A Parallel Implementation of Symmetric Band Reduction Using PLAPACK*

Yuan-Jye J. Wu,† Philip A. Alpatov,‡ Christian H. Bischof,§ and Robert A. van de Geijn¶

Abstract

Successive band reduction (SBR) is a two-phase approach for reducing a full symmetric matrix to tridiagonal (or narrow banded) form. In its simplest case, it consists of a full-to-band reduction followed by a band-to-tridiagonal reduction. Its richness in BLAS-3 operations makes it potentially more efficient on high-performance architectures than the traditional tridiagonalization method. However, a scalable, portable, general-purpose parallel implementation of SBR is still not available. In this article, we review some existing parallel tridiagonalization routines and describe the implementation of a full-to-band reduction routine using PLAPACK as a first step toward a parallel SBR toolbox. The PLAPACK-based routine turns out to be simple and efficient and, unlike the other existing packages, does not suffer restrictions on physical data layout or algorithmic block size.

1 Introduction

Reducing a full, dense symmetric matrix to tridiagonal form is one of the key steps in computing eigenvalues and eigenvectors of a symmetric matrix. Although serial tridiagonalization algorithms and implementations are well studied (e.g., LAPACK [1]), a scalable, portable, general-purpose parallel implementation remains a challenge. Existing software packages, like ScaLAPACK [2] or HJS [3], provide tridiagonalization routines that draw on low-level message-passing primitives and a serial BLAS library as their main buildings blocks. However, they all impose some constraints either on data layout or on the topology of the processor grid, which may cause difficulties in incorporating them into applications.

As an alternative to directly computing the tridiagonalization, a multistep approach called successive band reduction (SBR) was introduced by Bischof et al. [4, 5, 6]. Instead of reducing a full matrix to tridiagonal form directly, SBR first reduces the original matrix to banded form and then performs a band-to-tridiagonal reduction. This approach increases the potential for utilizing BLAS-3 operations. Therefore, we need routines for full-to-band and band-to-tridiagonal reductions, with tridiagonalization being a special case of the former. When we started the presented work, we have a full-to-band routine (called SBNDRD) based on MPI and serial BLAS.

Examining this existing SBNDRD and the other discussed tridiagonalization routines, which employ only message-passing communications and serial BLAS as the abstractions on which they are built, we found that the efficiency of both parallel tridiagonalization and full-to-band reduction is strongly affected by the following input options: the data layout, the final bandwidth, and the algorithmic block size for updating the matrix and accumulating the Householder transformations. However, the existing routines do not allow one to vary those factors independently. Hence, constraints are introduced on either the physical data layout or the algorithmic block size.

Our goal was to reimplement a full-to-band routine that does *not* suffer from these restrictions. Providing this generality requires considerable coding complexity to take care of partial data panels and index alignment. Thus, continuing in the previous low-level programming style was impractical because of the excessive amount of work that would be required.

Recently, van de Geijn et al. introduced a new parallel library, the Parallel Linear Algebra Package (PLAPACK) [7]. One of the goals of PLAPACK is to enable the rapid transformation of matrix algorithms

*This work is part of the PRISM project. It is supported in part by ARPA grant P-95006 and the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under contract W-31-109-Eng-38.

†Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439 (jwu@mcs.anl.gov).

‡Department of Physics, The University of Texas at Austin, Austin, TX 78712 (philip@physics.utexas.edu).

§Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439 (bischof@mcs.anl.gov).

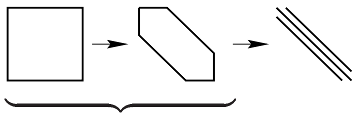
¶Department of Computer Sciences and Texas Institute for Computational and Applied Mathematics, The University of Texas at Austin, Austin, TX 78712 (rvdg@cs.utexas.edu).

into parallel codes while hiding the complicated local index alignment and data movement issues. Therefore, PLAPACK appeared to be an attractive infrastructure for rapidly prototyping our new approach. Conversely, the nontrivial algorithmic underpinnings of a blocked full-to-band reduction made it a good test case for validating the PLAPACK design goal.

In this article, we present our experiences in implementing a full-to-band reduction based on the PLAPACK infrastructure. We discuss the implications of PLAPACK on programming style and compare it with traditional parallel linear algebra libraries. Preliminary timing results on the IBM SP and Intel Paragon show that the portability and simplicity do not negatively impact performance.

2 Successive Band Reduction

A two-step SBR approach for tridiagonalization is illustrated in the following picture.



The first task (SBNDRD) is to reduce a full symmetric matrix to a banded matrix with arbitrary bandwidth. Typically, at each step we compute a Householder transformation, update part of the original matrix, and accumulate those transformations (if required).

2.1 Householder Transformation

Given a vector a, we can find a vector v such that

$$(I - \frac{2vv^T}{v^Tv})a = -\mathrm{sign}(\alpha_1)\|a\|_2 e_1, \tag{1}$$

where α_1 is the first component of a and e_1 is the first unit vector. Let us denote the Householder transformation H as

$$H = I + yw^T,$$

where

$$\begin{array}{l} y = v \quad \text{and} \\ w = -\frac{2v}{v^Tv}. \end{array}$$

The Householder transformation is a rank-one update, that is, a BLAS-2 operation.

Accumulating a certain number of Householder transformations into a block update is a common way to enrich BLAS-3 operations. Suppose that we have a sequence of k Householder transformations H_i, $i = 1, \ldots, k$. For one-sided updates, we can use the *WY representation* [8] of the product of these transformations to achieve one rank-k update instead of a sequence of rank-one updates. Let $Y_1 = y_1$ and $W_1 = w_1$. The WY representation,

$$H_k \cdots H_1 = I + Y_k W_k^T, \tag{2}$$

can be computed by the following recursion: for $i = 2, \ldots, k$,

$$\begin{aligned} Y_i &= [Y_{i-1}\ y_i] \text{ and} && (3)\\ W_i &= [W_{i-1}\ \ (I + Y_{i-1}W_{i-1}^T)w_i]. && (4) \end{aligned}$$

For symmetric Householder updates, the *YZ representation* [9] can be employed to perform a rank-$2k$ update. Given a matrix A, suppose that we wish to compute

$$(I + y_k w_k^T) \cdots (I + y_1 w_1^T) A (I + w_1 y_1^T) \cdots (I + w_k y_k^T). \tag{5}$$

With the initial conditions $Y_1 = y_1$ and $Z_1 = z_1 = Aw_1 + \frac{1}{2} y_1 w_1^T A w_1$, we can compute Y_k and Z_k as follows: for $i = 2, \ldots, k$,

$$\begin{aligned} A_{i-1}w_i &= Aw_i + Y_{i-1}Z_{i-1}^T w_i + Z_{i-1}Y_{i-1}^T w_i, && (6)\\ z_i &= A_{i-1}w_i + \frac{1}{2} y_i w_i^T A_{i-1} w_i, && (7)\\ Y_i &= [Y_{i-1}\ y_i], \quad \text{and} && (8)\\ Z_i &= [Z_{i-1}\ z_i]. && (9) \end{aligned}$$

Then, the two-sided transformation equation (5) can be replaced by

$$A + Y_k Z_k^T + Z_k Y_k^T. \tag{10}$$

2.2 Full-to-Band Algorithm

Given a symmetric matrix A, we wish to find an orthogonal matrix Q such that

$$QAQ^T = B\ ,$$

where B is a banded matrix. Let the current working matrix A_{cur} be the original matrix A initially. Like the block QR decomposition, the full-to-band algorithm first takes a skinny column panel of A_{cur} to perform a panel QR decomposition. Those Householder transformations produced in the panel QR are saved to form the required W_{cur}, Y_{cur}, and Z_{cur} matrices for updating the rest of the original matrix. Then the same process is applied to the newest updated submatrix until no panel is left. The only difference between

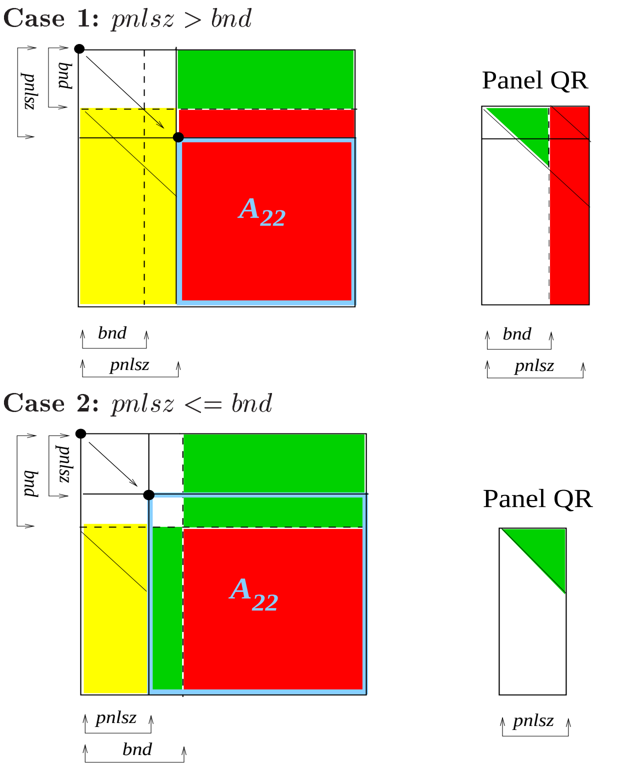


Figure 1: One step of the reduction procedure for the current working matrix A_{cur}

the full-to-band algorithm and block QR decomposition is that some updates on the original matrix are two sided.

Two parameters are given by the users in the algorithm: the final bandwidth (*bnd*) and the algorithmic block size (*nb*) for the block update. Figure 1 shows the detail of one reduction step on the current working matrix A_{cur}. The light-gray area represents the panel in A_{cur}. The mid-gray area denotes the portion to which one-sided Householder transformations have been applied, and the dark region requires two-sided Householder transformations. The panel size (*pnlsz*) is equal to the algorithmic block size *nb* (except possibly for the last panel).

In Case 1 ($pnlsz > bnd$), both one-sided (first *bnd* column) and two-sided (last $pnlsz - bnd$ columns) updates are required in the panel QR. The matrix A_{22} can be modified by one rank-$2k$ update operation using (10). However, since the panel QR in Case 2 ($pnlsz \leq bnd$) requires only one-sided transformations, the matrix Z is not formed, and A_{22} is updated by two rank-k updates using (2) from both sides.

Therefore, we can state the reduction algorithm as the following:

Reduction procedure:

While ($pnlsz > 0$) {
    Divide A_{cur} to get panel for QR and A_{22}
    Panel QR (light-gray)
    Update A_{22} (mid- & dark-gray)
    Replace A by A_{22} (black bullet)
    Compute the next $pnlsz$
}

For those problems requiring the accumulation of the Householder transformations, we compute the final matrix Q in a backward direction:

$$\underbrace{I(I+W_1Y_1^T)\cdots\underbrace{(I+W_{k-1}Y_{k-1}^T)\underbrace{(I+W_kY_k^T)}_{\texttt{1st step}}}_{\texttt{2nd step}}}_{\cdots\ \texttt{last step}}.$$

Since the orders of those WY matrices on the right sides are smaller than those on the left side, the backward accumulation fills in the identity matrix slowly thus requiring roughly one half of the operations that would be required for updating Q on the fly.

3 PLAPACK-based SBNDRD

Although the algorithm looks simple, a parallel implementation is nontrivial. (Indeed, a sequential implementation is nontrivial.) In general, a two-dimensional processor mesh and a two dimensional torus-wrap data mapping are preferable for this type of problem. For programs based on MPI-style message passing and serial BLAS, constraints are typically introduced to reduce the complicated local indices calculation and data movement. For example, the HJS tridiagonalization routine and an MPI version of the SBNDRD require a square-processor grid. The old parallel SBNDRD and the ScaLAPACK tridiagonalization allow data mapped in a block-torus wrapping, but the block size of the physical data layout must be equal to the algorithmic block size. Furthermore, the old parallel SBNDRD requires that the final bandwidth matches the algorithmic block size. Hence, using SBNDRD for tridiagonalization will only utilize BLAS-2 operations, and an implementation of Case 1 in Figure 1 is impossible. Moreover, the final bandwidth in the current SBNDRD has to divide the order of the original matrix evenly. These restrictions severely limit the applicability of this routine for a parallel SBR implementation.

```
    void prism_pla_sbndrd(PLA_LAObj A, PLA_LAObj Q, int nb, int bnd)
    {
     /********   declarations and definitions  ***********/
 (1) PLA_Matrix_global_all( A, &n, &n, &im, &in, &template);
 (2) PLA_Matrix_create(MPI_DOUBLE, n, n, template, im, in, &W);
 (3) PLA_Matrix_create(MPI_DOUBLE, n, n, template, im, in, &Y);
 (4) PLA_Matrix_create(MPI_DOUBLE, n, n, template, im, in, &Z);
 (5) PLA_Matrix_submatrix(A,PLA_DIM_ALL,PLA_DIM_ALL,PLA_ALIGN_FIRST,PLA_ALIGN_FIRST,&Acur);

 (6) m=n-bnd-1; /* the last global column index required by QR */
 (7) for (k=0; k< m; k+=nb) {  /* the first column index of each submatrix */
 (8)  pnlsz = (m-k > nb)? nb: (m-k);
 (9)  PLA_Matrix_submatrix(W, n-k, pnlsz, k, k, &Wcur);
(10)  PLA_Matrix_submatrix(Y, n-k, pnlsz, k, k, &Ycur);
(11)  PLA_Matrix_submatrix(Z, n-k, pnlsz, k, k, &Zcur);
      /* panel QR */
(12)  prism_pla_pnlqr(Acur, Wcur, Ycur, Zcur, bnd);

      /* Update A22 */
(13)  PLA_LAObj_split_4( Acur, pnlsz, pnlsz, &dummy, &dummy, &dummy, &A22);
(14)  if (pnlsz > bnd) {
```

$$A_{22} \leftarrow A_{22} + Z_{temp}Y_{temp}^T + Y_{temp}Z_{temp}^T$$

```
(15)   PLA_LAObj_horz_split_2( Ycur, pnlsz, &dummy, &Ytemp);
(16)   PLA_LAObj_horz_split_2( Zcur, pnlsz, &dummy, &Ztemp);
(17)   PLA_gemm(PLA_NOTRANS,PLA_TRANS, PLA_ONE , Ztemp, Ytemp,PLA_ONE, A22);
(18)   PLA_gemm(PLA_NOTRANS,PLA_TRANS, PLA_ONE , Ytemp, Ztemp,PLA_ONE, A22);
      }
```

$$A_{temp} \leftarrow (I + Y_{temp}W_{temp}^T)A_{temp}$$

```
(19)  else {
(20)   PLA_LAObj_horz_split_2( Ycur, bnd, &dummy, &Ytemp);
(21)   PLA_LAObj_horz_split_2( Wcur, bnd, &dummy, &Wtemp);
(22)   PLA_LAObj_horz_split_2( A22, bnd-pnlsz, &dummy, &Atemp);
(23)   PLA_Matrix_submatrix(Z, pnlsz, n-pnlsz-k, k, k+pnlsz, &work);
(24)   PLA_gemm(PLA_TRANS,PLA_NOTRANS, PLA_ONE, Wtemp, Atemp, PLA_ZERO, work);
(25)   PLA_gemm(PLA_NOTRANS,PLA_NOTRANS, PLA_ONE , Ytemp, work, PLA_ONE, Atemp);
```

$$A_{temp} \leftarrow A_{temp}(I + W_{temp}Y_{temp}^T)$$

```
(26)   PLA_LAObj_vert_split_2( A22, bnd-pnlsz, &dummy, &Atemp);
(27)   PLA_Matrix_submatrix(Z, n-pnlsz-k, pnlsz, k+pnlsz, k, &work);
(28)   PLA_gemm(PLA_NOTRANS,PLA_NOTRANS, PLA_ONE, Atemp, Wtemp, PLA_ZERO, work);
(29)   PLA_gemm(PLA_NOTRANS,PLA_TRANS, PLA_ONE , work, Ytemp, PLA_ONE, Atemp);
      }
(30)  PLA_LAObj_swap(&Acur, &A22);
     } /* end of A reduction */
```

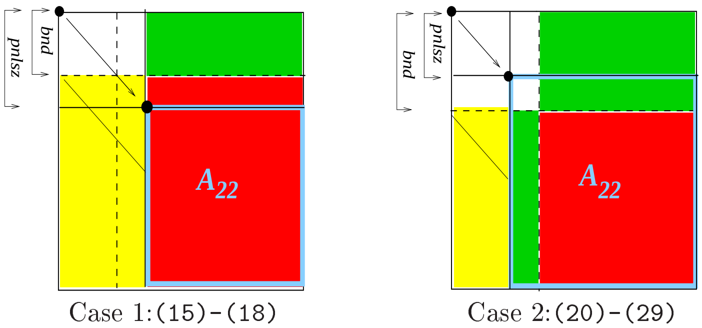


Figure 2: The actual C code for the reduction step (Note: some of the calling sequences reflect those used for an alpha-version of PLAPACK, and have since changed.)

```
(31) m=n-pnlsz-1;
(32) for (k=m; k>=bnd; k-=nb) {  order of each WY matrices */
(33)  PLA_LAObj_split_4(Q, k, k, &dummy, &dummy, &dummy, &Q2);
(34)  PLA_Matrix_submatrix(W, n-k, pnlsz, k, k-bnd, &Wtemp);
(35)  PLA_Matrix_submatrix(Y, n-k, pnlsz, k, k-bnd, &Ytemp);

(36)  if (k==m) {
(37)   PLA_gemm(PLA_NOTRANS,PLA_TRANS, PLA_ONE, Wtemp, Ytemp, PLA_ONE, Q2);
      }
(38)  else {
(39)   PLA_Matrix_submatrix(Z, pnlsz, n-k, k-bnd, k, &Ztemp);
(40)   if (nb>=1) {
(41)    PLA_gemm(PLA_TRANS,PLA_NOTRANS, PLA_ONE, Ytemp, Q2, PLA_ZERO, Ztemp);
(42)    PLA_gemm(PLA_NOTRANS,PLA_NOTRANS, PLA_ONE, Wtemp, Ztemp, PLA_ONE, Q2);
       }
      }
(43)  pnlsz = nb;
     } /* end of Q-update */

     /********  free LAObjs **********/
    } /* end of prism_pla_sbndrd */
```

$$\underbrace{(I+W_1Y_1^T)\underbrace{(I+W_2Y_2^T)\cdots\underbrace{(I+W_{k-1}Y_{k-1}^T)(I+W_kY_k^T)}}}_{\cdots}$$

Figure 3: The actual C code for the transformations accumulation (Note: some of the calling sequences reflect those used for an alpha-version of PLAPACK, and have since changed.)

PLAPACK, on the other hand, provides the infrastructure to implement SBR without those constraints. In the users' driver routine, an MPI communicator with a two-dimensional Cartesian topology and an integer for wrapping data in blocks are set to create a data layout template. Then, viewing each vector or matrix as a linear algebra object (LAObj) lying on the template, we are allowed to partition, split, or extract an LAObj in the usual mathematical sense. All local and global configuration of an LAObj is automatically created and computed in the data layout template. Thus, we do not need to worry about those complicated local data indices and alignments caused by the uneven data layout.

Figures 2 and 3 show the actual C code of the reduction step and transformations accumulation in PLAPACK-based SBNDRD routine, respectively. Corresponding mathematical equations and structure graphs in Figure 1 are also provided. We first extract the global information of the matrix A at line 1, such as the dimensions ($n \times n$), the data layout template, and the alignments (im, in) on the template. Then, at lines 2–4, we allocate space for the matrices W, Y, and Z properly aligned with A to save all the intermediate results in (3), (4), and (9). At the end, the matrices Y and Z are tridiagonal and W is block tridiagonal. The original matrix A is assigned to the working matrix A_{cur} at line 5. Note that this assignment just copies the LAObj's configuration locally, without allocating new space or copying actual entries.

The integer k at line 7 is the first global column index for the current working matrix A_{cur}. Then we extract the submatrices Y_{cur}, W_{cur}, and Z_{cur} before performing the panel QR. Those matrices are properly aligned on the first $pnlsz$ columns of A_{cur}, the light-gray and white areas in the graphs. The panel QR routine, `prism_pla_pnlqr`, has a coding format similar to this main routine, so we omit its description here. It generates the Householder transformations, computes the QR decomposition of the panel, and returns the computed Y_{cur}, W_{cur}, and Z_{cur}.

To update the rest of A_{cur}, we first extract the submatrix A_{22} at line 13. All the unwanted submatrices are assigned to the dummy LAObj `dummy`. For Case 1 in Figure 1, we get the proper YZ representations (Y_{temp} and Z_{temp}) by splitting Y_{cur} and Z_{cur} at lines 15–16 to compute the rank-$2k$ update

$$A_{22} + Z_{temp}Y_{temp}^T + Y_{temp}Z_{temp}^T$$

at lines 17–18.

For Case 2, in addition to extracting the proper WY representations (Y_{temp} and W_{temp}), we need to partition A_{22} to get a rectangular matrix A_{temp} for two different rank-k updates. Since the Z matrix is not computed in Case 2, we use part of its space for saving intermediate results. Note that we apply `submatrix` to ensure the proper global dimension and alignment for this temporary LAObj `work`. After setting up these LAObjs at lines 20–23 and 26–27, we compute

$$(I + Y_{temp}W_{temp}^T)A_{temp}$$

and

$$A_{temp}(I + W_{temp}Y_{temp}^T)$$

at lines 24–25 and 28–29. Finally, we use `swap` to exchange the configurations of the LAObjs A_{cur} and A_{22} at line 30 and continue the same process until the last panel is finished. The transformations accumulation in Figure 3 is similar to the reduction code, and we omit the description here.

In contrast to the previous SBNDRD and the HJS and ScaLAPACK implementations, this PLA_SBNDRD routine does not restrict choices of the algorithmic block size, the physical data layout, or the final bandwidth. One particularly notable feature of the code is that it lacks explicit index transformations between the global matrix view (at which the algorithm is expressed) and the local processor index view (at which computation is performed due to the distribution nature of data). Once the global matrix and vector partitions for PLAPACK are defined, no local indices need to be dealt with.

The simplicity of this code also makes it very easy to debug, document, maintain, and extend. In particular, we have added capabilities that are specifically addressed in the PRISM eigensolver [10]. For example, if the norm of a vector (or a column of a panel) a in (1) is small enough, we might skip computing the Householder transformation for a and continue the process on the next column in the panel. This skipping option is useful for PRISM because some matrices in the eigensolver are already close to the required banded form. To add this option requires only a simple "if" statement in our PLA_SBNDRD, but in the old code we need to take care of the changes of the number and length of the communicating messages caused by the skipping.

$n = 2048$	*bnd*			
$nb = 32$	4	8	16	32
PLA_SBNDRD	824.48	719.59	642.31	618.84
OLD_SBNDRD	65.07	51.68	45.00	41.98
PLA/OLD	12.67	13.92	14.23	14.74

Table 1: The first timing results on the IBM SP (in seconds)

4 Performance

We compare the performance of the PLA_SBNDRD code shown in Figures 2 and 3 with the current SBNDRD routine on the IBM SP and Intel Paragon platforms. The first test matrix is of order 2048 distributed on a 4×4 processor mesh. The wrapping block size (*tnb*) of the physical data layout is 24, and the algorithmic block update size (*nb*) is set to be 32. Partial timing results are shown in Table 1. In the worst case, the PLA_SBNDRD is slower than the OLD_SBNDRD by a factor of 15. However, both orthogonality and reduction residual tests for the new routine show that we have correct answers for any combinations of *bnd* and *nb* choosing from {1, 4, 8, 16, 32}. Thus, the PLAPACK infrastructure gets us up and running very fast. For algorithmic pieces that are not performance-critical, this rapid time is all that would be desired. However, for an efficient library routine, such performance is clearly unacceptable. Fortunately, the performance can be easily improved by tuning some LAObjs properties.

Figure 4(a) gives an example of setting the LAObj's orientation for one rank-k update for the reduction code in Figure 2. The mathematical equation can be broken into two steps:

$$\left\{ \begin{array}{lcl} X^T & \leftarrow & W_{temp}^T A_{temp} \\ A_{temp} & \leftarrow & A_{temp} + Y_{temp} X^T \end{array} \right. .$$

Although we already provide an LAObj `work` to save the intermediate matrix X^T, the property of transpose is not revealed because all LAObjs' orientations are set to be column panels initially. By setting the orientation of `work` to `ROW_PANEL` at the extra statement ($\star$), we are able to have `PLA_gemm` make a better data distribution and communication internally.

Another example is shown in Figure 4(b). Inside the panel QR, we need a certain small temporary space (single column or row) to save some intermediate LAObjs. Instead of memorizing the global alignments for these temporary LAObjs on a big working matrix, it is better to create the "conformal to" working LAObjs (line +) because it gives the proper global (a): Set LAObjs' orientations

$$A_{temp} \leftarrow (I + Y_{temp} W_{temp}^T) A_{temp}$$

```
(23)   PLA_Matrix_submatrix(Z, pnlsz, n-pnlsz-k, k, k+pnlsz, &work);
 (*)   PLA_LAObj_set_orientation(work, PLA_ROW_PANEL);
(24)   PLA_gemm(PLA_TRANS,PLA_NOTRANS, PLA_ONE, Wtemp, Atemp, PLA_ZERO, work);
(25)   PLA_gemm(PLA_NOTRANS,PLA_NOTRANS, PLA_ONE , Ytemp, work, PLA_ONE, Atemp);
```

(b): Create "conformal to" LAObjs

$$a \leftarrow (I + YW^T)a$$

```
 (+)   PLA_Pvector_create_conf_to(W, PLA_PROJ_ONTO_ROW,PLA_DUPLICATE,&work);
       PLA_gemv(PLA_TRANS, PLA_ONE, W, a, PLA_ZERO, work);
       PLA_gemv(PLA_NOTRANS, PLA_ONE, Y, work, PLA_ONE, a);
```

Figure 4: Examples for optimization

Routine	Panel QR	A_{22} Update	Q Update	Total
Original	103.6150	62.2850	11.6312	177.75
Optimized	13.4601	5.6688	2.1606	21.49

Table 2: Improvement on the Intel Paragon for $n = 750$, $tnb = 30$, $bnd = 30$, and $nb = 30$ on 3×3 processor mesh (in seconds)

alignments as well as orientations at once.

With simple adjustments, we significantly improved the performance of PLA_SBNDRD. Note that these adjustments are easily to made by users, based on usual mathematical sense, and they do not adversely affect the mapping from the algorithmic equations to the actual code. Table 2 shows the improvement on the Intel Paragon. The test matrix is of order 750 distributed on a 3×3 processor mesh with wrapping block size 30. Both the algorithmic block size and the final bandwidth are set to be 30. The improvement on the IBM SP is given in Table 3. The matrix size is 2048 on a 4×4 processor grid. The numbers in parentheses below are the Mflops per node. We see that PLA_SBNDRD is now only a factor of 1.5 slower than the OLD_SBNDRD (compared with 15 earlier), and the blocking enabled by Case 1 in Figure 1 results in a speedup for tridiagonalization.

Further possible items for optimization are as follows:

- In the `split` operation, the assignment on dummy LAObj `dummy` results in redundant computations on those useless configurations. To reduce the overhead, we can replace `split` by `view_shift`, which can directly recompute a required LAObj's configuration locally. The `view_shift` naturally supports the "sliding window" paradigm commonly used in matrix operations.
- We can exploit symmetry in the updates of A.
- We can spread the skinny panels, like Y_{cur}, to all the processors to balance the work.
- It should be noted that the PLAPACK infrastructure itself has not yet been tuned. All tuning performed to date involved changing the PLA_SBNDRD code itself.

5 Conclusions

We have presented a new version of the SBNDRD full-to-band routine based on PLAPACK infrastructure. The code is simple and clean, without any explicit global-local index transformations and communications. We also demonstrated that the portability and simplicity of this code do not impact performance.

Our experience indicates several benefits:

- Using PLAPACK, one can code at the global index level that is also used in the algorithmic formulation. There is no need for dealing with local indices.
- Unlike traditional parallel programming, coding focuses on the global matrices or vectors parti-

Parameters			Optimized PLA_SBNDRD				OLD_SBNDRD	NEW/OLD
nb	*bnd*	*tnb*	Panel QR	A_{22} Update	Q Update	Total	Timings	Ratio
32	32	16	22.55 (0.75)	28.17 (50.01)	11.39 (61.87)	62.74	41.98	1.49
1	1	32	15.10 (0.035)	165.00 (10.84)	65.62 (13.63)	247.75	134.15	1.85
32	1	32	85.71 (4.48)	11.87 (62.58)	12.30 (59.97)	110.44	134.15	0.82

Table 3: Improvement on the IBM SP for $n = 2048$ on 4×4 processor mesh (in seconds)

tions rather than local data movements and computations.

- With PLAPACK BLAS, the *correct* answer can be obtained with very little coding effort.

Further work will deal with fine-tuning PLA_SBNDRD and creating a parallel implementation of the band-to-tridiagonal algorithm.

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 Users' Guide*, 2nd edition. SIAM, Philadelphia, 1995.

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

[3] B. Hendrickson, E. Jessup, and C. Smith. A parallel eigensolver for dense symmetric matrices. Sandia National Laboratory Report, 1996.

[4] C. Bischof, M. Marques, and X. Sun. Parallel band reduction and tridiagonalization. *Proceedings of Sixth SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, 383–390, 1993.

[5] C. Bischof, B. Lang and X. Sun. A framework for symmetric band reduction. Mathematics and Computer Science Division Preprint ANL/MCS-P586-0496, Argonne National Laboratory, 1996.

[6] C. Bischof, B. Lang and X. Sun. The SBR toolbox – software for successive band reduction. Mathematics and Computer Science Division Preprint ANL/MCS-P587-0496, Argonne National Laboratory, 1996.

[7] R. van de Geijn et al. Parallel Linear Algebra Package (PLAPACK): Release R0.1 (Beta) Users' Guide. Unpublished manuscript, 1996.

[8] C. Bischof and C. Van Loan. The WY representation for products of Householder matrices. *SIAM J. Sci. Stat. Comput.*, 8:s2–s13, 1987.

[9] J. Dongarra, S. Hammarling, and D. Sorensen. Block reduction of matrices to condensed forms for eigenvalue computations. *J. Comp. and Appl. Math.*, 27:215–227, 1989.

[10] C. Bischof, S. Huss-Lederman, X. Sun, A. Tsao, and T. Turnbull. Parallel studies of the invariant subspace decomposition approach for banded symmetric matrices. *Proceedings of Seventh SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, 516–521, 1995.