Packed Storage Extension for ScaLAPACK

Ed D'Azevedo[*]*, Jack Dongarra*[†]

1 Introduction

We describe an extension to ScaLAPACK for computing with symmetric (and hermitian) matrices stored in a packed form. This is similar to the compact storage for symmetric (and hermitian) matrices available in LAPACK [2]. This enables more efficient use of memory by storing only the lower or upper triangular part of a symmetric matrix. The capabilities include Choleksy factorization (PxSPTRF) and solution (PxSPTRS) of symmetric (Hermitian) linear systems, the computation of eigenvalues and eigenvectors (PxSPEV), expert drivers (PxSPEVX) for generalized eigenvalue problem (PxSPGVX) for symmetric (Hermitian) matrices in packed storage. This work differs from an earlier work [5] on packed storage by considering wider block column panels of width `NB * NPCOL` instead of single block column of width `NB` in performance sensitive routines. The goal is to reduce the overhead in index calculation by increasing the granularity and operating on wider column panels.

The packed storage scheme (described in §2) resembles the ScaLAPACK data distribution but physically stores only the lower (or upper) *blocks*. Each block column panel of width `NB * NPCOL` can be considered as a trapezoidal submatirx in a fully dense ScaLAPACK matrix. Section 3 contains two concrete examples on how such an arrangement can be used with conventional ScaLAPACK routines for dense storage. For some performance critical PBLAS like routines such as PxTPSM (triangular solve) and PxTPMM (multiplication by triangular matrix), block diagonal submatrices are copied into fully dense ScaLAPACK matrices to reuse standard high perform parallel BLAS routines. The right-looking Cholesky factorization re-

[*]Computer Science and Mathematics Division, Oak Ridge National Laboratory, Bldg 6012, MS6367, Oak Ridge, TN37831

[†]Computer Science Department, University of Tennessee, Knoxville,TN 37996–3450

lies on efficient symmetric rank-k updates (PxSPRK) to the right unfactored submatrix. The algorithm uses two replicated row and column vectors to minimize the communication volume. The details for the BLAS like routines are in §4 Finally, results of numerical experiments on a Beowulf Linux cluster is presented and discussed in §5.

2 Data layout for packed storage

ScaLAPACK principally uses a two-dimensional block-cyclic data distribution (see Figure 1) for full dense in-core matrices [3, Chapter 4]. This distribution has the desirable properties of good load balancing where the computation is spread reasonably evenly among the processes, and can make use of highly efficient level 3 BLAS (Basic Linear Algebra Subroutines) at the process level. Each colored rectangle represents an $mb \times nb$ submatrix. Matrix entry (i, j) is mapped to matrix block $(ib, jb) = (1 + \lfloor (i-1)/mb \rfloor, 1 + \lfloor (j-1)/nb \rfloor)$ and is assigned to process $(p, q) = (\mathrm{mod}(ib - 1, P_r), \mathrm{mod}(jb - 1, P_c))$ on a $P_r \times P_c$ process grid. Thus the first entry $(1, 1)$ is mapped to process $(0, 0)$ and entry $(1 + mb, 1 + nb)$ is mapped to process $(1, 1)$.

The packed storage scheme resembles the ScaLAPACK two-dimensional block-cyclic data distribution but physically stores only the lower (or upper) *blocks*. For example, on a 2×3 process grid as shown in Figure 1, if only the lower blocks are stored, then process $(0, 0)$ holds blocks A_{11}, A_{31}, A_{51}, A_{71}, A_{54}, A_{74}, A_{77}. Process $(0, 2)$ holds blocks A_{33}, A_{53}, A_{73} and A_{76}. Similarly process $(1, 1)$ holds blocks A_{22}, A_{42}, A_{62}, A_{82} and A_{65}, A_{85} plus A_{88}. We note that each block in the packed storage scheme is assigned to the *same* process as in the fully two-dimensional block-cyclic data distribution. Note that each block column in the packed storage scheme may be considered a full ScaLAPACK matrix distributed across only one process column. Moreover, each column panel of width $nb * P_c$ may be considered (with some adjustment to starting address) a trapezoidal ScaLAPACK matrix. This treatment of a block column or column panel as a particular ScaLAPACK submatrix is a key characteristic to the reuse of ScaLAPACK and PBLAS library components.

If we consider the 'local' view in process $(0, 0)$, the first block column panel consists of A_{11}, A_{31}, A_{51} and A_{71}. This panel is stored in memory as a $(4 * mb) \times nb$ Fortran column-major matrix. The second block column panel consists of blocks A_{54} and A_{74}. It is stored in local memory as a $(2 * mb) \times nb$ Fortran column-major matrix. The first entry of the second panel follows the last entry of the first panel in memory, i.e. the first entry in block A_{54} follows the last entry in block A_{71}. Note that the entire diagonal block A_{11} is stored, even though only the lower triangular part is accessed. This incurs a small price in extra storage but greatly simplifies reuse of ScaLAPACK components.

3 Examples in the use of packed storage matrix

Here we illustrate by two examples the reuse of ScaLAPACK library components for matrices stored in packed form. The key idea is the treatment of each block

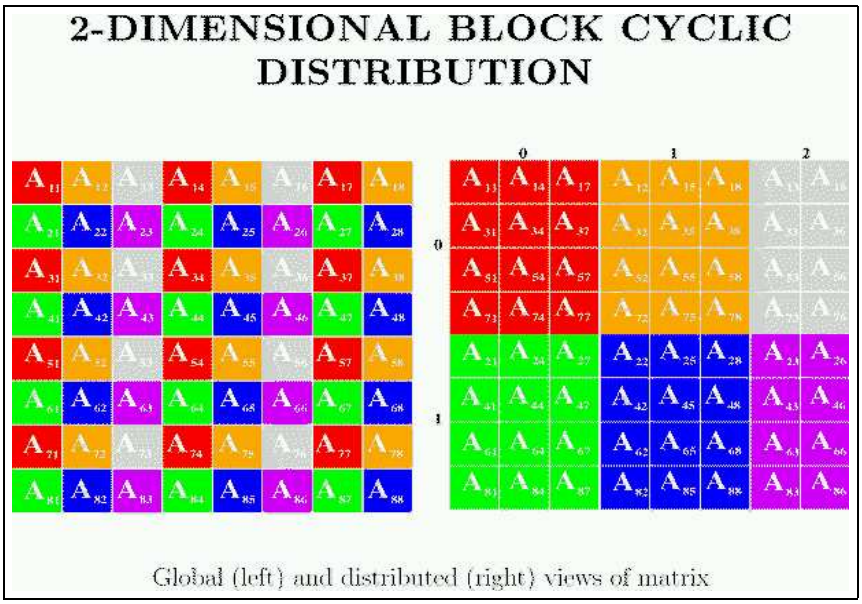

Figure 1. *Two-dimensional block-cyclic distribution.*

column or panel as a regular ScaLAPACK matrix distributed across the process grid. The routine `DESCINITT` (`DESCINITTW`) is provided to simplify the manipulation of indices by initializing a new matrix descriptor for a block column (panel). The routine interface can be described using Fortran 90 syntax as

```
 SUBROUTINE DESCINITT(UPLO, IA, JA, DESCA, &
                      IAP, JAP, LOFFSET, DESCAP)
CHARACTER,INTENT(IN) ::  UPLO
INTEGER, INTENT(IN) ::  IA, JA, DESCA(*)
INTEGER, INTENT(OUT) ::  IAP, JAP, LOFFSET, DESCAP(*)
END SUBROUTINE DESCINITT
```

For example, access to the global entry `A(IA, JA)` in full storage is obtained by the ScaLAPACK routine

```
 CALL PDELGET( SCOPE, TOP, ALPHA, A, IA, JA, DESCA )
```

The corresponding code to access the lower triangular entry in packed storage would be

```
 CALL DESCINITT('Lower', IA, JA, DESCA, &
                      IAP, JAP, LOFFSET, DESCAP)
CALL PDELGET(SCOPE, TOP, ALPHA, A(LOFFSET), IAP, JAP, DESCAP)
```

The routine `DESCINITT` generates a new matrix descriptor `DESCAP` that corresponds to the block column panel with new indices `(IAP, JAP)` relative to the new de-

```
1   ! Process grid is NPROW by NPCOL
2   CALL BLACS_GRIDINFO( DESCA(ICTXT_),NPROW,NPCOL,MYPROW,MYPCOL)
3   ANRM = ZERO
4   N = DESCA(N_)        ! Number of columns in matrix A
5   NB = DESCA(NB_)      ! Width of each block column
6   NNB = NB * NPCOL     ! Width of column panel
7   DO JA=1,N,NB*NPCOL
8      JB = MIN( NNB, N-JA+1 )
9      IA = JA
10     !
11     ! Generate new descriptor for wide column panel
12     !
13     CALL DESCINITTW('Lower',IA,JA,DESCA,IAP,JAP,LOFFSET,DESCAP)
14     !
15     ! Handle diagonal block
16     !
17     ANRM2 = PDLANSY('M','Lower',JB,A(LOFFSET),IAP,JAP,DESCAP,WORK)
18     ANRM = MAX( ANRM, ANRM2 )
19     !
20     ! Handle off-diagonal rectangular block
21     ! Use Lower triangular part
22     !
23     IA = IA + JB
24     IF (IA .LE. N) THEN
25       ANRM2 = PDLANGE('M',N-IA+1,JB,A(LOFFSET),IAP+JB,JAP,DESCAP,WORK)
26       ANRM = MAX( ANRM, ANRM2 )
27     ENDIF
28  ENDDO
```

Figure 2. *Example code to illustrate the use of* `DESCINITTW` *and reuse of ScaLAPACK components for matrices stored in packed storage.*

scriptor. It will also produce the correct value for `LOFFSET` to adjust for the beginning of the column panel.

Another more complicated example (see Figure 2) is computing the largest absolute value $(\max(|A(I,J)|))$ in a packed matrix. This is similar to computing with the `NORM='M'` option in `PDLANSY` for the full storage,

```
ANRM = PDLANSY( 'M', UPLO, N, A, 1, 1, DESCA, WORK )
```

This code uses the block column panel of width `NB * NPCOL` and reuses ScaLAPACK `PDLANSY` and `PDLANGE` for computing the maximum entry in each block column panel. Better performance may be obtained by using a wider `NB * NPCOL` panel over a single block column of width `NB`.

The code traverses each block column panel (line 7) and calls `DESCINITTW` to establish the appropriate matrix descriptor for a wide column panel. It calls

PDLANSY (line 17) to find the largest value in the diagonal block. Routine PDLANGE (line 25) computes the largest value in the remaining off-diagonal rectangular block. Although essentially the same computation is performed, the packed version has higher overhead in making several separate calls to PDLANSY and PDLANGE. The performance of packed storage relies on using the dense storage PDLANSY and PDLANGE that may introduce extra implicit synchronizations; moreover, the granularity of the algorithm is limited by the width of the column panel (NB * NPCOL).

4 Algorithms for packed storage

Many high level ScaLAPACK routines, such as Cholesky factorization, can be easily adapted to using the packed storage by using DESCINITT and DESCINITTW that provide a simple interface. Most of the subroutines for dense storage perform computations by looping over a a block column at a time and rely on efficient implementation of PBLAS for optimal performance. Thus, special attention was given to providing several performance critical PBLAS like routines such as PxTPSM (triangular solve), PxTPMM (triangular matrix multiply), PxSPRK (symmetric rank-K update), PxSPR2K (symmetric rank-2K update) for packed storage.

Both PxTPSM and PxTPMM rely on operating on a wider block column panel (NB * NPCOL) for increased performance. One technique copies the block diagonal triangular part into full dense ScaLAPACK storage to utilize PBLAS PxTRSM or PxTRMM. Note that no communication is required since the packed and dense storage format uses the same mapping of entries to to processors. On a square process grid, the additional storage is only one block per processor. Another technique considers the rectangular submatrix of the column panel as a regular dense ScaLAPACK matrix and uses the more efficient PxGEMM to perform updates to the right-hand matrix. This should yield good performance for a large number of righ-hand vectors. However, for a narrow right-hand matrix (NRHS $\leq$ NB), the underlying PBLAS implementation might broadcast the right-hand matrix across the column panel to perform the update. This broadcast would increase the communication volume by moving the right-hand matrix multiple times and might introduce unnecessary synchronization barriers.

PxSPRK ($C \leftarrow \beta C + \alpha A A^T$) is used in the right-looking Cholesky factorization and PxSPR2K ($C \leftarrow \beta C + \alpha A B^T + \alpha B A^t$) is used in the generalized eigen solvers. Here matrix C is in packed storage and matrices A (and B) are usually NB columns wide. The straightforward implementation of updating matrix C using block columns or column panels by PBLAS incurs a high communication cost. Essentially for each block column (or panel), the submatrices of matrix A are communicated across the process grid. We can reduce this communication cost by maintaining a replicated column copy A_c and a row copy A_r of matrix A. PBLAS version 2 supports a new capability for handling *replicated* matrices. The value of DESC(RSRC_) = -1 signals a copy of the matrix is replicated across the the processor rows. Similarly, DESC(CSRC_) = -1 replicates across the processor columns. PBLAS PxGEADD and PxTRAN can then perform the copy and transpose copy operations from A to A_r and A_c. This may be similar to a row or column broadcast

across the process grid. After this communication, no further communication is then required in updating the matrix C in compact storage. However, the broadcast across the process grid may be an additional implicit synchronization barrier.

5 Numerical experiments

We have developed the following prototype codes: PxPPTRF/PxPPTRS for Cholesky factorization and solution, simple driver PxSPEV (PxHPEV) routines for finding eigenvalues and eigenvectors of symmetric (Hermitian) matrices stored in packed form, expert drivers for symmetric (Hermitian) matrices PxSPEVX/PxHPEVX and generalized eigenvalue problems PxSPGVX/PxHPGVX.

We have compared the performance of the new routines in packed storage with ScaLAPACK routines in full storage. The goal is to determine the overhead cost of computation with packed storage over the existing routines for full storage. The new routines have higher overhead in index calculations and have algorithm granularity limited by the width of the block column panel. However, the packed storage may have better data locality and cache reuse.

The tests were performed on the TORC II* Beowulf Linux cluster. Each node consisted of a dual Pentium-II at 450Mhz with 512MBytes of memory running redhat linux 6.2 with smp kernel. MPIBLACS was used with LAM/MPI† version 6.3 with Gigabit ethernet. Job submission to the cluster was through the Portable Batch System (PBS)‡. Since the cluster was a shared resource, there was some variation in runtimes. Each case was run several times and the minimum time was taken. Two MPI tasks were spawned on each node to utilize both cpus. A single cpu can achieve about 300Mflops/s in DGEMM operations with optimized BLAS libraries produced by Automatically Tuned Linear Algebra Software (ATLAS)§ The experiments were performed with MB=NB=50 using PBLAS [4, 7] version 2¶. Results for upper case (UPLO='U') and lower case (UPLO='L') were very similar so results for only the upper case are presented.

Table 1 summarizes the times for the Cholesky factorization PDPOTRF for full storage and PDPPTRF for packed storage. Routines PDPPTRS and PDPOTRS were used to solve the factored system with 50 and 1000 (NRHS) right-hand vectors. For the small problem size (N=5000) considered, the times for factorization by PDPPTRF with packed storage were similar to to times taken by PDPOTRF with full storage. For the larger problem size (N=10000), the times for packed storage were 30-50% higher. Solution times for a narrow right-hand matrix (NRHS=50) show PDPPTRS for packed storage to be slower than PDPOTRSfor full storage for large problems (N=10000) and similar to full storage on the smaller problem (N=5000). Solution times for a wide right-hand matrix (NRHS=1000) show PDPPTRS for packed storage to be competitive with PDPPTRS, especially for the smaller problem size.

*Visit http://www.epm.ornl.gov/torc/ for details. Special thanks to Stephen Scott for arranging use of the cluster.

†Visit http://www.mpi.nd.edu/lam for details.

‡Visit http://www.openpbs.com/ for details.

§Visit http://www.netlib.org/atlas/ for details.

¶Visit http://www.netlib.org/scalapack/html/pblas_qref.html for details.

Table 2 summarizes the execution times for the symmetric eigensolvers PDSYEV with full storage and PDSPEV with packed storage. The computations were performed with JOBZ='N' to find all eigenvalues or with JOBZ='V' to find all eigenvectors and eigenvalues. Routine PDSPEV for packed storage incurred at most a 20% increase over PDSYEV for full storage in finding eigenvalues only. In some cases (N=2000) PDSPEV even performed slightly better than PDSYEVwith full storage. On closer examination and profiling, we find part of the extra time is incurred in a routine to perform a matrix vector multiply operation where the matrix is stored in packed storage. Performance of DSYMV and DGEMV for the packed version may be limited by the width of the block column panel and extra overhead in subroutine calls to dense routines. When both eigenvectors and eigenvalues are required, PDSPEV compared favorably with PDSYEV for full storage.

Table 3 summarizes the execution times for the expert drivers for the symmetric eigensolvers. Although the expert driver is capable of finding specific clusters of eigenvalues, all eigenvalues (RANGE='ALL') were requested. The routine PDSPEVX performs reorthogonalization of eigenvectors when there is sufficient temporary workspace. This reorthogonalization can cause the higher run times for finding all eigenvectors over the simple driver PDSYEV. In these runs, reorthogonalization is turned off by setting ORFAC=0 and ABSTOL=0 Performance analysis of PDSYEVX is described in [3, Chapter 5] and [6]. For both shorter runs with JOBZ='N' where only eigenvalues are sought, and longer running computation where both eigenvectors and eigenvalues were requested (JOBZ='V'), PDSPEVX for packed storage was comparable to PDSYEVX for full storage.

Table 4 and Table 5 summarize the times for the generalized symmetric eigensolvers PDSPGVX with packed storage and PDSYGVX with full storage for finding all eigenvalues with RANGE='All'. The input parameter IBTYPE describes the type of problem to be solved:

$$\texttt{IBTYPE} = \begin{cases} 1 & \text{solve } Ax = \lambda Bx, \\ 2 & \text{solve } ABx = \lambda x, \\ 3 & \text{solve } BAx = \lambda x. \end{cases} \tag{1}$$

The problem is reduced to canonical form by first performing a Cholesky factorization on B ($B = LL^H$ or U^HU) and then overwriting A with

$$\texttt{IBTYPE} = \begin{cases} 1 & A \longleftarrow U^{-H}AU^{-1} \text{ or } L^{-1}AL^{-H}, \\ 2 \text{ or } 3 & A \longleftarrow UAU^H \text{ or } L^HAL. \end{cases} \tag{2}$$

For the cases IBTYPE=2 and IBTYPE=3, the packed version incurs a significant extra overhead of about 25% compared to the version for full storage. The in-place conversion of matrix A to canonical form (2) may require access to block rows in matrix A or B. Since the packed storage is stored in a column panel oriented manner, traversal *across* block rows will be less efficient than traversal *down* columns.

The results suggest packed storage for Cholesky factorization with large matrices and few right-hand vectors still need improvement. Recently, a new recursive implementation of Cholesky factorization in LAPACK compact storage [1]

				NRHS=50		NRHS=1000	
$P_r \times P_c$	N	PDPOTRF	PDPPTRF	PDPOTRS	PDPPTRS	PDPOTRS	PDPPTRS
2×2	5000	47.1s	49.9s	4.0s	8.3s	56.5s	56.5s
4×4	5000	18.3s	20.4s	2.2s	4.3s	22.2s	22.8s
6×6	5000	27.9s	36.1s	5.3s	5.3s	39.6s	43.7s
8×8	5000	13.8s	13.8s	3.3s	3.3s	18.2s	18.2s
4×4	10000	109.4s	166.7s	6.0s	15.2s	81.7s	83.5s
6×6	10000	111.5s	145.8s	10.0s	21.7s	122.0s	132.7s
8×8	10000	47.8s	67.3s	7.7s	11.4s	49.8s	62.7s

Table 1. *Performance (in seconds) of Cholesky factorizations and solves.*

shows performance approaching the fully dense case. It would be interesting to consider a similar recursive packed format for parallel computing.

6 Acknowledgement

This work was supported in part by the National Science Foundation Grant No. ASC-9005933; by the Defense Advanced Research Projects Agency under contract DAAL03-91-C-0047, administered by the Army Research Office; by the Office of Scientific Computing, U.S. Department of Energy, under Contract DE-AC05-00OR22725; and by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615, and Center for Computational Sciences at Oak Ridge National Laboratory for the use of the computing facilities.

$P_r \times P_c$	N	JOBZ	PDSYEV	PDSPEV
2×2	1000	N	7.5s	8.1s
4×4	1000	N	7.7s	8.1s
6×6	1000	N	17.4s	18.3s
8×8	1000	N	23.6s	27.9s
2×2	2000	N	37.0s	38.6s
4×4	2000	N	24.1s	25.6s
6×6	2000	N	65.9s	59.5s
8×8	2000	N	94.8s	82.5s
2×2	4000	N	244.6s	249.9s
4×4	4000	N	101.9s	109.9s
6×6	4000	N	185.9s	190.1s
8×8	4000	N	300.9s	323.7s
2×2	1000	V	30.2s	30.8s
4×4	1000	V	14.7s	15.0s
6×6	1000	V	12.2s	12.5s
8×8	1000	V	13.0s	13.0s
2×2	2000	V	220.5s	220.1s
4×4	2000	V	76.2s	78.5s
6×6	2000	V	54.7s	56.9s
8×8	2000	V	40.2s	42.1s
2×2	4000	V	1784.4s	1793.2s
4×4	4000	V	486.4s	489.2s
6×6	4000	V	295.1s	297.6s
8×8	4000	V	213.3s	215.5s

Table 2. *Performance (in seconds) of simple drivers for symmetric eigensolvers.*

$P_r \times P_c$	N	JOBZ	PDSYEVX	PDSPEVX
2×2	1000	N	7.6s	8.0s
4×4	1000	N	6.1s	6.4s
6×6	1000	N	9.2s	9.8s
8×8	1000	N	20.4s	16.1s
2×2	2000	N	39.3s	39.7s
4×4	2000	N	19.8s	21.1s
6×6	2000	N	34.0s	33.4s
8×8	2000	N	51.4s	50.2s
2×2	4000	N	256.0s	256.8s
4×4	4000	N	91.6s	94.8s
6×6	4000	N	128.1s	131.4s
8×8	4000	N	178.0s	179.8s
2×2	1000	V	72.4s	72.5s
4×4	1000	V	71.3s	71.1s
6×6	1000	V	88.7s	91.2s
8×8	1000	V	74.2s	74.2s
2×2	2000	V	723.4s	718.6s
4×4	2000	V	706.6s	690.5s
6×6	2000	V	775.3s	758.8s
8×8	2000	V	722.4s	722.6s
4×4	4000	V	5655.5s	5688.0s
6×6	4000	V	5715.9s	5704.8s
8×8	4000	V	5587.2s	5538.8s

Table 3. *Performance (in seconds) of expert drivers for symmetric eigensolvers.*

$P_r \times P_c$	N	IBTYPE	JOBZ	PDSYGVX	PDSPGVX
2×2	400	1	N	1.7s	1.9s
2×2	400	2	N	1.6s	1.8s
2×2	400	3	N	1.6s	1.8s
2×2	1000	1	N	10.7s	12.5s
2×2	1000	2	N	10.2s	11.9s
2×2	1000	3	N	10.2s	11.9s
2×2	2000	1	N	57.0s	69.6s
2×2	2000	2	N	55.3s	64.8s
2×2	2000	3	N	55.5s	64.7s
4×4	1000	1	N	8.4s	9.7s
4×4	1000	2	N	7.8s	8.8s
4×4	1000	3	N	7.8s	8.8s
4×4	2000	1	N	31.5s	39.1s
4×4	2000	2	N	28.4s	34.0s
4×4	2000	3	N	28.1s	33.7s
4×4	4000	1	N	152.1s	212.0s
4×4	4000	2	N	139.0s	173.4s
4×4	4000	3	N	139.0s	174.3s
6×6	1000	1	N	40.1s	48.0s
6×6	1000	2	N	40.8s	43.4s
6×6	1000	3	N	26.1s	36.4s
6×6	2000	1	N	127.5s	138.4s
6×6	2000	2	N	113.2s	107.9s
6×6	2000	3	N	99.3s	108.6s
6×6	4000	1	N	347.9s	366.9s
6×6	4000	2	N	352.1s	369.5s
6×6	4000	3	N	292.4s	353.5s
8×8	1000	1	N	39.3s	55.0s
8×8	1000	2	N	42.5s	51.3s
8×8	1000	3	N	43.6s	53.9s
8×8	2000	1	N	132.3s	139.7s
8×8	2000	2	N	144.7s	151.4s
8×8	2000	3	N	148.3s	152.0s

Table 4. *Performance (in seconds) of expert drivers for generalized eigensolvers with option* JOBZ*='N'.*

$P_r \times P_c$	N	IBTYPE	JOBZ	PDSYGVX	PDSPGVX
2×2	1000	1	V	15.4s	16.6s
2×2	1000	2	V	14.8s	16.4s
2×2	1000	3	V	14.9s	16.5s
2×2	2000	1	V	85.7s	93.9s
2×2	2000	2	V	84.0s	93.5s
2×2	2000	3	V	84.7s	94.2s
2×2	4000	1	V	583.4s	634.5s
2×2	4000	2	V	580.4s	684.6s
2×2	4000	3	V	578.4s	684.5s
4×4	1000	1	V	10.6s	12.1s
4×4	1000	2	V	9.8s	11.1s
4×4	1000	3	V	9.9s	11.1s
4×4	2000	1	V	42.6s	50.0s
4×4	2000	2	V	39.5s	45.1s
4×4	2000	3	V	39.1s	45.5s
4×4	4000	1	V	216.8s	277.3s
4×4	4000	2	V	203.2s	239.3s
4×4	4000	3	V	203.0s	241.0s
6×6	1000	1	V	10.9s	12.2s
6×6	1000	2	V	10.9s	11.2s
6×6	1000	3	V	10.0s	11.3s
6×6	2000	1	V	36.5s	41.5s
6×6	2000	2	V	32.1s	39.2s
6×6	2000	3	V	33.2s	38.2s
6×6	4000	1	V	147.6s	178.4s
6×6	4000	2	V	136.2s	180.2s
6×6	4000	3	V	136.4s	183.2s
8×8	1000	1	V	18.4s	22.6s
8×8	1000	2	V	20.3s	24.6s
8×8	1000	3	V	18.6s	23.1s
8×8	2000	1	V	59.7s	68.7s
8×8	2000	2	V	54.8s	62.7s
8×8	2000	3	V	56.1s	61.5s

Table 5. *Performance (in seconds) of expert drivers for generalized eigensolvers with option JOBZ='V'.*

Bibliography

[1] B. ANDERSEN, F. GUSTAVSON, AND J. WASNIEWSKI, *A recursive formulation of cholesky factorization of a matrix in packed storage*, Tech. Report CS-00-441, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, 2000.

[2] 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 Users' Guide*, SIAM, second ed., 1995. Online version at `http://www.netlib.org/lapack/lug/lapack_lug.html`.

[3] L. S. BLACKFORD, J. CHOI, A. CLEARY, E. D'AZEVEDO, J. DEMMEL, I. DHILON, J. DONGARRA, S. HAMMARLING, G. HENRY, A. PETITET, K. STANLEY, D. WALKER, AND R. C. WHALEY, *ScaLAPACK Users' Guide*, SIAM, 1997. Online version at `http://www.netlib.org/scalapack/slug/scalapack_slug.html`.

[4] J. CHOI, J. DONGARRA, S. OSTROUCHOV, A. PETITET, D. WALKER, AND R. C. WHALEY, *A proposal for a set of parallel basic linear algebra subprograms*, Tech. Report CS-95-292, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, 1995. Also appears as LAPACK working note 100. Online version at `http://www.netlib.org/lapack/lawns/lawn100.ps`.

[5] E. D'AZEVEDO AND J. DONGARRA, *Packed storage extensions for scalapack*, Tech. Report CS-98-385, Computer Science Department, University of Tennessee, Knoxville, Tennessee, 1998.

[6] J. DEMMEL AND K. STANLEY, *The performance of finding eigenvalues and eigenvectors of dense symmetric matrices on distributed memory computers*, Tech. Report CS-94-254, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, 1994. Also appears as LAPACK working note 86. Online version at `http://www.netlib.org/lapack/lawns/lawn86.ps`.

[7] A. PETITET, *Algorithmic redistribution methods for block cyclic decompositions*, PhD thesis, University of Tennessee, Knoxville, Tennessee, 1996.