ORNL/TM-12472

Engineering Physics and Mathematics Division

Mathematical Sciences Section

THE DESIGN OF A PARALLEL DENSE LINEAR ALGEBRA SOFTWARE LIBRARY: REDUCTION TO HESSENBERG, TRIDIAGONAL, AND BIDIAGONAL FORM

Jaeyoung Choi [§]
Jack J. Dongarra [§†]
David W. Walker [†]

[§] Department of Computer Science
University of Tennessee at Knoxville
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

Date Published: January 1995

Research was supported in part by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy, by the Defense Advanced Research Projects Agency under contract DAAL03-91-C-0047, administered by the Army Research Office, and in part by the Center for Research on Parallel Computing

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400

Contents

THE DESIGN OF A PARALLEL DENSE LINEAR ALGEBRA SOFTWARE LIBRARY: REDUCTION TO HESSENBERG, TRIDIAGONAL, AND BIDIAGONAL FORM

Jaeyoung Choi
Jack J. Dongarra
David 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 BLAS (PBLAS) 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 Level 3 PBLAS operations, 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 BLAS routines appearing in the LAPACK routine (mostly matrix-vector and matrix-matrix multiplications) with the corresponding PBLAS 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.

A set of Basic Linear Algebra Subprograms (Level 1, 2, and 3 BLAS) [8, 11, 20] is available as a highly efficient machine-specific implementation on many modern high-performance computers. They provides high performance with portability and are used as the building blocks of a number of applications, including LAPACK. The Basic Linear Algebra Communication Subprograms (BLACS) [10] comprise a package that provides ease-of-use and portability for message-passing in parallel linear algebra applications. The Parallel BLAS (PBLAS), which provide a simplified interface around the Parallel Block BLAS (PB-BLAS) [7], are intermediate level routines based on the sequential BLAS and the BLACS. The PBLAS provide all the functionality supported by parallel versions of the Level 1, 2, and 3 BLAS on a restricted class of matrices having a block cyclic data distribution. The ScaLAPACK routines are built using the sequential BLAS, the BLACS, and the PBLAS modules. ScaLAPACK can be ported with minimal code modification to any machine on which the BLAS and the BLACS are available.

Of particular interest in this paper is the tradeoff between performance and modular algorithm design. This tradeoff will be illustrated using routines that use Householder transformations to 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.

Currently ScaLAPACK also includes LU, QR, and Cholesky factorization routines with their solvers. The implementation details, performance, and scalability of the ScaLAPACK factorization routines are presented in a separate paper [4].

The design philosophy of the ScaLAPACK library is addressed in Section 2. In Section 3, we induce the block equations of the reduction routines and describe the ScaLAPACK reduction routines by comparing them with the corresponding LAPACK routines. Section 4 presents performance results and scalability of the algorithms on the Intel family of computers: the iPSC/860, the Touchstone Delta, and the Paragon. In Section 5, conclusions and future work are presented.

2. Design Philosophy

In ScaLAPACK, algorithms are presented in terms of *processes*, rather than the processors of the physical hardware. A process is an independent thread of control with its own nonshared, distinct memory. Processes communicate by pairwise point-to-point communication, or by collective communication, as necessary. In general there may be several processes on a physical processor, in which case it is assumed that the runtime system handles the scheduling of processes. For example, execution of a process waiting to receive a message may be suspended and another process scheduled, thereby overlapping communication and calculation. In the absence of such a sophisticated operating system, ScaLAPACK has been developed and tested for the case of one process per processor.

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 or RISC 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 processes, thereby avoiding message startup latency. The sequential computations performed by each process 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 [16] 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 [17], 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 process. A number of researchers have made use of the block cyclic data distribution in parallel dense linear algebra algorithms [5, 6, 9, 13, 21]. The block cyclic data distribution is parameterized by the four numbers P, Q, r, and c, where $P \times Q$ is the process 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 $0, 1, \ldots, M-1$. 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

$$m \longmapsto \langle s \bmod P,\ \lfloor s/P \rfloor,\ m \bmod r \rangle \tag{1}$$

where $s = \lfloor m/r \rfloor$, 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

$$m \longmapsto \langle\, m \bmod P, \lfloor m/P \rfloor, 0 \,\rangle \tag{2}$$

for a cyclic data distribution, and

$$m \longmapsto \langle\, \lfloor m/L \rfloor, 0, m \bmod L \,\rangle, \tag{3}$$

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 [18, 24]) 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.

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, 14], that perform common matrix-oriented communication tasks, and the sequential Basic Linear Algebra Subprograms (BLAS) [8, 11, 20], 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 BLAS (PBLAS) provide a simplified interface to the Parallel Block BLAS (PB-BLAS) [7] — the PBLAS are essentially C wrappers around the PB-BLAS, which in turn are intermediate-level routines based on the BLACS and sequential BLAS. The BLACS, the sequential BLAS, and the PBLAS 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 PBLAS 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.

In the ScaLAPACK routines all interprocess communication takes place within the PBLAS 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 SP1 and SP2, the Cray T3D, and for PVM.

The PBLAS are distributed 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 PBLAS operation would be the multiplication of a matrix of $M \times N$ blocks by a "vector" of N blocks. The PBLAS make use of calls to the sequential BLAS for local computations, and calls to the BLACS for communication. The PBLAS are used, for example, to perform

Notation	Meaning	Type	Size
A	general matrix	matrix	$m \times n$
$A_j = [A]_{:,1:j}$	matrix which includes first j columns of A	matrix	$m \times j$
$a_j = [A]_{:,j}$	jth column of A	column vector	m
$a_{i,j} = [A]_{i,j}$	element (i,j) of A	scalar	1

Table 1: Notation relating to an $m \times n$ matrix A.

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, block-partitioned algorithms for reducing matrices to Hessenberg, tridiagonal, and bidiagonal form by applying a sequence of orthogonal similarity transforms are discussed. The basic approach for these algorithms is to aggregate Householder transforms [19] and to apply them in a blocked fashion [3, 22], thus achieving algorithms that are rich in matrix-matrix operations [12]. The sequential versions of the algorithms are derived and parallel versions 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.

The parallel algorithms described below extend and generalize previous work. Dongarra and van de Geijn [15] have presented a parallel, block-partitioned algorithm for reduction to Hessenberg form, and assumed a one-dimensional, block column data distribution. Their data distribution corresponds to a $1 \times Q$ process template in the terminology of Section 2.2. Smith, Hendrickson, and Jessup have described and implemented a parallel algorithm for Householder tridiagonalization on a square process template [23]. In the terminology of Section 2.2 this corresponds to the case $n_b = 1$ with a $P \times P$ process template.

Before describing the routines we shall first introduce some notation. For an $m \times n$ matrix A, $[A]_{i:j,k:l}$ denotes the submatrix of A consisting of elements of row $i, \cdots, j$ and columns $k, \cdots, l$. $[A]_{:,k:l}$ and $[A]_{i:j,:}$ will be used if all columns or rows of the matrix are involved, respectively. And the meaning of A_j, a_j, and $a_{i,j}$ is as given in Table 1, except where explicitly stated otherwise.

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, $Q =$

$Q^{(1)}Q^{(2)} \cdots Q^{(M-2)}$. Each of the matrices $Q^{(k)}$ for $k = 1, 2, \ldots, M-2$ is symmetric. Thus, we may write,

$$A^{(k+1)} = Q^{(k)}A^{(k)}Q^{(k)} = Q^{(k)}Q^{(k-1)} \cdots Q^{(1)}AQ^{(1)} \ldots Q^{(k-1)}Q^{(k)} \tag{4}$$

where $A^{(1)} = A$ and $A^{(M-1)} = H$. The Householder matrices have the form, $Q^{(k)} = I - \tau vv^T$, where $v = v_k$ and $\tau = \tau_k = 2/ \parallel v \parallel_2^2$, and we omit the (k) subscripts on v and τ for notational clarity. The Householder vector v is

$$v = \left(\frac{1}{a_{k+1,k}^{(k)} + \sigma} \right) (x + \sigma e_{k+1})$$

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

$$v = (0, \ldots, 0, 1, w_{k+2}, \ldots, w_M)^T, \quad \text{where } w_i = \frac{a_{i,k}^{(k)}}{a_{k+1,k}^{(k)} + \sigma} \text{ is a scalar,}$$

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

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 $k+1, \ldots, M$ 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, the kth column of A below the first subdiagonal is overwritten by the last $M-k-1$ elements of the Householder vector in step k. 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

We can rewrite the Eq. 4 as follows.

$$A^{(k+1)} = Q^{(k)} A^{(k)} Q^{(k)} = (I - \tau v v^T) A^{(k)} (I - \tau v v^T) = (I - \tau v v^T) \cdot (A^{(k)} - y v^T) \tag{5}$$

where $y = \tau A^{(k)} v$.

By mathematical induction, assume that

$$A^{(k)} = (I - V_{k-1} T_{k-1}^T V_{k-1}^T) \cdot (A^{(1)} - Y_{k-1} V_{k-1}^T) \tag{6}$$

where V_{k-1} and Y_{k-1} are $M \times (k-1)$ matrices such that $V_{k-1} = (v_1, v_2, \cdots, v_{k-1})$ and $Y_{k-1} = (y_1, y_2, \cdots, y_{k-1})$. And T_{k-1} is a $(k-1) \times (k-1)$ upper triangular matrix.

Then,

$$\begin{aligned}
A^{(k+1)} &= (I - \tau\, v\, v^T) \cdot A^{(k)} \cdot (I - \tau\, v\, v^T) \\
&= (I - \tau\, v\, v^T)\,(I - V_{k-1}\, T_{k-1}^T\, V_{k-1}^T) \cdot (A^{(1)} - Y_{k-1}\, V_{k-1}^T)\,(I - \tau\, v\, v^T) \\
&= (I - \tau\, v\, v^T - V_{k-1}\, T_{k-1}^T\, V_{k-1}^T + \tau\, v\, v^T\, V_{k-1}\, T_{k-1}^T\, V_{k-1}^T) \\
&\qquad \cdot \left(A^{(1)} - Y_{k-1}\, V_{k-1}^T - \tau\,(A^{(1)}\, v - Y_{k-1}\, V_{k-1}^T\, v) v^T\right) \\
&= \left(I - (V_{k-1},\, v) \begin{pmatrix} T_{k-1} & -\tau\, T_{k-1}\, V_{k-1}^T\, v \\ 0 & \tau \end{pmatrix}^T (V_{k-1},\, v)^T \right) \\
&\qquad \cdot \left(A^{(1)} - (Y_{k-1},\, y)\ (V_{k-1},\, v)^T\right) \\
&= (I - V_k\, T_k^T\, V_k^T) \cdot \left(A^{(1)} - Y_k\, V_k^T\right) \qquad (7)
\end{aligned}$$

where

$$y = \tau\,(A^{(1)}\, v - Y_{k-1}\, V_{k-1}^T\, v), \qquad (8)$$

$$T_k = \begin{pmatrix} T_{k-1} & -\tau\, T_{k-1}\, V_{k-1}^T\, v \\ 0 & \tau \end{pmatrix}. \qquad (9)$$

If we assumed that $T_1 = \tau_1$, Eq. 6 is true for $k = 2$. By Eq. 7, Eq. 6 is true for all k ($k \geq 2$).

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 kth 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.

Step k of the LAPACK routine, **DGEHRD**, proceeds in three main phases.

1. **DLAHRD**: Reduce the kth panel of the matrix and compute V, Y, and T.

   [Repeat n_b times for $i = 1, \cdots, n_b$ (let $k_i = (k-1)n_b + i$)]

   1. Compute the Householder vector v_i.
   2. Compute $y_i = \tau\,(A v_i - Y_{i-1} V_{i-1}^T v_i)$.
   3. Compute $[T_i]_{1:i-1,i} = -\tau\, T_{i-1} V_{i-1}^T v_i$.
   4. Update the k_{i+1}th column of A ($[A]_{:,k_{i+1}}$) if necessary.
      - Apply the block Householder vector from the right: $[A]_{:,k_{i+1}} \Leftarrow [A]_{:,k_{i+1}} - Y_i\, [V_i]_{k_{i+1},:}$.
      - Apply the block Householder vector from the left: $[A]_{:,k_{i+1}} \Leftarrow \left(I - V_i T_i^T V_i^T\right) [A]_{:,k_{i+1}}$.

2. **DGEMM**; Update A with Y and V, $A \Leftarrow A - Y\,V$.

3. **DLARFB**: Apply the block Householder vector from the left, $A \Leftarrow (I - VT^T V^T)A$.

where we omit the (n_b) subscripts on Y, V, and T for notational simplicity. And subroutines called by **DGEHRD** are specified in the front of each procedure.

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. As each column of a panel is processed a new column of V, Y, and T is constructed. At the start of processing the ith 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. In the LAPACK Hessenberg reduction algorithm, the routine **DLARFG** is called to evaluate the Householder vector v_i 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 $Av_i - Y_{i-1}(V_{i-1}^T v_i)$, which is then scaled by τ to give y_i according to Eq. 8. The first $(i-1)$ entries of the ith column of T_i are found by 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 $[T_i]_{1:i-1,i}$ by Eq. 9. This completes the evaluation of the ith 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**. We update the $(i+1)$th column in the panel by computing $[A]_{:,k_i} - Y_i\,[V_i]_{k_i,:}$ and then 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 number of rows and columns in a block of the data distribution are chosen to be equal to the block size of the computation n_b, 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 PBLAS routine **PDGEMM** is called to apply the block reflector for a panel from the left, and **PDLARFB** 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 the ith column of a panel, the routine PDLARFG is called to evaluate the Householder vector v_i and the value of τ. The Householder vector is distributed over the processes in one column of the process template.

The next step 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 these matrix-vector products are performed by the PBLAS routine, PDGEMV. Computing the next column of T is done on one process by calling DTRMV and DSCAL, which require no communication.

Evaluation of the $(i+1)$th column in the panel of $[A]_{:,k_i} - Y_i\,[V_i]_{k_i,:}$ requires matrix-vector multiplication which is performed by a single PDGEMV call. Next, $(I - V_i T_i^T V_i^T)$ is applied to the column. This involves a series of calls to general matrix-vector multiplications, and triangular matrix-vector multiplications.

The general matrix-vector multiplications are performed by calls to PBLAS routine, PDGEMV. The triangular matrices (T is upper triangular and the top part of V is unit lower triangular) lie in just one process, so the triangular matrix-vector multiplications are performed by the sequential 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. In this section, we describe the reduction algorithm for the symmetric lower triangular matrix. The algorithm for symmetric the upper triangular matrix is very similar.

3.2.1. Sequential Block Tridiagonal Reduction

As before, we assume A is partitioned into panels of width n_b columns, and in the kth step of the algorithm the kth panel is reduced. A series of the Householder reflectors 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. We describe first the unblocked version of the algorithm, and then expand the algorithm to the blocked version.

$$\begin{aligned}
A^{(k+1)} &= Q^{(k)}\,A^{(k)}\,Q^{(k)} = (I - \tau\,v\,v^T)\,A^{(k)}\,(I - \tau\,v\,v^T) \\
&= A^{(k)} - \tau v v^T A^{(k)} - \tau A v v^T + \tau^2 v v^T A^{(k)} v v^T \\
&= A^{(k)} - v x^T - x v^T + \tau (v^T x) v v^T
\end{aligned}$$

where $x = \tau A^{(k)} v$. Let $w = x - \tau v (v^T x)/2$, then

$$A^{(k+1)} = A^{(k)} - v w^T - w v^T. \tag{10}$$

By mathematical induction, assume that

$$A^{(k)} = A^{(1)} - V_{k-1}W_{k-1}^T - W_{k-1}V_{k-1}^T \tag{11}$$

where $V_{k-1} = (v_1, v_2, \cdots, v_{k-1})$ and $W_{k-1} = (w_1, w_2, \cdots, w_{k-1})$. Then

$$\begin{aligned} A^{(k+1)} &= A^{(k)} - vw^T - wv^T \\ &= A^{(1)} - V_{k-1}W_{k-1}^T - W_{k-1}V_{k-1}^T - vw^T - wv^T \\ &= A^{(1)} - (V_{k-1},\, v)\ (W_{k-1},\, w)^T - (W_{k-1},\, w)\ (V_{k-1},\, v)^T \\ &= A^{(1)} - V_kW_k^T - W_kV_k^T. \end{aligned} \tag{12}$$

And

$$\begin{aligned} x &= \tau A^{(k)}v = \tau \left(A^{(1)} - V_{k-1}W_{k-1}^T - W_{k-1}V_{k-1}^T \right) v \\ &= \tau \left(A^{(1)}v - V_{k-1}W_{k-1}^T v - W_{k-1}V_{k-1}^T v \right), \end{aligned} \tag{13}$$

$$w = x - \tau v(v^T x)/2. \tag{14}$$

By comparing Eq. 10 with Eq. 11, Eq. 11 is true for $k = 2$. And from Eq. 12, Eq. 11 is true for all $k \geq 2$.

In LAPACK, a real symmetric matrix is reduced to tridiagonal form by calling the routine **DSYTRD**. Step k of the block algorithm proceeds as follows:

1. **DLATRD**: Reduce the kth panel of the matrix and compute V and W.

   [Repeat n_b times for $i = 1, \cdots, n_b$ (let $k_i = (k-1)n_b + i$)]

   1. Compute the Householder vector v_i.
   2. Compute $x_i = \tau \left(A^{(1)}v_i - W_{i-1}(V_{i-1}^T v_i). - V_{i-1}(W_{i-1}^T v_i) \right)$.
   3. Compute $w_i = x_i - \tau v_i(v_i^T x_i)/2$.
   4. Update the k_{i+1}th column of A ($[A]_{:,k_{i+1}}$) if necessary.

2. **DSYR2K**: Apply a block rank-2 update, $A \Leftarrow A - V\,W^T - W\,V^T$.

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 ith 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.

The routine **DLARFG** is then called to evaluate the Householder transformation, (τ, v_i). v_i is the ith column of the matrix V, which overwrites the lower triangular portion of A. The vector x_i is found next by Eq.13. The symmetric matrix-vector multiplication Av_i is performed by a

Level 2 BLAS routine, **DSYMV**, and the other matrix-vector multiplications needed to evaluate x_i are performed by four calls to **DGEMV**. The evaluation of x_i is completed by a **DSCAL** call to scale x_i by τ. Then w_i is computed by calls to **DDOT** to evaluate $v_i^T x_i$, and then **DAXPY** to subtract the two terms on the righthand side of Eq. 14, and it is overwritten to x_i. Denoting column i of the panel by $[A]_{:,k_i}$, $[A]_{:,k_i}$ is updated from Eq. 11 as follows,

$$[A]_{:,k_i} \leftarrow [A]_{:,k_i} - V_{i-1}\, w_i^T - W_{i-1}\, v_i^T \,. \tag{15}$$

This update is performed by two calls to the Level 2 BLAS routine, **DGEMV**.

After the routine **DLATRD** has looped over the n_b columns of the panel, the construction of the $M \times n_b$ 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**, **DGEMV**, **DSCAL**, **DDOT**, and **DAXPY** by calls to the corresponding PBLAS routines, **PDSYMV**, **PDGEMV**, **PDSCAL**, **PDDOT**, and **PDAXPY** The call to **DLARFG** to evaluate the Householder transformation is replaced by a call to the equivalent parallel routine, **PDLARFG**.

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 **PDSYMV** to evaluate Av_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 $Q^T AP = B$. If $M \geq N$, the reduced matrix B is upper bidiagonal, and otherwise is lower bidiagonal. We describe below the reduction to upper bidiagonal form; the algorithm for reduction to lower bidiagonal form is very similar.

3.3.1. Sequential Bidiagonal Reduction

We describe first the unblocked version of the algorithm to reduce an $M \times N$ matrix to the bidiagonal form.

$$\begin{aligned}
A^{(k+1)} &= Q^{(k)} A^{(k)} P^{(k)} = (I - \tau_v vv^T) A^{(k)} (I - \tau_u uu^T) \\
&= A^{(k)} - \tau_v vv^T A^{(k)} - \tau_u A^{(k)} uu^T + \tau_v \tau_u vv^T A^{(k)} uu^T \\
&= A^{(k)} - vy^T - (z - \tau_u vy^T u)u^T
\end{aligned}$$

where $y = \tau_v {A^{(k)}}^T v$ and $z = \tau_u A^{(k)} u$. Let $x = z - \tau_u vy^T u$. Then,

$$A^{(k+1)} = A^{(k)} - vy^T - xu^T. \tag{16}$$

By mathematical induction, assume that

$$A^{(k)} = A^{(1)} - V_{k-1}Y_{k-1}^T - X_{k-1}U_{k-1}^T, \tag{17}$$

where $V_{k-1} = (v_1, \cdots, v_{k-1})$, $U_{k-1} = (u_1, \cdots, u_{k-1})$, $X_{k-1} = (x_1, \cdots, x_{k-1})$, and $Y_{k-1} = (y_1, \cdots, y_{k-1})$. Eq. 17 is true for $k = 2$.

$$\begin{aligned}
A^{(k+1)} &= A^{(k)} - vy^T - xu^T \\
&= A^{(1)} - V_{k-1}Y_{k-1}^T - X_{k-1}U_{k-1}^T - vy^T - xu^T \\
&= A^{(1)} - (V_{k-1},\, v)\ (Y_{k-1},\, y)^T - (X_{k-1},\, x)\ (U_{k-1},\, u)^T \\
&= A^{(1)} - V_k Y_k^T - X_k U_k^T.
\end{aligned} \tag{18}$$

And

$$\begin{aligned}
y^T &= \tau_v {A^{(k)}}^T v = \tau\, (A^{(1)} - V_{k-1}Y_{k-1}^T - X_{k-1}U_{k-1}^T)^T\, v \\
&= \tau_v\, ({A^{(1)}}^T v - Y_{k-1}V_{k-1}^T v - U_{k-1}X_{k-1}^T v), \qquad (19) \\
x &= z - \tau_u vy^T u = \tau_u A^{(k)} u - \tau_u vy^T u \\
&= \tau_u \left(A^{(1)} - X_{k-1}U_{k-1}^T - V_{k-1}Y_{k-1}^T\right) u - \tau_u vy^T u \\
&= \tau_u \left(A^{(1)} u - X_{k-1}U_{k-1}^T u - (V_{k-1},\, v)\, (Y_{k-1},\, y)^T\right) u
\end{aligned}$$

$$= \quad \tau_u \left(A^{(1)} u - X_{k-1} U_{k-1}^T u - V_k Y_k^T \right) u. \qquad (20)$$

From Eqs. 16 and 18, Eq. 17 is true for all $k \geq 2$. A is assumed to be partitioned into square blocks of size $n_b \times n_b$. In step k, the kth column of blocks (column panel) and the kth row of blocks (row panel) of A are reduced, after which the block reflectors are applied to the unprocessed trailing submatrix.

In LAPACK, a real matrix is reduced to bidiagonal form by calling the routine **DGEBRD**. Step k of the block algorithm proceeds as follows:

1. **DLABRD**: Reduce the kth panel of the matrix and compute V_{n_b}, U_{n_b}, Y_{n_b}, and X_{n_b}.

   [Repeat n_b times for $i = 1, \cdots, n_b$ (let $k_i = (k-1)n_b + i$)]

   1. Update the k_ith column of A.
   2. Compute the ith column Householder vector of A, v_i.
   3. Compute $y_i^T = \tau_v \left(A^T v_i - Y_{i-1} V_{i-1}^T v_i - U_{i-1} X_{i-1}^T v_i \right)$.
   4. Update the k_ith row of A.
   5. Compute the ith row Householder vector of A, u_i.
   6. Compute $x_i = \tau_u \left(A u_i - X_{i-1} U_{i-1}^T u_i - V_i Y_i^T u_i \right)$.

2. **DGEMM**: Update A with V and Y, $A \Leftarrow A - V\,Y^T$.

3. **DGEMM**: Update A with X and U, $A \Leftarrow A - X\,U^T$.

DGEBRD reduces each column panel and row panel of A in turn to generate the matrices V, U, Y, and X. 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 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 ith loop, two calls are made to **DGEMV** to reduce the ith column of the column panel $[A]_{:,k_i}$:

$$[A]_{:,k_i} \leftarrow [A]_{:,k_i} - V_{i-1}\, y_i^T - X_{i-1}\, u_i^T. \qquad (21)$$

Next, the routine **DLARFG** is called to generate the Householder transformation (τ_v, v_i) that introduces zeros below the diagonal in the ith column of the column panel. From Eq. 19, a sequence of five calls to the matrix-vector multiplication routine **DGEMV**, and a call to the scaling routine **DSCAL**, evaluates y_i.

Denoting the ith row of the current row panel of A as $[A]_{k_{i},:}$, it is reduced using two DGEMV calls:

$$[A]_{k_{i},:} \leftarrow [A]_{k_{i},:} - Y_i\, v_i^T - U_{i-1}\, x_{i-1}^T. \tag{22}$$

The routine DLARFG is called again to generate the Householder transformation (τ_u, u_i) that introduces zeros to the right of the superdiagonal in the ith 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 (τ_v, 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}. Then five calls to DGEMV, and one call to DSCAL, are used to evaluate x_i.

3.3.2. Parallel Bidiagonal Reduction

The conversion of the sequential routine for reducing a real matrix to bidiagonal form, DGEBRD, to the parallel ScaLAPACK version, PDGEBRD, is straightforward. The ScaLAPACK routine calls PDLABRD to reduce the kth 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, τ_v and τ_u. The unprocessed portion of the matrix is then updated as in Eq. 17 by two calls to the PBLAS matrix multiplication routine, PDGEMM.

The ScaLAPACK routine PDLABRD is implemented from the LAPACK routine DLABRD by replacing the calls to DLARFG, DGEMV, and DSCAL by calls to the corresponding parallel routines PDLARFG, PDGEMV, and PSCAL, respectively.

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 PBLAS, Y is stored in transposed form as an $n_b \times N$ matrix, in the same way that U is also stored. The ith 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 BLAS calls in the LAPACK code by corresponding calls to the PBLAS. 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 PBLAS.

Although replacing the sequential Level 1, 2, and 3 BLAS routines in LAPACK with the corresponding parallel PBLAS 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 PBLAS 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 PBLAS 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. 8) 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 PBLAS 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. Extensive performance evaluation has been done on the Intel iPSC/860, Delta, and Paragon computers. In Figure 1, we plot performance on the Intel Delta measured in Gflops (gigaflops per second) against number of processors while keeping the size of the matrix per processor fixed at 9 Mbytes. For an $N \times N$ matrix, 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

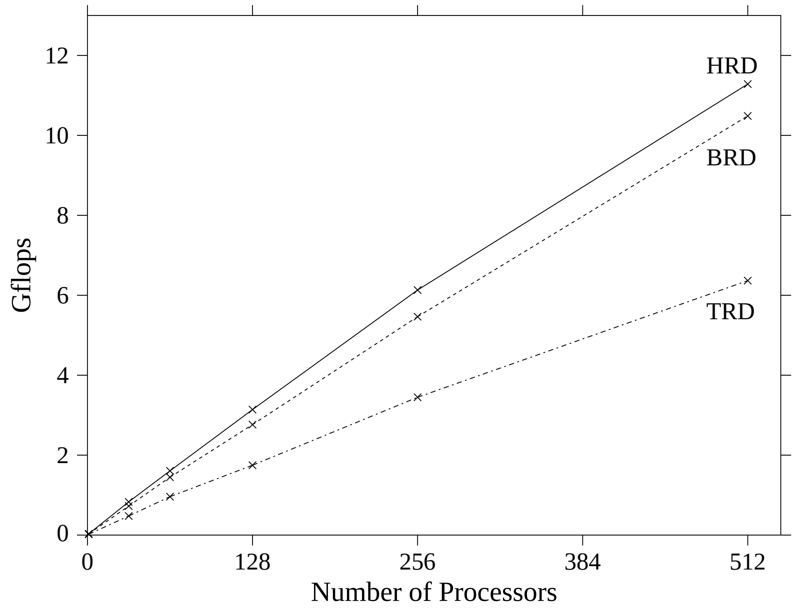

Figure 1: Isogranularity plots for the Hessenberg (HRD), tridiagonal (TRD), and bidiagonal reduction (BRD) routines on the Intel Delta. The matrix size per processor is fixed at 9 Mbytes.

form, and $\frac{4}{3}N^3$ for reduction to tridiagonal form. The algorithms for reduction to Hessenberg and bidiagonal form run at 11.5 and 10.5 Gflops on 512 processors, respectively, while that for reduction to tridiagonal form runs at about 6.5 Gflops. This difference is attributable to the fact that the tridiagonal reduction routine involves operations on a symmetric matrix, that is, the main updating computation routine, `PDSYR2K`, in Eq. 12 involves only the half of the matrix: upper or lower triangular part of the 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 [15]. The fact that the plots in Figure 1 are almost linear shows that the algorithms scale well on the Intel Delta at a granularity of 9 Mbytes/node. The isogranularity plots at 5 Mbytes/node are also almost linear, showing that good scalability is achieved when only about half of the available memory is used.

Figures 2, 3, and 4 show the performance of the three reduction algorithms as a function of matrix size for the 128-node Intel iPSC/860, the 512-node Intel Delta, and the 512-node Intel Paragon, respectively. Again, the differences in performance between the algorithms is largely attributable to their different floating-point operation counts.

Figure 5 compares the performance of the algorithm for reduction to Hessenberg form for

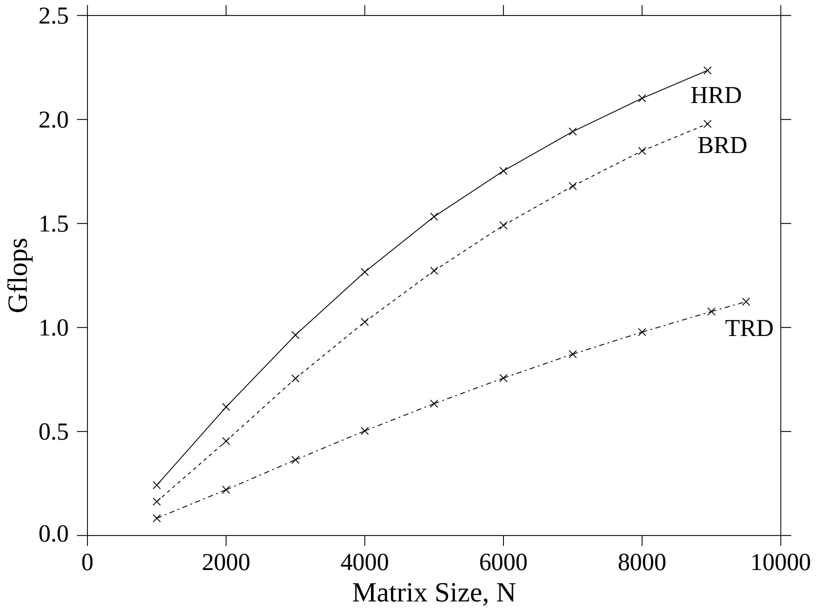

Figure 2: Perfomance of the Hessenberg (HRD), tridiagonal (TRD), and bidiagonal reduction (BRD) routines on the 128-node Intel iPSC/860 as a function of matrix size N. The optimium block size $n_b = 6$ was used.

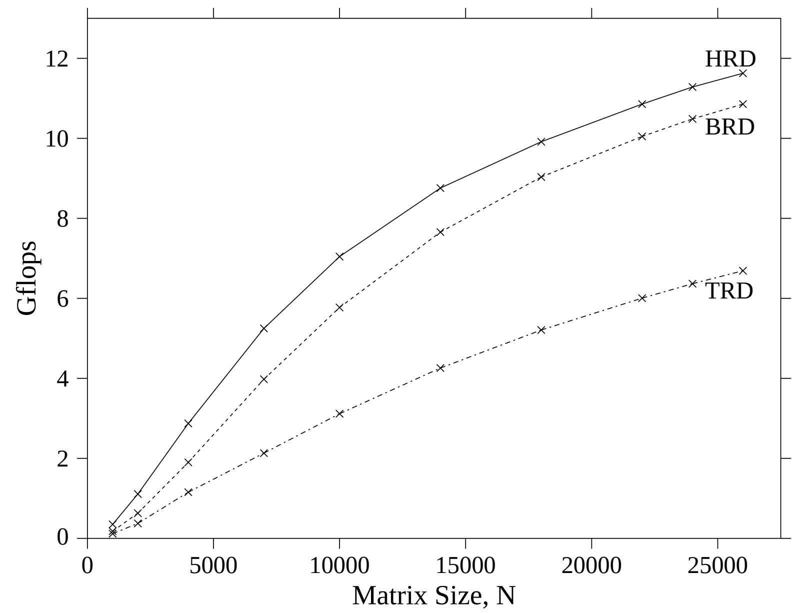

Figure 3: Perfomance of the Hessenberg (HRD), tridiagonal (TRD), and bidiagonal reduction (BRD) routines on the 512-node Intel Delta as a function of matrix size N. The optimium block size $n_b = 8$ was used.

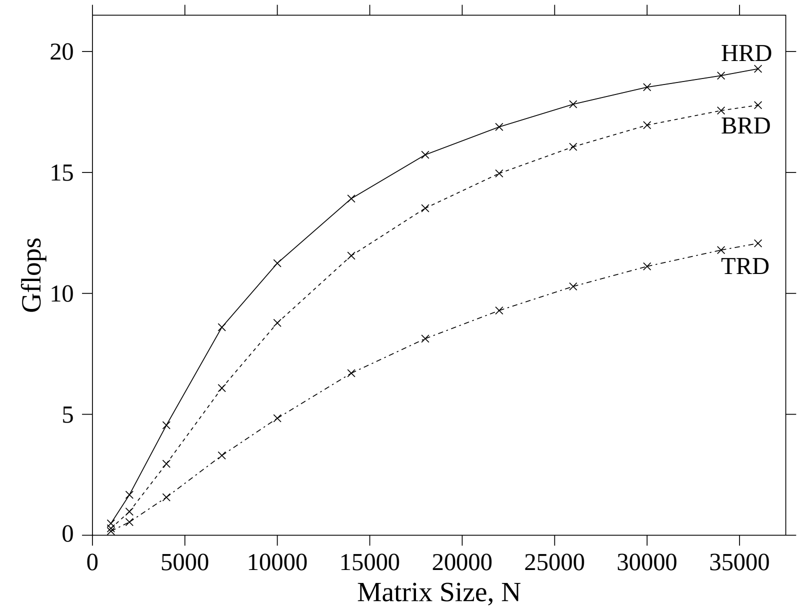

Figure 4: Perfomance of the Hessenberg (HRD), tridiagonal (TRD), and bidiagonal reduction (BRD) routines on the 512-node Intel Paragon as a function of matrix size N. The optimium block size $n_b = 6$ was used.

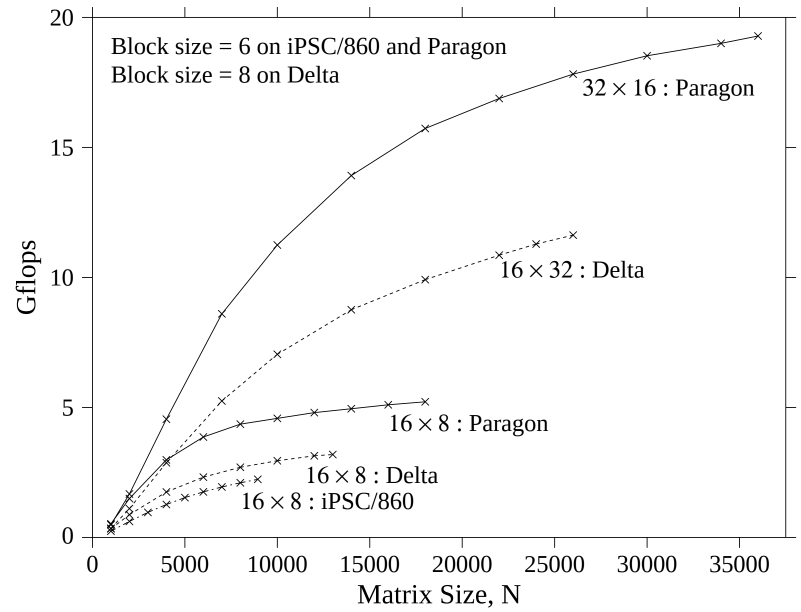

Figure 5: Performance of the algorithm for reduction to Hessenberg form as a function of matrix size N, on the Intel iPSC/860, Delta, and Paragon.

the three Intel computers. For each machine we choose the optimum layout of the process template and the optimum block size. The Intel iPSC/860 and Delta machines both use the same 40MHz i860 processor, so we attribute the better performance of the Delta compared with the iPSC/860 to its higher communication bandwidth. The Paragon uses the faster 50MHz i860XP processor, and has a larger communication bandwidth than the Delta and iPSC/860. Hence, the performance of the Paragon is significantly faster than the other two machines.

The Hessenberg, bidiagonal, and tridiagonal reduction routines attained 11.6, 10.9 and 6.7 Gflops for $N = 26000$ on the 512-node Intel Delta. This corresponds to 22.7, 21.2 and 13.1 Mflops per processor, respectively. The peak performance of the sequential assembly-coded BLAS routine, `DGEMM`, on the Delta is about 36.2 Mflops for a 400×400 matrix multiplication. Thus the routines achieve 62.7 %, 58.6 %, and 36.1 % of the maximum achieved performance for matrix multiply on the Intel Delta. On the Intel Paragon, the peak performance of `DGEMM` is about 46.3 Mflops for a 400×400 matrix multiplication, the communication bandwidth is higher, and there is more memory per processor. The ratio between the communication vs computations goes down as the problem size increases, as the communication cost is $O(n^2)$ and the computational cost is $O(n^3)$. Thus for a matrix problem of size $n = 36000$ on the Paragon, the routines achieve 81.3 %, 75.0 %, and 50.9 % of the peak, respectively.

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 PBLAS. The PBLAS, 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 1, 2, and 3 BLAS in the LAPACK code can be replaced on a one-for-one basis by the corresponding PBLAS routine.

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

The ScaLAPACK reduction routines are currently available through *netlib* for all numeric data types, such as single and double precision real and complex. To obtain the routines, send the message "`send index from scalapack`" to `netlib@ornl.gov`.

Acknowledgements

This research was performed in part using the Intel iPSC/860 hypercube and the Paragon computers at Oak Ridge National Laboratory, and in part using the Intel Touchstone Delta system operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to the Delta system was provided through the Center for Research on Parallel Computing.

6. References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du 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*, pages 1–10. IEEE Press, 1990.

[3] C. Bischof and C. Van Loan. The wy representation for products of householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8:s2–s13, 1987.

[4] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Submitted to Scientific Programming*, 1994. Also available on Oak Ridge National Laboratory Technical Reports, TM-12270, September, 1994.

[5] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In H. J. Siegel, editor, *Proceedings of the Fourth Symposium on Massively Parallel Computing*, pages 120–127, 1992.

[6] J. Choi, J. J. Dongarra, and D. W. Walker. The design of scalable software libraries for distributed memory concurrent computers. In J. J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 3–15, 1993. Proceedings of workshop held September 7-8, 1992, in Saint Hilaire du Touvet, France.

[7] J. Choi, J. J. Dongarra, and D. W. Walker. 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.

[8] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.

[9] J. Dongarra and S. Ostrouchov. LAPACK block factorization algorithms on the Intel iPSC/860. Technical Report 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(1):1–17, March 1988.

[12] J. J. Dongarra, S. J. Hammarling, and D. C. Sorensen. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27:215–227, 1989.

[13] J. J. Dongarra, R. van de Geijn, and D. W. Walker. A look at scalable dense linear algebra libraries. In IEEE, editor, *Proceedings of the Scalable High-Performance Computing Conference*, pages 372–379. IEEE Publishers, 1992.

[14] J. J. Dongarra and R. A. van de Geijn. Two-dimensional basic linear algebra communication subprograms. Technical Report LAPACK working note 37, Computer Science Department, University of Tennessee, Knoxville, TN, October 1991.

[15] J. J. Dongarra and R. A. van de Geijn. Reduction to condensed form for the eigenvalue problem on distributed memory architectures. *Parallel Computing*, 18:973–982, 1992.

[16] 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.

[17] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 1994. Accepted for publication.

[18] 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, N.J., 1988.

[19] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins Press, Baltimore, Maryland, 2nd edition, 1989.

[20] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5:308–323, 1979.

[21] W. Lichtenstein and S. L. Johnsson. Block cyclic dense linear algebra. *SIAM Journal on Scientific Computing*, 14(6):1259–1288, 1993.

[22] R. Schreiber and C. Van Loan. A storage efficient wy representation for products of housholder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10:53–57, 1989.

[23] C. Smith, B. Hendrickson, and E. Jessup. A parallel algorithm for Householder tridiagonalization. In *Proc. Fifth SIAM Conf. Appl. Linear Alg.*, pages 361–365, June 1994.

[24] E. F. Van de Velde. Data redistribution and concurrency. *Parallel Computing*, 16, December 1990.