Scientific Computing on Parallel Machines

Cálculo científico en ordenadores paralelos

Markus Uhlmann

CIEMAT

www.ciemat.es/sweb/comfos/personal/uhlmann

UCM – Abril 2008

Schedule

Part I	Introduction to parallel programming	lecture 1
Part II	Introduction to MPI	
	General introduction & MPI program structure	lecture 2
	Point-to-point communication	lecture 3,4
	Collective communication	lecture 5
	2D Poisson example	lecture 6,7
	Non-contiguous data & Mixed datatypes	lecture 7
	Virtual topologies & Communication subsets	lecture 8
	Use of linear algebra libraries	lecture 9
	Extensions	lecture 10

Part II Introduction to MPI

What is ScaLAPACK?

Scalable Linear Algebra PACKage

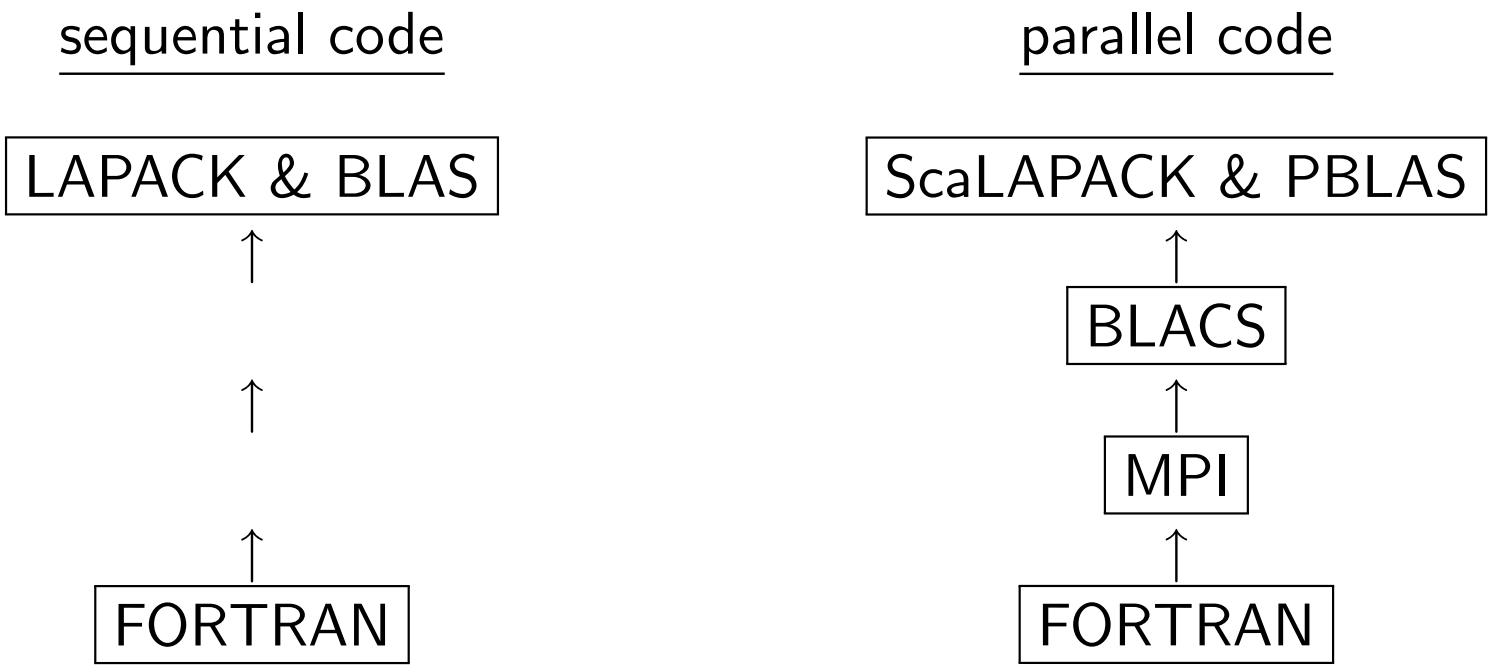
- analogous (but not completely) to LAPACK
- provides linear algebra operations:
  - linear system solution
  - eigenanalysis
  - singular value decomposition
  - ...
- distributed memory, based upon MPI
- freely available (e.g. from netlib)

Linear system solvers included in ScaLAPACK

Type of matrix and storage scheme	Operation	Single precision		Double precision	
		Real	Complex	Real	Complex
general (partial pivoting)	simple driver expert driver	PSGESV PSGESVX	PCGESV PCGESVX	PDGESV PDGESVX	PZGESV PZGESVX
general band (partial pivoting)	simple driver	PSGBSV	PCGBSV	PDGBSV	PZGBSV
general band (no pivoting)	simple driver	PSDBSV	PCDBSV	PDDBSV	PZDBSV
general tridiagonal (no pivoting)	simple driver	PSDTSV	PCDTSV	PDDTSV	PZDTSV
symmetric/Hermitian positive definite	simple driver expert driver	PSPOSV PSPOSVX	PCPOSV PCPOSVX	PDPOSV PDPOSVX	PZPOSV PZPOSVX
symmetric/Hermitian positive definite band	simple driver	PSPBSV	PCPBSV	PDPBSV	PZPBSV
symmetric/Hermitian positive definite tridiagonal	simple driver	PSPTSV	PCPTSV	PDPTSV	PZPTSV

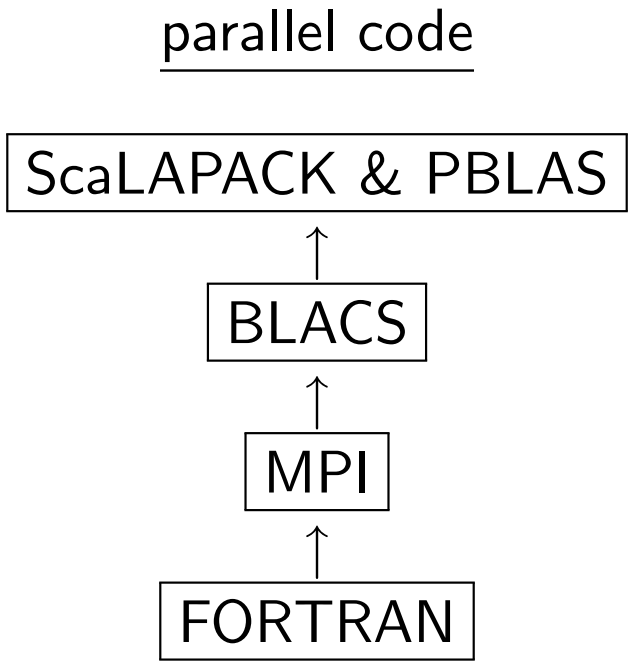
Hierarchy of libraries

Elements needed for linear algebra functionality



ScaLAPACK components

- ScaLAPACK
  - provides higher-level linear algebra functions
- PBLAS
  - provides basic linear algebra functions
- BLACS
  - provides basic communication functions
  - hides most of MPI from user



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,
   $\longrightarrow$ each processor assembles the local sub-array
4. call the desired ScaLAPACK routine
5. release the BLACS context

BLACS start-up

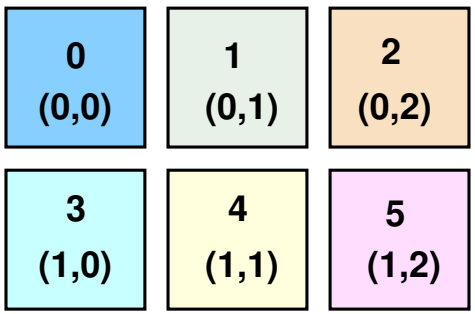
Initializing an nprow × npcol processor grid

- `call SL_INIT(icontxt,nprow,npcol)`

→ returns a BLACS context (similar to MPI communicator)

Query function

- `call BLACS_GRIDINFO(icontxt,nprow,npcol, myrow,mycol)`
- returns calling processor's Cartesian indices: `myrow`, `mycol`

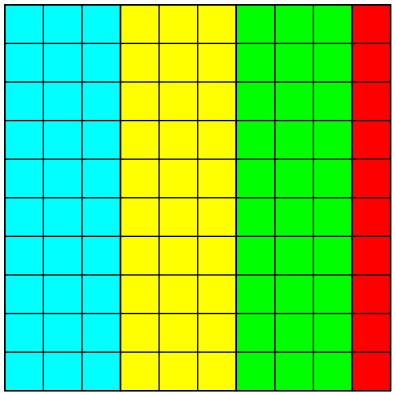


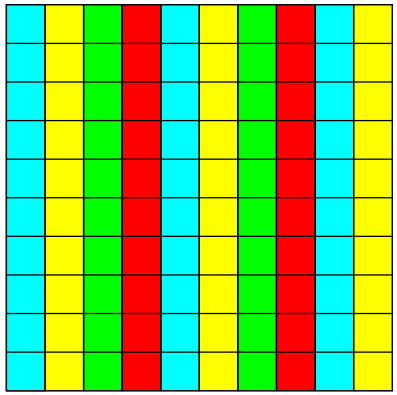
← rank

← (myrow,mycol)

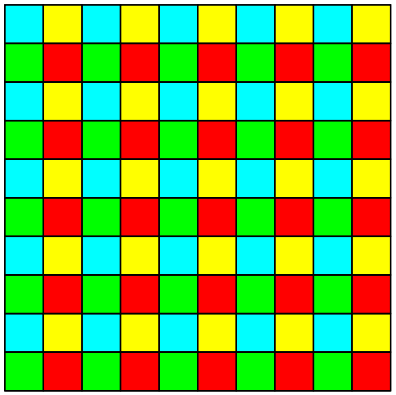
ScaLAPACK's Block-cyclic array distribution

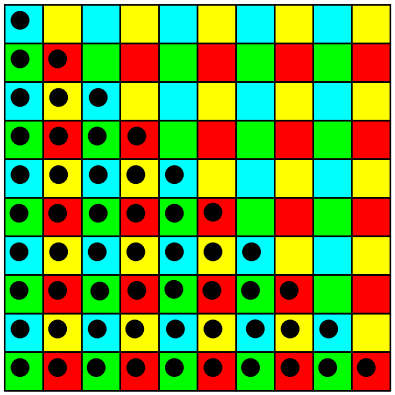
- 1d block distribution vs. 1d block-cyclic distribution

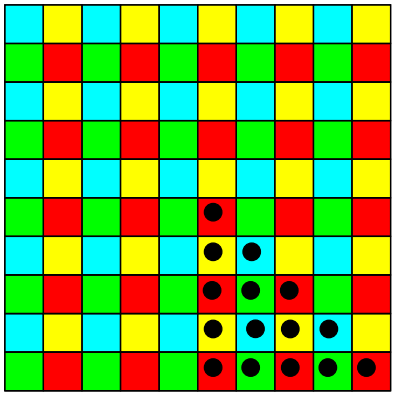




- 2d block-cyclic distribution







$\Rightarrow$ block-cyclic allows for optimal load-balancing

ScaLAPACK's Block-cyclic array distribution (2)

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

Array 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

Linear system solve with LAPACK

LU decomposition example – LAPACK

- general matrix, double precision

```
call assign_matrix(a,m,m)                    !user function
call assign_vector(b,m,1)                    !user function
call DGESV(m,1,a,m,iwork,b,m,ierr)
```

- returns solution to $A\,x = b$ in vector b

Linear system solve with ScaLAPACK

```
c     use 2*2=4 processors
      nprow=2
      npcol=2
      call SL_INIT(ictxt,nprow,npcol)
      call BLACS_GRIDINFO(ictxt,nprow_0,npcol_0,
     &                    my_row,my_col)
c     descriptor for matrix a
      call DESCINIT(idesca,m,m,mb,nb,0,0,ictxt,m,ierr)
c     descriptor for rhs vector b
      call DESCINIT(idescb,m,1,mb,nb,0,0,ictxt,m,ierr)
c     assign the local data
      call assign_matrix(aloc,idesca,my_row,my_col)
      call assign_vector(bloc,idescb,my_row,my_col)
c     solve the sytem
      call PDGESV(m,1,aloc,1,1,idesca,iwork,
     &            bloc,1,1,idescb,ierr)
```

Linear system solve with ScaLAPACK

Syntax of PDGESV

- `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

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

→ dynamically allocate with `(mloc,nloc)` or compare to static allocation size

Terminating ScaLAPACK

Release a BLACS grid context:

- `call BLACS_GRIDEXIT(ictxt)`

→ similar to deleting a Cartesian MPI communicator

Terminate all communication:

- `call BLACS_EXIT(0)`

→ similar to `MPI_FINALIZE`

- `call BLACS_EXIT(1)`

→ exit BLACS, but can still continue using MPI

I/O of distributed matrices

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

Strategies for parallelizing your algorithm

Alternatives:

- each processor assembles local part of system
  (the way to go)
- one master processor assembles the entire system, then distributes the data
  (not the best idea)
- all processors assemble the entire system redundantly
  (a good way to get started quickly)

$\longrightarrow$ then solve by call to ScaLAPACK

Obtaining more information on ScaLAPACK

Resources:

- ScaLAPACK user guide
- BLACS information pages on netlib
- NCSA online MPI course (chapter 10)

ScaLAPACK exercise

Problem:

- solve: $A\,x = b$, with $b_i = 207 - i$, $A_{i,j} = \begin{bmatrix} 10000 & i = j \\ i + j/2 & else \end{bmatrix}$

$\rightsquigarrow$ find the bug(s) in this source!
(linear system solver code – bugged)

- you will need the following in order to compile:
  - makefile for GNU/Linux
  - LAPACK library
  - BLAS library
- local storage scheme for block-cyclic mapping explained:
(ScaLAPACK user guide)
- reference to ScaLAPACK routine PDGESV
- reference to LAPACK routine DGESV

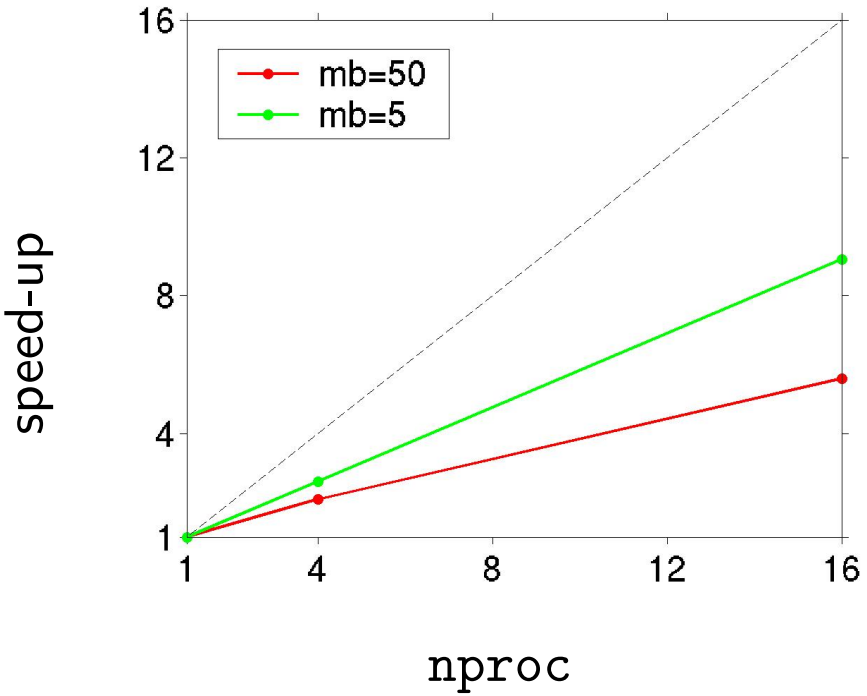
ScaLAPACK exercise: result

Size 10×10:

1	0.02052855547
2	0.02041159784
3	0.02029460511
4	0.02017757726
5	0.02006051428
6	0.01994341614
7	0.01982628284
8	0.01970911435
9	0.01959191067
10	0.01947467177

- MAPLE code for verification

ScaLAPACK exercise: parallel Speed-Up



linear dense matrix solution (PDGESV):

n=10000

SGI Origin 3000 shared memory

`fenix.ciemat.es`

- block size is compromise between communication/load-balancing

$\longrightarrow$ needs to be optimized for given problem/hardware!