TOWARDS AN EFFICIENT PARALLEL EIGENSOLVER FOR DENSE SYMMETRIC MATRICES *

BRUCE HENDRICKSON†, ELIZABETH JESSUP‡, AND CHRISTOPHER SMITH§

Abstract. We describe a parallel algorithm for finding the eigenvalues and eigenvectors of a dense symmetric matrix, with an emphasis on the dense linear algebra operations. We follow the traditional three step process: reduce to tridiagonal form, solve the tridiagonal problem then backtransform the result. Since the different steps have different algorithmic characteristics, this problem serves as a perfect vehicle for exploring some issues associated with parallel linear algebra calculations. In particular, we examine the effects of matrix distribution and blocking on the computational performance of tridiagonalization and backtransformation. Through experiments on an Intel Paragon, we demonstrate that block storage of the matrix is not necessary for a highly efficient block algorithm. The performance of our approach compares very favorably with that of the corresponding ScaLAPACK routines.

1. Introduction. Large order symmetric eigenvalue problems arise in a variety of contexts ranging from real-time signal processing [40] to the analysis of electrical networks [12] to the modeling of acoustic and electromagnetic waveguides [34]. Because determining the eigenvalues and eigenvectors of large matrices is a time-consuming process demanding extensive memory, such applications motivate the study of parallel implementations of symmetric eigensolvers. The increased computing and storage resources provided by multiprocessors can allow efficient solution of large order problems. This paper describes a scalable parallel approach to calculating eigenvectors and eigenvalues of dense symmetric matrices and presents results of its implementation on the Intel Paragon.

While the symmetric eigenproblem is an important practical problem, it also serves as a useful platform for examining algorithmic performance issues. In this paper, we use it to study the capabilities and limitations of parallel computers for dense linear algebra problems that are significantly more complex than the well-studied problem of LU factorization. Thus, in implementing our eigensolver, we intentionally chose a different algorithmic approach than that used in ScaLAPACK [11]. Our method differs from ScaLAPACK's in three fundamental ways. First, unlike ScaLAPACK's, our approach does not require that the matrix be stored in a blocked manner. As we discuss in §3, this choice has both advantages and disadvantages, and the symmetric eigenproblem is a perfect application with which to investigate them. Second, we use a more efficient set of communication operations than does ScaLAPACK, thereby reducing the communication overhead of our algorithms significantly. And third, unlike ScaLAPACK's routine, ours is able to keep all processors used busy during each computationally significant stage of the algorithm thus further improving overall performance.

Relative to ScaLAPACK we had the advantage of targeting a single architec-

*To appear in SIAM J. Sci. Comput.

† Applied & Numerical Math Dept., Sandia National Labs, Albuquerque, NM 87185–1110. bah@cs.sandia.gov. Hendrickson and Smith were funded by the Applied Mathematical Sciences program, U.S. DOE, Office of Energy Research, and performed research at Sandia National Labs, operated for DOE under contract No. DE-AL04-94AL8500.

‡Dept. Computer Science, Univ. Colorado, Boulder, CO 80309–0430. jessup@cs.colorado.edu. Jessup was funded by DOE contract DE-FG03-97ER25325 and by an NSF Young Investigator Award.

§Advanced Product & Process Technology, Tower Automotive, Milwaukee, WI 53216. smith.chris.b@towerautomotive.com.

ture and not worrying about portability. However, all our communication operations use the Intel csend/crecv and isend/irecv calls which have direct MPI counterparts. Consequently, porting to MPI would be straightforward. The intent of this study, however, was not to produce a public library routine, but rather to study some important questions about efficient parallel algorithm development.

The traditional method for determining the eigensystem of a real, dense symmetric matrix A employs a three step technique [22]. First, A is reduced to symmetric tridiagonal form using a series of Householder transformations. Next, the eigensystem of the tridiagonal matrix T is computed. The eigenvalues of A are the same as those of T, while the eigenvectors of A are found in the third step by backtransforming the eigenvectors of T via the reduction transformations [39]. Specifically, if the transformations are accumulated into the matrix Q, the tridiagonal matrix T is formed by the matrix product $T = Q^T AQ$. If the eigendecomposition of T is $T = XDX^T$, then the eigendecomposition of A is $A = (QX)D(QX)^T$.

Tridiagonalization and backtransformation are critical elements in an efficient parallel eigensolver. First, implementing these routines in parallel allows the dense matrix A to be distributed among processors. This distribution permits solution of large problems on storage-limited machines. Second, for large problems, the dense linear algebra routines usually dominate the overall runtime of the eigensolver. (The cost of the tridiagonal eigensolver we use can become quite large for pathological matrices. That problem could be overcome by means of a parallel implementation of the new tridiagonal eigensolver described in [13]–a method developed since the writing of this paper.) For these reasons, this paper focuses on the dense linear algebra routines.

This paper is organized as follows. We review the state of the art in §2. Then, in §3, we contrast the data mapping used in our dense linear algebra algorithms with that used in ScaLAPACK, and describe the communication patterns our algorithms require. In §4, we review the sequential algorithm for tridiagonalization and describe and analyze our parallel counterpart. This is the most complex parallel algorithm in this paper. In §5, we explain how we approach the tridiagonal eigenproblem. In §6, we describe our parallel implementation of backtransformation. In §7, we provide timings of our code on the Intel Paragon along with comparisons to ScaLAPACK, followed by conclusions in §8.

2. Previous Work. Although the sequential algorithms for the tridiagonalization and backtransformation steps are well understood, tridiagonal eigensolvers continue to be an area of active research (see [1, 22, 30, 31] for surveys.) With respect to parallel algorithms, the situation is somewhat reversed. There has been more work on the tridiagonal eigensolver than on parallel methods for tridiagonalization. Investigation of parallel tridiagonal eigensolvers begin with Huang's 1974 study of multisection on the ILLIAC IV [27] and continue with the implementation of several other tridiagonal eigensolvers on a variety of shared– and distributed–memory computers and their simulators [2, 3, 4, 5, 19, 29, 32, 36, 37, 38, 43].

Work on parallel tridiagonalization and backtransformation algorithms began more recently and has been confined to distributed–memory machines. Chang et al. described, but did not implement, two parallel versions of the Householder tridiagonalization algorithm for the hypercube in 1988 [9, 10]. One of these algorithms uses the one dimensional mapping of matrix elements to processors that was the common choice at the time. The second employs a two dimensional mapping that is now generally recognized as superior for dense linear algebra calculations. We explain why in §3.

Several parallel implementations of the traditional tridiagonalization algorithm have appeared recently. These include one dimensional mappings on the Intel iPSC/860 by Dongarra and van de Geijn [18] and by Fann, Littlefield, and Maschhoff [37], a two dimensional mapping on the nCUBE 2 by Hendrickson and Womble [26], and a two dimensional mapping by Choi, Dongarra and Walker as part of the ScaLAPACK project [11] which was developed simultaneously with the current work. This last reference is the most similar to ours. The differences are discussed in the next section.

A general approach to parallel band reduction has been proposed by Lang [35] and by Bischof, Lang and Sun [6, 7], but this approach is not suitable for the tridiagonalization of dense matrices if a backtransformation is required. An entirely different parallel eigensolver rich in efficient matrix-matrix (level 3 BLAS) operations has been proposed by Huss-Lederman, Tsao, and Zhang [28]. A parallel algorithm based on one-sided rotations has been described by Hegland in [23].

3. Preliminaries.

3.1. Data Layout.

The efficiency of a parallel algorithm depends largely on the cost of data movement between processors. This cost, in turn, depends both on the initial mapping of data to processors and on the algorithms used for interprocessor communication. In this section, we describe the data distribution we use in our algorithm and contrast it with the distributions supported in the ScaLAPACK project.

A critical first step in devising a parallel linear algebra algorithm is deciding which matrix elements are assigned to which processors. We chose to use a *square torus-wrap mapping*, a special case of the torus-wrap mappings that have been successfully applied to a variety of dense linear algebra problems.

One way to describe a square torus mapping is to consider a square tile of size $\sqrt{P} \times \sqrt{P}$, where P is the number of processors in the parallel machine. Each element of the tile is assigned a unique value between 0 and $P-1$. The matrix is then covered with these tiles, and a matrix element is assigned to the processor whose tile number overlays it. In this way, each row and each column of the matrix is distributed among $\sqrt{P}$ processors. The matrix elements owned by a particular processor lie at the intersection of a set of rows and a set of columns of the matrix. Note that for a symmetric matrix, the array of elements owned by a processor is the transpose of the array owned by the processor in the transpose location on the tile. Equation (3.1) shows how the elements of a 7×7 matrix A would be distributed among $9 = 3^2$ processors according to a square torus mapping. The number in position (i, j) of the matrix M shows the processor to which matrix element $A(i, j)$ is assigned.

$$M = \begin{pmatrix} 0 & 3 & 6 & 0 & 3 & 6 & 0 \\ 1 & 4 & 7 & 1 & 4 & 7 & 1 \\ 2 & 5 & 8 & 2 & 5 & 8 & 2 \\ 0 & 3 & 6 & 0 & 3 & 6 & 0 \\ 1 & 4 & 7 & 1 & 4 & 7 & 1 \\ 2 & 5 & 8 & 2 & 5 & 8 & 2 \\ 0 & 3 & 6 & 0 & 3 & 6 & 0 \end{pmatrix} \tag{3.1}$$

At first glance, it seems that this mapping precludes the use of block algorithms and level 3 BLAS since the elements owned by a processor are not contiguous in the matrix. But this widely held belief is wrong, as has been observed by several researchers [8, 45]. The algorithms we describe below for tridiagonalization and backtransformation use level 3 BLAS and the square torus-wrap mapping. A more detailed discussion of the properties of that mapping can be found in [26] and the references therein.

Our reasons for using a square torus-wrap mapping are threefold. First, compared to a row or column mapping, a torus mapping reduces the communication requirements by a factor of $\sqrt{P}$ [26]. Second, the square torus allows for an efficient implementation of matrix-vector multiplication [25] which is an important step in the tridiagonalization algorithm. And third, a square torus simplifies the exploitation of matrix symmetry. On the other side of the ledger, insisting on a square torus restricts our algorithm to run on a number of processors that is a perfect square. This restriction limits the machine sizes on which our software can run. For instance, the Intel Paragon at Sandia National Labs has 1840 processors, but we are only able to use 1764 of them – a loss of over 4%.

The ScaLAPACK eigensolver supports a range of data distributions that are in many ways more general than ours but in a crucial way more restrictive. First, ScaLAPACK does not insist upon a square tile but rather allows a tile of arbitrary rectangular shape. We chose not to allow this generality because it significantly complicates the code and because, in practice, the tile must be nearly square for optimal performance to be obtained [26].

The second manner in which our code differs from ScaLAPACK is that the latter employs a *block* torus mapping. In this decomposition, the matrix is first decomposed into square blocks, and these blocks are assigned to processors in a torus-wrap manner. In this case, each element in the tile described above covers a $d \times d$ block of the matrix instead of a single entry. Thus, the mapping our code supports is a special case of the block torus mapping in which $d = 1$. We say that ScaLAPACK employs *storage blocking* while our approach does not.

To obtain peak performance on processors with a memory hierarchy, it is necessary to avoid cache misses as much as possible. In dense linear algebra calculations, this is achieved by recasting algorithms to be rich in matrix–matrix operations carried out by level 3 BLAS routines. The archetypical level 3 routine is the `dgemm` operation in which an $m \times n$ matrix A is updated by the product of matrices U and V^T of respective sizes $m \times k$ and $k \times n$ ($A = A + UV^T$). Typically, k is much smaller than m and n, but making it larger than 1 allows for cache reuse and greater overall performance. In practice, this means that matrix updates have to be delayed and performed in blocks instead of singly. We refer to this organization of operations as *algorithmic blocking.*

In principle, the concepts of storage blocking and algorithmic blocking are completely independent. But as a practical matter, a code that completely decoupled them would be painfully complex. Our code and ScaLAPACK avoid this complexity in different ways. Both codes allow any algorithmic blocking, but ScaLAPACK requires that the storage blocking factor be equal to algorithmic blocking factor. We instead restrict storage blocking to be equal to 1.

Our approach has several advantages when compared to the ScaLAPACK approach and one principle disadvantage. The first and probably most important advantage is that our mapping is simpler from a user's perspective. Not only is there one less degree of freedom in the mapping, but also the user can alter algorithmic blocking without having to redistribute the matrix among processors. Of course, if our mapping is inconvenient for a particular application, than the broader range of ScaLAPACK schemes could prove to be advantageous. Second, in linear algebra operations involving several matrix transformations, the algorithmic blocking size can take on optimal values for each transformation independently. When the algorithmic blocking is constrained by the storage blocking, changing the algorithmic blocking factor requires a redistribution of the matrix. The symmetric eigenproblem is such a multistep operation, and so it is an excellent problem with which to investigate these tradeoffs. Third, for a *dtimesd* storage block, the amount of load imbalance is generally proportional to d [26]. If a processor needs large blocks to obtain peak performance, this load imbalance can significantly degrade performance, as was observed by Strazdins for LU factorization on the AP1000 [42].

The commonly stated advantage of block storage is that it is necessary for block algorithms. As we discussed above, this supposed advantage is a mirage. However, in some settings block storage does have a distinct, though not widely appreciated, advantage over our approach. Most linear algebra algorithms modify a contiguous block of matrix elements or a block of columns and then use this block to transform the remainder of the matrix. In some cases, the modification of the block does not effect the other matrix elements. If so, and if the block is owned by a single processor, then the block can be sent to other processors in a single message. In this way, the number of messages can be reduced (although not the total volume of communication), thereby speeding up the algorithm.

As a concrete example, consider LU factorization with and without pivoting. Without pivoting, a diagonal block of the transformation can be generated without any information about the remainder of the matrix. This block can then be sent to other processors in a single message and used to update the other matrix elements. With pivoting, the transformation is not independent of the remainder of the matrix. Updates must be applied before the next pivot element can be selected. Thus, messages cannot be bundled into a single large message but rather must be sent individually after each column is transformed.

In this respect, Householder tridiagonalization is like LU factorization with pivoting while backtransformation is like LU factorization without pivoting. Thus, while our storage scheme is advantageous for tridiagonalization, it is disadvantageous for backtransformation. Since we do not use block storage, our approach requires communication after each column operation in the backtransformation. In contrast, the ScaLAPACK implementation of backtransformation requires communication only at the end of each block. It is this fundamental difference between tridiagonalization and backtransformation that makes the symmetric eigenproblem an ideal vehicle by which to investigate the significance of these algorithmic tradeoffs.

3.2. Communication Operations. Our parallel algorithm uses five distinct communication patterns, three of which are basic and two slightly more complex. The first basic pattern is *point-to-point* communication in which one processor sends a message to another. In our algorithm the pair of interacting processors are those lying in transpose positions in the tile defining the square torus mapping. Point-to-point communication is a primitive operation on any message passing parallel computer.

The second communication pattern is a *broadcast* where a processor sends the same message to several others. Our broadcasts involve all the $\sqrt{P}$ processors in a single row or single column of the square torus tile. This operation is performed in $\lceil \log_2 \sqrt{P} \rceil$ stages and can generally be performed without any contention for wires. Consider, for example, the simple situation in which the $\sqrt{P}$ processors are connected by a single path of wires. In the first step, the processor originating the message sends it to the processor that is $\sqrt{P}/2$ wires away. In the next step, both processors send it to their counterparts which are $\sqrt{P}/4$ wires away, and so on.

Our third communication operation is *global summation* in which each processor begins with a value and ends with the sum of all the processors' values. In MPI, this is the `Allreduce` operation. We used our own implementation based on an exchange operation that requires $\lceil \log_2 P + 1 \rceil$ stages. Let $q = 2^k$ be the largest power of two no larger than P. In the first stage, each processor numbered at least q sends its value to a processor numbered less than q. Each receiving processor adds the arriving value to its own. In each of the subsequent k stages, each processor r numbered less than q exchanges its value with a processor whose number differs from r in a single binary bit and adds the received value to its own running total. At the end of these stages, all processors numbered less than q hold the sum of all the values. In a final stage, this sum is communicated back to the processors numbered at least q.

The fourth and fifth communication patterns required by our algorithm are special cases of the MPI functions `Reduce_scatter` and `Allgather`. They are also commonly known as *recursive halving* and *recursive doubling*. As with broadcasting, we only use these operations among processors in a row or a column of the square torus tile.

In the recursive halving operation, each of P processors begins with a vector of length n and completes with n/P elements of the element-wise sum of the P vectors. In the recursive doubling operation, each of P processors begins with a vector segment of length n/P and ends with the vector of length n formed by concatenating the P vector segments in order.

Previous descriptions of these algorithms rely on having a number of processors that is a power of two [21, 44]. In contrast, our algorithm allows for an arbitrary numbers of processors. In our approach, each processor sends and receives a total of n values, while using at most $2\log_2 P$ messages. If the number of processors is a power of two, then our algorithm reduces to the familiar one in which only $log_2 P$ messages are required. The details of our approach are beyond the scope of this paper but can be found in the more complete technical report [24].

4. Householder Tridiagonalization.

4.1. The Sequential Algorithm.

The best sequential algorithm for tridiagonalizing a symmetric matrix uses a sequence of Householder transformations. Let A be an $n \times n$ symmetric matrix, and let H_1^T be the Householder transformation that zeros elements in the first column of A below the subdiagonal by pivoting on the subdiagonal element. It is easy to see that $H_1^T A H_1$ also zeros the elements in the first row of A to the right of the superdiagonal. We can then construct H_2 to zero the subdiagonal elements of the second column, and so on. After $n-2$ steps of this process, the matrix is reduced to tridiagonal form. An efficient sequential algorithm for this calculation is shown in Fig. 4.1. In this algorithm, the normalization constants are calculated and applied in nonstandard fashion. Although this makes the presentation somewhat more involved than that found in [22], it simplifies the parallel algorithm.

Steps (1–5) of this algorithm comprise the construction of the Householder vector v. Step (1) is merely for notational convenience and is not executed. Also, the entries of the Householder vectors are actually stored in the lower triangle of A for later use in the backtransformation. The off–diagonal entries of the tridiagonal matrix are generated in step (4) and stored in the vector z while the diagonal entries are kept in the diagonal of A. In step (6), we save the norm of the Householder vector for later use while backtransforming the eigenvectors. The two computationally dominant steps in the algorithm are the matrix–vector multiplication in the generation of the updating vector p in step (7) and the rank–two update of A in step (10). Exploiting the symmetry of A allows us nearly to halve the operations required in the rank–two update (although it complicates the calculation of Av). The resulting algorithm

(∗ Symmetric $A \in \mathbb{R}^{n\times n}$; $x, v, w, p, z \in \mathbb{R}^n$. ∗)
For $i = 1 : n-2$
(1) $v = A(i+1:n, i)$
(2) $\mu = v^T v$
(3) $v(1) = v(1) + \mathrm{sign}(v(1))\sqrt{\mu}$
(4) $z(i+1) = -\mathrm{sign}(v(1))\sqrt{\mu}$
(5) $\phi = 2(\mu + |v(1)|\sqrt{\mu})$ (∗ $\phi = v^T v$ ∗)
(6) $\Phi(i) = \phi$
(7) $p = A(i+1:n, i+1:n)v$
(8) $\gamma = p^T v$
(9) $w = \frac{2}{\phi}(p - \frac{\gamma}{\phi} v)$
(10) $A(i+1:n, i+1:n) = A(i+1:n, i+1:n) - vw^T - wv^T$

FIG. 4.1. *Sequential algorithm for Householder tridiagonalization.*

requires $4n^3/3 + O(n^2)$ flops [22].

The algorithm in Fig. 4.1 employs only the vector–vector and matrix–vector operations known as level 1 and level 2 BLAS [16, 14]. On processors with a memory hierarchy, it is generally more efficient to recast algorithms to allow for greater memory reuse. Dongarra, Sorensen and Hammarling [17] have shown how to modify this algorithm to use some matrix–matrix operations or level 3 BLAS [15]. The basic idea is to save the v and w vectors from several steps and use them to update A simultaneously. If we save b pairs of vectors, we can perform a rank–$2b$ update of A. Postponing updates complicates the generation of the vector p, since the values stored in the matrix A are not always up–to–date, but it allows for higher performance in the outer–product update of A. We note, however, that the generation of the vector p still requires level 2 BLAS. This block algorithm is summarized in Fig. 4.2.

The single loop of the algorithm shown in Fig. 4.1 is now replaced with two loops, the outer one over blocks of columns and the inner one over columns within a block. Steps (1–6) of the algorithm in Fig. 4.2 precisely mimic steps (1–6) of the algorithm in Fig. 4.1 except that for completeness we have made the indices more explicit. We have broken out the matrix–vector multiplication into steps (7.1–7.3) to show how to use only the lower triangular portion of the matrix. Since the matrix L is not updated immediately, steps (7.1–7.3) use an out–of–date version of the matrix. We correct for that in step (7.4). In step (10.1), the columns of L that lie within the current block are updated, but the update of the rest of the matrix is delayed until step (10.4). That update can be performed with a single call to the level 3 BLAS routine `dsyr2k`. We note that the call to `dsyr2k` requires the matrix to consume n^2 space, even though only the lower triangular portion is used. Without BLAS for blocked storage, exploiting symmetry saves flops but not space.

4.2. The Parallel Algorithm. Besides its intrinsic complexity, there are several features of the algorithm in Fig. 4.2 that make it challenging to implement in parallel. First, the matrix–vector multiplication in the generation of the vector p is more complicated to parallelize than the simple updates required in LU or QR factorization [26]. Second, exploiting symmetry both increases the communication requirements of the algorithm and makes the design of an efficient matrix–vector multiplication step more difficult. Finally, the outer–product update required in our parallel algorithm is *not* included in the standard suite of level 3 BLAS. As we will see, the portion of A owned by a processor may be nonsymmetric, and there is no

(* $L \in \mathbb{R}^{n\times n}$ lower triangular part of symmetric matrix; $v, w, p, z \in \mathbb{R}^n$. *)
(* b is the number of columns in a block. *)
(* $V, W \in \mathbb{R}^{n\times b}$ are reused each outer iteration. *)
$N = \lceil (n-2)/b \rceil$
For $j = 1 : N$
    $V = [\emptyset]$, $W = [\emptyset]$
    $s = (j-1)b + 1$ (* First column in block. *)
    $t = \min(n-2, s+b-1)$ (* Last column in block. *)
    For $i = s : t$ (* Form the block of updates. *)
(1) $v(i+1:n) = L(i+1:n, i)$
(2) $\mu = v(i+1:n)^T v(i+1:n)$
(3) $v(i+1) = v(i+1) + \mathrm{sign}(v(i+1))\sqrt{\mu}$
(4) $z(i+1) = -\mathrm{sign}(L(i+1,i)\sqrt{\mu}$
(5) $\phi = 2(\mu + |v(i+1)|\sqrt{\mu})$ (* $\phi = v^T v$ *)
(6) $\Phi(i) = \phi$
(7.1) $x(i+1:n) = L(i+1:n, i+1:n)v(i+1:n)$ (* Without diagonal. *)
(7.2) $y(i+1:n) = v^T(i+1:n)L(i+1:n, i+1:n)$ (* With diagonal. *)
(7.3) $p = x + y$
        (* Correct for out–of–date entries of L *)
(7.4) $p(t+1:n) = p(t+1:n) - (V(t+1:n,*)W(t+1:n,*)^T +$
        $W(t+1:n,*)V(t+1:n,*)^T)v(t+1:n)$
(8) $\gamma = p(i+1:n)^T v(i+1:n)$
(9) $w(i+1:n) = \frac{2}{\phi}(p(i+1:n) - \frac{\gamma}{\phi}v(i+1:n))$
        (* Update remainder of block of L. *)
(10.1) $L(i+1:n, i+1:t) = L(i+1:n, i+1:t) - v(i+1:n)w(i+1:t)^T -$
        $w(i+1:n)v(i+1:t)^T$
(10.2) $V = [V \ v]$
(10.3) $W = [W \ w]$
    (* Perform symmetric rank–$2b$ update of the submatrix. *)
(10.4) $L(t+1:n, t+1:n) = L(t+1:n, t+1:n) - V(t+1:n,*)W(t+1:n,*)^T -$
        $W(t+1:n,*)V(t+1:n,*)^T$

FIG. 4.2. *Block algorithm for Householder tridiagonalization.*

level 3 BLAS routine to perform a nonsymmetric, outer–product update of a lower triangular matrix.

4.2.1. The Tridiagonalization Algorithm. Our parallel algorithm for Householder tridiagonalization for processor q is depicted in Fig. 4.3. The labels to the left of the executable statements correspond to the numbers of the corresponding operations in Fig. 4.2. Although the basic structure of this algorithm mimics that of the algorithm in Fig. 4.2, the parallelization adds considerable complexity. The parallel algorithm is complicated by the fact that the set of matrix elements owned by a processor are not symmetric. That is, a processor holds all the matrix elements in the intersection of a set of rows α and a set of columns β, but these two sets are not generally the same. However, because our tile is square, there is always another processor that contains elements at the intersection of rows β and columns α. These two processors thus own transpose portions of the matrix.

Since the matrix elements assigned to a processor do not form a symmetric submatrix, the processor needs two different sets of components of the Householder vector v in order to compute $L(\alpha,\beta)v_\beta$ and $v_\alpha^T L(\alpha,\beta)$ in steps (7.1) and (7.2). This leads to the communication operations in steps (Y1). Similarly, the results of the two matrix–vector products contribute to different sets of elements of the vector p, and the communication in step (7.3) realigns these indices. A similar communication of the vector w is required in steps (Y2) to prepare for the outer–product update of the matrix A.

Another consequence of processors having nonsymmetric portions of A is that the outer–product update is nonsymmetric. Because there is no level 3 BLAS call to update a lower triangular matrix with a nonsymmetric outer–product, we perform the update by covering the lower triangular matrix with a sequence of rectangular panels and use a standard rectangular outer product update on each panel. The full rank–$2b$ update of a panel can be performed with a single call to the level 3 BLAS routine `dgemm`. Note that this scheme modifies some memory locations outside the storage of the lower triangular matrix (and performs some corresponding unnecessary floating point operations.) For the tests described in this paper, we used panel widths of 2 seemed to give the best performance.

An additional detail of the parallel algorithm is shown in the steps labeled (X). Since only a subset of processors owns each row and column of the matrix, the row and column incrementing requires more attention than in the sequential code.

This algorithm requires several different types of communication operations. We describe and analyze them here for the case where L has $(k+1)$ unreduced rows and columns remaining. The total communication in the algorithm is the sum of these communication steps for $k = 1,\ldots,n-2$. In step (1), $\sqrt{P}$ broadcasts occur, each involving $\sqrt{P}$ processors and vectors of length $\lceil k/\sqrt{P}\rceil$. This requires $\lceil \log_2(P)/2\rceil$ startups and $\lceil k/\sqrt{P}\rceil\lceil \log_2(P)/2\rceil$ volume for the slowest processor. Steps (2) and (8) include global sums of scalars, while step (7.4) involves a global sum of a vector of length $i-s$. Each global sum requires $\lceil \log_2(P)\rceil$ startups for each processor. The messages are of size 1 in steps (2) and (8), and size $i-s$ in step (7.4). In steps (Y1), (Y2) and (7.3), a vector of length $\lceil k/\sqrt{P}\rceil$ is transposed. Each transpose operation requires a single startup and a volume of $\lceil k/\sqrt{P}\rceil$. The final two operations are the recursive halving in step (7.3) and the recursive doubling in step (9). Each of these operations requires up to $\lceil \log_2(P)\rceil$ startups and about $\lceil k/\sqrt{P}\rceil$ volume.

Counting up all these operations and summing over k, we find that the communication operations require time for about $n(5.5\log_2 P+3)$ message startups and the transmission of about $n^2(\log_2(P)+10)/4\sqrt{P}$ double precision values. The logarithmic term in the communication volume is due to the use of a binomial tree in the broadcast of Householder vectors. For the LU or QR decomposition, an alternative would be to use a ring shift approach, essentially piping the data through the processors. Processors would not be synchronized, but the communication cost could be reduced. Unfortunately, this is not a viable option for tridiagonalization because the matrix–vector multiplication serves to synchronize the processors. Another option would be to perform the broadcast via recursive halving followed by a recursive doubling. With this approach the communication volume is proportional to n. In practice, however, a large number of processors need to be involved before the logarithmic term dominates the communication cost.

The communication complexity of our algorithm compares quite favorably with that of ScaLAPACK [41]. The ScaLAPACK routine `PDSYTRD` requires time for about $8.5n\log_2(P)$ message startups and $2.5n^2(\log_2(P)+1)/\sqrt{P}$ volume. The dramatic difference in communication volume is principally due to our use of more efficient communication kernels. Another advantage of our approach is that we are able to use all P processors for every computationally significant step. In `PDSYTRD`, the computation of the Householder vectors is only distributed across $\sqrt{P}$ of the processors [41].

As stated, our algorithm requires a fairly large amount of space for temporary vectors. However, with a careful implementation, the space in V_α, V_β, W_α and W_β is sufficient to hold all the temporary vectors. What remains is the storage of these four arrays and the storage of L, z and Φ. This makes for a total of $\lceil n/\sqrt{P}\rceil(\lceil n/\sqrt{P}\rceil + 4b+2)$

This algorithm involves communication among rows and columns of the torus–wrap tile as well as between transpose processors. Without careful mapping to the parallel architecture, these operations may induce substantial contention for wires in the communication network. However, we note that on a hypercube, a judicious choice of processor assignments ensures that none of these operations induces congestion [25]. Although this isn't true on other topologies of parallel machines, we will assume that the communication cost is proportional to the volume calculated above.

A careful analysis reveals that there is only $\theta(nP)$ redundant numerical computation in this algorithm. Since the computation is well balanced, the total computation time should scale as $\frac{4}{3}n^3/P+O(n^2)$. The ratio of computation to communication thus is thus $n/\sqrt{P}(log_2(P)+10)$. To retain constant efficiency as the number of processors increases, n must increase slightly faster than $\sqrt{P}$, and the total amount of memory per processor must increase as $(log_2(P)+10)^2$.

5. The Tridiagonal Eigensolver. To solve the symmetric tridiagonal eigenvalue problem, we use the method of bisection to compute the eigenvalues of T followed by the method of inverse iteration to compute its eigenvectors [1]. This combination seems particularly well-suited to parallel implementation as the eigenpairs can often be computed essentially independently of one another. Indeed, if n/P eigenpairs are assigned to each processor, it appears that no interprocessor communication is required.

When any of the eigenvalues are close, however, inverse iteration cannot produce orthogonal eigenvectors. In this case, the modified Gram-Schmidt (MGS) procedure is used to reorthogonalize the computed eigenvectors. While bisection and inverse iteration are both $\mathrm{O}(n^2)$ algorithms, MGS is $\mathrm{O}(n^3)$. Furthermore, if the close eigenvalues are distributed between processors their eigenvectors cannot be reorthogonalized without communication [1]. Two parallel reorthogonalization schemes are described in [29, 20]. The ScaLAPACK implementation of inverse iteration (PDSTEIN) excludes interprocessor reorthogonalization but does attempt not to split close eigenvalues between processors.

Any variant of parallel reorthogonalization can represent a substantial bottleneck for some distributions of eigenvalues. A better approach to the symmetric tridiagonal problem would replace bisection and inverse iteration with the altogether different method described in [13]. Based on accurate matrix factorization techniques, this brand new method allows independent computation of eigenvalues and orthogonal eigenvectors regardless of the matrix spectrum. The serial implementation of this algorithm requires $\mathrm{O}(n^2)$ time. An LAPACK implementation is pending, and an efficient parallel implementation could follow almost trivially from that.

For the purpose of this paper, we use the naive approach to bisection and inverse iteration described in first paragraph of this section: each processor computes n/P

consecutive eigenvalues and their corresponding eigenvectors. This computation is carried out by independent calls to the LAPACK routines DSTEBZ and DSTEIN [1]. No interprocessor reorthogonalization is performed. The calls to LAPACK routines are preceded by a global exchange of the tridiagonal matrix so that it is in place on all processors. They are followed by a redistribution of the eigenvectors from the block column arrangement in which they are transformed to the torus–wrap distribution required by the backtransformation algorithm.

We recognize that our approach to the tridiagonal eigenproblem is not robust, and we plan to replace our tridiagonal eigensolver with improved code as it becomes available.

6. Backtransformation.

6.1. The Sequential Algorithm.

The purpose of the backtransformation is to compute the eigenvectors QX of A from the eigenvectors X of T. Since Q is comprised of a sequence of Householder transformations, we need merely apply these transformations to T in the reverse of the order in which they were applied to A [22]. Because the eigenvector matrix X is nonsymmetric and the basic operations are less complex, the algorithm for backtransformation is much simpler than that for tridiagonalization.

As with our algorithm for tridiagonalization, we would like to use level 3 BLAS to achieve higher performance. As before, this can be accomplished by saving several updates and applying them all at once. Doing so is considerably simpler for backtransformation than it was for tridiagonalization. In particular, we use the WY representation of the sequence of Householder transformations [22]. The sequential algorithm is sketched in Fig. 6.1.

In steps (1–2), the sequential algorithm merely retrieves information it generated in the tridiagonalization. The WY representation of the sequence of Householder transformations is generated in steps (3–5) and applied to the matrix of eigenvectors in step (6). Note that we use a nonstandard version of the WY representation in which the Householder operations are combined in the reverse of the usual order. Since the individual Householder matrices are symmetric, reversing the order in this way is equivalent to transposing the product. This allows us to apply YW^T instead of the more familiar WY^T in steps (4) and (6). The advantage of our approach is that our Y and W matrices are lower triangular so step (6) can be performed with two invocations of the level 3 BLAS routine `dtrmm`, consuming a total of $2kn^2$ flops, where k is the number of eigenvectors being transformed. For our implementation, we actually chose not to exploit this efficiency because `dgemm` on the Paragon is significantly faster than the BLAS routines for triangular matrices. Presumably because of its importance in the LINPACK benchmark, `dgemm` has received more attention from the library developers.

When k is significantly larger than one, step (6) dominates the computation time. Thus, in contrast to the tridiagonalization routine, the bulk of the flops in backtransformation can be handled by calls to a level 3 BLAS routine. We also note that we need never explicitly form Y since the Householder vectors are already stored properly in L. This will not be the case in the parallel algorithm.

6.1.1. The Parallel Algorithm.

Although our parallel backtransformation algorithm has the same structure as the sequential algorithm in Fig. 6.1, the parallelism inevitably adds some complexity. We use the Householder vectors and their norms as generated and stored by the parallel tridiagonalization algorithm in Fig. 4.3. We store the eigenvector matrix X in the same square torus–wrap manner in which we distribute L. This arrangement keeps the communication cost of the algorithm low. Unfortunately, both the tridiagonal eigensolvers described in §5 leaves the eigenvector matrix X distributed by block columns so that it must be redistributed before the backtransformation begins. This redistribution is conceptually straightforward so we will not describe it in detail except to make the following observation. The operations in backtransformation are independent of the ordering of the eigenvectors in X, so we are not required to keep them in any particular order. If we let a particular column set β correspond to the eigenvectors computed by $\sqrt{P}$ processors, then these $\sqrt{P}$ processors need only communicate among themselves to remap X. Exploiting this observation allows us to perform the redistribution by having communication among the $\sqrt{P}$ processors in each of $\sqrt{P}$ sets, instead of requiring communication between all processors.

Our parallel backtransformation algorithm is depicted in Fig. 6.2, with the statement labels matching those in Fig. 6.1. The algorithm blocksize is denoted B to differentiate it from that used in the tridiagonalization algorithm. As mentioned in §3, our code allows these two blocksizes to be tuned independently.

The first notable difference between the parallel and sequential algorithms is indicated by the steps marked (X) which have no sequential counterparts. These steps keep track of the rows of the eigenvector matrix owned by a particular processor that are modified by the Householder transformation. In the sequential case, this number of active rows simply increases by one with each step.

The second and more substantial complication induced by parallelism occurs in the generation of the vector z_α in step (4). This computation proceeds in two phases: first, the vector $r = W^T v$ is computed and then z_α is obtained by $z_\alpha = (-2/\phi)(v_\alpha + Y_\alpha r)$. Note that each processor owns entire rows of W and Y and that these are distributed so that $\sqrt{P}$ processors could compute the same product $W_\alpha^T v_\alpha$. To avoid this redundant computation, we assign each processor a distinct subset of α, and each computes an independent contribution to the vector r. These results are then summed among all the processors. Computation of the product $Y_\alpha r$ is similarly distributed. Each processor computes a different set of elements of the product, and these results are concatenated within processor rows to produce the vector z_α.

The final change in the parallel algorithm happens in step (6), the update of the eigenvectors. As with step (4), this operation is broken into two steps, the multiplication by W_α^T followed by the multiplication by Y_α. Since each processor owns a distinct set of elements of X, there is no concern here with duplicating computation. Each processor first computes its own values for the $B \times (k/\sqrt{P})$ matrix product $W_\alpha^T X$ using a call to `dgemm`. Although the complete W matrix is lower triangular, the portion owned by a particular processor does not have such a nice structure, so we can't use `dtrmm`. The values in this product must be summed among the processors owning a column of X to yield Z_β. Since there may be a large number of values being summed, we perform this with a recursive halving, followed by a recursive doubling to avoid duplication of numerical operations. Each processor can then update its elements of X independently via another call to `dgemm`.

Although this organization does not allow us to exploit the triangular structure of W and Y, the total amount of unnecessary computation is $O(kB^2)$ when k eigenvectors are backtransformed. As long as the blocksize B is much less than n, the impact of the redundant computation is not significant, and typical values of B are less than 10. The total number of floating point operations in the backtransformation algorithm is $2kn^2 + O(kB^2)$ (we assume that $n \gg B$). The flops are well balanced among processors, and the dominant calculation is entirely within level 3 BLAS.

The communication occurs in steps (1&2), (4) and (6). As with the parallel tridiagonalization algorithm, this backtransformation approach requires the processors to remain synchronized, so we can't perform the broadcast of Householder vectors as a ring shift efficiently. Instead, we use a binomial tree which requires a total of $n \log_2(P)/2$ message startups and $n^2 \log_2(P)/4\sqrt{P}$ volume. In step (4), the summation of r vectors requires a total of $n \log_2(P)$ messages and $nB \log_2(P)/2$ volume, while the recursive doubling costs about $n \log_2(P)$ startups and $n^2/2\sqrt{P}$ volume. The recursive halving and doubling in step (6) requires about $2n \log_2(P)$ messages and a total communication volume of $2nk/\sqrt{P}$. Summing these terms, we find overall communication requirements about $4.5n \log_2(P)$ message startups and $n(n \log_2(P)/4 + n/2 + 2k)/\sqrt{P}$ volume. As with our tridiagonalization algorithm, the logarithmic broadcast of Householder vectors is the dominant communication cost asymptotically.

`PDORMTR` is the ScaLAPACK routine for backtransformation. As discussed in §3.1, `PDORMTR` exploits the block format to reduce the number of messages, but not the total volume. Consequently, for the case in which all the eigenvectors are computed (k equals n), the number of messages is about $2.5n \log_2(P)/B$ while the total volume is $n^2(\log_2(P)+1)/sqrtP$ [41]. Although our communication volume is substantially less, this is paid for with a larger number of messages. Note in particular that for `PRORMTR` the number of messages is inversely proportional to the blocking factor. As the results in §7 will indicate, this difference in the number of messages has a significant impact upon performance.

Like the tridiagonalization algorithm, the temporary vectors v_α and z_α can use space inside Y_α and W_α. Besides W and Y, the algorithm needs space for L, Φ, X and Z, consuming a total of $\lceil n/\sqrt{P} \rceil (\lceil n/\sqrt{P} \rceil + \lceil k/\sqrt{P} \rceil + 3B + 1)$ double precision values.

7. Experimental Results. We have implemented our algorithm on the Intel Paragon, a mesh-connected MIMD computer at Sandia's Massively Parallel Computing Research Laboratory, running the SUNMOS operating system. Although each logical node in the Paragon has two processors, the results reported below for our code and its counterpart in ScaLAPACK Version 1.0 use only a single processor per node. Our results are for random matrices with entries selected uniformly between -1 and 1. In all tests, all eigenvectors of the tridiagonal matrix are computed and backtransformed (i.e., $k = n$), although no times for the tridiagonal problem are reported. All reported times for backtransformation also include the time for remapping the eigenvectors of the tridiagonal matrix into the distribution required by the backtransformation routine used. We developed our code for experimental purposes only, and it is not available for public distribution.

We first ran a sequence of factorizations of a 4000×4000 matrix on 64 processors in which we varied the algorithmic blocksize (from 1 through 50). The run times for our Householder tridiagonalization and backtransformation routines (HJS) and for the ScaLAPACK tridiagonalization and backtransformation routines (SLP) are shown in Fig. 7.1. All four routines reach a minimum runtime for blocksizes near 10, and the HJS routines show little variation in runtime for larger blocksizes. The ScaLapack routines, on the other hand, show a marked increase in runtime for the larger blocksizes. In particular, at a blocksize of 10, the ratio of the time for SLP to the time for HJS is 1.56. This ratio grows to 1.78 for a blocksize of 50. This increase is due to the load imbalance which accompanies ScaLAPACK's use of storage blocking.

Clearly, an implementation with algorithm blocking but no storage blocking (like HJS) scales better as the blocksize is increased.

Another notable feature of the figure is the variation in optimal block size for the different algorithms. In these experiments, the best blocksize for the backtransformation is smaller than that for the tridiagonalization. However, the code performance is very similar over a range of blocksizes. So the freedom within HJS to use different blocksizes for the different algorithms seems to be of only modest value. For fairness in comparisons, we used a blocksize of 10 in all subsequent experiments.

Next, we ran two different fixed-size problems on a range of machine sizes (with blocksize of 10). The tridiagonalization results are shown in Fig. 7.2, while those for backtransformation are in Fig. 7.3. The most notable point about the tridiagonalization times is that HJS is significantly faster than SLP, even on a single processor. We do not have an adequate explanation for this difference, but it may be due to SLP's need to work for an arbitrary $q \times r$ tile of processors, while HJS is limited to a square tile. The SLP generality induces code complexity which could lead to reduced performance. The shape of the tridiagonalization curves is quite similar, indicating similar scalability in the two implementations.

In contrast to the tridiagonalization runtimes, the difference in backtransformation performance revealed in Fig. 7.3 is relatively small on modest numbers of processors. This similar performance provides evidence that the software overheads incurred by SLP for portability are small, at least when the problem size per processor is fairly large. As expected, the reduction in message number allowed by the block torus-wrap mapping translates to a better performance for SLP as the number of processors increases. For the 1500×1500 matrix on 100 processors, the SLP backtransformation code takes only 64% of the time required by HJS.

Although serial backtransformation requires 1.5 times the number of floating point operations of serial tridiagonalization the tridiagonalization takes much more time than the backtransformation for both HJS and SLP. This is due to several factors. While all the backtransformation flops are level 3, half the tridiagonalization flops are level 2. Also, the level 3 operation in the HJS implementation is performed in an awkward manner since the appropriate BLAS operation is missing. Finally, the amount of communication in the parallel tridiagonalization algorithm is significantly greater than the amount in backtransformation.

Thus, for block sizes of 10 and a range of problem sizes, HJS tridiagonalization is substantially faster than SLP while HJS backtransformation is somewhat slower. But since tridiagonalization is a much more costly operation, the overall performance of HJS is significantly better than that of SLP. Fig. 7.4 shows the total time for tridiagonalization and backtransformation versus number of processors for both HJS and ScaLapack for matrix order $n = 1500$. This plot does not include time for solution of the tridiagonal eigenproblem. The total time for ScaLapack is about 1.33 times greater than the time for HJS for all machine sizes.

Next, we ran a series of experiments in which the problem size grew with the number of processors so that the submatrix owned by each processor remained of size 500×500. Figs. 7.5 and 7.6 show the performance of the tridiagonalization and backtransformation codes respectively for this set of problems. In computing the flop/s numbers, we have assumed the optimal sequential flop count of $\frac{4}{3}n^3$ for tridiagonalization and $2n^3$ for backtransformation. The dotted diagonal line in the two figures corresponds to a perfect (scaled) speedup of the HJS code from its single processor level. Both codes scale fairly well. On a single processor, the HJS tridiagonalization runs at 31.0 Mflop/s, while it runs at 5.06 Gflop/s on 256 processors. This yields a scaled efficiency of 63.7%. The corresponding speeds for SLP are 18.6 Mflop/s on one processors and 3.34 Gflop/s on 256, implying a scaled efficiency of 70.0%. The performance of the HJS tridiagonalization is consistently 1.5 to 1.6 times that attained by SLP.

It is worth noting that the models of communication complexity only explain about half of the observed performance difference between the two codes. The remainder is presumably due to ScaLAPACK's inability to keep all processors busy, and to software overheads. Among existing parallel machines, the Paragon has atypically fast communication relative to processor speeds. Consequently, the performance gain of our approach due to communication reduction should be even larger on other platforms.

The comparatively simple backtransformation code scales better than the more complex tridiagonalization. On one processor the HJS backtransformation attains 45.4 Mflop/s, and on 256 it runs at 9.42 Gflop/s, implying a scaled efficiency of 81.1%. For SLP backtransformation, the performance is 42.4 Mflop/s on one processor and 9.37 Gflop/s on 256; a scaled efficiency of 86.3%.

8. Conclusions. We have described a parallel methodology for finding eigenvalues and eigenvectors of dense, symmetric matrices, focusing on the reduction to tridiagonal form and the backtransformation of the eigenvectors. The symmetric eigenproblem is an ideal vehicle for investigating the tradeoffs associated with algorithmic variants because it involves several computational stages with quite different properties. The efficiency of our approach is a result of three factors.

1. Minimizing the communication cost of the algorithm by the careful use of efficient communication kernels.
2. Using the full set of processors at all times.
3. Maximizing the use of level 3 BLAS kernels.

Although dense linear algebra operations have been well-studied, our results show that there is still room for new ideas.

Our experimental comparisons with the ScaLAPACK algorithms lead us to the following observations:

- Block storage is not necessary for block algorithms and level 3 performance. Indeed, the use of block storage leads to significant load imbalance when the blocksize is large. This is not a concern on the Paragon, but may be problematic for machines requiring larger blocksizes for optimal BLAS performance [42].
- Although unnecessary for block algorithms, block storage does enable a reduction in the number of messages (but not the total message volume) for some matrix transformations. Specifically, the ScaLAPACK backtransformation algorithm is superior to ours due to the use of block storage.
- It is essential to devise algorithms in which all the significant computation is divided among all the processors. We believe that this is one reason for the superior performance of our tridiagonalization routine.
- We believe that another reason is that we constrained our implementation to a square torus, thereby simplifying the code significantly. With parallel software, there is a tradeoff between performance and generality, and our results argue that the cost of generality can be significant.

While our implementation runs faster than other available software for a variety of problem and machine sizes, the efficiency of tridiagonalization is limited by the need for level 2 BLAS operations in forming the updating vector p. Its performance is further constrained by the absence of a BLAS 3 routine for performing a nonsymmetric outer product update of a lower triangular matrix. The most efficient alternative is to apply the dense matrix routine to panels which cover the lower triangular matrix.

Note that the nonsymmetric outer product update of a lower triangular matrix is not an operation unique to the parallel tridiagonalization routine. It also arises, for example, in the indefinite linear system solver of [33]. The software for that problem is similarly slowed by the lack of the appropriate BLAS 3 routine.

Acknowledgements. We are deeply indebted to Ken Stanley for insights into the tradeoffs associated with the use of block storage, and also for describing to us the ScaLAPACK eigensolver routines. We also appreciate the helpful discussions we had with Robert van de Geijn, Xiaobai Sun and David Womble about the tridiagonalization algorithm.

(* Processor q holds rows α and columns β of L, which is
the lower triangular part of $A \in \mathbb{R}^{n\times n}$. *)
(* $\Phi \in \mathbb{R}^{n/\sqrt{P}}$ saves norms of Householder vectors for backtransformation. *)
(* b is the number of columns in a block. *)
(* $V_\alpha, V_\beta, W_\alpha, W_\beta \in \mathbb{R}^{\frac{n}{\sqrt{P}}\times b}$ are reused each outer iteration. *)
(* $x_\alpha, y_\beta, y_\beta, p_\alpha, v_\alpha, v_\beta, w_\alpha, w_\beta \in \mathbb{R}^{n/\sqrt{P}}$. *)
$N = \lceil (n-2)/b \rceil$
For $j = 1 : N$
$V_\alpha = [\emptyset]$; $W_\alpha = [\emptyset]$; $V_\beta = [\emptyset]$; $W_\beta = [\emptyset]$
$s = (j-1)b + 1$ (* First column in block. *)
$t = \min(N-2, s+b-1)$ (* Last column in block. *)
For $i = s : t$ (* Form the block of updates. *)
(X) **If** $i \in \alpha$ **Then** $\alpha = \alpha \setminus \{i\}$ (* Remove i from active rows. *)
(* Generate and broadcast next Householder vector. *)
(1) **If** $i \in \beta$ **Then**
(1) $v_\alpha = L(\alpha, i)$
(1) Broadcast v_α to processors sharing rows α
(1) **Else** Receive v_α
(2) μ^q = independent contribution to $v^T v$
(2) Sum μ^q contributions to form μ, and append $v(i+1)$ to communication
(3) **If** $i+1 \in \alpha$ **Then** (* Update first element of v. *)
(3) $v(i+1) = v(i+1) + \mathrm{sign}(v(i+1))\sqrt{\mu}$
(4) **If** $i \in \beta$ **Then** (* Compute subdiagonal element of T. *)
(4) $L(i+1, i) = -\mathrm{sign}(L(i+1, i))\sqrt{\mu}$
(5) $\phi = 2(\mu + |v(i+1)|\sqrt{\mu})$
If $i \in \beta$ **Then**
(6) $\Phi(\lceil i/\sqrt{P} \rceil) = \phi$ (* Save norm for backtransformation. *)
(X) $\beta = \beta \setminus \{i\}$ (* Remove i from active columns. *)
(* Get elements of v^T to the correct processors. *)
(Y1) **Send** v_α to transpose processor
(Y1) **Receive** v_β from transpose processor
(7.1) $x_\alpha = L(\alpha, \beta) v_\beta$ (* Without diagonal. *)
(7.2) $y_\beta = v_\alpha^T L(\alpha, \beta)$ (* With diagonal. *)
(7.3) **Send** y_β to transpose processor
(7.3) **Receive** y_α from transpose processor
(7.3) $p_\alpha = x_\alpha + y_\alpha$
(7.3) Recursive halve p_α within rows yielding p_η
(* Correct for out–of–date entries of L. *)
(7.4) $\delta^q = V_\eta^T v_\eta$
(7.4) $\epsilon^q = W_\eta^T v_\eta$
(7.4) Sum δ^q and ϵ^q among all processors yielding δ and ϵ
(7.4) $\zeta = \eta \cap (t+1 : N)$ (* Just indices outside block. *)
(7.4) $p_\zeta = p_\zeta - \delta V_\zeta - \epsilon W_\zeta$
(* Compute w. *)
(8) $\gamma^q = p_\eta^T v_\eta$
(8) Sum γ^q among all processors yielding γ
(9) $w_\eta = \frac{2}{\phi}[p_\eta - \frac{\gamma}{\phi} v_\eta]$
(9) Recursive double w_η within rows yielding w_α
(Y2) **Send** w_α to transpose processor
(Y2) **Receive** w_β from transpose processor
(* Update remainder of block of L. *)
(10.1) $\nu = \beta \cap (i+1 : t)$ (* Columns within current block. *)
(10.1) $L(\alpha, \nu) = L(\alpha, \nu) - v_\alpha w_\nu^T - w_\alpha v_\nu^T$
(10.2) $V_\alpha = [V_\alpha \;\; v_\alpha]$; $V_\beta = [V_\beta \;\; v_\beta]$
(10.3) $W_\alpha = [W_\alpha \;\; w_\alpha]$; $W_\beta = [W_\beta \;\; w_\beta]$
(* Update remainder of matrix. *)
(10.4) $L(\alpha, \beta) = L(\alpha, \beta) - V_\alpha W_\beta^T - W_\alpha V_\beta^T$

FIG. 4.3. *Parallel block algorithm for Householder tridiagonalization for processor q.*

(* $X \in \mathbb{R}^{n\times k}$ matrix of eigenvectors; $v, w, p \in \mathbb{R}^n$. *)
(* $L \in \mathbb{R}^{n\times n}$ matrix of Householder vectors. $\Phi \in \mathbb{R}^n$ Householder norms. *)
(* $W, Y \in \mathbb{R}^{n\times b}$ are reused each outer iteration. *)
(* b is the number of columns in a block. *)
$N = \lceil (n-2)/b \rceil$
For $j = N : 1 : -1$
    $W = [\emptyset]$; $Y = [\emptyset]$
    $s = (j-1)b + 1$
    $t = \min(n-2, s+b-1)$
    For $i = t : s : -1$ (* Form the block of updates. *)
(1) $\phi = \Phi(i)$ (* $\phi = v^T v$, saved from tridiagonalization. *)
(2) $v = L(i+1:n, i)$ (* Householder vector from column i of L. *)
        If $(i = t)$ **Then**
(3) $z = \frac{-2}{\phi} v$
        Else
(4) $z = \frac{-2}{\phi}(I + YW^T)v$
(5) $W = [z \;\; W]$; $Y = [v \;\; Y]$
    (* Update eigenvectors. *)
(6) $X = (I + YW^T)X$

FIG. 6.1. *Sequential block algorithm for backtransformation*

(* Processor q owns row set α_0 and column set β of
eigenvectors $X \in \mathbb{R}^{n\times k}$ and Householder vectors $L \in \mathbb{R}^{n\times n}$. *)
(* $\Phi \in \mathbb{R}^{n/\sqrt{P}}$ Householder norms; B is the number of columns in a block. *)
(* $Y_\alpha, W_\alpha \in \mathbb{R}^{\frac{n}{\sqrt{P}}\times B}$ reused each outer iteration; $v_\alpha, z_\alpha \in \mathbb{R}^{n/\sqrt{P}}$. *)
(* $Z_\beta \in \mathbb{R}^{B\times\frac{n}{\sqrt{P}}}$. *)
$N = \lceil (n-2)/B \rceil$
(X) $\alpha = \emptyset$
(X) **If** $(n \in \alpha_0)$ $\alpha = \alpha \cup n$
For $j = N : 1 : -1$
$W = [\emptyset]$; $Y = [\emptyset]$
$s = (j-1)B + 1$
$t = \min(n-2, s+B-1)$
For $i = t : s : -1$ (* Form the block of updates. *)
(X) **If** $(i+1 \in \alpha_0)$ $\alpha = \alpha \cup i+1$ (* Add $i+1$ to active rows. *)
(* Retrieve and broadcast Householder vector & norm. *)
(1&2) **If** $(i \in \beta)$ **Then**
(1) $v_\alpha := L(\alpha, i)$
(2) $\phi = \Phi(\lceil i/\sqrt{P} \rceil)$ (* Retrieve Householder norm.*)
(1&2) Broadcast v_α and ϕ to processors sharing rows α
(1&2) **Else** Receive v_α and ϕ
If $(i = s)$ **Then** (* First pass through inner loop. *)
(3) $z_\alpha = \frac{-2}{\phi} v_\alpha$
Else (* Compute $z_\alpha = \frac{-2}{\phi}(I + YW^T)v$. *)
(4) r^q = independent contribution to $W_\alpha^T v_\alpha$ (* $r^q \in \mathbb{R}^{t-i}$. *)
(4) Sum r^q among all processors yielding r
(4) z_α^q = independent set of rows of $\frac{-2}{\phi}(v_\alpha + Y_\alpha r)$
(4) Recursive double z_α^q within rows yielding z_α
(5) $W_\alpha = [z_\alpha \;\; W_\alpha]$; $Y_\alpha = [v_\alpha \;\; Y_\alpha]$
(* Update eigenvectors. *)
(6) $Z_\beta^q = W_\alpha^T X(\alpha, \beta)$
(6) Sum Z_β^q within columns yielding Z_β
(6) $X(\alpha,\beta) = X(\alpha,\beta) + Y_\alpha Z_\beta$

FIG. 6.2. *A parallel block backtransformation algorithm for processor q.*

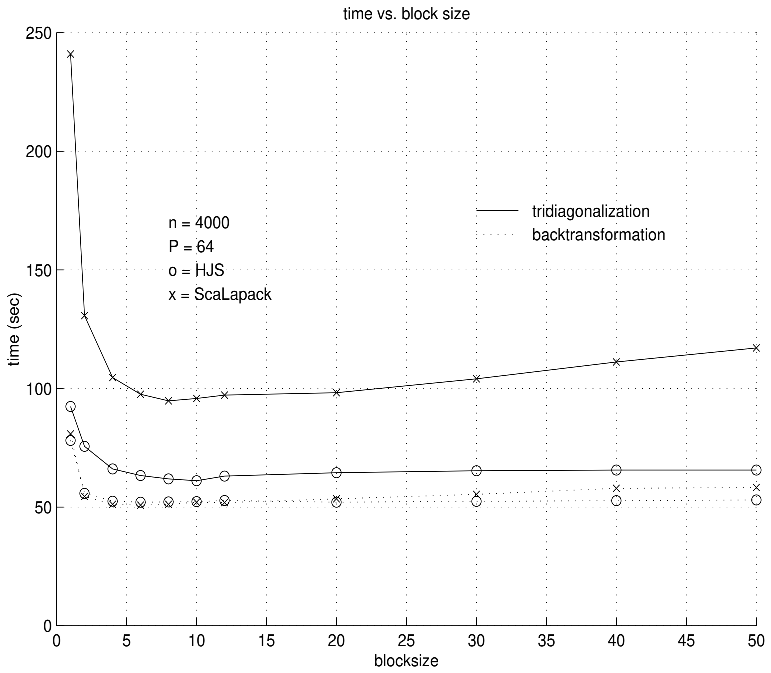

FIG. 7.1. *Time for the HJS and ScaLapack tridiagonalization and backtransformation routines versus block size.*

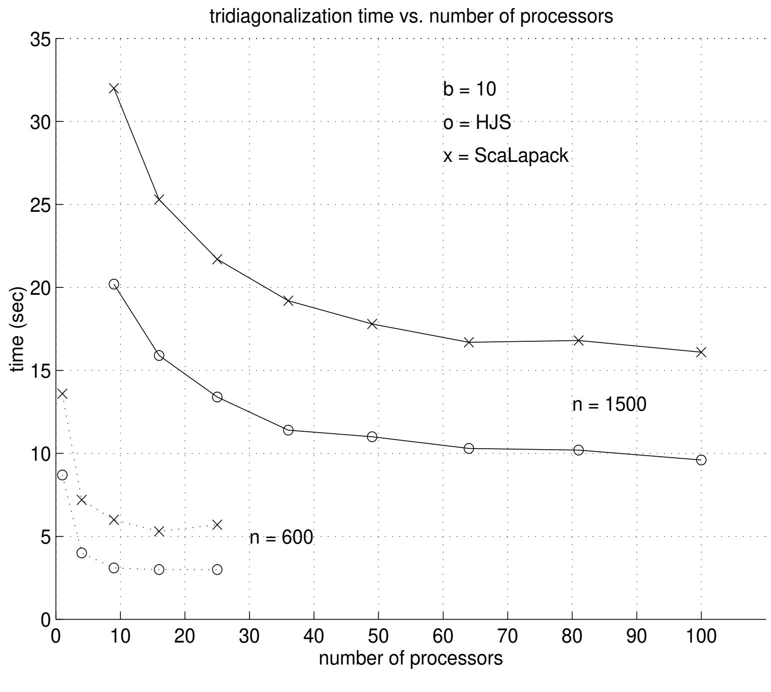

FIG. 7.2. *Time for the HJS and ScaLapack tridiagonalization routines versus number of processors for matrix orders $n = 600$ and $n = 1500$.*

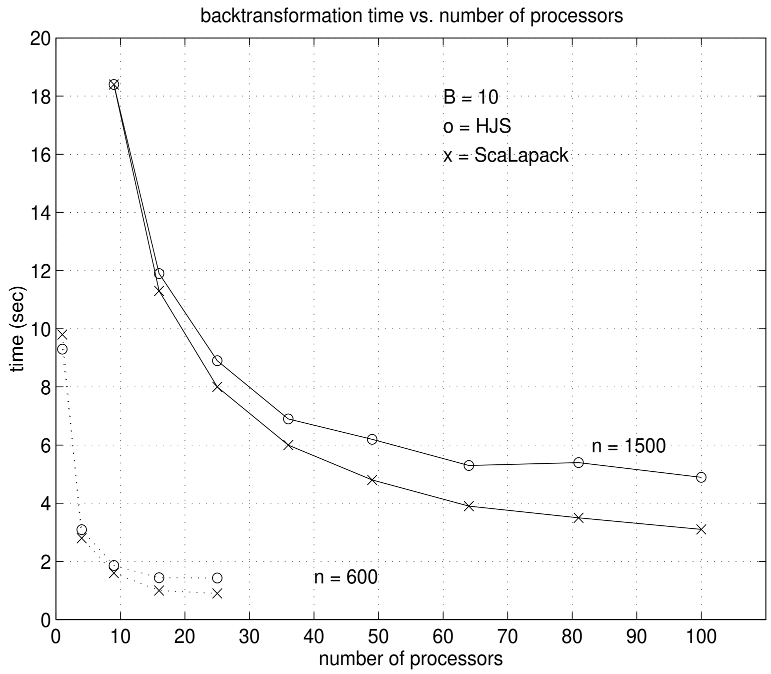

FIG. 7.3. *Time for the HJS and ScaLapack backtransformation routines versus number of processors for matrix orders $n = 600$ and $n = 1500$.*

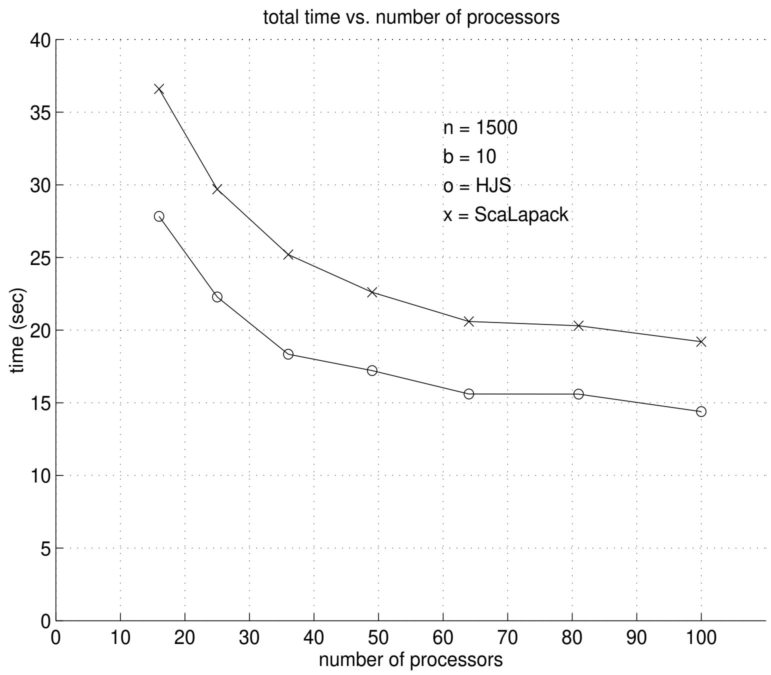

FIG. 7.4. *Time for the HJS and ScaLapack tridiagonalization and backtransformation routines versus number of processors for matrix order $n = 1500$.*

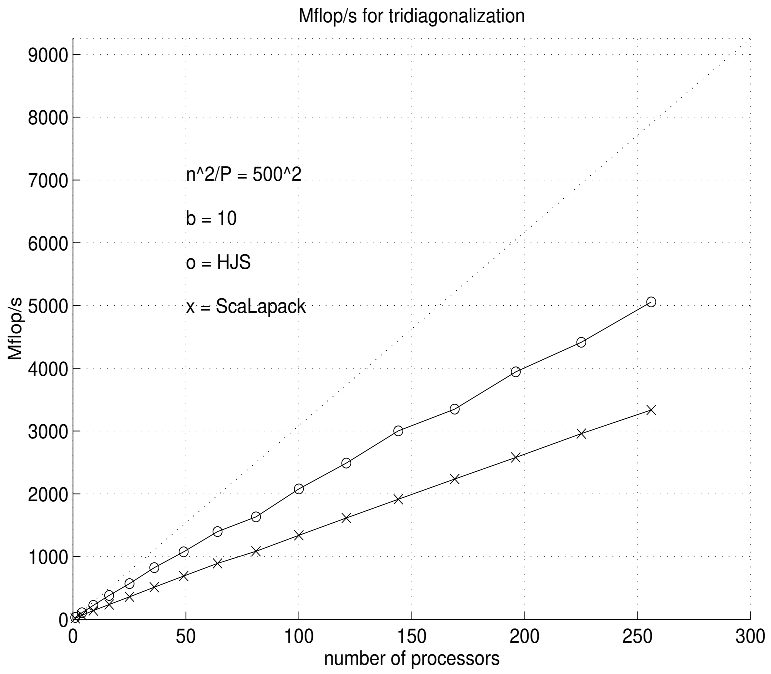

FIG. 7.5. *Mflop/s for HJS and ScaLapack tridiagonalization versus number of processors when the matrix order is varied so that there are* 500×500 *matrix elements per processor.*

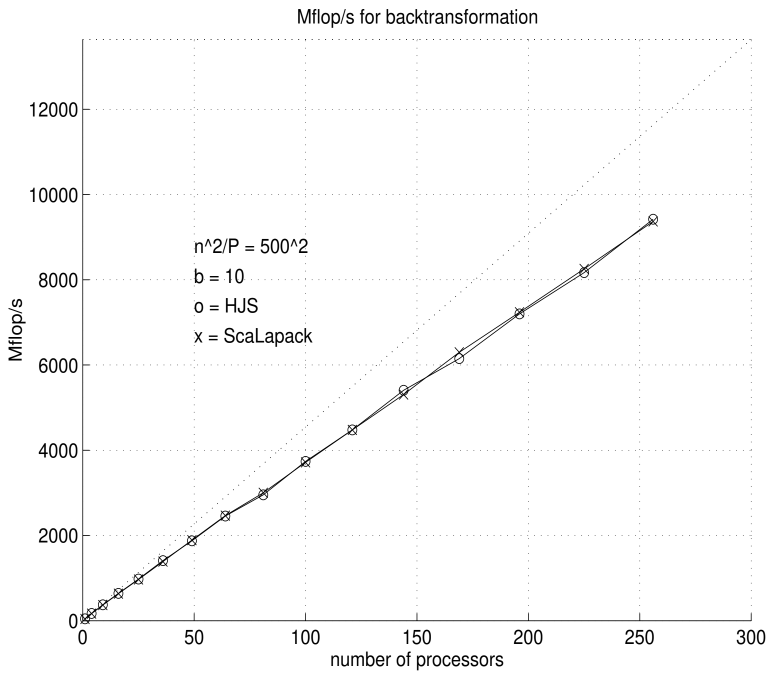

FIG. 7.6. *Mflop/s for HJS and ScaLapack backtransformation versus number of processors when the matrix order is varied so that there are* 500×500 *matrix elements per processor.*

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK User's Guide*, SIAM, 2nd ed., 1995.

[2] R. H. Barlow and D. Evans, *A parallel organization of the bisection algorithm*, The Computer Journal, 22 (1977), pp. 267–69.

[3] R. H. Barlow, D. J. Evans, and J. Shanehchi, *Parallel multisection applied to the eigenvalue problem*, The Computer Journal, 26 (1983), pp. 6–9.

[4] H. J. Bernstein and M. Goldstein, *Parallel implementation of bisection for the calculation of eigenvalues of tridiagonal symmetric matrices*, Computing, 37 (1986), pp. 85–91.

[5] ———, *Optimizing Givens' algorithm for multiprocessors*, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 601–602.

[6] C. Bischof, B. Lang, and X. Sun, *A framework for symmetric band reduction*, Tech. Rep. Preprint ANL/MCS–P586–0496, Math & Comp. Sci. Div. Argonne Natl. Lab., Argonne, IL 60439, 1996.

[7] ———, *The sbr toolbox–software for successive band reduction*, Tech. Rep. Preprint ANL/MCS–P587–0496, Math & Comp. Sci. Div. Argonne Natl. Lab., Argonne, IL 60439, 1996.

[8] R. Bisseling, *Personal communication*, 1992.

[9] H. Y. Chang, S. Utku, M. Salama, and D. Rapp, *A parallel Householder tridiagonalization stratagem using scattered square decomposition*, Parallel Comput., 6 (1988), pp. 297–311.

[10] ———, *A parallel Householder tridiagonalization strategem using scattered row decomposition*, Intl. J. Num. Meth. Engin., 26 (1988), pp. 857–873.

[11] J. Choi, J. Dongarra, and D. Walker, *The design of a parallel, dense linear algebra software library: Reduction to Hessenberg, tridiagonal, and bidiagonal form*, Numerical Algorithms, 10 (1995), pp. 379–400.

[12] J. Cullum and R. A. Willoughby, *Lanczos and the computation of eigenvalues in specified intervals of the spectrum of large, sparse, real symmetric matrices*, in Sparse Matrix Proceedings 1978, SIAM, 1978.

[13] I. Dhillon, *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*, PhD thesis, University of California, Berkeley, 1997.

[14] J. Dongarra, , J. DuCroz, S. Hammarling, and R. Hanson, *An extended set of fortran basic linear algebra subprograms: Model implementation and test programs*, ACM Trans. Math. Software, 14 (1988), pp. 18–32.

[15] J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling, *A set of level 3 basic linear algebra subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 1–17.

[16] J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, *An extended set of Fortran Basic Linear Algebra Subprograms*, ACM Trans. Math. Software, 14 (1988), pp. 1–17.

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

[18] J. Dongarra and R. van de Geijn, *LAPACK working note #30: Reduction to condensed form for the eigenvalue problem on distributed memory architectures*, Dept. of Computer Science, University of Tennessee, 1991.

[19] J. J. Dongarra and D. C. Sorensen, *A fully parallel algorithm for the symmetric eigenvalue problem*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. s139–s154.

[20] G. I. Fann and R. J. Littlefield, *Parallel inverse iteration with reorthogonalization*, in Parallel Processing for Scientific Computing, SIAM, 1993, pp. 409–413.

[21] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving problems on concurrent processors: Volume 1*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[22] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins Press, Baltimore, MD, 2nd ed., 1989.

[23] M. Hegland, *A distribution independent algorithm for the reduction to tridiagonal form using one-sided rotations*, in Proceedings of the First IEEE International Conference on Algorithms and Architectures for Parallel Processing, IEEE Computer Society Press, 1995, pp. 286–289.

[24] B. Hendrickson, E. Jessup, and C. Smith, *A parallel eigensolver for dense symmetric matrices*, Tech. Rep. SAND96-0822, Sandia National Laboratories, 1996.

[25] B. Hendrickson, R. Leland, and S. Plimpton, *An efficient parallel algorithm for matrix–vector multiplication*, Int. J. High Speed Comput., 7 (1995), pp. 73–88.

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

[27] H. Huang, *A parallel algorithm for symmetric tridiagonal eigenvalue problems*, CAC Document No. 109, Center for Advanced Computation, University of Illinois, 1974.

[28] S. Huss-Lederman, A. Tsao, and G. Zhang, *A parallel implementation of the invariant subspace decomposition algorithm for dense symmetric matrices*, in Proc. 6th SIAM Conf. Parallel Proc. Sci. Comput., SIAM, 1993, pp. 367–374.

[29] I. C. F. Ipsen and E. R. Jessup, *Solving the symmetric tridiagonal eigenvalue problem on the hypercube*, SIAM J. Sci. Stat. Comput., Vol. 11, No. 2, (1990), pp. 203–229.

[30] E. R. Jessup, *Parallel Solution of the Symmetric Tridiagonal Eigenproblem*, PhD thesis, Dept of Computer Science, Yale University, 1989.

[31] E. R. Jessup and I. C. F. Ipsen, *Improving the accuracy of inverse iteration*, SIAM J. Sci. Stat. Comput., 13 (1992), pp. 550–571.

[32] L. Kaufman, *A parallel QR algorithm for the symmetric tridiagonal eigenvalue problem*, J. Parallel and Distributed Computing, 23 (1994), pp. 429–434.

[33] ———, *Computing the MDM^T decomposition*, ACM Trans. Math. Software, 21 (1995), pp. 476–489.

[34] J. R. Kuttler and V. G. Sigillito, *Eigenvalues of the Laplacian in two dimensions*, SIAM Review, 26 (1984), pp. 163–193.

[35] B. Lang, *A parallel algorithm for reducing symmetric banded matrices to tridiagonal form*, SIAM J. Sci. Comput., 14 (1993), pp. 1320–1338.

[36] T. Li, H. Zhang, and X. Sun, *Parallel homotopy algorithm for symmetric tridiagonal eigenvalue probl em*, SIAM J. Sci. Stat. Comput., 12 (1991), pp. 464–485.

[37] R. J. Littlefield and K. J. Maschhoff, *Investigating the performance of parallel eigensolvers for large processor counts*, Theoretica Chimica Acta, 84 (1993), pp. 457–473.

[38] S. Lo, B. Phillipe, and A. Sameh, *A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem*, SIAM J. Sci. Stat. Comput., 8 (1987), pp. s155–s165.

[39] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines–EISPACK Guide, Lecture Notes in Computer Science, Vol. 6, 2nd edition*, Springer-Verlag, 1976.

[40] J. Speiser and H. Whitehouse, *Parallel processing algorithms and architectures for real-time signal processing*, in Proc. SPIE Real Time Signal Processing IV, SPIE, 1981, pp. 2–9.

[41] K. Stanley, *Execution time of Symmetric Eigensolvers*, PhD thesis, Dept. Computer Science, University of California at Berkeley, 1997.

[42] P. E. Strazdins, *Matrix factorization using distributed panels on the Fujitsu AP1000*, in Proc. IEEE First Intl. Conf. Algorithms & Architectures for Parallel Processing, IEEE, April 1995.

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

[44] R. A. van de Geijn, *Efficient global combine operations*, in Proc. 6th Distributed Memory Computing Conf., IEEE Computer Society Press, 1991, pp. 291–294.

[45] ———, *Personal communication*, 1992.