Using Parallel Numerical Libraries

Science & Technology Support
High Performance Computing

Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212-1163

OSC

Table of Contents

- Introduction
- Serial and Parallel Mathematical Libraries
- How to use a ScaLAPACK routine: step-by-step
- Matrix-Vector Multiplication
- Processor Grid Creation
- ScaLAPACK Data Distribution
- Using subarrays in ScaLAPACK routines
- Matrix-Matrix Multiplication
- Solving a Set of Simultaneous Linear Equations
- Tips for using the ScaLAPACK library

Introduction

- Course Goals
- ScaLAPACK Features
- References
- Acknowledgements

Course Goals

- Spread the Word to OSC users!
  - Since the advent of Massively Parallel Processing (MPP) supercomputers users have wanted a stable, efficient, library of useful mathematical routines that would run in parallel
  - ScaLAPACK is such a library for linear algebra calculations. It has reached such a high-level of portability and popularity that it has become the de facto standard
- Use a "how-to" teaching style to gradually expose the details of using ScaLAPACK through successively more complex sample programs and laboratory exercises
  - Complete programs are shown because subtle problems are often hidden when just code fragments are presented
  - The four sample programs developed cover popular linear algebra procedures and illustrate various programming styles/approaches
- Teach the four tasks needed to be performed before any ScaLAPACK routine is called
- Illustrate the use of tool subroutines built into the ScaLAPACK library to aid new users of the library
- Provide performance, debugging, and system-level tips for using the ScaLAPACK routines

ScaLAPACK Features

- Portability
  - ScaLAPACK library successfully runs on a variety of MPP platforms (Cray T3E, Origin 2000, IBM SP2, Workstation Clusters, Intel Paragon, etc.)
- Extensive set of routines for a variety of linear algebra problems
- Scalability
  - If number of processors used by a ScaLAPACK routines increases, the wall clock time for the calculations decreases
  - If the size of the problem (array dimensions) increases, parallel efficiency remains constant
- Efficient library structure
  - Built on top of existing, well-optimized serial library LAPACK
  - Includes separate libraries PBLACS for communication between processors and PBLAS for optimized low-level linear algebra tasks that insulate the particular machine characteristics from the user
- "User-Friendly" routine structure
  - ScaLAPACK routine names mirror LAPACK routine names (prefix a "P")
  - ScaLAPACK routine arguments as similar as possible to LAPACK arguments

Last but far from Least!

- User does not have to write their own parallel processing code *or* research a good parallel algorithm for linear algebra procedures
  - User needs no knowledge of messing-passing parallel programming (MPI, Shmem, PVM, …)
  - User needs no knowledge of data-parallel directive-based parallel programming (HPF, OpenMP, CRAFT,…)

Documentation for Parallel Numerical Libraries

- THE SOURCE: The netlib repository -> http://www.netlib.org
  - Source of **FREE** mathematical libraries for installation on a variety of machines
  - Site includes libraries themselves, **subroutine descriptions**, manuals, man pages, quick-reference cards, etc.
  - In particular, manuals of interest for this course are:
    - ScaLAPACK User's Guide
    - A Proposal for a Set of Parallel Basic Linear Algebra Subprograms (PBLAS User's Guide)
    - A User's Guide to the BLACS
- On Cray T3* MPP systems: useful introductory man pages
  - man intro_scalapack
  - man intro_blas1, man intro_blas2, man intro_blas3
  - man intro_blacs
  - individual man pages on routines listed in the library introductions

Acknowledgements

- To the computer scientists who developed the libraries, documentation, and support material found in the NETLIB depository
- Primary Institutions:
  - University of Tennessee, Knoxville
  - Oak Ridge National Laboratory
  - University of California: Berkeley

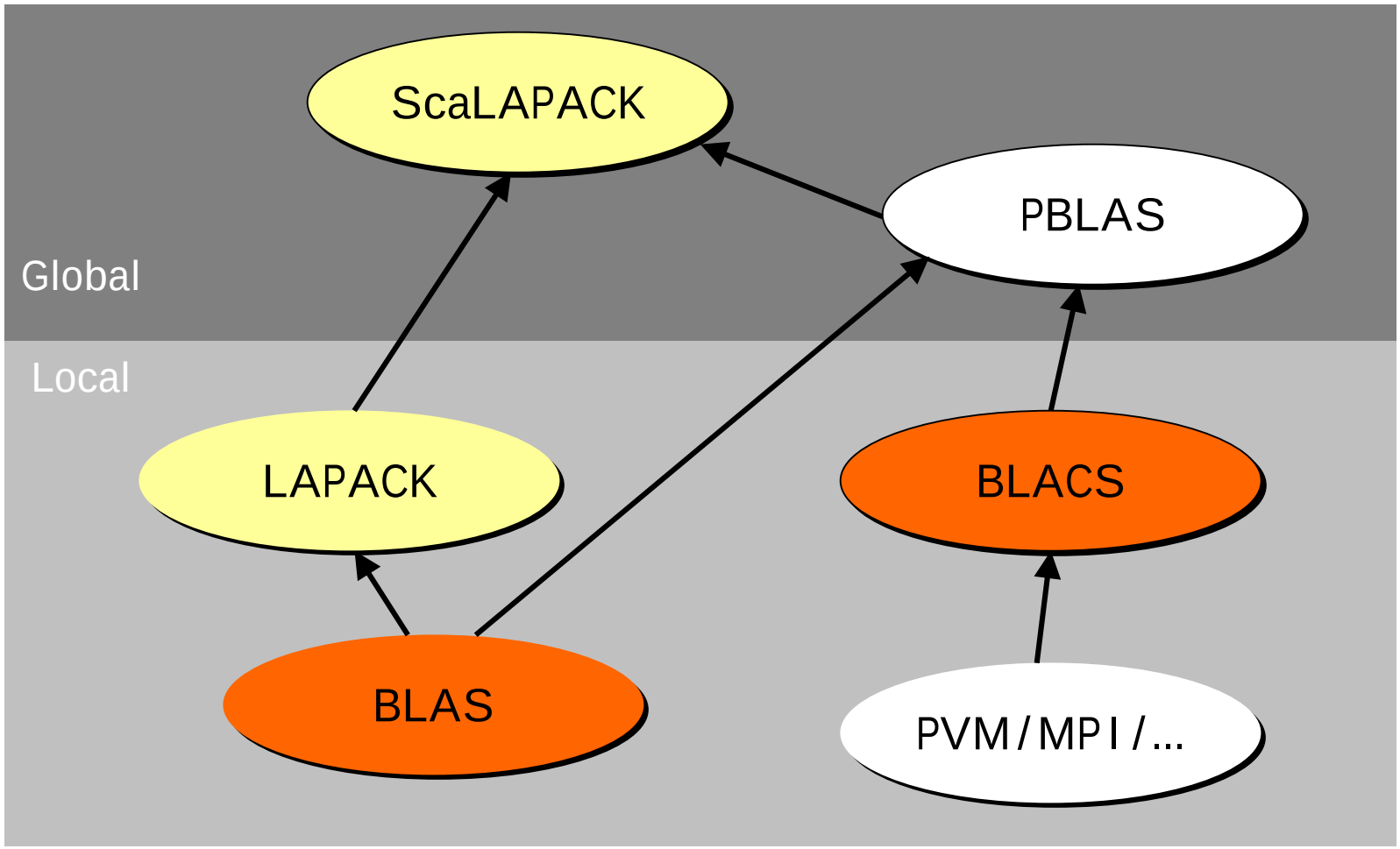
Serial and Parallel Mathematical Libraries

- Acronyms
- Mathematical Library Hierarchy
- Library Interconnections
- Overall Vision
- BLAS/PBLAS
- LAPACK/ScaLAPACK
- BLACS
- Processor Grid Example

First: The Acronyms

- BLAS: Basic Linear Algebra Subprograms
- PBLAS: Parallel Basic Linear Algebra Subprograms
- BLACS: Basic Linear Algebra Communication Subprograms
- LAPACK: Linear Algebra PACKage
- ScaLAPACK: Scalable Linear Algebra PACKage

The Mathematical Library Hierarchy



Library Interconnections

- The libraries lying below the “Local” keyword are serial libraries
  - LAPACK contains and is built on BLAS
  - Vendor optimized and only work on single processors: vector supercomputers, workstations, or one processor of an MPP system
  - Use blocking algorithms to keep and reuse data in the lowest levels of the memory hierarchy: registers -> primary cache -> secondary cache -> local memory
  - Routines contained in vendor’s own math library (e.g., Cray: LIBSci and IBM SP: ESSL)
- The libraries lying above the “Global” keyword are parallel libraries
  - ScaLAPACK contains and is built on PBLAS
  - Used for MPP systems to perform linear algebra tasks in parallel
  - ScaLAPACK also needs to built on BLACS in order to transfer local data from one processor to others
- The BLACS library is built on top of a message-passing library
  - Install the version of BLACS that matches the message-passing library present on a given machine
  - MPI is the industry standard and, therefore the most common library BLACS interfaces to but older message-passing libraries supported (PVM, MPL, Intel NX,…)

Overall Vision

- ScaLAPACK routines rely on
  - BLAS routines for optimized calculations for local arrays on single processors (the local arrays containing pieces of the global array actually being worked on in parallel)
  - BLACS routines for optimized communication of data between local arrays when needed
  - Only the BLAS and BLACS libraries depend on the characteristics of the real computer being used (cache/memory sizes, interconnection topology). Thus the SCALAPACK user is isolated from having to know about/program for a given machine's specifications.

BLAS/PBLAS

- Serial and parallel versions of basic linear algebra procedures
- Three Levels
  - Level 1: Vector-Vector Operations
    - Examples: swap, copy, addition, dot product, …
  - Level 2: Matrix-Vector Operations
    - Examples: multiply,rank-updates, …
  - Level 3: Matrix-Matrix Operations
    - Examples: multiply, transpose, rank-updates, ...
- Work with general, symmetric, complex, Hermitian, & triangular matrices
  - Supercomputer dependent

LAPACK/ScaLAPACK

- Serial and parallel routines for more advanced linear algebra calculations
- Three major classes
  - Solution of a set of simultaneous linear equations
  - Eigenvaluve/Eigenvector problems
  - Linear Least squares fit
- Factorization algorithms used:
  - LU, Cholesky, QR, LQ, Orthogonal, …
- Matrix types allowed:
  - General, complex, symmetric, banded, tridiagonal, Hermitian, orthogonal, triangular, ...

BLACS

- Used primarily for transferring data between processors
  - Point-Point routines for data sent from one processor and received by another
  - Broadcast routines for data broadcast from one processor and received by all the other processors being used
  - All communication routines are array-based: data that is transferred is entire arrays or subarrays
- Also has “combining” routines for doing global calculations with pieces of data in each local processors memory
  - Operations: summation, maximum, & minimum
- Most important for ScaLAPACK users: routines for creating and examining the processor grid
  - All Scalpack routines based on a 2-D processor grid whose size and shape is controlled by the user
  - Processor is identified not by its traditional ID number, but rather by its row and column number in the processor grid

Processor Grid Example

- Eight processors to be used in the program arranged in a 2x4 grid
  - Processor ID, row numbering, and column numbering all begin with 0
  - Have chosen to insert the processors into the grid “by-row”
  - Example: Processor 6 has grid coordinates (p,q) = (1,2)

	0	1	2	3
0	0	1	2	3
1	4	5	6	7

Step-by-Step Procedure for using a ScaLAPACK routine

- ScaLAPACK Checklist
- Running a ScaLAPACK Program

ScaLAPACK Checklist

① Write a scaled-down program using the equivalent LAPACK routine
   - → Recommended, not required
   - → Run on a single processor
   - → Good for familiarizing user with the proper routine name and arguments. Parallel routine name and arguments will be similar
   - → Good for syntax/logical debugging

② Initialize the BLACS library for its use in the program

③ Create and use the BLACS processor grid

④ Distribute pieces of each global array over the processors in the grid
   - → User does this by creating an array descriptor vector for each global array
   - → Global array elements mapped in a 2-D blocked-cyclic manner onto the processor grid

⑤ Have each processor initialize its local array with the correct values of the pieces of the global array it owns

⑥ Call the ScaLAPACK routine!

⑦ Confirm/use the output of the ScaLAPACK routine

⑧ Release the processor grid and exit the BLACS library

Running a ScaLAPACK program

- Compile loading the ScaLAPACK, BLACS, and communication primitives (i.e. MPI) libraries
- Run your code using the procedure required for the communication primitives library
  - It is at this stage that you set the number of processors your program will use
- For the OSC T3E:

  ```
  module load scalapack
  f90 prog.f90
  mpprun -n 8 a.out
  ```

- For the OSC Origin 2000

  ```
  f90 ${SCALAPACK} ${FBLACS}
  mpirun -n 8 a.out
  ```

- C versions of the libraries and compilation flags also available

Matrix-Vector Multiplication

- Problem Description
- Serial
  - LAPACK Routine: SGEMV
  - Example Program
- Parallel
  - ScaLAPACK Routine: PSGEMV
  - Example Program
- Summary

Problem Description

- The vector b(16) is to be multiplied by the 16 x 16 matrix A
- A and b are filled with random numbers in the range -50:50
- The operands are given by:

A=

-45	49	-45	48	-38	-17	5	17	29	30	0	10	-31	43	-12	-3
9	-10	0	-38	-38	-31	4	0	28	-3	-40	11	-17	49	7	-19
45	18	-16	-6	-38	0	-24	25	-7	-31	-30	-29	-45	31	31	33
9	-12	7	-20	-11	-26	-23	1	40	27	-33	-19	-18	22	-42	0
-6	18	-15	46	22	29	-45	-25	-33	-24	1	-26	-47	-30	-47	50
-17	-7	-31	-48	-10	25	-23	18	39	-15	37	43	26	24	36	15
11	-6	-23	50	47	48	-17	9	40	-20	-43	-38	-1	20	-11	38
5	-14	23	23	40	-24	18	26	-10	17	45	22	-32	-34	-12	12
-43	-31	49	48	21	-47	30	-33	47	26	-33	-15	43	11	-2	-25
0	15	-50	35	18	9	-12	45	-16	-25	11	-47	14	-2	41	47
26	-17	-7	2	13	-45	1	-42	-28	27	30	-45	48	-33	-44	-19
21	-28	10	25	-46	5	7	-8	49	41	22	-49	-6	24	-17	8
-41	-37	-15	26	29	-43	-4	-49	13	7	9	-38	-10	48	-28	50
35	25	6	39	42	-28	47	-34	24	40	44	20	39	26	-42	-28
-32	-50	-4	20	21	-7	39	0	28	41	5	41	-44	18	6	-33
49	-25	0	39	-10	41	13	-34	23	49	15	27	-49	16	-40	28

b= [-40 48 38 -32 -24 -10 49 -28 38 30 -7 42 -4 -4 -15 10]^T

LAPACK routine: SGEMV

- The first program shown will perform the matrix-vector multiplication on a single processor (step 1)
- Will use the BLAS Level 2 routine SGEMV
  - Template to BLAS/LAPACK routine names: S -> Single Precision, GE -> general matrix, and MV -> matrix-vector multiplication
- SGEMV performs the operation $y = \alpha Ab + \beta y$
- SGEMV(trans, m, n, α, A, lda, b, incb, β, y, incy)

  trans='n', use normal matrix A in the calculations; 'y'=> use transpose

  m= number of rows in A to be used (does not have to be all the rows)

  n= number of columns of A to be used (does not have to be all the columns)

  α= scaling factor (usually 1.0)

  A= address of the element of A that is first element of the submatrix of A used [to use all of A, can enter just A, or A(1,1)]

  lda= total number of rows of A when it was declared (leading dimension)

  b= address of the element of b that is the first element of subvector of b used

  incb= stride for b (typically 1)

  β= scaling/initialization factor (typically 0.0)

  y= on exit form routine, vector y **contains the result** of the multiplication

  incy= stride for y (typically 1)

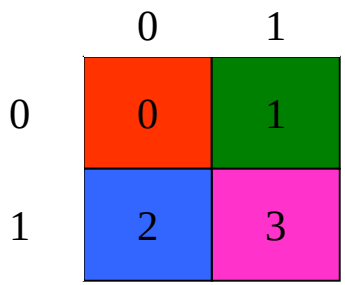
Serial Matrix-Vector Multiply Program

```
program serial_mv
       real, dimension(16,16) :: a
       real, dimension(16) :: b,y

       open(unit=12,file="a.data")
       read(12,*) a
       open(unit=13,file="b.data")
       read(13,*) b

       call sgemv('n',16,16,1.0,a,16,b,1,0.0,y,1)

       print *,"product is ",y

 end program serial_mv
```

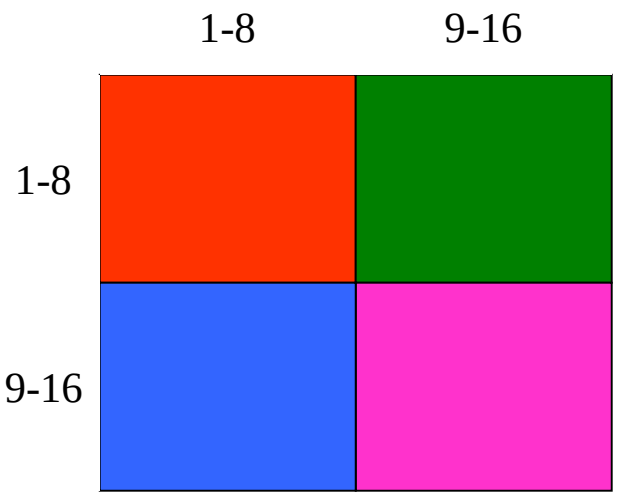
```
-------------------------------------------------
 product is  4341.,   3467.,  -5259.,  1716.,  -5821.,   750., -6561.,  -237.,
             5149.,  -9212.,  -2387.,     0.,  -1815.,  4601.,  3890.,   386.
```

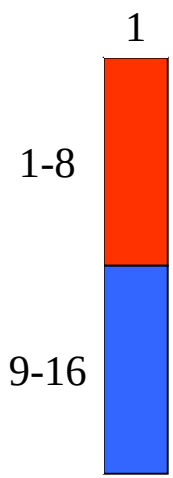
Parallel Matrix-Vector Multiply

- Next program will call the PSGEMV routine for parallel multiplication
- The 4 processors used will be arranged in a 2 x2 grid



- Blocks of A and b are distributed to the processors as follows:





ScaLAPACK routine: PSGEMV

- Program uses the "traditional" call to PSGEMV which assumes
  - Want to use the entire matrix A as an operand, not a submatrix
  - Want to use the entire, stand-alone, vector b as an operand, not a subvector
  - "Advanced" explanation of **all** the arguments of PSGEMV will be discussed later
- PSGEMV(trans,m,n,α,la,1,1,desca,lb,1,1,descb,1,β,ly,1,1descy,1)

  trans= if is 'n', work on normal matrix A, if 'y', work on the A^T

  m= the number of row in the global array A

  n= the number of columns of the global array A

  α= scaling factor (typically 1.0)

  la= local array containing the pieces of the distributed, global array A (shape and contents depend on the processor)

  desca= array descriptor vector for global array A (used by the routine to know what pieces of global array are on what processors)

  lb= local array containing the pieces of the distributed, global vector b

  descb= array descriptor vector for the global vector b

  β= scaling/initialization factor (typically 0.0)

  ly=local array containing pieces of the resultant product vector y

  descy= array descriptor for the global, product vector y

Parallel Matrix-Vector multiply program

```
program parallel_mv
  ! global arrays
  real, dimension(16,16) :: a
  real, dimension(16) :: b,y

  ! variables for BLACS initialization and processor grid creation
  integer iam,nprocs,ictxt,nprow,npcol,myrow,mycol

  ! variables needed for distributing global arrays across the proc grid
  integer desca(9),descb(9),descy(9),m,n,mb,nb,rsrc,csrc
  integer llda,lldb,info

   ! local arrays
  real, dimension(8,8) :: la
  real, dimension(8) :: lb,ly

  ! Initializing the BLACS library (STEP 2)
  call blacs_pinfo(iam,nprocs)
  call blacs_get(-1,0,ictxt)

  ! Creating and using the processor grid (STEP 3)
  nprow=2; npcol=2
  call blacs_gridinit(ictxt,'r',nprow,npcol)
  call blacs_gridinfo(ictxt,nprow,npcol,myrow,mycol)
```

Parallel MV* program (2)

```
! Making the array descriptor vectors (STEP 4)
  m=16; n=16
  mb=8; nb=8
  rsrc=0; csrc=0
  llda=8
  call descinit(desca,m,n,mb,nb,rsrc,csrc,ictxt,llda,info)
  n=1; nb=1; lldb=8
  call descinit(descb,m,n,mb,nb,rsrc,csrc,ictxt,lldb,info)
  call descinit(descy,m,n,mb,nb,rsrc,csrc,ictxt,lldb,info)

  ! Filling the global arrays A,b
  open(unit=12,file="a.data")
  read(12,*) a
  open(unit=13,file="b.data")
  read(13,*) b

  ! Each processors fills in its local arrays with correct elements
  ! from the global arrays (STEP 5)
  if(myrow.eq.0.and.mycol.eq.0) then
    do i_loc=1,8
      do j_loc=1,8
        la(i_loc,j_loc)=a(i_loc,j_loc)
      end do
      lb(i_loc)=b(i_loc)
    end do
  end if
```

Parallel MV* program (3)

```
if(myrow.eq.1.and.mycol.eq.0) then
    do i_loc=1,8
      do j_loc=1,8
        la(i_loc,j_loc)=a(i_loc+llda,j_loc)
      end do
      lb(i_loc)=b(i_loc+lldb)
    end do
  end if

  if(myrow.eq.0.and.mycol.eq.1) then
    do i_loc=1,8
      do j_loc=1,8
        la(i_loc,j_loc)=a(i_loc,j_loc+llda)
      end do
    end do
  end if

  if(myrow.eq.1.and.mycol.eq.1) then
    do i_loc=1,8
      do j_loc=1,8
        la(i_loc,j_loc)=a(i_loc+llda,j_loc+llda)
      end do
    end do
  end if
```

Parallel MV* program (4)

```
  ! Call the ScaLAPACK routine (STEP 6)
  n=16
  call psgemv('n',m,n,1.0,la,1,1,desca,lb,1,1,descb,1,0.0,ly,1,1,descy,1)

  ! Each processor prints out its part of the product vector y (STEP 7)
  if(myrow.eq.0.and.mycol.eq.0) then
    do i=1,8
      print *,'PE:',myrow,mycol,' y(',i,')=',ly(i)
    end do
  end if

  if(myrow.eq.1.and.mycol.eq.0) then
    do i=1,8
      print *,'PE:',myrow,mycol,' y(',i+lldb,')=',ly(i)
    end do
  end if

  ! Release the proc gird and BLACS library (STEP 8)
   call blacs_gridexit(ictxt)
   call blacs_exit(0)

end program parallel_mv
```

Parallel MV* program output

```
PE: 0,0  y( 1 )= 4341.
PE: 0,0  y( 2 )= 3467.
PE: 0,0  y( 3 )= -5259.
PE: 0,0  y( 4 )= 1716.
PE: 0,0  y( 5 )= -5821.
PE: 0,0  y( 6 )= 750.
PE: 0,0  y( 7 )= -6561.
PE: 0,0  y( 8 )= -237.
PE: 1,0  y( 9 )= 5149.
PE: 1,0  y( 10 )= -9212.
PE: 1,0  y( 11 )= -2387.
PE: 1,0  y( 12 )= 0.
PE: 1,0  y( 13 )= -1815.
PE: 1,0  y( 14 )= 4601.
PE: 1,0  y( 15 )= 3890.
PE: 1,0  y( 16 )= 836.
```

Key points summary of the Parallel MV* program

- Need to do all the preparatory steps!
- Each processor must call and contribute to the ScaLAPACK routine
- When comparing the LAPACK and ScaLAPACK routine calls find
  - Global arrays used A,b, and y used in the SGEMV call
  - Replaced by local arrays la, lb, and ly **AND** their array descriptor vectors in PSGEMV call
- Each processor will have different local arrays resulting from the distribution of the global arrays over the processor grid
- Resultant array (y) also is distributed meaning parts of the answer are found on different processors
- Something to ponder: Do we, in general, need to declare the global arrays? …

Processor Grid Formation

- BLACS Initialization Routines
- BLACS Grid Routines
- BLACS Tool Routines
- Example Program and Output

BLACS initialization routines

- For all the BLACS routines will follow the convention that underlined arguments are output from the routine, all others are input
- BLACS_PINFO(myid, nprocs)
  - primary purpose of calling this routine is to get the number of processors that the program will use (specified on parallel run command line)
  - Arguments
    - myid =the ID number of the processor that called the routine (zero-based)
    - nprocs= the total number of processors
- BLACS_GET(-1,0,ictxt)
  - Blacs_get is actually a general-purpose, utility routine to get values for a set of BLACS internal parameters.
  - With these arguments, the purpose of the blacs_get call is to retrieve the context (third argument) for the parallel processing program.
    - N.B. First argument is ignored in a context-getting call
  - Context of a parallel data communication library uniquely identifies all the BLACS data transfer calls as being part of a particular SCALAPCK routine. No other message-passing routines in the program can interfere with the BLACS communication
  - For MPI users, context is equivalent to an MPI Communicator value. It represents a message passing “universe”

BLACS Grid Routines

- BLACS_GRIDINIT (icontxt, order, nprow, npcol)
  - This routines creates the virtual processor grid according to the specifications contained in its arguments
  - Arguments:
    - icontxt= BLACS context for its message-passing (first argument of almost every BLACS routine)
    - order= Sets how processor ID numbers are mapping to the grid
      - 'R' => mapping done by rows
      - 'C' => mapping done by columns
    - nprow= sets the number of rows in the processor grid
    - npcol= sets the number of columns in the processor grid
- BLACS_GRIDINFO (icontxt, nprow, npcol, myrow, mycol)
  - Primary use is for each processor to obtain its coordinates (myrow, mycol) in the processor grid. Get in the habit of using these coordinates to identify a processor!
  - Notice that nprow and npcol are echoed back to make sure they are what you wanted
  - New arguments:
    - myrow =calling processor's row number in the processor grid (zero-based)
    - mycol= calling processor's column number in the processor grid (zero-based)

BLACS Tool Routines

- How do you know the grid was created correctly?
- BLACS_PCOORD (icontxt, myid, myrow, mycol)
  - Answers question: What grid position does a given processor have?
  - Used to retrieve the processor coordinates (myrow, mycol) of a processor with traditional ID number myid.
- integer BLACS_PNUM (icontxt, myrow, mycol)
  - Complimentary function to BLACS_PCOORD
  - Answers question: What processor is at a specified grid position?
  - **Returns** the processor ID number at processor grid coordinates (myrow, mycol)
- Program on the next slide demonstrates the use of these tool routines in the creation of a 2x3 processor grid using 7 processors

BLACS tools program

```
program tools
   integer ictxt,myid,nprocs,nprow,npcol
   integer p,q,myrow,mycol
   integer blacs_pnum
   call blacs_pinfo(myid,nprocs)
   call blacs_get(-1,0,ictxt)
   ! Calculations to make the most square-like proc grid possible
   nprow=int(sqrt(real(nprocs)))
   npcol=nprocs/nprow
   call blacs_gridinit(ictxt,'c',nprow,npcol)
   call blacs_gridinfo(ictxt,nprow,npcol,myrow,mycol)
   if (myid==0) then
     do i=0,(nprow*npcol)-1
       call blacs_pcoord(ictxt,i,p,q)
       print *,'Processor ',i,' is at grid position (p,q)=',p,q
     end do
     do p=0,nprow-1
       do q=0,npcol-1
         print *,'At grid position (',p,',',q,') is processor',&
                     blacs_pnum(ictxt,p,q)
       end do
     end do
   end if
   call blacs_gridexit(ictxt)
   call blacs_exit(0)
 end program tools
```

BLACS tools program output

```
Processor  0  is at grid position (p,q)= 0,0
Processor  1  is at grid position (p,q)= 1,0
Processor  2  is at grid position (p,q)= 0,1
Processor  3  is at grid position (p,q)= 1,1
Processor  4  is at grid position (p,q)= 0,2
Processor  5  is at grid position (p,q)= 1,2
At grid position ( 0 , 0 ) is processor 0
At grid position ( 0 , 1 ) is processor 2
At grid position ( 0 , 2 ) is processor 4
At grid position ( 1 , 0 ) is processor 1
At grid position ( 1 , 1 ) is processor 3
At grid position ( 1 , 2 ) is processor 5
```

ScaLAPACK Data Distribution

- Data Distribution Method
- Array Descriptor Vector
- Data Distribution Tool Routines
- Program and Output

Data Distribution Method

- ScaLAPACK uses a two-dimensional block cyclic distribution technique
- Distributes global array elements onto the processor grid
- Chosen because it gives best load balance and efficient use of Level 3 BLAS single-processor computation routines (data locality) for most ScaLAPACK algorithms
- Procedure steps:
  - Divide up the global array into blocks with **mb** rows and **nb** columns
  - From now on, think of the global array as composed only of these blocks
  - Distribute first row of array blocks across the first row of the processor grid in order. If you run out processor grid columns cycle back to first column
  - Repeat for the second row of array blocks, with the second row of the processor grid.
  - Continue for remaining rows of array blocks
  - If you run out of processor grid rows, cycle back to the first processor row and repeat
- The diagram on the next slide illustrates a 2-D block-cyclic distribution of a 9x9 global array with 2x2 blocks over a 2x3 processor grid (The colors represent the 6 different processors)

Distribution Example

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}	a_{17}	a_{18}	a_{19}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}	a_{27}	a_{28}	a_{29}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}	a_{36}	a_{37}	a_{38}	a_{39}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}	a_{46}	a_{47}	a_{48}	a_{49}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}	a_{56}	a_{57}	a_{58}	a_{59}
a_{61}	a_{62}	a_{63}	a_{64}	a_{65}	a_{66}	a_{67}	a_{68}	a_{69}
a_{71}	a_{72}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}	a_{78}	a_{79}
a_{81}	a_{82}	a_{83}	a_{84}	a_{85}	a_{86}	a_{87}	a_{88}	a_{89}
a_{91}	a_{92}	a_{93}	a_{94}	a_{95}	a_{96}	a_{97}	a_{98}	a_{99}

Global View

	0				**1**			**2**
	a_{11}	a_{12}	a_{17}	a_{18}	a_{13}	a_{14}	a_{19}	a_{15}	a_{16}
	a_{21}	a_{22}	a_{27}	a_{28}	a_{23}	a_{24}	a_{29}	a_{25}	a_{26}
0	a_{51}	a_{52}	a_{57}	a_{58}	a_{53}	a_{54}	a_{59}	a_{55}	a_{56}
	a_{61}	a_{62}	a_{67}	a_{68}	a_{63}	a_{64}	a_{69}	a_{65}	a_{66}
	a_{91}	a_{92}	a_{97}	a_{98}	a_{93}	a_{94}	a_{99}	a_{95}	a_{96}
	a_{31}	a_{32}	a_{37}	a_{38}	a_{33}	a_{34}	a_{39}	a_{35}	a_{36}
	a_{41}	a_{42}	a_{47}	a_{48}	a_{43}	a_{44}	a_{49}	a_{45}	a_{46}
1	a_{71}	a_{72}	a_{77}	a_{78}	a_{73}	a_{74}	a_{79}	a_{75}	a_{76}
	a_{81}	a_{82}	a_{87}	a_{88}	a_{83}	a_{84}	a_{89}	a_{85}	a_{86}

Local (distributed) View

Distribution Example (2)

- Notice that each processor has a different sized local array
  - Processor (0,0): local array dimensions 5x4
  - Processor (0,1): local array dimensions 5x3
  - Processor (0,2): local array dimensions 5x2
  - Processor (1,0): local array dimensions 4x4
  - Processor (1,1): local array dimensions 4x3
  - Processor (1,2): local array dimensions 4x2
- Notice that contents of each local array consists of a non-contiguous collection of global array elements (a real mishmash …)

Array Descriptor Vector

- Mechanism by which each ScaLAPACK routine determines the distribution of global array elements into local arrays owned by each processor
- For the global array A, traditional to name the nine-element array descriptor vector DESCA
- ScaLPACK notation: "_A" read as "of the distributed global array A
- Symbolic name and definition of each element of DESCA:

  DESCA(1) = dtype_A -> type of matrix being distributed (1 for dense matrix)

  DESCA(2) = ictxt -> BLACS context for the program

  DESCA(3) = m_A -> number of rows in the global array A

  DESCA(4) = n_A -> number of columns in the global array A

  DESCA(5) = mb_A -> number of rows in a block of A

  DESCA(6) = nb_A -> number of columns in a block of A

  DESCA(7) = rsrc_A -> processor grid row which has the first block of A (typically 0)

  DESCA(8) = csrc_A -> processor grid col which has the first block of A (typically 0)

  DESCA(9) = lld -> number of rows of the local array that stores the blocks of A

  (local leading dimension)

DESCINIT Utility Routine

- Never have to initialize the descriptor vector explicitly
- Just need to call the DESCINIT routine which will create the descriptor vector from is arguments
- DESCINIT(desc, m, n, mb, nb, rsrc, csrc, ictxt, lld, info)
  - Arguments
    - desc = the “filled-in” descriptor vector returned by the routine
    - arguments 2-9 = values for elements 2-9 of the descriptor vector (order different)
    - info= integer returned to indicate whether the DESCINIT call worked

      info=0 -> routine was successful

      info=-i -> the ith argument had an illegal value

Data Distribution Tool Routines

- Each processor has several questions it needs answers to:
  - How many rows should be in my local array?
  - How many columns should be in my local array?
  - What global elements should I put in my local array?
- ScaLAPACK function NUMROC will answer the first two
  - Read as “Number of Rows Or Columns”
- ScaLAPACK function INDXG2P will answer the third
  - Read as “Index: global to processor grid”

NUMROC utility function

- Returns the number of rows or columns of a local array containing blocks of a distributed global array
- Will show the arguments for computing the local number of rows. Just switch row to column everywhere to get the number of local columns
- Returned values are dependent on the calling process
- integer NUMROC(m_, mb, myrow, rsrc, nprow)
  - Arguments
    - m_ = number of rows in the global array
    - mb = number of rows in each block
    - myrow = row coordinate of the calling processor
    - rsrc = row coordinate of the processor containing the first block
    - nprow = number of rows in the processor grid

INDXG2P utility routine

- Given the global indices (i,j) of a certain element of a global array, returns the processor grid coordinates (*p*,*q*) that element was distributed to
- Will show the arguments to INDXG2P in which i is provided to the routine and *p* is returned. To get *q*, substitute j for i, and column for row.
- integer INDXG2P(i, mb, iproc, rsrc, nprow)
  - Arguments:
    - i = row index of the global array element (i,j)
    - mb = number of rows in each block
    - iproc = dummy argument
    - rsrc = processor grid row coordinate of the processor containing the first block
    - nprow = number of rows in the processor grid

- Following program illustrates the use of these utility routines for the 9x9 array distributed onto the 2x3 processor grid shown in the Distribution Example page.

Data Distribution program

```
program dtools
  integer ictxt,myid,nprocs,nprow,npcol
  integer p,q,myrow,mycol

  call blacs_pinfo(myid,nprocs)
  call blacs_get(-1,0,ictxt)
  nprow=2; npcol=3
  call blacs_gridinit(ictxt,'r',nprow,npcol)
  call blacs_gridinfo(ictxt,nprow,npcol,myrow,mycol)

  !Find out the size of the local array for proc(0,1)
  if (myrow==0.and.mycol==1) then
    print * ,'The number of rows in my local array is ',&
              numroc(9,2,myrow,0,2)
    print * ,'The number of cols in my local array is ',&
              numroc(9,2,mycol,0,3)
  !Find out what processors got the diagonal elements
    do k=1,9
      p=indxg2p(k,2,0,0,2)
      q=indxg2p(k,2,0,0,3)
      print *,'Element (',k,',',k,') is on proc (',p,',',q,')'
    end do
  end if

  call blacs_gridexit(ictxt)
  call blacs_exit(0)
end program dtools
```

Data Distribution program output

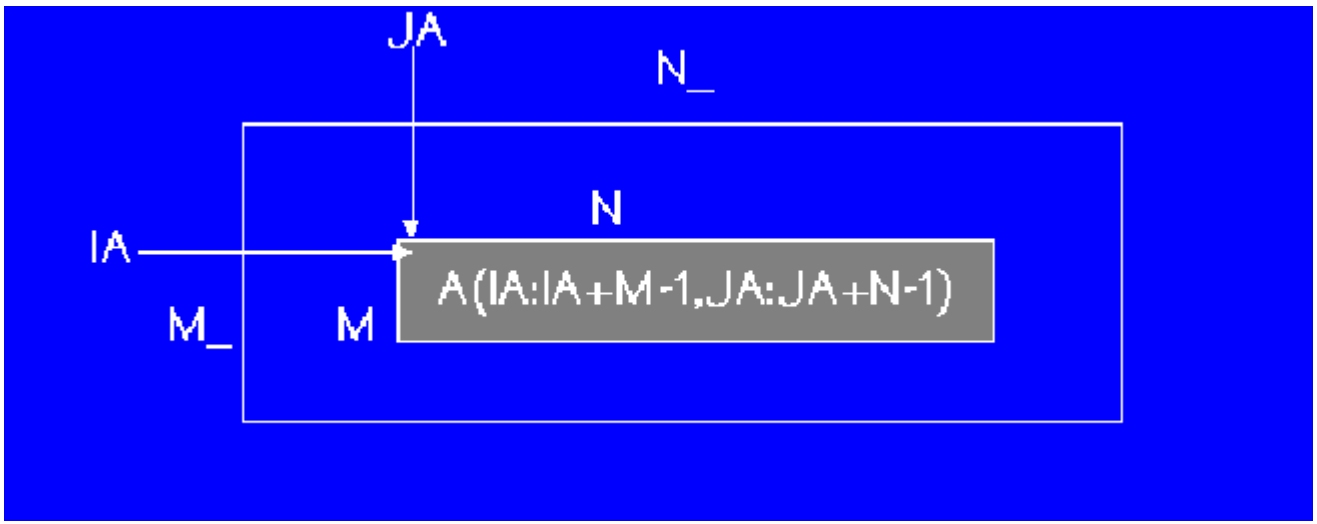
```
The number of rows in my local array is  5
The number of cols in my local array is  3
Element ( 1 , 1 ) is on proc ( 0 , 0 )
Element ( 2 , 2 ) is on proc ( 0 , 0 )
Element ( 3 , 3 ) is on proc ( 1 , 1 )
Element ( 4 , 4 ) is on proc ( 1 , 1 )
Element ( 5 , 5 ) is on proc ( 0 , 2 )
Element ( 6 , 6 ) is on proc ( 0 , 2 )
Element ( 7 , 7 ) is on proc ( 1 , 0 )
Element ( 8 , 8 ) is on proc ( 1 , 0 )
Element ( 9 , 9 ) is on proc ( 0 , 1 )
```

Using subrarrays in ScaLAPACK routines

Subarray Syntax

- A long-standing design feature of single-processor LAPACK/BLAS routines has been the ability of the user to only use a section of an array - a subarray- in the call to a calculation subroutine.
  - The entire array must still be declared and initialized before the library call.
  - The syntax for specifying a subarray has always followed the same pattern
    - call LAPACK_routine (…, M, N, A(I,J), …)
  - This trio of arguments tells the routine **do not** use the entire array A, but use a subarrray starting at element A(I,J) and extending for M rows and N columns
- The developers of the parallel ScaLAPACK/PBLAS libraries decided to also allow users the ability to use subarrays as the operands for parallel calculations…. But the syntax is different
  - **All** the preparatory steps must still being carried out before the call to the ScaLAPACK routines
  - Syntax for ScaLAPACK routines now involves the following arguments:
    - call ScaLPACK_routine (…, M, N, la, IA, JA, desca, …)
  - These arguments indicate to use a subarray of the global array A starting at element A(IA,JA) and extending for M rows and N columns.
  - Notice that the actual array argument - la - is still the local array holding blocks of the global array A and that the array descriptor vector is (as always) needed

Illustration of a ScaLAPACK subarray



Matrix-Matrix Multiplication

- Problem Description
- Serial
  - Example Program
- Parallel Approach
  - INDXL2G
  - PSGEMM
  - Example Program
  - Observations

Problem Description

- Create and initialize two 5x5 matrices A and B given by the equations:

  $A(i,j) = 10.0*i + j$

  $B(i,j) = A(i,j) + 5.0$

- Calculate the product C=A*B where A, B, and C are all 3x3 subarrays
  - Subarray of A that will be used as an operand will start at A(3,3)
  - Subarray of B that will be used as an operand will start at B(1,3)
  - Subarray of C that will hold the result will start at C(3,1)
- Will use the BLAS routine SGEMM
  - Performs the operation $C = \alpha AB + \beta C$
  - SGEMM (transa, transb, m, n, k, α, A, lda, B, ldb, β, C, ldc)
    - transa, transb = character arguments to choose to use transposes
    - m x k = dimensions of subarray A
    - k x n = dimensions of subarray B
    - m x n = dimensions of subarray C
    - lda, ldb,ldc = leading dimensions at declaration of full arrays A, B, and C
    - α,β = scaling factors

Serial MM* program

```
program mm
  real, dimension(5,5) :: a,b,c
  do i=1,5
    do j=1,5
      a(i,j)=10.0*i+j
    end do
  end do
  b=a+5.0; c=0.0
  print *,"a is "
  call show2d(a)
  print *,"b is "
  call show2d(b)
  call sgemm('n','n',3,3,3,1.0, a(3,3) ,5, b(1,3) ,5,0.0, c(3,1) ,5)
  print *,"product c is "
  call show2d(c)
contains
  subroutine show2d(a)
    real, dimension(:,:) :: a
    integer :: nrows,ncols,i
    nrows=size(a,1)
    ncols=size(a,2)
    do i=1,nrows
      print *, a(i,1:ncols)
    end do
  end subroutine show2d
end program mm
```

Serial MM* program output

```
a is
11.,  12.,  13.,  14.,  15.
21.,  22.,  23.,  24.,  25.
31.,  32.,  33.,  34.,  35.
41.,  42.,  43.,  44.,  45.
51.,  52.,  53.,  54.,  55.
b is
16.,  17.,  18.,  19.,  20.
26.,  27.,  28.,  29.,  30.
36.,  37.,  38.,  39.,  40.
46.,  47.,  48.,  49.,  50.
56.,  57.,  58.,  59.,  60.
product c is
 0.,   0.,   0.,   0.,   0.
 0.,   0.,   0.,   0.,   0.
2876.,  2978.,  3080.,   0.
3716.,  3848.,  3980.,   0.
4556.,  4718.,  4880.,   0.
```

Parallel Matrix-Matrix Multiply Approach

- The parallel MM* program will distribute the 5x5 global arrays A,B, & C over a 2x2 processor grid made from the 4 processors it will be run on
- The blocks used for the distribution will be 2x2 resulting in the following mapping:

a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}
a_{31}	a_{32}	a_{33}	a_{34}	a_{35}
a_{41}	a_{42}	a_{43}	a_{44}	a_{45}
a_{51}	a_{52}	a_{53}	a_{54}	a_{55}

5x5 matrix partitioned in 2x2 blocks

	0		**1**	
	a_{11}	a_{12}	a_{15}	a_{13}	a_{14}
0	a_{21}	a_{22}	a_{25}	a_{23}	a_{24}
	a_{51}	a_{52}	a_{55}	a_{53}	a_{54}
1	a_{31}	a_{32}	a_{35}	a_{33}	a_{34}
	a_{41}	a_{42}	a_{45}	a_{43}	a_{44}

2x2 process grid point of view

- As with the serial program the same three subarrays of A,B, & C will be used as the actual arguments in the parallel library routine call PSGEMM

INDXL2G utility routine

- A difficulty that we will encounter in this particular parallel programming example (and it arises in many others as well) is dealing with the fact that the local arrays must be filled with a diverse collection of blocks of elements from the global arrays.
- I.e., when processor (1,1) initializes its local array la, la(2,1) must be set to the contents of the global array element A(4,3). And the global array elements are determined by a formula using the global indices i,j.
- What is needed is INDXL2G: a utility function that converts local array coordinates to their global equivalents
  - Routine name should be read as Index: local-to-global
- Will show the arguments for INDXL2G when iloc is provided to the routine and global i is returned. To convert a local column index, switch j for i and col for row everywhere
- Integer INDXL2G (iloc, mb, myrow, rsrc, nprow)
  - iloc = row index of the local array
  - mb = number of rows in each block
  - myrow = *p* = row coordinate of the proc grid for the calling processor
  - rsrc = row coordinate of the proc grid holding the first block
  - nprow = number of rows in the proc grid

PSGEMM PBLAS2 routine

- PSGEMM (transa, transb, m, n, k, α, la, IA, JA, desca, lb, IB, JB, descb, β, lc, IC, JC, desc)
- Performs the same calculation as SGEMM but in parallel
- Arguments:
  - transa, transb = character arguments to choose to use transposes
  - m x k = shape of the subarray of A
  - k x n = shape of the subarray of B
  - m x n = shape of the subarray of C
  - la = local array contains the pieces of A owned by the calling processor
  - IA, JA = global indices indicating where the operand subarray A starts
  - desca = array descriptor vector for the distribution of A
  - α,β = scaling constants

Parallel MM* program

```
program pmm
  ! Declare only local arrays
  real, dimension(3,3) :: la,lb,lc

  integer iam,nprocs,ictxt,nprow,npcol,myrow,mycol
  integer desca(9),descb(9),descc(9)
  integer m,n,mb,nb,rsrc,csrc,ictxt,llda,lldb,lldc,info
  integer lm,ln
  integer i,j,iloc,jloc

  ! Initialize and use BLACS proc grid (STEPS 2 & 3)
  nprow=2; npcol=2
  call blacs_pinfo(iam,nprocs)
  call blacs_get(-1,0,ictxt)
  call blacs_gridinit(ictxt,'r',nprow,npcol)
  call blacs_gridinfo(ictxt,nprow,npcol,myrow,mycol)

  ! Make the array descriptor vectors (STEP 4)
  m=5; n=5
  mb=2; nb=2
  rsrc=0; csrc=0
  llda=3; lldb=3; lldc=3
  call descinit(desca,m,n,mb,nb,rsrc,csrc,ictxt,llda,info)
  call descinit(descb,m,n,mb,nb,rsrc,csrc,ictxt,lldb,info)
  call descinit(descc,m,n,mb,nb,rsrc,csrc,ictxt,lldc,info)
```

Parallel MM* program (2)

```
! Initialize local arrays (STEP 5)
 lm=numroc(5,2,myrow,0,nprow)
 ln=numroc(5,2,mycol,0,npcol)
 do iloc=1,lm
   do jloc=1,ln
     i=indxl2g(iloc,2,myrow,0,nprow)
     j=indxl2g(jloc,2,mycol,0,npcol)
     la(iloc,jloc)=10.0*i+j
     lb(iloc,jloc)=la(iloc,jloc)+5.0
     lc(iloc,jloc)=0.0
   end do
 end do
 ! Call the ScaLAPACK routine (STEP 6)
 call psgemm('n','n',3,3,3,1.0,la,3,3,desca,lb,1,3,descb,&
                0.0,lc,3,1,descc)
 ! Print out the matrix product (STEP 7)
 do iloc=1,lm
   do jloc=1,ln
     i=indxl2g(iloc,2,myrow,0,nprow)
     j=indxl2g(jloc,2,mycol,0,npcol)
     print *,'PE:',myrow,mycol,' c(',i,',',j,')=',lc(iloc,jloc)
   end do
 end do
 ! Release BLACS procs grid (STEP 8)
 call blacs_gridexit(ictxt)
 call blacs_exit(0)
end program pmm
```

Parallel MM* program output

```
PE: 0,1  c( 1 , 3 )= 0.
PE: 0,1  c( 1 , 4 )= 0.
PE: 0,1  c( 2 , 3 )= 0.
PE: 0,1  c( 2 , 4 )= 0.
PE: 0,1  c( 5 , 3 )= 4880.
PE: 0,1  c( 5 , 4 )= 0.
PE: 1,0  c( 3 , 1 )= 2876.
PE: 1,0  c( 3 , 2 )= 2978.
PE: 1,0  c( 3 , 5 )= 0.
PE: 1,0  c( 4 , 1 )= 3716.
PE: 1,0  c( 4 , 2 )= 3848.
PE: 1,0  c( 4 , 5 )= 0.
PE: 0,0  c( 1 , 1 )= 0.
PE: 0,0  c( 1 , 2 )= 0.
PE: 0,0  c( 1 , 5 )= 0.
PE: 0,0  c( 2 , 1 )= 0.
PE: 0,0  c( 2 , 2 )= 0.
PE: 0,0  c( 2 , 5 )= 0.
PE: 0,0  c( 5 , 1 )= 4556.
PE: 0,0  c( 5 , 2 )= 4718.
PE: 0,0  c( 5 , 5 )= 0.
PE: 1,1  c( 3 , 3 )= 3080.
PE: 1,1  c( 3 , 4 )= 0.
PE: 1,1  c( 4 , 3 )= 3980.
PE: 1,1  c( 4 , 4 )= 0.
```

Observations on the parallel MM* program

- Good that declared only local arrays (should only do that!)
  - Bad that each processor declared the same size local array (the largest needed)
- Good that each processor only filled up necessary elements of their local arrays through the use of the numroc function
- Good demonstration of the handiness of the indxg2l function

Solving a Set of Simultaneous Linear Equations

- Problem Description
- Serial
  - LAPACK LU Factorization Routine
  - LAPACK Solver
  - Example Program
- Parallel
  - ScaLAPACK Routines
  - Example Program
  - Observations

Problem Description

- Will solve the classic linear algebra equation Ax = b for x
- Two-step approach using two separate library routines
  - First, perform LU factorization of A in which it is converted to a product of L: a lower triangular matrix, and U: an upper triangular matrix
  - Second, solve for x using the LU form of A with two backward substitutions
- For our particular problem A will be an 8x8 matrix and b an 8x1 column vector given by:

$$A(i,j) = 1.0 + (|\, i-j \,|/8)^{1/2} \quad \text{and} \quad b(i) = i$$

- Input parameters will be read in from an ASCII file consisting of two lines

  8

  2
- Will use double-precision variables allowing the use of the OSC Origin 2000
- Will use Fortran 90 dynamic memory allocation of arrays involved

LAPACK LU Factorization routine

- DGETRF(m, n, A, lda, ipiv, info)
- Arguments

  m = number of rows of A to use

  n = number of columns of A to use

  A = when call the routine, A is the initialized array (or subarray) to be factored

  when routine is exited, A has been transformed to LU structure

  lda = total number of rows of A when declared

  ipiv = array of pivot indices

  info = error parameter

  - 0 => routine worked successfully
  - -i => ith argument has illegal value
  - i => U(i,i) is exactly zero, division by zero will result if try to solve the system of equations

LAPACK Solver Routine

- DGETRS(transa, n, nrhs, A, lda, ipiv, b, ldb, info)
- New arguments

  nrhs = number of right hand sides (also number of columns of b)

  ldb = number of rows of b at declaration

  b = matrix of size (ldb,nrhs)

  When call the routine, b is composed of right-hand side vectors

  When exit the routine, b is x - the solution vectors

  A = the factors L and U of the original matrix A

  info = error indicator (meaning of output same as descinit routine)

Serial Ax=b program

```
program serial_lu_solve
  implicit none
  integer, parameter :: wp = kind(1.d0)
  real(kind=wp), allocatable, dimension(:,:) :: A
  real(kind=wp), allocatable, dimension(:) :: b
  integer :: sys_order

  ! read input and rank of the matrices
  call get_parameters(sys_order)

  ! allocate memory for matrices
  allocate(A(sys_order,sys_order),b(sys_order))

  ! initialize and solve system
  call init_system(A,b)
  call solver(A,b)

  ! deallocate memory
  deallocate(A,b)

contains

  subroutine get_parameters(sys_order)
    integer, intent(out) :: sys_order
    open(10,file='lu_solve.in',status='old')
    read(10,*) sys_order
    close(10)
  end subroutine get_parameters
```

Serial Ax=b program (2)

```
 subroutine init_system(A,b)
   real(kind=wp), dimension(:,:), intent(out) :: A
   real(kind=wp), dimension(:), intent(out) :: b
   integer :: sys_order, i, j, nrows, ncols
   sys_order = size(b)
   do j = 1, sys_order
     do i = 1, sys_order
       A(i,j) = 1.0_wp + sqrt(dble(abs(i-j))/dble(sys_order))
     end do
     b(j) = j
   end do
  end subroutine init_system
  subroutine solver(A,b)
    real(kind=wp), dimension(:,:), intent(inout) :: A
    real(kind=wp), dimension(:), intent(inout) :: b
    integer :: sys_order, nrhs=1, info
    integer, dimension(:), allocatable :: ipvt
    sys_order = size(b)
    allocate(ipvt(sys_order))
    call dgetrf(sys_order,sys_order,A,sys_order,ipvt,info)
    if (info .ne. 0) print*,'dgetrf info: ',info
    call dgetrs('N',sys_order,nrhs,A,sys_order,ipvt,b,sys_order,info)
    if (info .ne. 0) print*,'dgetrs info: ',info
    print *, "The solutions is ",b
    deallocate(ipvt)
  end subroutine solver
end program serial_lu_solve
```

Serial Ax=b program output

```
The solution is  3.8336897379972403,  1.2859533851648366,  0.60781423200748153,
    0.29060541561231096,
            6.07382194034837147E-2, -0.19533620761377216,  -0.66577303130579069,
    -2.3075968294420841
```

Parallel solution to Ax=b

- Will run on 4 processors arranged in 1x4 processor "grid"
- Block size for global array A will be 2x2, for global vector b will be 2x1
- Given these parameters Block-cyclic mapping will results in the following distribution:
  - Proc (0,0) will have cols 1 & 2 of A and all of b
  - Proc (0,1) will have cols 3 & 4 of A
  - Proc (0,2) will have cols 5 & 6 of A
  - Proc (0,3) will have cols 7 & 8 of A
- But we don't have to work out this distribution ourselves, the program will determine it and each processor will dynamically allocate the exactly correct-sized local arrays
- Will implement parallel LU factorization and parallel solver with the routines PDGETRF and PDGETRS respectively
- As with the serial program, will operate on the entire arrays A and b, not on subarrays

ScaLAPACK routines used

- PDGETRF(m, n, la, IA, JA, desca, ipiv, info)
- PDGETRS(transa, n, nrhs, la, IA, JA, desca, ipiv, lb, IB, JB, descb, info)
- Again, we see that arguments have the same name and meaning as in the LAPACK routines, except for how the operand arrays are specified.
- The same translation for these operand arrays occurs
  - LAPACK routine (…, m, n, A(i,j), …) where A is global (only one processor)

  - SCALPACK routine (…, m,n, la, IA, JA, desca, ..) where la is local to a specific processor and desca tells the routine how the global array is distributed among the processors

Parallel Ax=b program

```
program parallel_lu_solve

implicit none
integer, parameter :: wp = kind(1.d0)
real(kind=wp), allocatable, dimension(:,:) :: A
real(kind=wp), allocatable, dimension(:,:) :: b
integer, dimension(9) :: desc_A, desc_b
integer :: sys_order, block_size, nproc
integer :: nprow, npcol, icontxt, myrow, mycol
integer :: mloc_A, nloc_A, mloc_b, nloc_b

! read input: get number of processors and block size
call get_parameters(sys_order,block_size,nproc)
! initialize BLACS grid and array descriptor vectors
call init_grid(sys_order,block_size,nprow,npcol,myrow,mycol,icontxt, &
               desc_A,desc_b,mloc_A,nloc_A,mloc_b,nloc_b)
! allocate space for matrices
allocate(A(mloc_A,nloc_A),b(mloc_b,nloc_b))
! initialize and solve system
call init_system(A,b,sys_order,block_size,mloc_A,nloc_A,mloc_b,nloc_b, &
                 myrow,mycol,nprow,npcol)
call solver(A,b,sys_order,desc_A,desc_b)
! deallocate space
deallocate(A,b)
! exit BLACS
call blacs_exit(0)
contains
```

Parallel Ax=b program (2)

```
subroutine get_parameters(sys_order,block_size,nproc)
  integer, intent(out) :: sys_order, block_size, nproc
  integer :: mypnum
  open(10,file='lu_solve.in',status='old')
  read(10,*) sys_order; read(10,*) block_size
  close(10)
  call blacs_pinfo(mypnum,nproc)
end subroutine get_parameters
subroutine init_grid(sys_order,block_size,nprow,npcol,myrow,mycol,icontxt, &
                     desc_A,desc_b,mloc_A,nloc_A,mloc_b,nloc_b)
  integer, intent(in) :: sys_order, block_size
  integer, intent(out) :: nprow, npcol, myrow, mycol, icontxt
  integer, intent(out) :: mloc_A, nloc_A, mloc_b,nloc_b
  integer, dimension(9), intent(out) :: desc_A, desc_b
  nprow = 1; npcol=nproc
  call blacs_get(0,0,icontxt)
  call blacs_gridinit(icontxt,'R',nprow,npcol)
  call blacs_gridinfo(icontxt,nprow,npcol,myrow,mycol)
  mloc_A = numroc(sys_order,block_size,myrow,0,nprow)
  nloc_A = numroc(sys_order,block_size,mycol,0,npcol)
  mloc_b = numroc(sys_order,block_size,myrow,0,nprow)
  nloc_b = numroc(1,1,mycol,0,npcol)
  ! set up descriptor vectors for matrix a and vector b
  call descinit(desc_A, sys_order, sys_order, block_size, block_size, 0, 0, &
              icontxt, mloc_A, info)
  call descinit(desc_b, sys_order, 1, block_size, 1, 0, 0, &
              icontxt, mloc_b, info)
end subroutine init_grid
```

Parallel Ax=b program (3)

```
subroutine init_system(A,b,sys_order,block_size,mloc_A,nloc_A,mloc_b,nloc_b, &
                       myrow,mycol,nprow,npcol)
  implicit none
  real(kind=wp), dimension(:,:), intent(out) :: A
  real(kind=wp), dimension(:,:), intent(out) :: b
  integer, intent(in) :: sys_order, mloc_A, nloc_A, mloc_b, nloc_b,block_size
  integer, intent(in) :: myrow, mycol, nprow, npcol
  integer :: indxl2g, i, i_loc, j, j_loc
  ! set up portion of system matrix A owned by each processor
  do j_loc = 1, nloc_A
    j = indxl2g(j_loc,block_size,mycol,0,npcol)
    do i_loc = 1, mloc_A
      i = indxl2g(i_loc,block_size,myrow,0,nprow)
      A(i_loc,j_loc) = 1.0_wp + sqrt(dble(abs(i-j))/dble(sys_order))
    end do
  end do

  do j_loc = 1, nloc_b
    j = indxl2g(j_loc,1,mycol,0,npcol)
    do i_loc = 1, mloc_b
      i = indxl2g(i_loc,block_size,myrow,0,nprow)
      b(i_loc,j_loc) = i
    end do
  end do
end subroutine init_system
```

Parallel Ax=b program output

```
subroutine solver(A,b,sys_order,desc_A,desc_b)
  real(kind=wp), dimension(:,:), intent(inout) :: A
  real(kind=wp), dimension(:,:), intent(inout) :: b
  integer, intent(in) :: sys_order
  integer, dimension(9), intent(in) :: desc_A, desc_b
  integer :: iA, jA, ib, jb, nrhs
  integer :: info
  integer, dimension(sys_order+block_size) :: ipvt
  iA=1; jA=1; ib=1; jb=1; nrhs=1
  call blacs_barrier(icontxt,'ALL')
  !  Call LU factorization routine
  call pdgetrf(sys_order,sys_order,A,iA,jA,desc_A,ipvt,info)
  if (info .ne. 0) print*,'pdgetrf info: ',info
  !  Call LU solver routine
  call pdgetrs('N',sys_order,nrhs,A,iA,jA,desc_A,ipvt,b,ib,jb,desc_b,info)
  if (info .ne. 0) print*,'pdgetrs info: ',info
  print *, 'PE:',myrow,mycol,' b=',b
end subroutine solver
end program parallel_lu_solve
-------------------------------------------------------
 PE: 0,1  b=
 PE: 0,2  b=
 PE: 0,3  b=
 PE: 0,0  b= 3.8336897379972394,  1.2859533851648375,  0.60781423200748175,
    0.29060541561231029,
       6.07382194034839298E-2,     -0.19533620761377282,     -0.66577303130578991,
    -2.3075968294420841
```

Observations on parallel Ax=b program

- Excellent modular, top-down programming design
- Can scale up easily in terms of all parameters: array rank, block size, and number of processors
  - This due to file input and dynamic memory allocation of local arrays
- High-level ScaLAPACK task
- Argument checking and global variable advantages of “contained” F90 subroutines
- Controllable, portable precision
- Drawback: for larger number of processors, use a square-like proc grid
- Acknowledgement: MHPCC

Tips for using the ScaLAPACK library

Useful ScaLAPACK tips

- Massively Parallelize: use a large number of processors
  - Rule of Thumb nprocs=(m x n)/1000000
  - Gives good-sized local arrays (rank in 1000s)
- Conversely, recall Amdahl's Law: time to solve a small problem plateaus for too large a number of processors
- Be aware of local memory each processor has on an MPP machine
  - Cray T3E= 128 MB = 16 MW (1 Word = 8 bytes)
  - O2K = 256 MB = 32 MW
  - Determines how large your local arrays can be
- Use the most square-like processor grid possible
  - Provides good load balance for most of the parallel algorithms used
- Use a large (32-64) block size
  - Allows for efficient use of caches in BLAS calls made by ScaLAPACK routines
- For a specific ScaLAPACK routine and a fixed number of processors, experiment with processor grid and global array distribution
- Use specific, machine-tuned BLAS and BLACS libraries

Useful ScaLAPACK tips (2)

- Use utility routines and dynamic memory allocation to make the processor-specific local arrays and to determine which blocks of the global arrays they hold
- Being a good citizen: freeing internal memory buffers when no longer needed
  - BLACS routine: BLACS_FREEBUF
  - PBLAS routine: PBFREEBUF
- Can use stand-alone BLACS routines for your own message-passing parallel codes
  - Compact set of routines: communication & calculations
  - Send and Receive: SGESD2D & SGERV2D
  - Broadcast Send & Receive: SGEBS2D & SGEBR2D
  - Unit of data that is sent is a general 2D array or trapezoidal 2D array
    - Don't have to use MPI routines to make you own new datatype
- Can control the topology used by the BLACS broadcast routines: some better for some parallel algorithms
  - Routine is PTOPSET
  - Two main classes of topologies: ring and tree
  - **Warning:** Library developers have already tried to pick best topology for you

Useful ScaLAPACK tips (3)

- Always check error information arguments (if they are present) returned by ScaLAPACK routines. Should be zero, if all went well
  - Have already seen INFO argument for checking validity of other arguments in the routine
  - For advanced ScaLAPACK routines there is an LWORK to indicate whether there was enough workspace for the routine to do its calculations
- We have always used the BLACS_GRIDINT routine to set up our processor grid. It only give you two choices - row and column - to what grid coordinates (*p*,*q*) a processor with a given ID gets.
  - More general routine BLACS_GRIDMAP in which you specify the exact mapping between processor ID number and (*p*,*q*)
  - Could be useful for having he virtual topology of the processor grid match more exactly the actual physical topology of the machine you are using. I.e., processors that are physical neighbors on the machine can be made to be neighbors on the processor grid