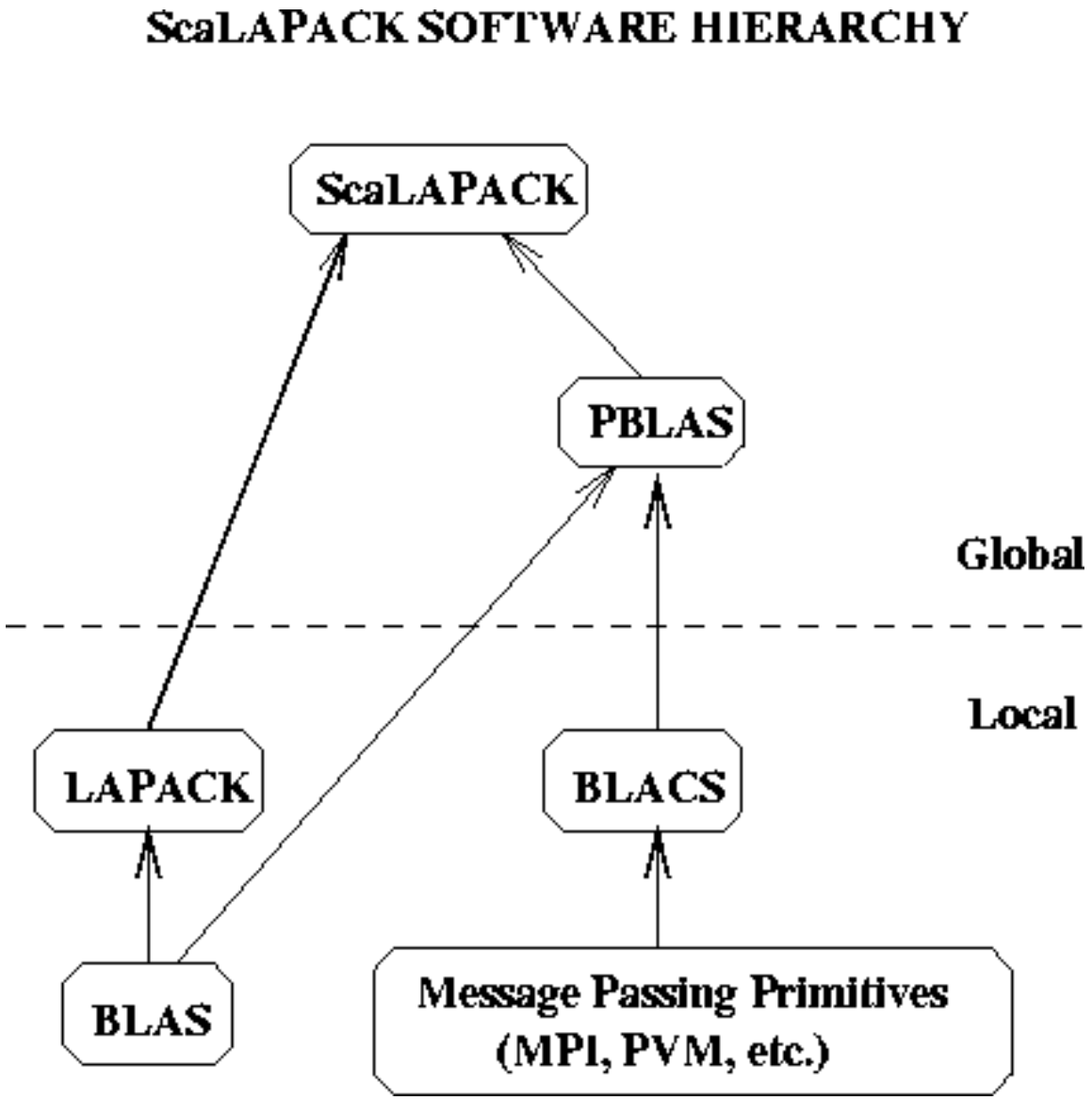
Exercises 7:
Dense Linear Algebra Libraries

Today

- Look at and understand sample code calling ScaLAPACK functions
- Understand how distribution of the data matters

ScaLAPACK Parallel Library



PBLAS

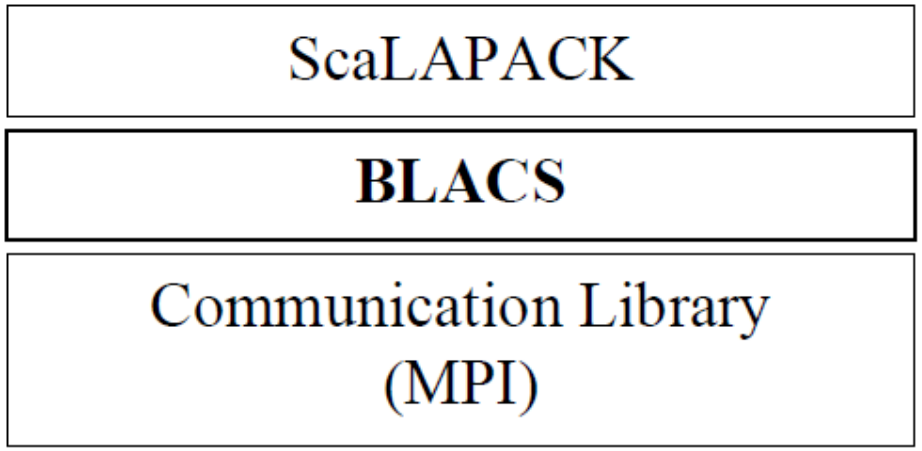
- Parallel Basic Linear Algebra Subroutines
- Level 1: vector - vector operations
- Level 2: matrix - vector operations
- Level 3: matrix - matrix operations

ScaLAPACK

- Matrix decompositions
- Solving linear systems of equations
- Eigenvalue equations
- Linear Least Squares
- For dense, banded, triangular, general, real, complex, etc.

BLACS

- (**B**asic **L**inear **A**lgebra **C**ommunication **S**ubprograms)
- The BLACS project is an ongoing investigation whose purpose is to create a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms



Structure of a code using ScaLAPACK

1. initialize the BLACS context
2. create a BLACS processor grid
3. decompose a global array in local sub-arrays,
   $\rightarrow$ each processor assembles the local sub-array
4. call the desired ScaLAPACK routine
5. release the BLACS context

Calling ScaLAPACK routines

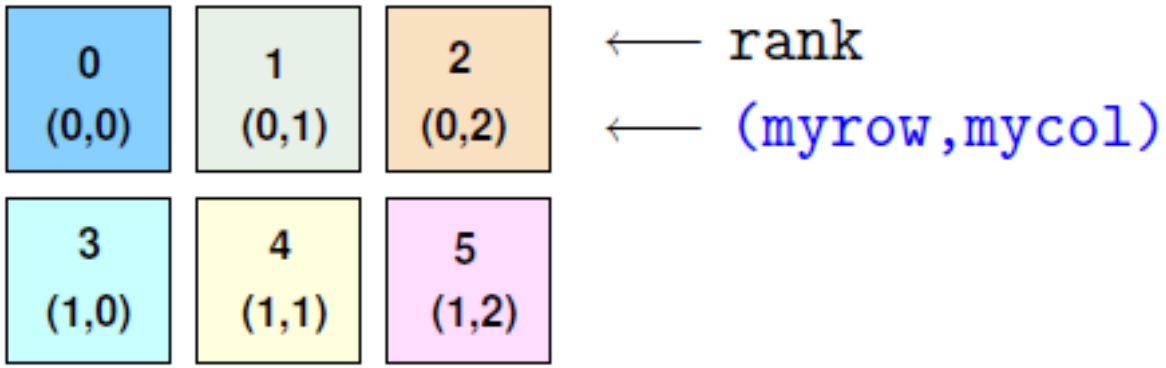
- It's the responsibility of the programmer to correctly distribute a global matrix before calling ScaLAPACK routines
- ScaLAPACK routines are written using a message passing paradigm, therefore each subroutine access directly ONLY local data
- Each process of a given CONTEXT must call the same ScaLAPACK routine...
  - ... providing in input its local portion of the global matrix

BLACS startup

```
//Returns the number of processes available for use
Cblacs_pinfo( &myrank_mpi, &nprocs_mpi ) ;

 //Assigns available processes to BLACS process grid
Cblacs_gridinit( &ictxt, "Row", nprow, npcol );

//Returns information on the current grid - my rank's Cartesian indices
Cblacs_gridinfo( ictxt, &nprow, &nprow, &myrow, &mycol );
```



Block-cyclic Array Distribution

Example of 5×5 array:

A11	A12	A13	A14	A15
A21	A22	A23	A24	A25
A31	A32	A33	A34	A35
A41	A42	A43	A44	A45
A51	A52	A53	A54	A55

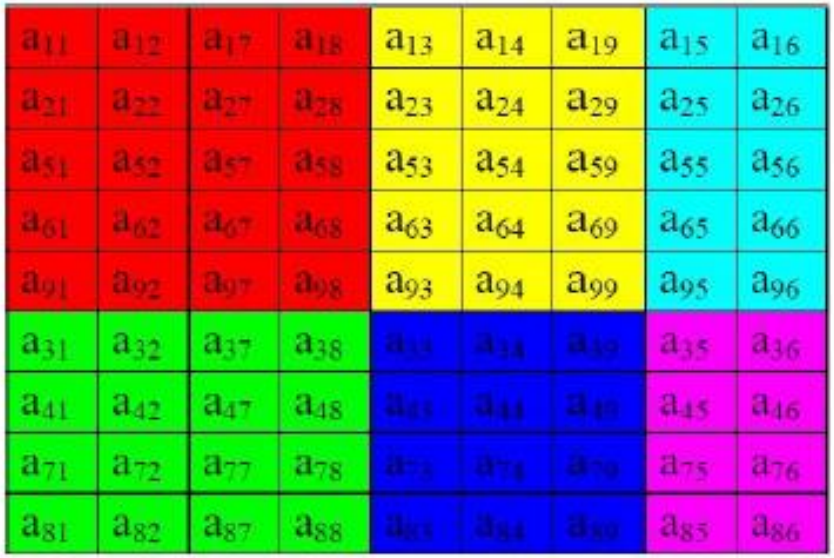
$\Rightarrow$

2×3 processor grid, 2×2 blocks

- need to account for different local sizes

Another example

Logical View (Matrix)



Local View (CPUs)

ScaLAPACK naming scheme

- The ScaLAPACK routine names follow a simple scheme. Each name has the structure PXYYZZZ, where the components have various meanings.
- X the second letter indicates the data type (real or complex) and precision
  - D real, double precision (in Fortran, DOUBLE PRECISION)
  - Z complex, double precision (in Fortran, COMPLEX*16)
  - (for real and complex, single precision, S and C respectively)
- YY the third and fourth letters indicate the type of the matrix A
  - GE general matrix
  - PO symmetric or Hermitian positive-definite dense matrix
  - TR triangular matrix
  - PT symmetric or Hermitian positive-definite tridiagonal matrix
  - PB symmetric or Hermitian positive-definite banded matrix
- ZZZ the last three letters indicate the computation performed
  - TRF triangular factorization
  - TRS solution of linear equations, using

PDGETRF

```
void pdgetrf (MKL_INT *m , MKL_INT *n , double *a , MK
L_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *ip
iv , MKL_INT *info );
```

The pdgetrf function forms the LU factorization of a general m-by-n distributed matrix sub(A) = A(ia:ia+m-1, ja:ja+n-1) as

A = P*L*U

where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if m>n) and U is upper triangular (upper trapezoidal if m < n). L and U are stored in sub(A).

The function uses partial pivoting, with row interchanges.

https://software.intel.com/en-us/mkl-developer-reference-c-p-getrf

PDGETRF Input Parameters

m	(global) The number of rows in the distributed matrix sub(*A*); *m*≥0.
n	(global) The number of columns in the distributed matrix sub(*A*); *n*≥0.
a	(local) Pointer into the local memory to an array of local size *lld_a***LOCc*(*ja*+*n*-1). Contains the local pieces of the distributed matrix sub(*A*) to be factored.
ia, *ja*	(global) The row and column indices in the global matrix *A* indicating the first row and the first column of the matrix sub(*A*), respectively.
desca	(global and local) array of size *dlen_*. The array descriptor for the distributed matrix *A*.

https://software.intel.com/en-us/mkl-developer-reference-c-p-getrf

PDGETRF Output Parameters

a	Overwritten by local pieces of the factors *L* and *U* from the factorization *A* = *P*L*U*. The unit diagonal elements of *L* are not stored.
ipiv	(local) Array of size *LOCr(m_a)* + *mb_a*. Contains the pivoting information: local row *i* was interchanged with global row *ipiv*[*i*-1]. This array is tied to the distributed matrix *A*.
info	(global) If *info*=0, the execution is successful. *info* < 0: if the *i*-th argument is an array and the *j*–th entry, indexed *j* - 1, had an illegal value, then *info* = -(*i**100+*j*); if the *i*–th argument is a scalar and had an illegal value, then *info* = –*i*. If *info* = *i* > 0, $u_{ia+i,\, ja+j-1}$ is 0. The factorization has been completed, but the factor *U* is exactly singular. Division by zero will occur if you use the factor *U* for solving a system of linear equations.

https://software.intel.com/en-us/mkl-developer-reference-c-p-getrf

PDGETRS

```
void pdgetrs (char *trans , MKL_INT *n , MKL_INT *nrhs , double
 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv,
double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb, MKL_INT
 *info );
```

The pdgetrs function solves a system of distributed linear equations with a general *n*-by-*n* distributed matrix sub(A) = A(*ia*:*ia*+*n*-1, *ja*:*ja*+*n*-1) using the LU factorization computed by pdgetrf.

The system has one of the following forms specified by *trans*:

sub(A)*X = sub(B) (no transpose),

sub(A)^T*X = sub(B) (transpose),

sub(A)^H*X = sub(B) (conjugate transpose),

where sub(B) = B(*ib*:*ib*+*n*-1, *jb*:*jb*+*nrhs*-1).

Before calling this function, you must call pdgetrf to compute the LU factorization of sub(A).

https://software.intel.com/en-us/mkl-developer-reference-c-p-getrs

PDGESV

- Computes the solution to the system of linear equations with a square distributed matrix and multiple right-hand sides.

  - `call PDGESV(m,nrhs,aloc,ia,ja,idesca,iwork, bloc,ib,jb,idescb,ierr)`
  - `m` – global array dimension (square matrix)
  - `nrhs` – number of r.h.s. vectors to be solved for
  - `aloc` – address of local array
  - `ia,ja,ib,jb` – start index of global matrix (in general 1)
  - `idesca` – array descriptor of system matrix
  - `iwork` – work array for pivoting
  - `bloc` – address of local r.h.s. vector (in/output: result)
  - `idescb` – array descriptor of r.h.s. vector

https://software.intel.com/en-us/mkl-developer-reference-c-p-gesv

The array descriptor

- The Descriptor is an integer array that stores the information required to establish the mapping between each global array entry and its corresponding process and memory location.
- Each matrix MUST be associated with a Descriptor.

Constructing an array descriptor

- `call DESCINIT(desc,m,n,mb,nb,rsrc,csrc,icontxt, lld,ierr)`
- `desc(1:9)` – array descriptor (integer, output)
- `m,n` – global array dimensions
- `mb,nb` – block size
- `rsrc,csrc` – processor row/column getting the first block (in general: `rsrc=csrc=0`)
- `icontxt` – BLACS context
- `lld` – leading dimension of local array

→ `desc` is used in calls to ScaLAPACK functions

What are the local array dimensions?

ScaLAPACK utility function:

```
mloc=NUMROC(m,mb,my_row,rsrc,nprow)
nloc=NUMROC(n,nb,my_col,csrc,npcol)
```

- `m` – global dimension in first direction
- `mb` – block size in first direction
- `my_row` – calling process' row index
- `rsrc` – source process for "dealing out" blocks per row
- `nprow` – number of processes per row

I/O

ScaLAPACK utility function (global call):

```
call PDLAWRITE(file,m,n,aloc,ia,ja,idesca,
               irwrite,icwrite, work)
```

- `file` – filename (character) for output
- `m,n` – global matrix dimension
- `aloc` – address of local storage
- `ia,ja` – first indices of sub-matrix (in general `1,1`)
- `idesca` – arrray descriptor
- `irwrite,icwrite` – processor grid indices of process which performs i/o
- `work` – work array (size $\geq$ `mb`)

$\rightarrow$ writes global matrix to file

Terminating ScaLAPACK

- Release BLACS grid context

```
Cblacs_gridexit(ictxt)
```

- Terminate all communication

```
Cblacs_exit(0)
```

  - Similar to MPI_Finalize()

```
Cblacs_exit(1)
```

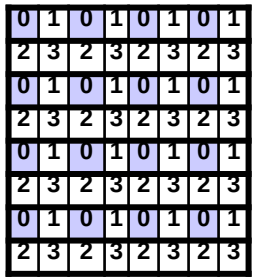
  - exit BLACS but continue using MPI

Resources

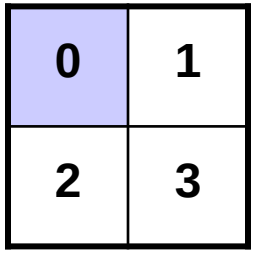
- BLAS quick guide: http://www.netlib.org/lapack/lug/node145.html
- LAPACK quick guide: http://www.netlib.org/lapack/lug/node142.html
- BLACS quick guide: http://www.netlib.org/blacs/BLACS/QRef.html
- PBLAS quick guide: http://www.netlib.org/scalapack/slug/node184.html
- ScaLAPACK user guide: http://www.netlib.org/scalapack/slug/

Example: Choosing the Block Size

- Get the file LUex.c from Moodle and put it in your local directory on the cluster
- Generates an NxN matrix distributed amongst 4 processes with block size nb
  - Uses a 2x2 processor grid - must call with 4 processes
- nb = 1 corresponds to 2D cyclic layout



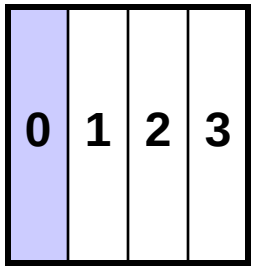
- nb = N/2 corresponds to 2D blocked layout



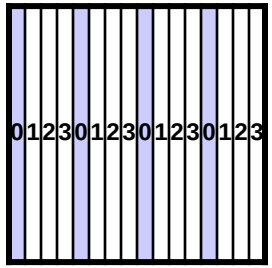
- Something in between will be best

Different Data Layouts for Parallel GE

Bad load balance:
P0 idle after first
n/4 steps



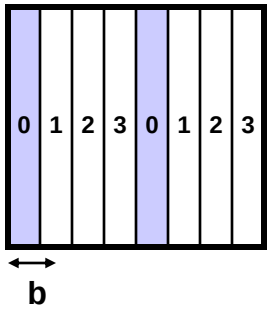
1) 1D Column Blocked Layout



Load balanced, but can't
easily use BLAS3

2) 1D Column Cyclic Layout

Can trade load balance
and BLAS3
performance by
choosing b, but
factorization of block
column is a bottleneck



3) 1D Column Block Cyclic Layout

0	1	2	3
3	0	1	2
2	3	0	1
1	2	3	0

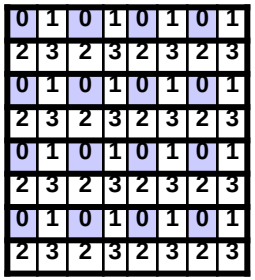
Complicated addressing,
May not want full parallelism
In each column, row

4) Block Skewed Layout

Bad load balance:
P0 idle after first
n/2 steps

0	1
2	3

5) 2D Row and Column Blocked Layout



The winner!

6) 2D Row and Column
Block Cyclic Layout

Setup on cluster

- Start an interactive session with 4 processors (or use a batch script if you prefer):

```
srun -n 4 -p express3 --pty /bin/bash -i
```

- Load MPI and ScaLAPACK libraries:

```
module load openmpi
module load scalapack
```

- compile the file

```
mpicc -o LUex LUex.c -lscalapack –lm
```

- To run:

```
mpirun -n 4 ./LUex N nb
```

Task: Performance vs. Block Size

- Run the code with N = 8192
- Try nb = 2^i for i = 0,...,12
  - Make a plot of LU factorization time versus block size

Questions to consider:

- What happens at the extremes (nb = 1, 4096)?
- What block size is best?

Example of how your plot should look