LAPACK Working Note 134
Department of Computer Science Technical Report CS-97-347

Packed storage extension for ScaLAPACK*

E. F. D'Azevedo
Mathematical Sciences Section
Oak Ridge National Laboratory
P.O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367

J. J. Dongarra,
Department of Computer Science
University of Tennessee
Knoxville, Tennessee 37996-1301

VERSION 1.6 ALPHA, January 1998

Abstract

We describe a new extension to ScaLAPACK [2] for computing with symmetric (Hermitian) matrices stored in a packed form. The new code is built upon the ScaLAPACK routines for full dense storage for a high degree of software reuse. The original ScaLAPACK stores a symmetric matrix as a full matrix but accesses only the lower or upper triangular part. The new code enables more efficient use of memory by storing only the lower or upper triangular part of a symmetric (Hermitian) matrix. The packed storage scheme distributes the matrix by block column panels. Within each panel, the matrix is stored as a regular ScaLAPACK matrix. This storage arrangement simplifies the subroutine interface and code reuse. Routines `PxPPTRF`/`PxPPTRS` implement the Cholesky factorization and solution for symmetric (Hermitian) linear systems in packed storage. Routines `PxSPEV`/`PxSPEVX` (`PxHPEV`/`PxHPEVX`) implement the computation of eigenvalues and eigenvectors for symmetric (Hermitian) matrices in packed storage. Routines `PxSPGVX (PxHPGVX)` implement the expert driver for the generalized eigenvalue problem for symmetric (Hermitian) matrices in packed storage. Routines `PFxSPGVX/PFxSPEVX (PFxHPGVX/PFxHPEVX)` uses the packed storage and perform out-of-core computation of eigenvectors. Performance results on the Intel Paragon suggest that the packed storage scheme incurs only a small time overhead over the full storage scheme.

*This work was supported in part by the National Science Foundation Grant No. ASC-9005933; by the Defense Advanced Research Projects Agency under contract DAAH04-95-1-0077, administered by the Army Research Office; by the Office of Scientific Computing, U.S. Department of Energy, under Contract DE-AC05-84OR21400; 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.

Contents

1 Introduction

This paper describes a new extension to ScaLAPACK [2] for computing with symmetric (Hermitian) matrices stored in a packed form. ScaLAPACK is an acronym for Scalable Linear Algebra PACKage, or Scalable LAPACK. ScaLAPACK is a library of high-performance linear algebra routines for distributed-memory message-passing MIMD (Multiple Instruction Multiple Data) computers and networks of workstations. Capability of ScaLAPACK is described in the ScaLAPACK Users' Guide [2].

The new code is built upon the ScaLAPACK routines for full dense storage for maximum portability. The original ScaLAPACK stores a symmetric matrix as a full matrix but accesses only the lower or upper triangular part. This design allows the reuse of Level 3 PBLAS (Parallel Basic Linear Algebra Subroutines) [3] without modification. However, almost half of the storage is holding redundant information. The new code enables more efficient use of memory by storing the submatrix blocks associated with only the lower or upper triangular part of a symmetric (Hermitian) matrix.

Although current computers have unprecedented storage and computation speed, they are also called upon to tackle ever larger problems. Let $N \times N$ be the largest symmetric (Hermitian) problem that can be stored in memory, then a larger approximately $\sqrt{2}N \times \sqrt{2}N$ symmetric matrix can be stored in the same memory using the packed storage scheme. Linear solution of symmetric (Hermitian) matrices by Cholesky factorization and computing eigenvalues and eigenvectors by the QR algorithm both have $O(N^3)$ complexities. With an $O(N^3)$ complexity, the runtime for solving the larger problem will be approximately $\sqrt{2}^3 \approx 2.8$ times longer.

A symmetric eigensolver for packed storage is adapted for use with out-of-core algorithms for solving large eigenvalue problems. The initial stage in the classical algorithm for finding eigenvalues and eigenvectors is to first reduce the original symmetric matrix into a tridiagonal matrix by orthogonal similarity Householder transformations. The original matrix is overwritten by these Householder transformations. One of the key steps is the frequent need for computing a matrix-vector multiply. An out-of-core algorithm that stores the symmetric matrix on disk would be highly inefficient since the matrix must be read in from disk for each matrix-vector multiply operation. A solution suggested by Ken Stanley is to hold in memory the symmetric matrix in packed storage and store the eigenvectors on disk. This approach would require $O(N^2/2)$ memory for the symmetric matrix in packed storage instead of $O(2N^2)$ memory for holding the symmetric matrix and eigenvectors in full storage, and would allow larger problems to be solved using the same limited amount of memory.

We have developed prototype codes `PxPPTRF/PxPPTRS` for Cholesky factorization and solution, and simple driver routines `PxSPEV` (`PxHPEV`) for finding eigenvalues and optionally eigenvectors of symmetric (Hermitian) matrices in packed storage. Expert drivers for symmetric (Hermitian) matrices `PxSPEVX` (`PxHPEVX`) and generalized eigenvalue problems `PxSPGVX` (`PxHPGVX`) are also available as prototype code. The out-of-core drivers [4] `PFxSPGVX/PFxSPEVX (PFxHPGVX/PFxHPEVX)` are based on the packed storage eigenvalue routines but stores the eigenvectors on disk. The names for the new routines follow the convention used in LAPACK [1] of using a 'P' to represent packed storage. Thus 'SY' ('HE') represents a symmetric (Hermitian) matrix and 'SP' ('HP') represents a symmetric (Hermitian) matrix in packed storage; similarly, 'PO' denotes a symmetric positive definite matrix and 'PP' denotes the symmetric positive definite matrix in packed storage.

Section 2 describes the layout of the packed storage scheme. Section 3 shows by a simple example how other ScaLAPACK routines can be modified for use with packed matrices. Section 4 summarizes the performance of `PDPPTRF/PDPPTRS`, `PDSPEV`, `PDSPEVX`, `PDSPGVX`, `PFDSPEVX` and `PFDSPGVX` on the Intel Paragon. Finally, Section 5 contains the summary.

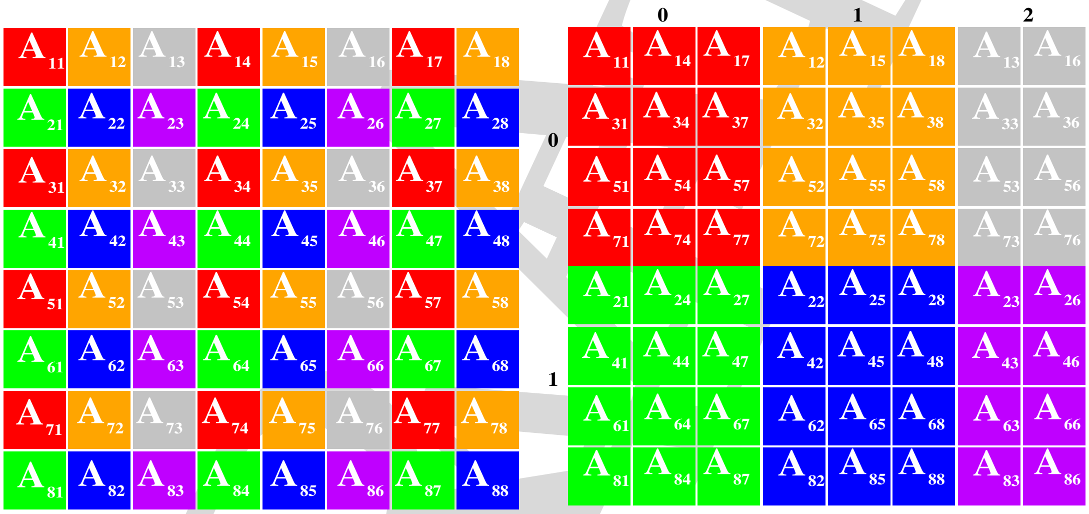
2 Data layout for packed storage

ScaLAPACK principally uses a two-dimensional block-cyclic data distribution (see Figure 1) for full dense in-core matrices [2, 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 $(p, q) = (\text{mod}(ib-1, P_r), \text{mod}(jb-1, P_c))$ 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. Moreover, each block column or panel in the packed storage scheme may be considered a full ScaLAPACK matrix distributed across only one process column. This treatment of a block 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.

2-DIMENSIONAL BLOCK CYCLIC DISTRIBUTION



Global (left) and distributed (right) views of matrix

Figure 1: Two-dimensional block-cyclic distribution.

3 Examples in the use of packed storage matrix

Here we illustrate by examples the reuse of ScaLAPACK library components for matrices stored in packed form. The key idea is the treatment of each block column or panel as a regular ScaLAPACK matrix distributed across a process column. The routine DESCINITT 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 descriptor. 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 )
```

The new code reuses ScaLAPACK PDLANSY and PDLANGE for computing the maximum entry in each block column panel.

The code traverses each block column (line 4) and calls DESCINITT to establish the appropriate matrix descriptor. It calls PDLANSY (line 11) to find the largest value in the diagonal block. Routine PDLANGE (line 19) 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. Moreover, the granularity of the algorithm is limited by the width of the column panel (NB=DESCA(NB_)).

```
N = DESCA(N_)         ! Number of columns in matrix A
NB = DESCA(NB_)       ! Width of each block column
ANRM = ZERO
DO JA=1,N,NB
   JB = MIN( NB, N-JA+1 )
   IA = JA
   CALL DESCINITT('Lower',IA,JA,DESCA,IAP,JAP,LOFFSET,DESCAP)
   !
   ! Handle diagonal block
   !
   ANRM2 = PDLANSY('M','Lower',JB,A(LOFFSET),IAP,JAP,DESCAP,WORK)
   ANRM = MAX( ANRM, ANRM2 )
   !
   ! Handle off-diagonal rectangular block
   ! Use Lower triangular part
   !
   IA = IA + JB
   IF (IA .LE. N) THEN
     ANRM2 = PDLANGE('M',N-IA+1,JB,A(LOFFSET),IAP+JB,JAP,DESCAP,WORK)
     ANRM = MAX( ANRM, ANRM2 )
   ENDIF
ENDDO
```

Figure 2: Example code to illustrate the reuse of ScaLAPACK components for matrices stored in packed storage.

4 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`. Out-of-core drivers [4] `PFxSPGVX/PFxSPEVX (PFxHPGVX/PFxHPEVX)` are based on the packed storage eigenvalue routines but stores the eigenvectors on disk.

We have compared the performance of the new routines in packed storage with ScaLAPACK routines in full storage. The goal is to demonstrate that the new version with packed storage has little or no overhead cost 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 XPS/35 Intel Paragon at the Center for Computational Sciences at the Oak Ridge National Laboratory. The XPS/35 has 512 GP nodes arranged in a 16×32 rectangular mesh. Each GP node has 32MBytes of memory. The runs were performed in a time-shared multi-user (non-dedicated) environment using a $P_r \times P_c$ *logical* process grid. Matrix block $mb = nb = 50$ was used for all tests. Results for upper case (`UPLO='U'`) and lower case (`UPLO='L'`) are very similar so results for only the lower case are presented. The latest version of PBLAS (version 2.0 alpha) was compiled with '-O3 -Mvect -Knoieee'† and linked with '-lkmath', the highly optimized CLASSPACK serial BLAS library. The new version of PBLAS incorporates automatic algorithmic blocking with block size set to 50‡. The PBLAS version 2.0 alpha release is still undergoing performance tuning.

Table 1 summarizes the times for the Cholesky factorization `PDPOTRF` for full storage and `PDPPTRF` for packed storage. The relative increase in runtime with packed storage over full storage is also displayed in the table. Routines `PDPPTRS` and `PDPOTRS` are used to solve the factored system with 50 and 1000 (`NRHS`) right-hand vectors. For the cases considered, the times for factorization by `PDPPTRF` with packed storage is comparable (at most two seconds difference) to times taken by `PDPOTRF` with full storage. Solution times for a narrow right-hand matrix (`NRHS`=50) show `PDPPTRS` for packed storage to be slower than `PDPOTRS`for full storage for large problems ($N \geq 2000$). The difference is about 3 seconds. Solution times for a wide right-hand matrix (`NRHS`=1000) show `PDPPTRS` for packed storage to be competitive with `PDPPTRS`. Routine `PDPPTRS` is slightly faster than `PDPOTRS` for cases $N = 1000$ and $N = 4000$, whereas for $N = 2000$, `PDPPTRS` is slower by 36%.

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 incurs at most a 11% increase over `PDSYEV` for full storage in finding eigenvalues only. 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

†Option -Knoieee turns off software emulation of IEEE arithmetic in divisions or operations on denormalized numbers to use the faster (but slightly less accurate) hardware units.

‡value return by routine `PILAENV` in PBLAS.

version may be limited by the width of the block column panel and by the block column by block column nature of the algorithm. When both eigenvectors and eigenvalues are required, `PDSPEV` compares 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'`) are 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, reorthoginalization is turned off by setting `ORFAC=0` and `ABSTOL=0` Performance analysis of `PDSYEVX` is described in [2, Chapter 5] and [5]. When only eigenvalues are requested (`JOBZ='N'`), `PDSPEVX` for packed storage is slower than `PDSYEVX` for full storage by 5 to 9 seconds. For longer running computation when both eigenvectors and eigenvalues are requested (`JOBZ='V'`), `PDSPEVX` for packed storage is comparable to `PDSYEVX` for full storage.

Table 4 summarizes 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 $U^H U$) and then overwriting A with

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

For the cases `IBTYPE=2` and `IBTYPE=3`, the packed version incurs a significant extra overhead 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.

Table 5 summarizes the times for the out-of-core eigensolvers `PFDSPEVX` with `RANGE='All'`. Routine `PFDSPEVX` is based on the packed storage `PDSPEVX` to compute groups of the eigenvectors using `RANGE='Interval'` and store them to disk. The results suggest the out-of-core version `PFDSPEVX` is almost competitive with `PDSYEVX`.

Table 6 summaries the times for the out-of-core eigensolver `PFDSPGVX` that calls `PFDSPEVX` to compute subsets of the eigenvectors. ScaLAPACK routine `PDSYGVX` calls `PDSYEVX` for the computation of eigenvectors, and performs a modification of eigenvectors by Cholesky factors of B. Unlike `PDSYGVX`, routine `PFDSPGVX` calls a routine similar to `PFDSPEVX` but also computes the modification of eigenvectors by B before they are stored to disk. If `PFDSPEVX` were used, there would be unnecessary input/output overhead to bring eigenvectors from disk to be modified by B and then write the vectors back out. However, even with this arrangement, the out-of-core packed routines still impose a significant overhead. For the

					NRHS=50			NRHS=1000		
$P_r \times P_c$	N	PDPOTRF	PDPPTRF	change	PDPOTRS	PDPPTRS	change	PDPOTRS	PDPPTRS	change
8×8	1000	2.3s	0.9s	-63%	1.6s	1.5s	-1%	4.1s	3.8s	-8%
8×8	2000	2.5s	2.9s	14%	1.3s	2.2s	73%	6.9s	8.5s	24%
8×8	4000	14.7s	13.9s	-5%	3.9s	7.6s	93%	62.6s	27.8s	-56%
10×10	1000	2.3s	0.8s	-63%	1.9s	1.6s	-20%	4.2s	3.2s	-25%
10×10	2000	2.0s	2.5s	21%	1.3s	2.1s	68%	4.8s	6.5s	36%
10×10	4000	11.3s	10.8s	-4%	4.1s	6.8s	67%	58.1s	20.2s	-65%

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

$P_r \times P_c$	N	JOBZ	PDSYEV	PDSPEV	Change
8×8	1000	N	25.3s	27.6s	9%
8×8	2000	N	81.4s	90.5s	11%
8×8	4000	N	317.0s	341.1s	8%
10×10	1000	N	25.4s	27.7s	9%
10×10	2000	N	79.0s	87.8s	11%
10×10	4000	N	304.2s	321.6s	6%
8×8	1000	V	64.6s	62.8s	-3%
8×8	2000	V	239.6s	226.8s	-5%
8×8	4000	V	1336.1s	1342.3s	0%
10×10	1000	V	65.3s	62.3s	-5%
10×10	2000	V	217.4s	221.7s	2%
10×10	4000	V	866.4s	843.2s	-3%

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

large problem `N=6000`, `PDSYGVX` did not complete in a reasonable amount of time due to excessive paging whereas `PFDSPGVX` completes in about 1500s since `PFDSPGVX` require only $O(N^2)$ instead of $O(3N^2)$ amount of memory.

5 Summary

The overall results suggest that for a reasonably large block size ($nb = 50$), the packed storage incurs only a small time overhead over the full storage routines. The difference may be as large as 20 seconds for short runs that complete in about a minute and approximately extra 10% overhead for larger problems. In some cases, the packed storage may even yield slightly better performance due to better data locality and cache reuse. The generalized eigensolver with `IBTYPE=2` or `IBTYPE=3` may require traversal across block rows and this leads to higher overhead (about 30% – 50%) for packed storage over full storage. The out-of-core generalized eigensolver incurs substantial time overhead (about 60% – 70%). A more efficient version is under development.

The design of the packed storage data layout to be a dense ScaLAPACK matrix in each block column panel also facilitates the reuse of PBLAS and ScaLAPACK library components for good performance.

$P_r \times P_c$	N	`JOBZ`	`PDSYEVX`	`PDSPEVX`	Change
8×8	1000	N	11.7s	15.2s	29%
8×8	2000	N	36.4s	44.9s	23%
8×8	4000	N	137.6s	157.2s	14%
8×8	1000	V	14.8s	18.2s	23%
8×8	2000	V	49.2s	57.8s	17%
8×8	4000	V	222.3s	228.1s	3%
10×10	1000	N	10.8s	14.3s	33%
10×10	2000	N	30.5s	38.8s	27%
10×10	4000	N	107.6s	126.6s	18%
10×10	1000	V	13.0s	16.6s	28%
10×10	2000	V	40.1s	48.4s	21%
10×10	4000	V	172.0s	176.9s	3%

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

$P_r \times P_c$	N	IBTYPE	JOBZ	PDSYGVX	PDSPGVX	Change
8×8	500	1	N	9s	8s	-11%
8×8	1000	1	N	15s	21s	34%
8×8	2000	1	N	54s	70s	30%
8×8	500	2	N	6s	7s	15%
8×8	1000	2	N	15s	21s	38%
8×8	2000	2	N	48s	75s	55%
8×8	500	3	N	5s	7s	31%
8×8	1000	3	N	15s	21s	39%
8×8	2000	3	N	48s	75s	55%
10×10	500	1	N	10s	8s	-16%
10×10	1000	1	N	14s	20s	39%
10×10	2000	1	N	47s	62s	32%
10×10	500	2	N	6s	7s	14%
10×10	1000	2	N	13s	19s	41%
10×10	2000	2	N	40s	65s	62%
10×10	500	3	N	5s	7s	36%
10×10	1000	3	N	13s	19s	44%
10×10	2000	3	N	40s	65s	61%
8×8	500	1	V	12s	10s	-19%
8×8	1000	1	V	21s	25s	18%
8×8	2000	1	V	76s	89s	16%
8×8	500	2	V	7s	8s	25%
8×8	1000	2	V	19s	25s	33%
8×8	2000	2	V	66s	94s	42%
8×8	500	3	V	6s	8s	29%
8×8	1000	3	V	18s	25s	34%
8×8	2000	3	V	65s	93s	44%
10×10	500	1	V	13s	10s	-25%
10×10	1000	1	V	18s	23s	27%
10×10	2000	1	V	64s	76s	19%
10×10	500	2	V	6s	8s	28%
10×10	1000	2	V	16s	22s	38%
10×10	2000	2	V	53s	79s	49%
10×10	500	3	V	6s	8s	34%
10×10	1000	3	V	16s	22s	39%
10×10	2000	3	V	52s	78s	51%

Table 4: Performance (in seconds) of expert drivers for generalized eigensolvers.

$P_r \times P_c$	N	JOBZ	PDSYEVX	PFDSPEVX	Change
8×8	1000	V	21	21	-1%
8×8	2000	V	55	63	15%
8×8	4000	V	220	260	18%
8×8	6000	V	1123	1056	-6%

Table 5: Performance (in seconds) of expert drivers for out-of-core symmetric eigensolvers.

$P_r \times P_c$	N	IBTYPE	JOBZ	PDSYGVX	PFDSPGVX	Change
8×8	1000	1	V	21s	37s	77%
8×8	2000	1	V	73s	99s	35%
8×8	4000	1	V	336s	444s	32%
8×8	6000	1	V	N/A[a]	1511s	N/A
8×8	1000	2	V	19s	27s	43%
8×8	2000	2	V	66s	98s	49%
8×8	4000	2	V	296s	487s	64%
8×8	6000	2	V	N/A	1503s	N/A
8×8	1000	3	V	19s	27s	42%
8×8	2000	3	V	65s	100s	54%
8×8	4000	3	V	292s	487s	67%
8×8	6000	3	V	N/A	1500s	N/A

Table 6: Performance (in seconds) of expert drivers for out-of-core generalized eigensolvers.

[a]Did not complete due to excessive paging.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, SIAM, second ed., 1995. Online version at http://www.netlib.org/lapack/lug/lapack_lug.html.

[2] 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.

[3] 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. Rep. 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.

[4] E. D'Azevedo and J. Dongarra, *The design and implementation of parallel out-of-core ScaLAPACK LU, QR and Cholesky factorization routines*, Tech. Rep. ORNL/TM-13372, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1997. Also available as LAPACK working note 118. Online version at http://www.netlib.org/lapack/lawns/lawn118.ps.

[5] J. Demmel and K. Stanley, *The performance of finding eigenvalues and eigenvectors of dense symmetric matrices on distributed memory computers*, Tech. Rep. 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.