AIM

Association for Information and Image Management

1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910
301/587-8202

Centimeter

Inches

MANUFACTURED TO AIM STANDARDS
BY APPLIED IMAGE, INC.

1 of 1

CONTRIBUTED PAPER
Workshop on Environments and Tools
For Parallel Scientific Computing II
May 26-27, 1994, Walland, TN

**The Design of a
Parallel, Dense Linear Algebra Software Library:
Reduction to Hessenberg, Tridiagonal,
and Bidiagonal Form¹**

Jaeyoung Choi [‡]
Jack J. Dongarra ^{‡§}
David W. Walker [§]

[‡] Department of Computer Science
University of Tennessee
107 Ayres Hall
Knoxville, TN 37996-1301

[§] Mathematical Sciences Section
Oak Ridge National Laboratory
P. O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

Corresponding author:
David W. Walker
Oak Ridge National Laboratory
P. O. Box 2008
Oak Ridge, TN 37831-6367
(615) 574-7401 (office)
(615) 574-0680 (fax)
walker@msr.epm.ornl.gov

"The submitted manuscript has been authored by a contractor of the U.S. Government under contract No. DE-AC05-84OR21400. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes."

MASTER

¹This work was supported in part by ARPA under contract number DAAL03-91-C-0047 administered by ARO, and in part by DOE under contract number DE-AC05-84OR21400 and by National Science Foundation Grant number ASC-ASC-9005933.

The Design of a Parallel, Dense Linear Algebra Software Library: Reduction to Hessenberg, Tridiagonal, and Bidiagonal Form[†]

J. Choi[‡] J. J. Dongarra[‡] D. W. Walker[§]

Abstract

This paper discusses issues in the design of ScaLAPACK, a software library for performing dense linear algebra computations on distributed memory concurrent computers. These issues are illustrated using the ScaLAPACK routines for reducing matrices to Hessenberg, tridiagonal, and bidiagonal forms. These routines are important in the solution of eigenproblems. The paper focuses on how building blocks are used to create higher-level library routines. Results are presented that demonstrate the scalability of the reduction routines. The most commonly-used building blocks used in ScaLAPACK are the sequential BLAS, the Parallel Block BLAS (PB-BLAS) and the Basic Linear Algebra Communication Subprograms (BLACS). Each of the matrix reduction algorithms consists of a series of steps in each of which one block column (or *panel*), and/or block row, of the matrix is reduced, followed by an update of the portion of the matrix that has not been factorized so far. This latter phase is performed using distributed Level 3 BLAS routines, and contains the bulk of the computation. However, the panel reduction phase involves a significant amount of communication, and is important in determining the scalability of the algorithm. The simplest way to parallelize the panel reduction phase is to replace the appropriate Level 2 and Level 3 BLAS routines appearing in the LAPACK routine (mostly matrix-vector and matrix-matrix multiplications) with PB-BLAS routines. However, in some cases it is possible to reduce communication startup costs by performing the communication necessary for consecutive BLAS operations in a single communication using a BLACS call. Thus, there is a tradeoff between efficiency and software engineering considerations, such as ease of programming and simplicity of code.

1 Introduction

This paper addresses issues in the design and implementation of ScaLAPACK, a software library for performing dense linear algebra computations on distributed memory concurrent computers. Upon completion, ScaLAPACK ("Scalable LAPACK") will make available on distributed memory machines the same set of library routines that LAPACK [1,2] provides for vector and shared memory architectures. Of particular interest in this paper is the tradeoff between performance and modular algorithm design. This tradeoff will

[†]This work was supported in part by ARPA under contract number DAAL03-91-C-0047 administered by ARO, and in part by DOE under contract number DE-AC05-84OR21400 and by National Science Foundation Grant number ASC-ASC-9005933.

[‡]Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301

[§]Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831-6367

be illustrated using routines that reduce a real general matrix to Hessenberg or bidiagonal form, and a symmetric matrix to tridiagonal form. The reduction of a matrix to Hessenberg form is an important computational component in the unsymmetric eigenvalue problem. The reduction to tridiagonal form plays a similar role in the symmetric eigenvalue problem. Reduction to bidiagonal form is important in evaluating the singular value decomposition (SVD) of a matrix, which in turn is used in the least-squares solution of overdetermined systems of linear equations.

2 Design Philosophy

2.1 Factors Affecting Performance

Two key factors in ensuring that the ScaLAPACK algorithms have good scalability and performance characteristics are maintaining long vector lengths, and maximizing data reuse in the upper levels of memory. Long vector lengths result in more effective use of the vector processors found in many parallel computers. Thus, in implementing ScaLAPACK we must avoid performing operations on small matrices and vectors. By reusing data in the upper levels of memory (registers and cache) the longer latencies associated with accesses to lower levels of memory (main memory, off-processor memory) are avoided. In ScaLAPACK, high levels of data reuse are ensured by the use of block partitioned algorithms that exploit locality of reference. This reduces the frequency of communication between processors, thereby avoiding message startup latency. The sequential computations performed on each processor are mostly expressed in terms of Level 2 and Level 3 Basic Linear Algebra Subprograms (BLAS) [8,11]. These computations are done using commercially available assembly coded routines that have good data reuse characteristics, and make efficient use of the target chip architecture.

In many of the ScaLAPACK routines, such as the factorization routines discussed in [15] and the reduction routines in this paper, columns and/or rows of the matrix are eliminated as the computation progresses. This leads to a tradeoff between data reuse and load balance. This tradeoff has been discussed in an earlier paper [16], and may be controlled at the user level by varying the parameters of the data distribution, as discussed in the next subsection.

2.2 Data Distribution

In many linear algebra algorithms the distribution of work may become uneven as the algorithm progresses, as in LU factorization in which rows and columns become eliminated from the computation. ScaLAPACK, therefore, makes use of the block cyclic data distribution in which matrix blocks separated by a fixed stride in the row and column directions are assigned to the same processor. A number of researchers have made use of the block cyclic data distribution in parallel dense linear algebra algorithms [3,4,9,12,19]. The block cyclic data distribution is parameterized by the four numbers P , Q , r , and c , where $P \times Q$ is the processor template and $r \times c$ is the block size. All ScaLAPACK routines work for arbitrary values of these parameters, subject to certain "compatibility conditions." Thus, for example, in the LU factorization routine we require that the blocks be square, since nonsquare blocks would lead to additional software complexity and communication overhead. When multiplying two matrices, $C = AB$, we require that all three matrices are distributed over the same $P \times Q$ process template; rectangular blocks are permitted, but we require that if the blocks of matrix A are $r \times t$, then those of B and C must be $t \times c$ and $r \times c$, respectively, so it is possible to multiply the individual blocks of A and B to form blocks of C .

Suppose we have M objects indexed by the integers $0, 1, \dots, M-1$. In the block cyclic data distribution the mapping of the global index, m , can be expressed as $m \mapsto (p, b, i)$, where p is the logical process number, b is the block number in process p , and i is the index within block b to which m is mapped. Thus, if the number of data objects in a block is r , the block cyclic data distribution may be written,

$$(1) \quad m \mapsto \left(\left\lfloor \frac{m \bmod T}{r} \right\rfloor, \left\lfloor \frac{m}{T} \right\rfloor, m \bmod r \right)$$

where $T = rP$, and P is the number of processes. The distribution of a block-partitioned matrix can be regarded as the tensor product of two such mappings, one that distributes the rows of the matrix over P processes, and another that distributes the columns over Q processes. It should be noted that Eq. 1 reverts to the cyclic distribution when $r = 1$, with local index $i = 0$ for all blocks. A block distribution is recovered when $r = \lceil M/P \rceil$, in which case there is a single block in each process with block number $b = 0$. Thus, we have

$$(2) \quad m \mapsto (m \bmod P, \lfloor m/P \rfloor, 0)$$

for a cyclic data distribution, and

$$(3) \quad m \mapsto (\lfloor m/L \rfloor, 0, m \bmod L),$$

for a block distribution, where $L = \lceil M/P \rceil$. A subtle distinction between the block distribution given by Eq. 3 and that often used elsewhere (see for example [17,20]) should be noted. Consider the block distribution of 6 items over 4 processes. This is commonly distributed as (2,2,1,1), i.e., 2 items in two of the processes and 1 item in the other two processes. The block distribution given by Eq. 3 results in the distribution (2,2,2,0), so that one of the processes contains no data items. Clearly, since the load imbalance is measured by the difference between the maximum and the average loads, both distribution schemes have the same degree of load imbalance. We prefer the block distribution given by Eq. 3 because the arithmetic needed to convert between global and local indices is simpler, and because of the symmetry between the equations for the block and cyclic distributions (compare Eqs. 2 and 3). There appear to be no other compelling reasons why one of the above forms of block distribution should be preferred to the other in all cases.

The form of the block cyclic data distribution given by Eq. 1 ensures that the block with global index 0 is placed in process 0, the next block is placed in process 1, and so on. However, it is sometimes necessary to offset the processes relative to the global block index so that, in general, the first block is placed in process p_0 , the next in process $p_0 + 1$, and so on. For example, in the parallel, block LU factorization algorithm described in [15] a rank- r update is applied to the unfactored portion of the matrix, E , in each step by multiplying a column of blocks, L_1 , by a row of blocks, U_1 , i.e., $E = E - L_1 U_1$. Here r is the block size. The three matrices involved in this update each have their (0,0) block in a different location of the process template. We, therefore, generalize the block cyclic data distribution by replacing m on the righthand side of Eq. 1 by $m' = m + rp_0$ to give,

$$(4) \quad \begin{aligned} m &\mapsto \left(\left\lfloor \frac{m' \bmod T}{r} \right\rfloor, \left\lfloor \frac{m'}{T} \right\rfloor, m' \bmod r \right) \\ &= \left(\left(\left\lfloor \frac{m \bmod T}{r} \right\rfloor + p_0 \right) \bmod P, \left\lfloor \frac{m + rp_0}{T} \right\rfloor, m \bmod r \right). \end{aligned}$$

The inverse mapping is given by

$$(5) \quad (p, b, i) \mapsto Br + i = (p - p_0)r + bT + i$$

where the global block number is given by $B = (p - p_0) + bP$.

The block cyclic data distribution is the only data distribution supported by the ScaLAPACK routines, and in its most general form is parameterized by P , Q , r , c , p_0 , and q_0 , where $P \times Q$ is the size of the process template, $r \times c$ is the block size, and (p_0, q_0) is the location in the template of the $(0, 0)$ block of the matrix. The block cyclic data distribution can reproduce most of the data distributions used in linear algebra computations. For example, one-dimensional distributions over rows or columns are obtained by choosing Q or P to be 1. Similarly, an $M \times N$ matrix can be decomposed into (nonscattered) blocks by choosing $r = \lceil M/P \rceil$ and $c = \lceil N/Q \rceil$. In algorithms, such as LU factorization, in which the distribution of work becomes uneven, a larger block size results in greater load imbalance, but reduces the frequency of communication between processors. There is, therefore, a tradeoff between load imbalance and communication startup cost which can be controlled by varying the block size.

In addition to the load imbalance that arises as distributed data are eliminated from a computation, load imbalance may also arise due to computational "hot spots" where certain processes have more work to do between synchronization points than others. This is the case in the LU factorization algorithm in which partial pivoting is performed over rows, and only a single column of the process template is involved in the pivot search while the other processes are idle [15]. Similarly, the evaluation of each block row of the U matrix requires the solution of a lower triangular system that involves only processes in a single row of the process template. The effect of this type of load imbalance can be minimized through the choice of P and Q . Another factor to be considered in choosing P and Q is the performance of collective communication routines, such as reduction and broadcast operations, that may be performed over the rows and columns of the process template. Load imbalance of this type also arises in the reduction algorithms presented in this paper — processes in a single column or row of the template are active in processing a column or row of blocks while the other processes are idle.

2.3 Building Blocks

The ScaLAPACK routines are built out of a small number of modules. The most fundamental of these are the Basic Linear Algebra Communication Subprograms (BLACS) [10,13], that perform common matrix-oriented communication tasks, and the sequential Basic Linear Algebra Subprograms (BLAS) [8,11,18], in particular the Level 2 and 3 BLAS. ScaLAPACK can be ported with minimal code modification to any machine on which the BLACS and the BLAS are available. The Parallel Block BLAS (PB-BLAS) [6] are intermediate-level routines based on the BLACS and sequential BLAS. The BLACS, the sequential BLAS, and the PB-BLAS are the modules from which the higher level ScaLAPACK routines are built. Thus, the entire ScaLAPACK package contains modules at a number of different levels. For many users the top level ScaLAPACK routines will be sufficient to build applications. However, more expert users may make use of the lower level routines to build customized routines not provided in ScaLAPACK.

The BLACS package attempts to provide the same ease of use and portability for MIMD message-passing linear algebra communication that the BLAS provide for linear algebra computation. Therefore, future software for dense linear algebra on MIMD platforms could consist of calls to the BLAS for computation and calls to the BLACS for communication. Since both packages will have been optimized for each particular platform, good performance should be achieved with relatively little effort.

Notation	Meaning	Type	Size
B_i	matrix whose first i columns are same as for B , and the rest zero	matrix	$M \times N$
b_i	column i of B	column vector	M
b'_i	column i of B^T	column vector	N
$b_{i,j}$	element (i, j) of B	scalar	1

TABLE 1
Notation relating to an $M \times N$ matrix B .

In the ScaLAPACK routines all interprocessor communication takes place within the PB-BLAS and the BLACS, so the source code of the top software layer of ScaLAPACK looks very similar to that of LAPACK. The BLACS have been implemented for the Intel family of computers, the TMC CM-5, the IBM SP-1, and for PVM. The PVM version of the BLACS has been used to implement ScaLAPACK on the CRAY T3D.

The PB-BLAS are distributed Level 2 and 3 BLAS routines in which at least one of the matrix sizes is limited to the block size. That is, at least one of the matrices consists of a single row or column of blocks, and is located in a single row or column of the process template. An example of a PB-BLAS operation would be the multiplication of a matrix of $M \times N$ blocks by a "vector" of N blocks. The PB-BLAS make use of calls to the sequential BLAS for local computations, and calls to the BLACS for communication. The PB-BLAS are used, for example, to perform block-oriented matrix/vector multiplications when reducing a column of blocks in the parallel reduction algorithms described in Section 3.

3 Dense Reduction Routines

In this section, algorithms for reducing matrices to Hessenberg, tridiagonal, and bidiagonal form by applying a sequence of orthogonal similarity transforms are discussed. The sequential and parallel versions of the block-partitioned algorithms are presented. The block reduction to Hessenberg form algorithms are examined in detail to show how the ScaLAPACK building blocks are used to parallelize the algorithm. We do not go into such detail for the reduction to tridiagonal and bidiagonal forms since the same approach and remarks apply as in the case of Hessenberg reduction.

Before describing the routines we shall first introduce some notation. Given an $M \times N$ matrix, B , the meaning of $b_{i,j}$, b_i , b'_i , and B_i is as given in Table 1, except where explicitly stated otherwise. We shall adopt the convention that B_i^T means $(B_i)^T$. It should be noted that $B_i^T \equiv (B_i)^T \neq (B^T)_i$.

3.1 Reduction to Hessenberg Form

A nonsymmetric $M \times M$ matrix A may be reduced to Hessenberg form, H , by an orthogonal similarity transform, $Q^T A Q = H$. The (upper) Hessenberg form has zeros below the first subdiagonal. The transformation matrix, Q , is a product of Householder transformations, $Q = Q^{(1)} Q^{(2)} \dots Q^{(M-2)}$. Each of the matrices $Q^{(i)}$ is symmetric. Thus, we may write,

$$(6) \quad A^{(k+1)} = Q^{(k)} A^{(k)} Q^{(k)} = Q^{(k)} Q^{(k-1)} \dots Q^{(1)} A Q^{(1)} \dots Q^{(k-1)} Q^{(k)}$$

for $k = 1, 2, \dots, M-2$, where $A^{(1)} = A$ and $A^{(M-1)} = H$. The Householder matrices have the form $Q^{(k)} = I - \tau u u^T$, where $\tau = 2 / \|u\|_2^2$. The Householder vector, u , is

$$(7) \quad u = \left(\frac{1}{a_{k+1,k}^{(k)} + \sigma} \right) (x + \sigma z),$$

and $\sigma = \text{sign}(a_{k+1,k}^{(k)}) \|x\|_2$. Here, $a_{i,j}^{(k)}$ denotes the (i, j) th element of $A^{(k)}$. The vector x is the k th column of $A^{(k)}$ with the first k entries set to zero. The vector z is zero, except for the $(k+1)$ th entry which is 1. Thus,

$$(8) \quad u = (0, \dots, 0, 1, u_{k+2}, \dots, u_M)^T, \quad \text{where } u_i = \frac{a_{i,k}^{(k)}}{a_{k+1,k}^{(k)} + \sigma} \text{ is a scalar.}$$

for $i = k+2, \dots, M$.

Applying the matrix $Q^{(k)}$ to $A^{(k)}$ from the right, and then $Q^{(k)}$ from the left, introduces zeros below the first subdiagonal of column k , and updates columns $k+1, \dots, M$ of $A^{(k)}$ to give $A^{(k+1)}$. Usually the algorithm is performed in-place, so $A^{(k+1)}$ overwrites $A^{(k)}$, and after the $M-2$ steps of the algorithm are completed, the original matrix A has been overwritten by the Hessenberg form, H . Furthermore, in step k , the k th column of A below the first subdiagonal is overwritten by the last $M-k-1$ elements of the Householder vector for this step. Since the $(k+1)$ th entry of the Householder vector is unity it does not have to be explicitly stored. The values of τ for each step are stored in a vector, making it possible to reapply the Householder transformations, $Q^{(k)}$.

3.1.1 Sequential Block Hessenberg Reduction Suppose the matrix A is partitioned into panels, with each panel consisting of n_b consecutive columns of A . In step k of the block-partitioned version of the Hessenberg reduction algorithm the k th panel is reduced. The Householder vectors for each column of the panel are found, and are used to update the next column of the panel, but the updating of panels to the right is deferred until the reduction of the current panel is completed. We shall refer to the product of the k Householder transformations generated as the *block reflector*, $B^{(k)}$. After reducing the k th panel, the block reflector $B^{(k)}$ is applied to A from the right, and then the transpose is applied from the left, thereby updating the panels to the right of panel k .

It can be shown that the block reflector may be written as,

$$(9) \quad B^{(k)} = I - V T V^T$$

where for notational clarity we omit the (k) superscripts on V and T . V is an $M \times n_b$ matrix whose columns are the Householder vectors for panel k . T is a $n_b \times n_b$ upper triangular matrix with the τ values for each column of the panel lying on the diagonal. We may write the update to A performed in step k of the block algorithm as

$$(10) \quad A \leftarrow (I - V T V^T)^T A (I - V T V^T) = (I - V T^T V^T)(A - Y V^T)$$

where $Y = A V T$. Step k of the block algorithm proceeds in three main phases.

1. Reduce panel k , overwriting the last $M - n_b(k-1) - i - 1$ elements of the i th column of the panel with the corresponding portion of the Householder vector for column i . In this step, T and Y are also generated and stored in auxiliary matrices. In LAPACK these computation are performed by calling the routine DLAHRD.

2. Evaluate $A - YV^T$. This operation updates only the panels of A lying to the right of panel k , and is performed in LAPACK by the matrix multiplication routine DGEMM.
3. Pre-multiply A by $I - VT^TV^T$. This operation only updates the last $(M - n_b k)$ rows of the panels of A lying to the right of panel k , and is performed in LAPACK by the routine DLAFRB.

To understand better how the parallel version of the algorithm is implemented we shall examine the first of these 3 phases in more detail. The reduction of each $M \times n_b$ panel is similar to the unblocked algorithm described in Section 3.1. The Householder vector for each column in the panel is evaluated in turn, and all such vectors computed so far are used to update the next column in the panel. The main difference from the unblocked algorithm is that in processing column i of panel k , zeros must be introduced below the $(n_b(k-1) + i)$ th subdiagonal, rather than below the first subdiagonal. As each column of a panel is processed a new column of V , T , and Y is constructed. At the start of processing the i th column of some panel the first $i-1$ columns of V , Y , and T are known. The columns of V are simply the Householder vectors. The i th column of T are given by the vector,

$$(11) \quad t_i = \tau(z^{(i)} - T_{i-1}V_{i-1}^T v_i),$$

where $z^{(i)}$ is a column vector of length n_b , of which all the entries are zero, except the i th which is 1. Thus, the i th element of t_i is simply τ . It should be noted here, and in similar computations, that the last $n_b - i + 1$ elements of $V_{i-1}^T v_i$ are identically zero, and so do not need to be explicitly calculated or stored in an implementation.

The i th column of Y is given by

$$(12) \quad y_i = \tau(-Y_{i-1}V_{i-1}^T v_i + Av_i).$$

In the LAPACK Hessenberg reduction algorithm the routine DLARFG is called to evaluate the Householder vector, v_i , for the i th column of a block, and the value of τ . Calls to the Level 2 BLAS routine DGEMV, which multiplies a matrix by a vector, are then used to evaluate Av_i and $V_{i-1}^T v_i$. A third call to DGEMV evaluates $-Y_{i-1}V_{i-1}^T v_i + Av_i$, which is then scaled by τ to give column i of Y according to Eq. 12. The first $(i-1)$ th entries in t_i are then found by first scaling $V_{i-1}^T v_i$ by τ , and then calling the Level 2 BLAS routine DTRMV, which multiplies a triangular matrix by a vector, to give $-\tau T_{i-1}V_{i-1}^T v_i$. This completes the evaluation of the i th columns of V , Y , and T .

The next task is to update the $(i+1)$ th column of the panel of A by applying the effects of the i Householder vectors evaluated so far for this panel. This involves a series of calls to the Level 2 BLAS routines DGEMV and DTRMV, to first compute the $(i+1)$ th column in the panel of $A - Y_i V_i^T$, and then to apply $(I - V_i T_i^T V_i^T)$ to this column.

We have described how each column in a panel is processed and updated by calls to Level 2 BLAS routines. In the next section we shall consider how the same operations are performed using the building blocks of the ScaLAPACK library.

3.1.2 Parallel Hessenberg Reduction The block size, n_b , of the computation, and the number of rows and columns in a block of the data distribution are chosen to be equal, i.e., $r = c = n_b$. An important consequence of this is that each panel lies in a single column of the process template. Moreover, the triangular matrix T lies in just one process.

The general structure of the parallel Hessenberg reduction algorithm is the same as in the sequential case. The routine PDLAHRD is called to reduce each panel. The PB-BLAS routine PBDGEMM is called to apply the block reflector for a panel from the left, and

PDLAFRB applies the block reflector from the right. We shall now examine PDLAHRD in more detail.

The structure of PDLAHRD is also very similar to that of DLAHRD. For each column of a panel, the routine PDLARFG is called to evaluate the Householder vector, v_i , and the τ value. This involves the evaluation of the scale factor $a_{k+1,k}^{(k)} + \sigma$, and the scalar product $\|v_i\|_2^2$, both of which require $\|x\|_2$ to be computed, where x is the k th column of $A^{(k)}$ with the first k elements set to zero. The column of the process template responsible for reducing a panel performs the summation required to find $\|x\|_2$ by calling the BLACS routine DGSUM2D. This routine returns the value of $\|x\|_2$ to the process holding $a_{k+1,k}^{(k)}$, which then evaluates τ and the scale factor $a_{k+1,k}^{(k)} + \sigma$. These two quantities are then broadcast to the other processes in the column by calls to the BLACS routines DGEBS2D and DGEBR2D. The Householder vector, which is distributed over the processes in one column of the process template, can then be found.

The next step in both the parallel and sequential algorithms is the evaluation of Av_i and $V_{i-1}^T v_i$ as preliminary steps in finding the next column of Y and T . In the parallel algorithm the first of these matrix-vector products is performed by the PB-BLAS routine PBDGEMV. However, since V and v_i are distributed over the rows of the process template in the same way, $V_{i-1}^T v_i$ can be formed by a local call to DGEMV followed by a summation over the processes in the column of the template. The summation is done by calling DGSUM2D, and leaves $V_{i-1}^T v_i$ in a single process. Similarly, the evaluation of the next column of Y , as given by Eq. 12, is completed by first broadcasting $V_{i-1}^T v_i$ over the template column (using DGEBS2D and DGEBR2D), and then calling DGEMV, followed by a local scaling operation. These last two operations require no communication. Computing the next column of T is done on one process by calling DTRMV, and also requires no communication.

Evaluation of the $(i+1)$ th column in the panel of $A - Y_i V_i^T$ requires matrix-vector multiplication which is performed by a single PB-BLAS call.

Next, $(I - V_i V_i^T)$ is applied to the column of $(A - Y_i V_i^T)$ just found. This involves a general matrix-vector multiplication, followed by a triangular matrix-vector multiplication, and finally another general matrix-vector multiplication. The general matrix-vector multiplications are performed by calls to PB-BLAS routines. The triangular matrix T lies in just one process, so the triangular matrix-vector multiplication is a local operation on one process, and is performed using the sequential Level 3 BLAS routine DTRMV.

3.2 Reduction to Tridiagonal Form

If A is a symmetric $M \times M$ matrix, then application of the Householder transformations described in Section 3.1 reduces A to tridiagonal form.

3.2.1 Sequential Block Tridiagonal Reduction As before, we assume A is partitioned into panels of width n_b columns, and in the k th step of the algorithm the k th panel is reduced. The block reflector given in Eq. 9 is applied to A , but in this case we make use of the symmetry of A to express the update as a block update of rank 2. Thus,

$$\begin{aligned} (I - VTV^T)^T A (I - VTV^T) &= A - YV^T - VY^T + V(T^T V^T Y)V^T \\ (13) \qquad \qquad \qquad &= A - WW^T - WW^T \end{aligned}$$

where $Y = AVT$ and,

$$(14) \qquad \qquad \qquad W = Y - \frac{1}{2}V(T^T V^T Y).$$

Y , V , and W are $M \times n_b$ matrices, and T is an $n_b \times n_b$ upper triangular matrix.

In LAPACK, a real symmetric matrix is reduced to tridiagonal form by calling the routine DSYTRD. DSYTRD reduces each panel of A in turn by first calling DLATRD to generate V and W , and then calling DSYR2K to apply the block rank 2 update. The routine DLATRD loops over columns of the panel and in the i th pass applies the previous $(i - 1)$ Householder vectors to update column i of the panel, and adds a new column i to the matrices V and W . Denoting column i of the panel by c_i , then by Eq. 13 c_i is updated as follows,

$$(15) \quad c_i \leftarrow c_i - V_{i-1} w'_i - W_{i-1} v'_i$$

This update is performed by two calls to the Level 2 BLAS routine DGEMV, and has no effect on the first pass through the loop ($i = 1$). The routine DLARFG is then called to evaluate the Householder transformation, (τ, v_i) . v_i is the i th column of the matrix V , which overwrites the lower triangular portion of A . The vector w_i is found next, and is given by,

$$(16) \quad w_i = y_i - \frac{\tau}{2} (y_i^T \cdot c_i) v_i$$

where $y_i = \tau(A - V_{i-1}W_{i-1}^T - W_{i-1}V_{i-1}^T)v_i$. The symmetric matrix-vector multiplication, Av_i is performed by a call to DSYMV, and the other matrix-vector multiplications needed to evaluate y_i are performed by four calls to DGEMV. The evaluation of w_i is completed by calls to DSCAL to scale y_i by τ , DDOT to evaluate $y_i^T \cdot v_i$, and DAXPY to subtract the two terms on the righthand side of Eq. 16.

After the routine DLATRD has looped over the n_b columns of the panel, the construction of the $M \times n$ matrices V and W is complete. Upon return from DLATRD, V and W are passed to the routine DSYR2K which applies a block rank 2 update to the unprocessed panels of A . This update is a Level 3 BLAS operation, and is the main computational task in the reduction to tridiagonal form.

3.2.2 Parallel Block Tridiagonal Reduction The conversion of the sequential routine for reduction to tridiagonal form, DSYTRD, to the parallel version, PDSYTRD, is quite straightforward. The parallel routine calls PDLATRD to reduce a panel and to evaluate the corresponding matrices V and W . Then the routine PDSYR2K uses V and W to apply the Householder transformations for the panel to the unprocessed part of the matrix.

The routine DLATRD is parallelized by replacing the calls to the Level 2 BLAS routines DSYMV and DGEMV by calls to the corresponding PB-BLAS routines, PBDSYMV and PBDGEMV. The call to DLARFG to evaluate the Householder transformation is replaced by a call to the equivalent parallel routine, PDLARFG. There is no PB-BLAS routine for performing the scalar product operation $y_i^T \cdot v_i$ in Eq. 16. Instead, this Level 1 BLAS operation is done by evaluating the local scalar product in each process and then calling the BLACS routine DGSUM2D to sum these contributions over the column of the process template. On exit from DLATRD, the diagonal elements of the reduced matrix are returned in the separate vector, d . All the processes in a column of the process template hold the portions of d that they were involved in computing, i.e., d is block cyclically distributed over the columns of the template. This requires the process containing the diagonal block of the matrix A to communicate the n_b values of d evaluated by a call to PDLATRD to the other process in the template column before returning from PDLATRD. This is done by calls to the BLACS routines DGEBS2D and DGEBR2D.

In reducing a panel in PDLATRD, all processes are involved in the call to PBDSYMV to evaluate Au_i . However, all the other computation in reducing a panel involves processes

in a single column of the process template. Thus, the panel reduction phase suffers from load imbalance. In general all processes are involved in updating the unprocessed portion of the matrix in PDSYR2K, and this phase of the computation is well load balanced.

3.3 Reduction to Bidiagonal Form

If A is a $M \times N$ matrix then Householder transformations can be used to reduce it to bidiagonal form, $Q^T A P = B$. If $M \geq N$, the reduced matrix, B , is upper bidiagonal, and otherwise is lower bidiagonal. We describe below just the reduction to upper bidiagonal form; the algorithm for reduction to lower bidiagonal form is very similar.

3.3.1 Sequential Bidiagonal Reduction A is assumed to be partitioned into square blocks of size $n_b \times n_b$, and in step k , the k th column of blocks (column panel) and the k th row of blocks (row panel) of A are reduced, after which the block reflectors are applied to the unprocessed trailing submatrix. Following Eq. 9, we write the block reflectors corresponding to n_b Householder transformations applied from the left and the right as $B_1 = I - VT^{(1)}V^T$ and $B_2 = I - UT^{(2)}U^T$, respectively. Then we have

$$\begin{aligned} (I - VT^{(1)}V^T)^T A (I - UT^{(2)}U^T) &= A - V(T^{(1)})^T V^T A - A U T^{(2)} U^T \\ &\quad + V(T^{(1)})^T V^T A U T^{(2)} U^T \\ (17) \qquad \qquad \qquad &= A - VY^T - XU^T, \end{aligned}$$

where $Y = A^T V T^{(1)}$, $X = Z - V Y^T U T^{(2)}$, and $Z = A U T^{(2)}$. The matrices V , X , and Z are $M \times n_b$ matrices, Y and U are $N \times n_b$ matrices, and $T^{(1)}$ and $T^{(2)}$ are $n_b \times n_b$ upper triangular matrices.

In LAPACK, a real matrix is reduced to bidiagonal form by calling the routine DGEBRD. DGEBRD reduces each column panel and row panel of A in turn, to generate the matrices V , U , Y , and X , and the Householder scale factors that lie along the diagonals of $T^{(1)}$ and $T^{(2)}$. The diagonal and off-diagonal elements of the reduced matrix are returned in two vectors. DGEBRD calls the routine DLABRD to do the column and row panel reductions, and then makes two calls to the general matrix multiplication routine, DGEMM, to apply the updates to the trailing submatrix part of A .

In DLABRD, n_b loops are performed in each of which a new column of V , U , Y , and X is evaluated. V and U^T overwrite the lower and upper triangular portions of A , respectively. X and Y are stored in $M \times n_b$ and $N \times n_b$ work arrays, respectively. In the i th loop, two calls are made to DGEMV to reduce the i th column of the column panel.

$$(18) \qquad \qquad \qquad c_i = c_i - V_{i-1} y'_i - X_{i-1} u'_i$$

Next, the routine DLARFG is called to generate the Householder transformation $(\tau^{(1)}, v_i)$ that introduces zeros below the diagonal in the i th column of the column panel. A sequence of five calls to the matrix-vector multiplication routine, DGEMV, and a call to the scaling routine, DSCAL, evaluates the i th column of Y ,

$$\begin{aligned} y_i &= \tau^{(1)} \left(A - V_{i-1} Y_{i-1}^T - X_{i-1} U_{i-1}^T \right)^T v_i \\ (19) \qquad \qquad \qquad &= \tau^{(1)} \left(A^T - Y_{i-1} V_{i-1}^T - U_{i-1} X_{i-1}^T \right) v_i \end{aligned}$$

If we denote the current row panel by the matrix R , then according to the notation introduced in Table 1, the i th row of R expressed as a row vector is $(r'_i)^T$. This row is next

reduced using two calls to DGEMV.

$$(20) \quad (r'_i)^T = (r'_i)^T - (v'_i)^T Y_i^T - (x'_{i-1})^T U_{i-1}^T$$

The routine DLARFG is then called to generate the Householder transformation $(\tau^{(2)}, u_i)$ that introduces zeros to the right of the superdiagonal in the i th row of the row panel. It should be noted that this reduction is performed after applying the transformations for the previous $i-1$ loops, and the transformation $(\tau^{(1)}, v_i)$ for the current loop. Thus, in this algorithm Householder transformations are applied first on the lefthand side, and then from the righthand side. This is why Y_i , which has i nonzero columns, is used in Eq. 20, rather than Y_{i-1} .

Five calls to DGEMV, and one call to DSCAL, are used to evaluate the i th column of the matrix X .

$$(21) \quad x_i = \tau^{(2)}(A - U_i Y_i^T - X_{i-1} U_{i-1}^T) u_i$$

This completes step k .

3.3.2 Parallel Bidiagonal Reduction The conversion of the sequential routine for reducing a real matrix to bidiagonal form, DGEBCD, to the parallel ScaLAPACK version, PDGEBRD, is straightforward. The ScaLAPACK routine calls PDLABRD to reduce the k th column and row panels. This routine also returns the matrices X and Y needed to update the unprocessed portion of the matrix, and the scalar variables, $\tau^{(1)}$ and $\tau^{(2)}$. The unprocessed portion of the matrix is then updated as in Eq. 17 by two calls to the PB-BLAS matrix multiplication routine, PBDGEMM. There is one complicating factor related to how columns of the matrix Y are computed and stored. The matrix Y is an $N \times n_b$ matrix, and for a particular panel reduction phase, it lies in a single row of the process template. Thus, to conform to the data layout requirements of the PB-BLAS, Y is stored in transposed form as an $n_b \times N$ matrix, in the same way that U is also stored. The i th column of Y evaluated in Eq. 19 is stored as row i of Y^T . In our Fortran code better performance is obtained if this row is evaluated as a temporary column vector of contiguous elements, stored in working space, and then transposed to be stored in Y^T .

4 Results and Discussion

In the ScaLAPACK versions of the three reduction routines the block size of the block cyclic data distribution is taken as $n_b \times n_b$. Thus, each column (row) panel lies in one column (row) of the process template. All $M \times n_b$ matrices lie within one column of the process template, all $n_b \times N$ matrices (i.e., U^T and Y^T in the algorithm for reduction to bidiagonal form) lie within one row of the process template, and all $n_b \times n_b$ matrices lie in just one process.

In the panel reductions most of the Level 2 BLAS operations involve only processes in a single row or column of the process template. Thus, the panel reduction phase suffers from load imbalance. In general all processes are involved in the Level 3 BLAS operations that update the unprocessed portion of the matrix, and this phase of the computation is well load balanced.

The ScaLAPACK reduction routines were produced by parallelizing the corresponding LAPACK routines. This involved 3 basic tasks: (1) writing a parallel version of the routine DLARFG to compute the Householder transformation for a given vector; (2) inserting control statements to control which columns and rows of the process template are involved in different phases of the algorithms; (3) replacing the Level 2 and 3 BLAS calls in the

LAPACK code by corresponding calls to the PB-BLAS. Note that we do not have to replace the calls to DTRMV since these involve a $n_b \times n_b$ matrix on a single process. All three of these tasks are quite straightforward, thus parallelizing the reduction routines was rather easy. The ease with which the reduction algorithms could be parallelized is largely due to the availability of well-designed, lower-level modules from which to construct them, in particular the PB-BLAS.

Although replacing the sequential Level 2 and 3 BLAS routines in LAPACK with the corresponding parallel PB-BLAS routines is a simple strategy for parallelization, in some cases better performance may be obtained by directly using the sequential BLAS and BLACS. The tradeoff between performance and software modularity arises in the restructuring of algorithms to reduce communication startup costs. Consider, for example, two successive independent calls to PB-BLAS routines in which the same pattern of communication is performed in each routine. Rather than sending two messages, it would be more efficient to combine them, and perform the communication with just one message. To “piggyback” messages in this way we would need to replace the PB-BLAS calls with calls to the BLACS and sequential BLAS. This situation arises in the parallel algorithm for reduction to Hessenberg form discussed in Section 3.1.1. In evaluating y_i in step k of the algorithm (see Eq. 12) we must first find $Y_{i-1}V_{i-1}^T v_i$. This requires $V_{i-1}^T v_i$ to be broadcast over a column of the process template. The subsequent evaluation of the $(i+1)$ th column of $A - Y_i V_i^T$ requires row $n_b(k-1) + i$ of V to be broadcast in the same way. Thus, the two broadcasts can be combined. In this instance, however, we have found the performance gain to be small, and so have chosen to use calls to the PB-BLAS for these operations, rather than piggybacking messages and using lower level calls to the BLACS and the sequential BLAS.

The three ScaLAPACK reduction routines were developed on a 128-node Intel iPSC/860 hypercube. In Figure 1 we plot performance in Gflop/s against number of processors while keeping the size of the matrix per processor fixed. The floating point operation count was assumed to be $\frac{10}{3}N^3$ for reduction to Hessenberg form, $\frac{8}{3}N^3$ for reduction to bidiagonal form, and $\frac{4}{3}N^3$ for reduction to tridiagonal form. The algorithms for reduction to Hessenberg and bidiagonal form run at just over 2 Gflop/s on 128 processors, while that for reduction to tridiagonal form runs at just over 1 Gflop/s. This difference is attributable to the fact that the tridiagonal reduction routine involves operations on a symmetric matrix. Thus, the total number of floating point operations is less than in the Hessenberg and bidiagonal reduction algorithms. The communication overhead, however, is similar in all cases, and so the ratio of computation to communication is lower for the tridiagonal reduction algorithm, and its performance is consequently poorer [14]. The fact that the plots in Figure 1 are almost linear shows that the algorithms scale well on the Intel iPSC/860.

5 Conclusions

We have shown how dense matrix reduction algorithms can be parallelized fairly easily using a small set of low-level modules, namely the sequential BLAS, the BLACS, and the PB-BLAS. The PB-BLAS, which themselves are built using the sequential BLAS and BLACS, are particularly useful in simplifying the task of parallelizing dense linear algebra algorithms. In general, calls to the Level 2 and Level 3 BLAS in the LAPACK code can be replaced on a one-for-one basis by the corresponding PB-BLAS routine.

The tradeoff between performance and software design considerations, such as modularity and clarity, is particularly important in the design software libraries. In Section 3.1.2,

FIG. 1. Isogranularity plots for the Hessenberg (HRD), tridiagonal (TRD), and bidiagonal reduction (BRD) routines on the Intel iPSC/860 hypercube. The matrix size per processor is fixed at 5 Mbytes.

we have discussed how nonstandard storage schemes for the matrix Y can result in better performance. We have also discussed, in Section 4, how the piggybacking of messages can reduce communication costs, again at the cost of replacing calls to the PB-BLAS by calls to the lower level BLACS and sequential BLAS. Here we have found the gain in performance too small to justify the loss in software modularity, and so do not piggyback messages.

Our results on the Intel family of parallel computers show that the ScaLAPACK reduction routines have good performance and scalability characteristics on these machines. Future work will involve similar performance studies on more recent machines, such as the CRAY T3D and the IBM SP1 and SP2.

References

- [1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK User's Guide*, SIAM, Philadelphia, PA, 1992.
- [2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *Lapack: A portable linear algebra library for high-performance computers*, in Proceedings of Supercomputing '90, IEEE Press, 1990, pp. 1-10.
- [3] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, *ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers*, in Proceedings of the Fourth Symposium on Massively Parallel Computing, H. J. Siegel, ed., 1992, pp. 120-127.
- [4] J. Choi, J. J. Dongarra, and D. W. Walker, *The design of scalable software libraries for distributed memory concurrent computers*, in Environments and Tools for Parallel Scientific Computing, J. J. Dongarra and B. Tourancheau, eds., 1993, pp. 3-15. Proceedings of workshop held September 7-8, 1992, in Saint Hilaire du Touvet, France.
- [5] —, *Level 3 BLAS for distributed memory concurrent computers*, in Environments and Tools for Parallel Scientific Computing, J. J. Dongarra and B. Tourancheau, eds., 1993, pp. 17-29. Proceedings of workshop held September 7-8, 1992, in Saint Hilaire du Touvet, France.
- [6] —, *PB-BLAS: A set of parallel block basic linear algebra subprograms*, in Proceedings of the 1994 Scalable High Performance Computing Conference, IEEE Computer Society, 1994.
- [7] —, *PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers*, Concurrency: Practice and Experience, (1994). To appear. Also available as Oak Ridge National Laboratory report TM-12252, November, 1992.

- [8] J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling, *A set of level 3 basic linear algebra subprograms*, ACM Transactions on Mathematical Software, 16 (1990), pp. 1-17.
- [9] J. Dongarra and S. Ostrouchov, *LAPACK block factorization algorithms on the Intel iPSC/860*, Tech. Rep. CS-90-115, University of Tennessee at Knoxville, Computer Science Department, October 1990.
- [10] J. J. Dongarra, *LAPACK Working Note 34: Workshop on the BLACS*, computer Science Dept. Technical Report CS-91-134, University of Tennessee, Knoxville, TN, May 1991. (LAPACK Working Note #34).
- [11] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, *An extended set of Fortran basic linear algebra subroutines*, ACM Transactions on Mathematical Software, 14 (1988), pp. 1-17.
- [12] J. J. Dongarra, R. van de Geijn, and D. W. Walker, *A look at scalable dense linear algebra libraries*, in Proceedings of the Scalable High-Performance Computing Conference, IEEE, ed., IEEE Publishers, 1992, pp. 372-379.
- [13] J. J. Dongarra and R. A. van de Geijn, *Two-dimensional basic linear algebra communication subprograms*, Tech. Rep. LAPACK working note 37, Computer Science Department, University of Tennessee, Knoxville, TN, October 1991.
- [14] ———, *Reduction to condensed form for the eigenvalue problem on distributed memory architectures*, Parallel Computing, 18 (1992), pp. 973-982.
- [15] J. J. Dongarra, R. A. van de Geijn, and D. W. Walker, *Scalability issues affecting the design of dense linear algebra library*, Journal of Parallel and Distributed Computing, (1994). Accepted for publication.
- [16] J. J. Dongarra and D. W. Walker, *Software libraries for linear algebra computations on high performance computers*, SIAM Review, (1994). Accepted for publication.
- [17] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors*, vol. 1, Prentice Hall, Englewood Cliffs, N.J., 1988.
- [18] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, *Basic linear algebra subprograms for Fortran usage*, ACM Trans. Math. Softw., 5 (1979), pp. 308-323.
- [19] W. Lichtenstein and S. L. Johnsson, *Block cyclic dense linear algebra*, SIAM Journal on Scientific Computing, 14 (1993), pp. 1259-1288.
- [20] E. F. Van de Velde, *Data redistribution and concurrency*, Parallel Computing, 16 (1990).

**DATE
FILMED**

10/17/94

END