a relatively small number of processors. However, we also show theoretically that it is impossible for the standard QR algorithm to be scalable. Performance of a parallel implementation of the LAPACK* `DLAHQR` *routine on the Intel Paragon™ system is reported.*

1. Introduction. Distributed memory parallel algorithms for the unsymmetric eigenvalue problem have been allusive. There are several matrix multiply-based methods currently being studied. Auslander and Tsao [2] and Lederman, Tsao, and Turnbull [24] have a matrix multiply-based parallel algorithm, which uses a polynomial mapping of the eigenvalues. Bai and Demmel [4] have another parallel algorithm based on bisection with the matrix sign function. Matrix tearing methods for finding the eigensystem of an unsymmetric Hessenberg matrix have been proposed by Dongarra and Sidani [9]. These involve doing a rank one change to the Hessenberg matrix to make two separate submatrices and then finding the eigenpairs of each submatrix, followed by performing Newton's method on these values to find the solution to the original problem. None of the above algorithms are particularly competitive in the serial case. They suffer from requiring more floating point operations (flops) and/or yield a loss of accuracy when compared to efficient implementations of the QR algorithm.

Efficient sequential (and shared memory parallel) implementations of the QR algorithm use a blocked version of the Francis double implicit shifted algorithm [13] or a variant thereof [20]. There have also been attempts at improving data reuse by increasing the number of shifts either by using a multi-implicited shifted QR algorithm [3] or pipelining several double shifts simultaneously [31, 32].

A number of attempts at parallelizing the QR algorithm have been made (see Boley and Maier [7], Geist, Ward, Davis, and Funderlic [14], and Stewart [27].) Distributing the work evenly amongst the processors has proven difficult for conventional storage schemes, especially when compared to the parallel solution of dense linear systems [22, 23]. Communication also becomes a more significant bottleneck for the parallel QR algorithm. As noted by van de Geijn [29] and van de Geijn and Hudson [30], the use of a block Hankel-wrapped storage scheme can alleviate some of the problems involved in parallelizing the QR algorithm.

In this paper, we present a number of results of theoretical significance on the subject. We reexamine the results on the Hankel-wrapped storage schemes in the setting of a parallel im-

* Intel Supercomputer Systems Division, 14924 N.W. Greenbrier Pkwy, Beaverton, OR 97006, `henry@ssd.intel.com`

† Department of Computer Sciences, The University of Texas, Austin, Texas 78712, `rvdg@cs.utexas.edu`

plementation of a state-of-the-art sequential implementation. Theoretically we can show that under certain conditions the described approach is asymptotically 100% efficient: if the number of processors is fixed and the problem size grows arbitrarily large, perfect speedup can be approached. However, we also show that our approach is not scalable in the following sense: To maintain a given level of efficiency, the dimension of the matrix must grow linearly with the number of processors. As a result, it will be impossible to maintain performance as processors are added, since memory requirements grow with the square of the dimension, and physical memory grows only with the number of processors. While this could be a deficiency attributable to our implementation, we also show that for the standard implementations of the sequential QR algorithm, it is impossible to find an implementation with better scalability properties. Finally, we show that these techniques can indeed be incorporated into a real code by giving details of a prototype distributed memory implementation of the serial algorithm `DLAHQR` [1], the LAPACK version of the double implicit shifted QR algorithm. Full functionality of the LAPACK code can be supported. That is, the techniques can be extended to allow for the cases of computing the Schur vectors, computing the Schur decomposition of H, or just computing the eigenvalues alone. We have implemented a subset of this functionality, for which the code is described and performance results are given.

Thus this paper makes four contributions: It describes a data decomposition that allows, at least conceptually, straight-forward implementation of the QR algorithm; It gives theoretical limitations for parallelizing the standard QR algorithm; It describes a parallel implementation based on the proposed techniques; It reports performance results of this proof-of-concept implementation obtained on the Intel Paragon™ system.

2. Sequential QR Algorithm.

While we assume the reader of this paper to be fully versed in the intricate details of the QR algorithm, we briefly review the basics in this section.

The Francis double implicit shifted QR algorithm has been a successful serial method for computing the Schur decomposition $H = QTQ^T$. Here T is upper pseudo-triangular with $1x1$ or $2x2$ blocks along the diagonal and Q is orthogonal. We assume for simplicity that our initial matrix H is Hessenberg. The parallelization of the reduction to Hessenberg form is a well understood problem, and unlike the eigenvalue problem, the Hessenberg reduction has been shown to parallelize well [5, 10].

One step of the Francis double shift Schur decomposition is in Figure 1. 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

$$v_j = \begin{cases} 0 & \text{if } j < i+1 \text{ or } j > i+3 \\ 1 & \text{if } j = i+1 \end{cases}$$

We assume the Hessenberg matrix is unreduced, and if not, find the largest unreduced submatrix of H. Suppose this submatrix is $H(k:l, k:l)$. We then apply the Francis HQR Step to the rows

```
**Francis HQR Step**
$e = \mathrm{eig}(H(n-1:n, n-1:n))$
Let $x = (H - e(1)I_n) * (H - e(2)I_n)$
Let $P_0 \in \Re^{n\times n}$ be a Householder matrix
s.t.
    $P_0 x$ 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
                $(i+3, i)$ entries.
        Update $H \leftarrow P_i H P_i$
        Update $Q \leftarrow Q P_i$
**endfor**
```

Fig. 1. *Sequential Francis HQR Step*

and columns of H corresponding to the submatrix; that is,

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

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

The double implicit shifts in this case are chosen to be the eigenvalues of

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

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

3. Parallel QR Algorithms.

3.1. Data Decomposition.

We briefly describe the storage scheme presented in [30]. Suppose $A \in \Re^{n\times n}$ where $n = mhp$ for some integers h and m and p is the number of processors. Suppose A is partitioned as follows

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,mp-1} & A_{1,mp} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,mp-1} & A_{2,mp} \\ A_{3,1} & A_{3,2} & \cdots & A_{3,mp-1} & A_{3,mp} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{mp,1} & A_{mp,2} & \cdots & A_{mp,mp-1} & A_{mp,mp} \end{bmatrix},$$

where $A_{i,j} \in \Re^{h\times h}$. We denote processor k owning block $A_{i,j}$ with a superscript $A_{ij}^{(k)}$. The one dimensional block Hankel-wrapped storage for $m = 1$ assigns submatrix $A_{i,j}$ to processor

$$(i+j-2) \ \mathbf{mod}\ p.$$

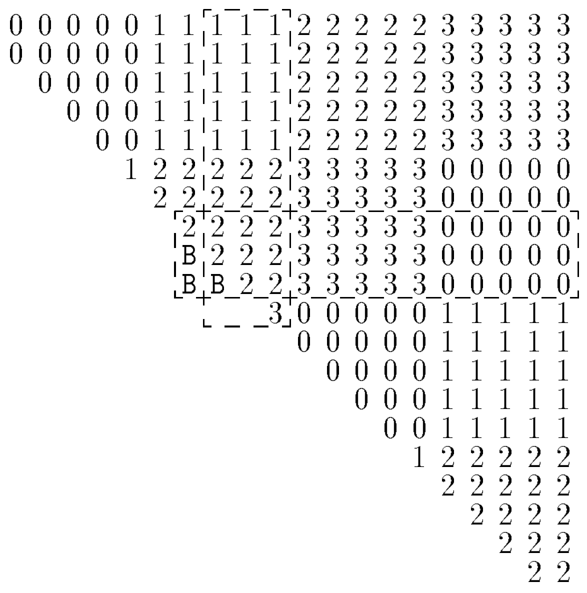

FIG. 2. *Data decomposition and chasing of the bulge.*

That is, the distribution amongst the processors is as follows [30]:

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

where the superscript indicates the processor assignment.

3.2. Basic Parallel Algorithm. We start describing the basic parallel QR algorithm by considering the Francis HQR Step. The basic loop in Figure 1, indexed by i, is generally referred to as "chasing the bulge". Figure 2 gives an arbitrary example of $i = 7$, in a 20×20 matrix, where the integers indicate nonzero elements of the upper Hessenberg matrix, and the B's indicate the fill in at that stage of the loop. In the next step of Francis HQR step, a Householder transformation P_i is computed and applied from the left and the right. The affected elements are within the dashed lines. After this application of the Householder transformation, the bulge moves down the diagonal one position, and the next iteration starts.

Figure 2 also demonstrates the possible parallelism in the Francis HQR step: The application of P_i to the appropriate elements of H, referred to as a reflection, is perfectly distributed among the processors, except for the processor that owns the bulge and the fill-in elements. This processor must do slightly more work (associated with the intersection of the two boxes). *It is the parallelism within the application of each Householder transformation that is the key to the success of the approach.*

The entire Francis HQR step now parallelizes as follows:

1. The processor(s) that holds eig($H(n-1:n, n-1:n)$ computes the shifts, and sends them to the first processor. Alternatively, all processors could compute the shifts, since it is a lower order operation.
2. The processor that owns the data required to compute P_0 computes it, and then broadcasts it to all other processors.
3. All processors update their portions of the matrix according to P_0.
4. **for** $i = 1, \dots, n-2$
   (a) The processor that owns the data required to compute P_i computes it and then broadcasts it to all other processors.
   (b) All processors update their portions of the matrix according to P_i.

Naturally, everytime the computation hits data that lies on the boundaries between processors, a limited amount of communication is required to bring appropriate rows and columns together.

3.3. Analysis of the Simple Parallel Algorithm. Before discussing the finer details of the parallel code, let us analyze the simple algorithm given above. This will give us an idea of how well the basic approach works. We will restrict our analysis to a single Francis step, and ignore shift derivation, convergence, or decoupling set up costs. We show later how this extra $O(n)$ work can be done without incurring excessive overhead.

For our analysis, we will use the following model: Performing a floating point computation requires time γ. Sending a message of length n between two nodes requires time $\alpha + n\beta$, where α represents the latency, and β the penalty per double precision number transferred. For convenience, we shall assume all communication is nearest neighbor, and hence no network conflicts occur. Processors can send and receive simultaneously.

We start by computing the sequential expense of the algorithm: Computing a Householder transformation from a vector of length three requires C, some small constant, flops. Applying such a Householder transformation of size three to three rows or columns of length n requires $10n$ flops. Within the loop indexed by i, each transformation is applied to $n-i+1$ triplets of rows and $i+3$ triplets of columns. Applying the transformations to the Hessenberg matrix therefore requires $10(n+4)$ flops and applying the transformations to the Schur matrix requires $10n$ flops. The total cost of executing a single Francis step thus becomes approximately

$$\sum_{i=1}^{n-1} \left(C + 10(n+4) + 10n\right)\gamma = 20n^2\gamma + O(n)$$

Turning now to the parallel implementation, contributions to the critical path are given roughly as follows: Computing a transformation still contributes C flops. Broadcasting this transformation requires time $\alpha + 3\beta$, since the parallel implementation can be arranged to pipeline around the logical ring of processors. After this, the processors that computed the transformation must update its part of the rows and columns. The row and column vectors are of length approximately n/p, with a few extra introduced by the bulge. This contributes the equivalent of about $20(n/p+4)$ flops. Now, it can compute the next transformation, and the process repeats itself $n-1$ times. The total contribution becomes

$$(1)\quad \sum_{i=1}^{n-1}\left(C\gamma + 10(\frac{n}{p}+4) + 10\frac{n}{p}\gamma + \alpha + 3\beta\right) = Cn\gamma + 20\frac{n^2}{p}\gamma + 40n\gamma + n\alpha + 3n\beta + O(1)$$

Here we cannot ignore $O(n)$ terms, since if p is comparable to n, these terms matter.

Equation 1 ignores the communication required when a boundary of processors is reached, which happens every h iterations. Each "border" row contains roughly $(n-i)/p$ elements per processor, and each "border" column contains roughly i/p elements per processor. Using transforms of size three implies that when the boundary of processors is reached, the next transform will also cross the boundary. Hence $2n/p$ items must be communicated to the right and returned. This means that the total communication volume for "border" or boundary data is

$$\frac{n}{h}(4\frac{n}{p}) \tag{2}$$

The communication volume in Equation 2 can be done in anywhere from one to eight messages depending on the implementation. Eight messages are required when both rows and both columns for the two iterations are sent separately, and returned again. One message is possible if data is only sent rightward around the ring, and the results accumulated into a buffer and sent back in $O(1)$ communications at the end of the Francis step. For the purpose of our model, we assume two communications, requiring time

$$2\frac{n}{h}\left(\alpha + 2\frac{n}{p}\beta\right) \tag{3}$$

for a total estimated parallel time of

$$T_{\text{est}}(n,p,h) = Cn\gamma + 20\frac{n^2}{p}\gamma + 40n\gamma + n\alpha + 3n\beta + 2\frac{n}{h}\alpha + 4\frac{n^2}{hp}\beta + O(1) \tag{4}$$

On current generation architectures, the above model shows that the bulk of inefficiency comes from the *number* of messages generated, since α is typically several orders of magnitude greater than either β or γ. To investigate the scalability of the approach, we start by computing the estimated speedup

$$\text{Speedup}_{\text{est}}(n,p,h) = \frac{T_{\text{seq}}(h)}{T_{\text{est}}(n,p,h)} = \frac{20n^2}{20\frac{n^2}{p}\gamma + 4\frac{n^2}{hp} + O(n)} = \frac{p}{1+\frac{4}{h}+O(\frac{p}{n})} \tag{5}$$

If h is chosen big enough, e.g. $h = n/p$, the second term in the denominator also becomes $O(p/n)$, and we can make the following observations: if p is fixed, and n is allowed to grow, eventually perfect speedup will be approached. Furthermore, the estimated efficiency attained is given by

$$\text{Efficiency}_{\text{est}}(n,p,h) = \text{Speedup}_{\text{est}}(n,p,h) \quad = \quad \frac{1}{1+O(\frac{p}{n})} \tag{6}$$

From this last equation, we can say something about the scalability of the implementation: In order to maintain efficiency, we must grow n linearly with p. This poses a serious problem: memory grows linearly with p, but memory requirements grow with n^2. We conclude that this approach to parallelization will not scale, since memory constraints dictate that eventually efficiency cannot be maintained, even if the problem is allowed to grow to fill the combined memories.

The previous modeling extends to the overall algorithm in a simple manner. On average, to find the eigenvalues of a matrix of size n requires $2n$ HQR iterations. Every other iteration, there is usually a deflation of size one or two. Since deflation occurs less frequently in the beginning than the end, our overall model satisfies:

$$\text{Overall Flops } = \sum_{k=1}^{n} 3 * 20k^2 = 20n^3,$$

where 3 is a heuristic fudge factor. Compare, for example, [15] which uses a similar heuristic, but different fudge factor, to suggest overall flops at $25n^3$.

Similarly, we can extend any of the formulas in the previous subsection. In Equation 4, the number of communication start-ups is $n + 2n/h$. For the overall algorithm, before the bundling described in § 3.5.4, we can estimate

$$\text{Communication Start-ups } = \sum_{k=1}^{n} 3 * (k + 2k/h) = (\frac{3}{2} + \frac{3}{h})n^2$$

3.4. Theoretical Limitations. For many dense linear algebra algorithms, better scalability can be obtained by using a so-called two dimensional data decomposition. Examples include the standard decomposition algorithms, like LU, QR, and Cholesky factorization [11, 16], as well as the reduction to condensed form required for reducing a general matrix to upper Hessenberg form [5, 10]. It is therefore natural to try to reformulate the parallel QR algorithm in an attempt to extract an algorithm that has better scalability properties. In this section we show *theoretically* that there is an *inherent* limitation to the scalability of the QR algorithm that indicates that a more complicated data and work decomposition alone *cannot* improve the scalability of the parallel implementation.

We present a simple analysis of the theoretical limitations of the QR algorithm. There is a clear dependence in the chasing of the bulge that requires at least $O(n)$ steps to complete one Francis HQR step. The associated computation can be thought of as running down the diagonal of the matrix. Moreover, the last element of the matrix must be computed before the next Francis step can start, since it is needed to compute the shift. Since there are $O(n^2)$ computations performed in one such step, the speedup is limited to $O(n)$, which indicates that the maximal number of processors that can be usefully employed is $O(n)$. Hence, $p \leq Cn$ for some constant C.

Let us now analyze the implications of this: If efficient use of the processors is to be attained, the equation $p \leq Cn$ must hold for some constant C. The total memory requirements for a problem of size n is Kn^2, for some constant K. Hence

$$\text{Memory Requirements} = Kn^2 \geq \frac{K}{C^2}p^2 = O(p^2).$$

Notice however that memory only grows linearly with the number of processors. We conclude that due to memory restrictions, it is impossible to solve a problem large enough to maintain efficiency as processors are added. In the conclusion, we indicate how it may be possible to overcome this apparent negative result.

3.5. Refinements of the Parallel Algorithm. The above analysis shows that there is reason to believe that trying to generalize the Hankel-wrapped mapping to yield the equivalent of a two dimensional wrapping will not result in the same kinds of benefits as it would for a factorization algorithm. However, there are a number of refinements that can be made that will improve the performance of the parallel QR algorithm. Moreover, the algorithm as it was described in Section 3.2 is far from a complete implementation.

3.5.1. Accumulating Q. The algorithm in Section 3.2 ignores the accumulation of the orthogonal matrix Q. There is a perfect parallelism in this operation, requiring no more communication than must already be performed as part of the updating of H.

Each stage of the Francis HQR step in Figure 1 requires the computation $Q \leftarrow QP_i$. Since P_i is broadcasted during the update of the Hessenberg matrix, all processors already have this information. The computation $Q \leftarrow QP_i$ reflects three columns of Q. Which three get reflected depends on i. Since Q is not accessed elsewhere within the HQR kernel, we can assume Q has the best data storage possible for reflecting three columns. This data storage is a one dimensional row mapping. With a one dimensional row mapping, the data in any row is always contained on a single processor. Each row in a column transform can be computed independently. So there is no extra communication required to complete the column transform, and this extra work is embarrassingly parallel.

3.5.2. Deflation. We previously mentioned that as subdiagonal elements become numerically zero, the process proceeds with unreduced submatrices on the diagonal. This process is known as deflation. One convenient way to determine the points where such a block start and end is to broadcast enough diagonal information during the bulge chase so that at the end of a complete HQR step, all processors hold all information required to determine when breakpoints occur. This requires sending six elements instead of three per transformation. While this creates a certain redundancy, the overhead is $O(n)$, and hence in line with the cost of the total computation.

3.5.3. Determining shifts. Related to the refinement mentioned above concerning deflation, it is convenient to also broadcast all entries of the tridiagonal of the newly formed Hessenberg matrix, since this allows all processors access to the data required to compute the next shifts.

3.5.4. Bundling transformations. As is argued in Section 3.3, the bulk of the communication overhead is due to communication startup cost. This cost can be amortized over several computations and applications of Householder transformations by bundling several Householder transformations before broadcasting.

We direct the reader's attention to Figure 3.

This picture represents the data in the block that currently holds the bulge. This block is assumed to be completely assigned to one processor. In chasing the bulge a few positions, to where it moves from the position indicated by "B"s to the position indicated by "b"s, a number of Householder transformations are computed. To do so, only the computation associated with the

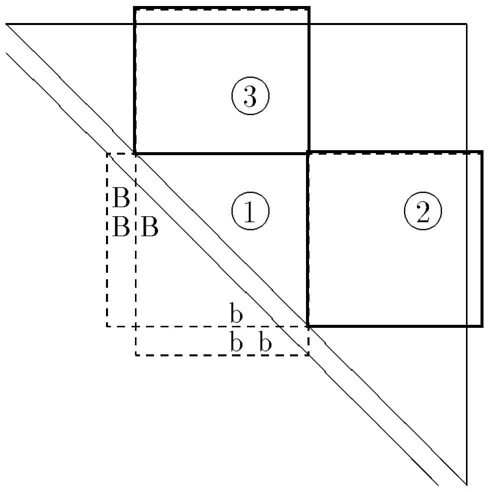

FIG. 3. *Bundling Householder transformations and order of updates*

region marked by "1" needs to be performed. [1] Thus these transformations can be bundled by performing this minimal computation, followed by the broadcast, after which the other regions, marked by "2" and "3" can be updated. The net effect is that the number of startups is reduced by the number of transformations that are bundled. Naturally, bundling across processor boundaries becomes complicated, and has only marginal benefit.

Repeating the analysis in Subsection 3.3 above, we get the following estimates: Computing r transformations, updating only region "1" in Figure 3 requires approximately

$$r(C + 10(r + 4))\gamma$$

time. Broadcasting the transformations requires time

$$\alpha + 3r\beta.$$

Next, updating regions "2" and "3" on the processor that computes the transformations requires time

$$10r(\frac{n}{p} - r)\gamma$$

to update matrix A, and

$$10r\frac{n}{p}\gamma$$

to update matrix Q. This is repeated n/r times, and border information must be communicated every h transformations, giving a (very rough) total estimated time of

$$20\frac{n^2}{p}\gamma + Cn\gamma + 40n\gamma + \frac{n}{r}\alpha + 3n\beta + 2\frac{n}{h}\alpha + 4\frac{n^2}{hp}\beta + O(1).$$

[1] As is noted in Henry [17, 20], the minimal information needed to determine the upcoming transforms is roughly eighty percent of the total flops required to actually complete the entire update in region "1" of Figure 3. This "partial" step is referred to as a lookahead step and can also be used to determine better shifts (cf. Henry [19].)

Clearly the above estimate is only a rough estimate: It suggests that increasing bundling factor r arbitrarily will continue to improve total time. Notice that this estimate is only reasonable when the pipe is undisturbed. However, everytime a boundary is encountered, the boundary will be disturbed and therefore larger bundling factors will increase load imbalance. Nonetheless, our model gives us insight.

4. Performance Results.

In this section, we report the performance of a prototype working implementation of the described parallel implementation. Our implementation is a parallelization of the LAPACK [1] routine `DLAHQR`, and is mathematically exact except that the sequential routine applies one final rotation per converged eigenpair so that the resulting "Schur" form has some regularity. Hence there is no need to report in detail on the numerical accuracy of the implementation: all numerical properties of the LAPACK apply to this parallel implementation.

We report performance obtained on an Intel Paragon™ XP/S Model 140 Supercomputer, running SUNMOS version S1.4.8. Since finding all eigenvalues of a very large matrix takes a considerable amount of time (measured in days) we report the performance of the algorithm during the first five iterations. For problems of size 2750 or less, and for 64 processors or less, we saw five iterations was always within three percent of predicting overall performance. In addition, it is hard to measure speedup with respect to a single processor, due to insufficient memory to hold a large matrix. As a result, we measured the performance of the best known sequential QR algorithm, and report the parallel performance with respect to that: The single node performance of the fastest QR algorithm (not necessarily a standard double shift algorithm from LAPACK) peaks at 10 Mflops for the five iterations. We report scaled speedup as being the speedup obtained with respect to the calculated time required to run a given problem on a single node performing at 10 Mflops. In this case, actual flops within the algorithm were calculated and used for results, and not heuristics.

Many parameters are used to describe the parallel implementation, including the number of nodes used, p, the matrix size, n, the blocking factor, h, and the bundling factor, r. All the runs in Table 1 were for problems of size approximately 10 to 14 Mbytes, all with blocking factors h given by n/p, with bundling factors constant at $r = 2$. Our models predict that varying h and r impacts communication overhead incurred. However, in practice the major source of inefficiency appears to come from the fact that row and column updates do not have the same memory access patterns, and therefore execute at different rates. Our experiments indicate row transforms are 13 percent faster than column transforms. At each step there is a roughly even amount of work for all the nodes, but there is not the same amount of row or column work. This means that at *each* step there is a delay for a few of the nodes to complete the ring broadcast. Hence although in our model predicts the load balance to be quite good, the different rates of execution make it much less well balanced. This shows up in the timings given in Table 1. Notice that at best, about 65 percent efficiency is attained.

5. Conclusion.

The research described in this paper was meant as a response to claims that no parallelism exists in the QR algorithm. The theoretical results in this paper show there are approaches to implementing the QR algorithm that allow parallelism to be extracted in a natural way. The actual implementation of this method on a high performance parallel computer

p	n	*Scaled Speedup*	*Efficiency*
4	2000	2.5	.63
8	2800	5.2	.65
16	4000	10.0	.63
32	4800	17.1	.53
48	6000	23.0	.48
64	8000	30.1	.47
96	9600	37.4	.39

Table 1
Parallel HQR

shows that in practice, parallelism can be extracted as well.

We caution the reader against misinterpreting the results in this paper as largely supporting the notion that new methods like those developed by Bai and Demmel [4] and Dongarra and Sidani [9] must be pursued if nonsymmetric eigenvalue problems are to be solved on massively parallel computers. Let us address some of the arguments that can be made to support such an interpretation and how these arguments are somewhat unsatisfactory.

No parallelism exists in the nonsymmetric QR algorithm.

We believe the major results in this paper are a counter example to this statement, both in theory and practice.

The performance of even the sequential algorithm leaves something to be desired.

One argument for matrix-matrix multiplication based methods is that matrix multiplication can achieve much higher performance rates than a kernel that applies a Householder transformation. This more than offsets the added floating point operations required for such novel solutions. This argument is true only because the matrix-matrix multiplication is recognized as an important kernel that warrants highly optimized implementation. It is the data reuse available in the matrix-matrix multiplication that allows near-peak performance to be achieved on a large number of platforms. However, a careful analysis of reuse of data when *multiple* Householder transformations are applied, which could be exploited with bundling, shows that the performance of the sequential QR algorithm could be greatly improved if such a kernel were assembly coded. Clearly a user who can afford a massively parallel computer would be able to afford to assembly code such a kernel.

One could argue that assembly coding this kernel would only improve the 40 percent of total time spent in useful computation and hence would actually *decrease* efficiency. However, since we argue that the bulk of overhead is due to load imbalance, the total time should benefit, not just the time spent in useful computation.

Parallel QR algorithms cannot be scalable.

One could draw this conclusion from Section 3.4. However, notice that the analysis holds for the algorithm when only a single double-shift is used during each iteration. If a method is parallelized that uses s shifts, the total computation per iteration is increased to $O(sn^2)$, while the dependence during the bulge chase only increases to $O(n+s)$ if multiple bulges are chased in a pipelined fashion. We conclude that the limit on speedup is now given by $O(sn^2)/O(n+s) \approx O(sn)$. $O(sn)$ processors can be usefully employed, leading to memory requirements

$$\text{Memory Requirements} = Kn^2 >= O(\frac{p^2}{s^2})$$

As long as $p/s^2 = O(1)$, i.e. $s = O(\sqrt{p})$, scalability can be achieved while meeting the fixed memory per processor criteria.

Scalable implementations of the QR algorithm may be achievable for parallel implementations that use s shifts simultaneously, provided the number of shifts grows with the square-root of the number of processors.

We must note that our data decomposition inherently requires n to grow with p and therefore it will not provide for the required parallel implementation. Therefore, generalizations to two dimensional data decompositions will be necessary.

Notice that some of the above comments indicate that many of the observations made from the earlier sections were merely due to the fact that we restricted ourselves in this paper to discussing the parallel implementation of widely used QR algorithms.

We conclude by saying that the reason parallel QR algorithm implementations may not be competitive is because insufficient resources have been allocated to study their implementation. Those resources are currently being used to instead explore novel algorithms. If someone has the required resources, our research points clearly to what properties a parallel implementation must have to be successful.

Acknowledgements. Our final runs were made on Sandia's Paragon™ XP/S Model 140 Supercomputer running SUNMOS S1.4.8, and we are grateful for this resource and operating system. SUNMOS stands for Sandia and University of New Mexico Operating System, and is a joint collaboration between Sandia National Laboratories and UNM. SUNMOS is copyrighted by Sandia National Laboratories. Most of the development was done on various Paragon™ supercomputers at Intel Supercomputer Systems Division. We thank Ken Stanley and David Watkins for communications regarding this work.

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, J., *On a Block Implementation of Hessenberg Multishift QR Iteration*, International Journal of High Speed Computing, Vol. 1, pp. 97–112, 1989

[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, SIAM Publications, Philadelphia, PA, 1993

[5] Berry, M. W., Dongarra, J. J., and Kim, Y., *A Highly Parallel Algorithm for the Reduction of a Nonsymmetric Matrix to Block Upper-Hessenberg Form*, LAPACK working note 68, University of Tennessee, CS-94-221, Feb. 1994

[6] Boley, D., *Solving the Generalized Eigenvalue Problem on a Synchronous Linear Processor Array*, Parallel Computing, Vol. 3, pp. 123–166, 1986

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

[8] Demmel, J. W., *LAPACK Working Note 47: Open Problems in Numerical Linear Algebra*, University of Tennessee, Technical Report CS-92-164, May 1992

[9] Dongarra, J. J., and Sidani, M., *A Parallel Algorithm for the Nonsymmetric Eigenvalue Problem*, Technical Report Number ORNL/TM-12003, ORNL, Oak Ridge Tennessee, 1991

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

[11] Dongarra, J. J., van de Geijn, R. A., and Walker, D. , *Scalability Issues Affecting the Design of a Dense Linear Algebra Library*, Journal of Parallel and Distributed Computing,Vol. 22, No. 3, Sept. 1994.

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

[13] Francis, J. G. F., *The QR Transformation: A Unitary Analogue to the LR Transformation, Parts I and II*, Comp. J., Vol. 4, pp. 332–345, 1961

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

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

[16] Hendrickson, B.A., Womble, D.E., *The torus-wrap mapping for dense matrix calculations on massively parallel computers.*, SIAM J. Sci. Stat. Comput., 1994

[17] Henry, G., *Improving the Unsymmetric Parallel QR Algorithm on Vector Machines*, SIAM 6th Parallel Conference Proceedings, 3/93

[18] Henry, G., *Increasing Data Reuse in the Unsymmetric QR Algorithm*, Theory Center Technical Report, CTC92TR100, 7/92

[19] Henry, G., *A New Approach to the Schur Decomposition*, Theory Center Technical Report in Progress

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

[21] Ipsen, I.C.F., Saad, Y., *The Impact of Parallel Architectures on the Solution of Eigenvalue Problems*, Large Scale Eigenvalue Problems, Editors J. Cullum and R. Willoughby, Elsevier Science Publishers, 1986

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

[23] Juszczak, J.W., van de Geijn, R.A., *An Experiment in Coding Portable Parallel Matrix Algorithms*, Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications, 1989

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

[25] Moler, C.B., *MATLAB User's Guide* Technical Report CS81-1, Department of Computer Science, University of New Mexico, Albuquerque, New Mexico, 1980

[26] Purkayastha, A., *A Parallel algorithm for the Sylvester-Observer Equations*, Ph.D. dissertation, Northern Illinois University, DeKalb, Illinois, Jan. 1993

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

[28] 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

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

[30] 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, 1989

[31] Watkins, D., *Shifting strategies for the parallel QR algorithm*, SIAM J. Sci. Comput., Vol. 15, July 1994

[32] Watkins, D., *Transmission of shifts in the multishift QR algorithm*, Proceedings of the 5th SIAM Conference on Applied Linear Algebra, Snowbird, Utah, June 1994

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