Packed Storage Extension for ScaLAPACK

Eduardo D'Azevedo
Computer Science and Mathematics Division
Oak Ridge National Laboratory

Jack Dongarra
Computer Science Department
University of Tennessee

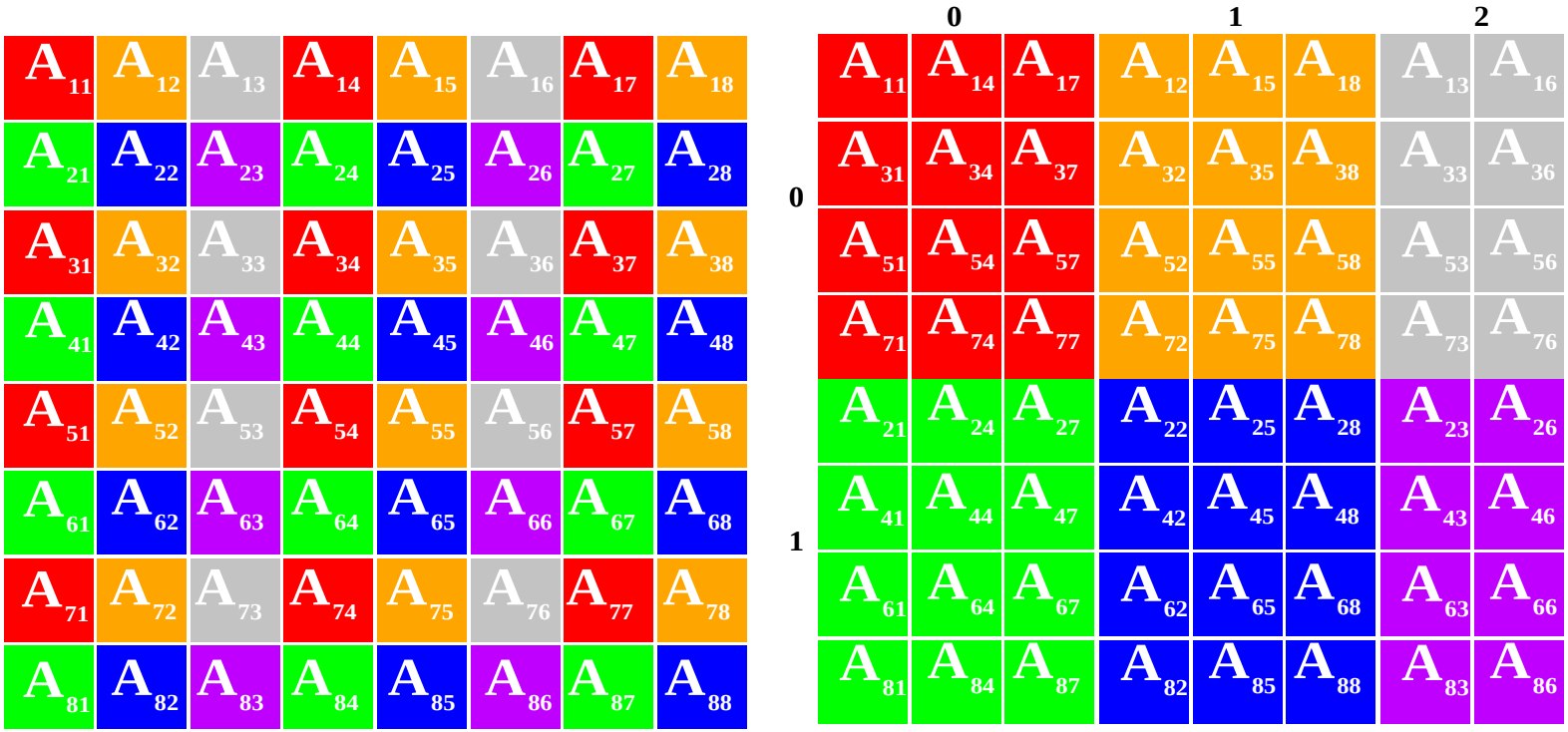
Packed Storage

- LAPACK has support for compact storage for symmetric matrices where only lower or upper triangular part is stored.
- Extension of ScaLAPACK to support compact storage:
  - Cholesky factorization: PxPPTRF,PxPPTRS
  - Eigen solver: PxSPEV, PxSPEVX
  - Generalized Eigen solver: PxSPGVX
  - PBLAS: PxTPMM, PxTPSM, PxSPRK, PxSPR2K

Data Layout

- ScaLAPACK uses 2D block cyclic matrix distribution
- Data is organized as $(\texttt{MB} \times \texttt{NB})$ blocks on $(P_r \times P_c)$ processor grid.
- Matrix entry (i, j) is mapped to matrix block $(ib, jb) = (1 + \lfloor (i-1)/\texttt{MB} \rfloor, 1 + \lfloor (j-1)/\texttt{NB} \rfloor)$.
- (ib, jb) block is assigned to processors $(0..P_r - 1, 0..P_c - 1)$, $(p, q) = (\mathrm{mod}(ib - 1, P_r), \mathrm{mod}(jb - 1, P_c))$
- Compact storage has blocks in triangular part on same processors to reuse PBLAS and ScaLAPACK routines.

2-DIMENSIONAL BLOCK CYCLIC DISTRIBUTION



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

Approach

- Treat NB wide block column as regular ScaLAPACK matrix.
- Create new descriptor with appropriate adjustment of array offset.
- Treat NB $\times P_c$ wide block column (trapezoidal shaped) as regular ScaLAPACK matrix.
- Wider panel for larger granularity and higher efficiency.

Example

- Get (IA, JA) entry in full storage:

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

- (IAP, JAP) are new index into "fake" ScaLAPACK matrix with descriptor DESCAP,

```
CALL DESCINITT('Lower', IA, JA, DESCA, &

IAP, JAP, LOFFSET, DESCAP)
CALL PDELGET(SCOPE, TOP, ALPHA, &
             A(LOFFSET), IAP, JAP, DESCAP)
```

- Each processor may get a different value for LOFFSET.

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

Performance critical PBLAS

- Cholesky factorization performs updates to right-looking part, need efficient rank update for compact storage.
- PxSPRK performs $C \leftarrow \beta C + \alpha A A^t$
- PxSPR2K performs $C \leftarrow \beta C + \alpha A B^t + \alpha B A^t$
- A,B are NB columns wide, C in packed storage.
- Very high communication cost if performed by looping over NB columns in C with PBLAS.

PxSPRK, PxSPR2K

- Copy data into row (column) replicated vectors A_r (A_c) aligned to submatrix to be updated.
- New feature in PBLAS V2 that support replicated vectors using `DESCA(CSRC_) = -1` or `DESCA( RSRC_) = -1`.
- PBLAS `PxGEADD` (copy with add) , `PxTRAN` (transpose copy) handles the row (column) broadcast and no further communication is needed.

PxTPMM, PxTPSM

- Multiply with compact triangular matrix PxTPMM:

$$B \leftarrow \alpha op(A)B, \quad B \leftarrow Bop(A)$$

- Solve with compact triangular matrix PxTPSM:

$$B \leftarrow \alpha op(A^{-1})B, \quad B \leftarrow \alpha Bop(A^{-1})$$

- Consider copying NB $*$ P_c triangular matrix to dense storage to use regular PxTRSM or PxTRMM.

DOT vs AXPY

- Solve (1) by (1) $x_3 = L_{33}^t \, b_3$, (2) $x_2 = L_{22}^t \, (b_2 - L_{32}^t x_3)$, (3) $x_1 = L_{11}^t \, (b_1 - \boxed{(L_{21}^t x_2 + L_{31}^t x_3)})$ uses "dot product" with no copy necessary.

$$\begin{pmatrix} L_{11}^t & L_{21}^t & L_{31}^t \\ & L_{22}^t & L_{32}^t \\ & & L_{33}^t \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \qquad (1)$$

- "axpy" operation require copying (L_{31}, L_{32}) block row into temporary storage. Step (2) is

$$\begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \leftarrow \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} - \boxed{\begin{pmatrix} L_{31}^t \\ L_{32}^t \end{pmatrix} x_3}$$

Beowulf cluster

- High TORC (Tennessee Oak Ridge Cluster Project) Linux cluster at ORNL (`www.epm.ornl.gov/torc`)
- 64 nodes, each node is a dual 450Mhz Pentium-II with 512MB, 8GB IDE disk, with switched gigabit ethernet connection.
- Node allocation controlled by PBS (Portable Batch System).
- LAM/MPI 6.3.2 with 2 MPI tasks per node
- ATLAS BLAS library capable of about 300Mflops on a single cpu.
- `MB = NB = 50`

PxSPGVX

- Parameter IBTYPE determines the type of problem

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

- Both A and B are in compact storage.
- Cholesky factorization of B and in-place modification of A to canonical form:

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

Cholesky factorization

- Factorization slower by about 40% on large problem.
- Triangular solves are slower for `NRHS=50` and about the same for `NRHS=1000`.

				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.

Eigen solver

- Similar performance between PDSYEV and PDSPEV.

$P_r \times P_c$	N	JOBZ	PDSYEV	PDSPEV
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	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.

Generalized Eigen Solver

- Compact version slower by 20-30%.

$P_r \times P_c$	N	IBTYPE	JOBZ	PDSYGVX	PDSPGVX
2×2	4000	1	V	583.4s	634.5s
4×4	4000	1	V	216.8s	277.3s
6×6	4000	1	V	147.6s	178.4s
2×2	4000	2	V	580.4s	684.6s
4×4	4000	2	V	203.2s	239.3s
6×6	4000	2	V	136.2s	180.2s
2×2	4000	3	V	578.4s	684.5s
4×4	4000	3	V	203.0s	241.0s
6×6	4000	3	V	136.4s	183.2s

Table 3: Performance (in seconds) of expert drivers for generalized eigensolvers with option JOBZ='V'.

Generalize Eigen Solver

$P_r \times P_c$	N	IBTYPE	JOBZ	PDSYGVX	PDSPGVX
2×2	2000	1	N	57.0s	69.6s
4×4	2000	1	N	31.5s	39.1s
6×6	2000	1	N	127.5s	138.4s
8×8	2000	1	N	132.3s	139.7s
2×2	2000	2	N	55.3s	64.8s
4×4	2000	2	N	28.4s	34.0s
6×6	2000	2	N	113.2s	107.9s
8×8	2000	2	N	144.7s	151.4s
2×2	2000	3	N	55.5s	64.7s
4×4	2000	3	N	28.1s	33.7s
6×6	2000	3	N	99.3s	108.6s
8×8	2000	3	N	148.3s	152.0s

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