Lecture 20 – Blocked Algorithms - LU Factorization (Linpack)

- **Consider the factorization of Matrix A that is decomposed into sub-blocks (sub-matrices) A_{11}, A_{12}, …..A_{nn}**

$$A = LU = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & U_{22} & U_{23} \\ & & U_{33} \end{pmatrix}$$

$$A_{11} = L_{11}U_{11} \qquad A_{12} = L_{11}U_{12} \qquad A_{13} = L_{11}U_{13}$$

$$A_{21} = L_{21}U_{11} \qquad A_{22} = L_{21}U_{12} + L_{22}U_{22} \qquad A_{23} = L_{21}U_{13} + L_{22}U_{23}$$

$$A_{31} = L_{31}U_{11} \qquad A_{32} = L_{31}U_{12} + L_{32}U_{22} \qquad A_{33} = L_{31}U_{13} + L_{32}U_{23} + L_{33}U_{33}$$

See "Numerical Linear Algebra for High-Performance Computers" by Dongarra, Duff, Sorensen, and van der Vorst

Blocked Algorithms - LU Factorization

$$A_{11} = L_{11}U_{11} \qquad A_{12} = L_{11}U_{12} \qquad A_{13} = L_{11}U_{13}$$
$$A_{21} = L_{21}U_{11} \qquad A_{22} = L_{21}U_{12} + L_{22}U_{22} \qquad A_{23} = L_{21}U_{13} + L_{22}U_{23}$$
$$A_{31} = L_{31}U_{11} \qquad A_{32} = L_{31}U_{12} + L_{32}U_{22} \qquad A_{33} = L_{31}U_{13} + L_{32}U_{23} + L_{33}U_{33}$$

- **With these relationships, we can develop different algorithms by choosing the order in which operations are performed.**
- **Blocksize, *b (column width)*, needs to be chosen carefully.**
  - *b=1* produces the usual single-processor algorithm
  - *b>1* will improve performance on a single-processor
  - Choice of b will depend on computer-dependent parameters such as cache size, memory bandwidth, and number of vector registers
- **Three natural variants:**
  - Left-Looking LU
  - Right-Looking LU, implemented in LAPACK and ScaLAPACK
  - Crout LU

Blocked Algorithms - LU Factorization

- **Assume you have already done the first row and column of the GEPP**

$$A = LU = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & & \\ L_{21} & & \\ L_{31} & & \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & U_{13} \\ & & \\ & & \end{pmatrix}$$

- **And you have the sub-block below left to work on**

$$\begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix}$$

- **Notice that the LU decomposition in this sub-block is independent of the portion you have already completed**

Blocked Algorithms - LU Factorization

- **For simplicity, change notation of sub-block to**

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = P \begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ & U_{22} \end{pmatrix} = P \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{pmatrix}$$

- **Notice that, once you have done Gaussian Elimination on A_{11} and A_{21} you have already obtained L_{11}, L_{21}, and U_{11}**

$$A_{12} = L_{11}U_{12} \qquad A_{22} = L_{21}U_{12} + L_{22}U_{22}$$
$$= L_{21}U_{12} + \tilde{A}_{22}$$

- **Now you can re-arrange the block equations by substituting:**

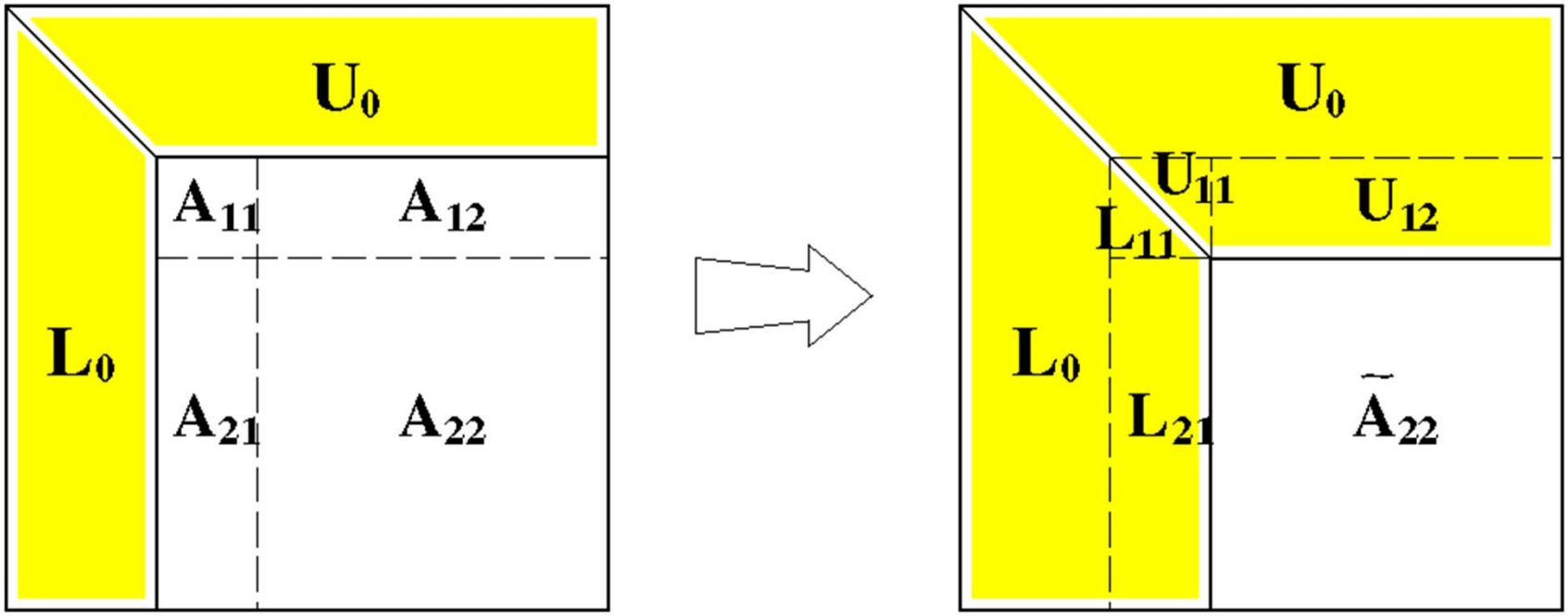
$$U_{12} \leftarrow (L_{11})^{-1} A_{12}$$

$$\tilde{A}_{22} \leftarrow A_{22} - L_{21}U_{12}$$

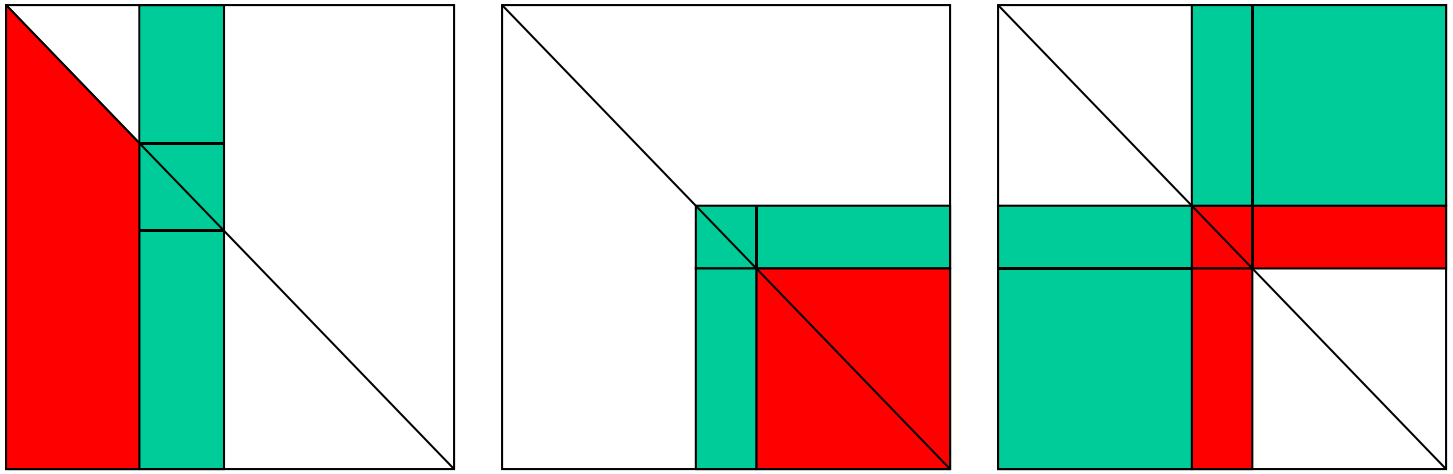
- **And repeat the procedure recursively moving down and to the right in the matrix**

Blocked Algorithms - LU Factorization
(right-looking)

- **A graphical view of what is going on is given by:**



Blocked Algorithms - LU Factorization



Left-Looking LU Right-Looking LU Crout LU

- **Variations in algorithm are due to the order in which sub-matrix operations are performed. Slight advantages to Crout's algorithm (hybrid of the first two) for vector machines with large memory bandwidth; requires fewer memory references**

Pre-computed sub-blocks

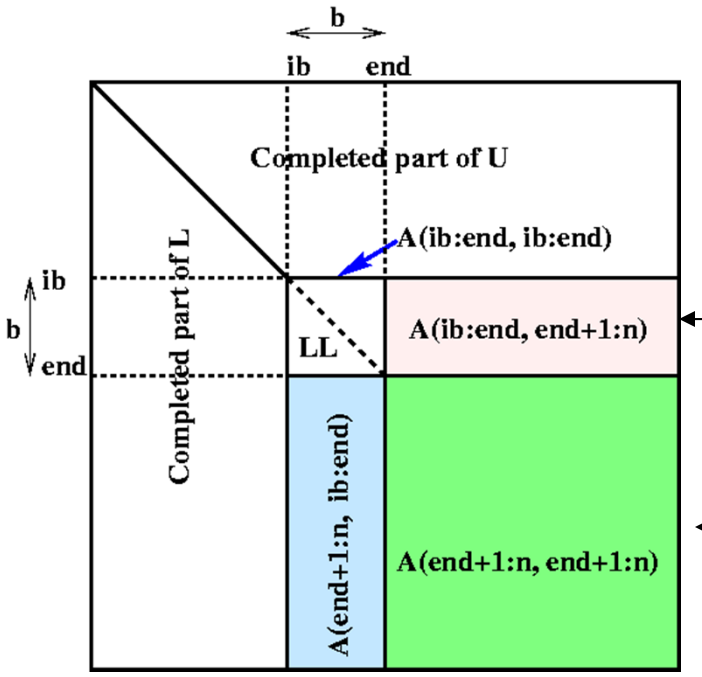
Currently being operated on sub-blocks

BLAS 3 (Blocked) GEPP with Right-Looking

```
for   ib = 1 to n-1 step b     … Process matrix b (block size) columns at a time
    end = ib + b-1             … Point to end of block of b columns
    apply BLAS2 version of GEPP to  get A(ib:n , ib:end) = P' * L' * U'
    … let LL denote the strict lower triangular part of A(ib:end , ib:end) + I
    A(ib:end , end+1:n) = LL^-1 * A(ib:end , end+1:n)        … update next b rows of U
    A(end+1:n , end+1:n ) = A(end+1:n , end+1:n )
              - A(end+1:n , ib:end) * A(ib:end , end+1:n)
                              … apply delayed updates with single matrix-multiply
                              … with inner dimension b
```

BLAS 3

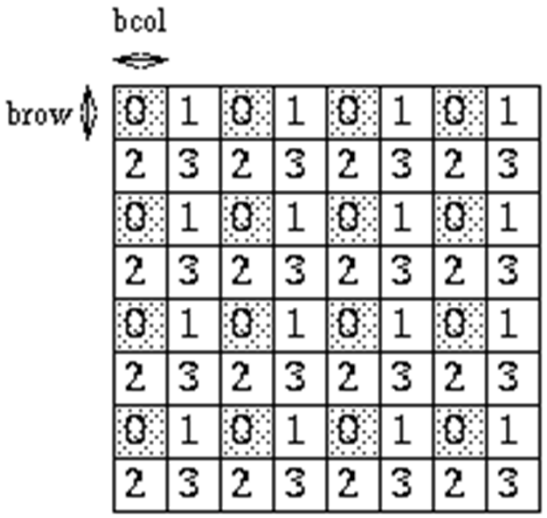
Gaussian Elimination using BLAS 3



$$U_{12} \leftarrow (L_{11})^{-1} A_{12}$$

$$\tilde{A}_{22} \leftarrow A_{22} - L_{21} U_{12}$$

Row and Column Block Cyclic Layout



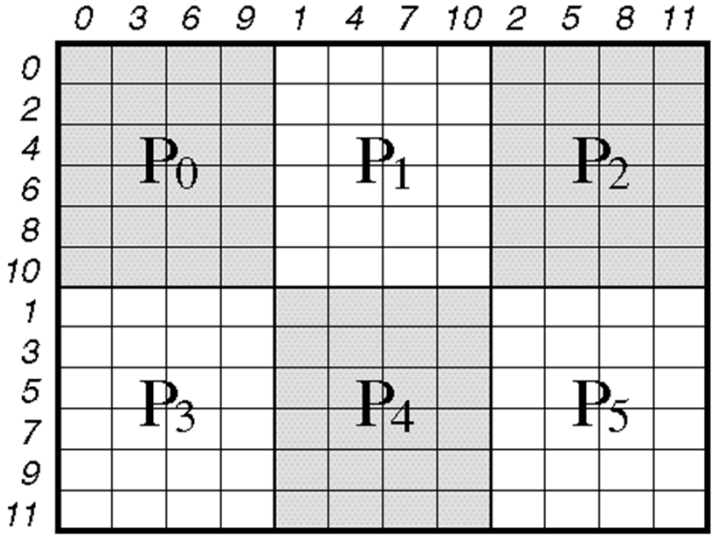
4) Row and Column Block Cyclic Layout

- **Let's examine block cyclic decomposition**
- **Processors (prow-by-pcol) and matrix blocks (brow-by-bcol) are distributed in a 2D array**
- **Pcol-fold parallelism in any column, and calls to the BLAS2 and BLAS3 on matrices of size brow-by-bcol**
- **Sub-matricies need not be symmetric in rows and columns**

Row and Column Block Cyclic Layout

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	2	0	1	2	0	1	2	0	1	2
1	3	4	5	3	4	5	3	4	5	3	4	5
2	0	1	2	0	1	2	0	1	2	0	1	2
3	3	4	5	3	4	5	3	4	5	3	4	5
4	0	1	2	0	1	2	0	1	2	0	1	2
5	3	4	5	3	4	5	3	4	5	3	4	5
6	0	1	2	0	1	2	0	1	2	0	1	2
7	3	4	5	3	4	5	3	4	5	3	4	5
8	0	1	2	0	1	2	0	1	2	0	1	2
9	3	4	5	3	4	5	3	4	5	3	4	5
10	0	1	2	0	1	2	0	1	2	0	1	2
11	3	4	5	3	4	5	3	4	5	3	4	5

(a) block distribution over 2 x 3 grid.



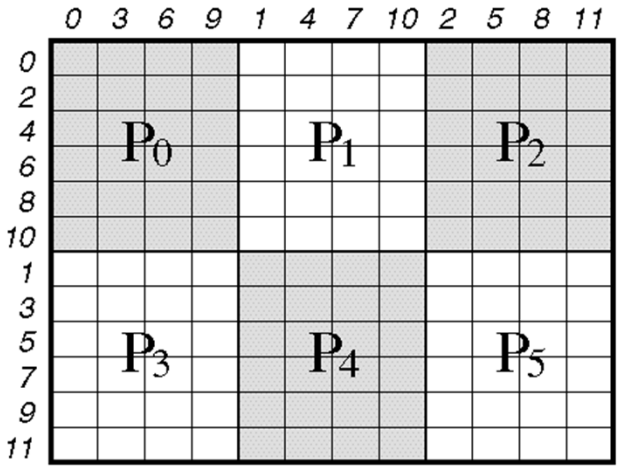
(b) data distribution from processor point-of-view.

- **In LU factorization, distribution of work becomes uneven as the computation progresses**
- **Larger block sizes result in greater load imbalance but reduce frequency of communication between processes**
- **Block size controls these tradeoffs**

Row and Column Block Cyclic Layout

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	2	0	1	2	0	1	2	0	1	2
1	3	4	5	3	4	5	3	4	5	3	4	5
2	0	1	2	0	1	2	0	1	2	0	1	2
3	3	4	5	3	4	5	3	4	5	3	4	5
4	0	1	2	0	1	2	0	1	2	0	1	2
5	3	4	5	3	4	5	3	4	5	3	4	5
6	0	1	2	0	1	2	0	1	2	0	1	2
7	3	4	5	3	4	5	3	4	5	3	4	5
8	0	1	2	0	1	2	0	1	2	0	1	2
9	3	4	5	3	4	5	3	4	5	3	4	5
10	0	1	2	0	1	2	0	1	2	0	1	2
11	3	4	5	3	4	5	3	4	5	3	4	5

(a) block distribution over 2 x 3 grid.



(b) data distribution from processor point-of-view.

- **Also, some hot spots arise in situations where some processors need to do more work between synchronization points than others (e.g. partial pivoting over rows in a single block-column…other processors stay idle. Also, the computation of each block row of the U factorization requires the solution of a lower triangular system across processes in a single row)**
- **Processor decomposition controls this type of tradeoff**

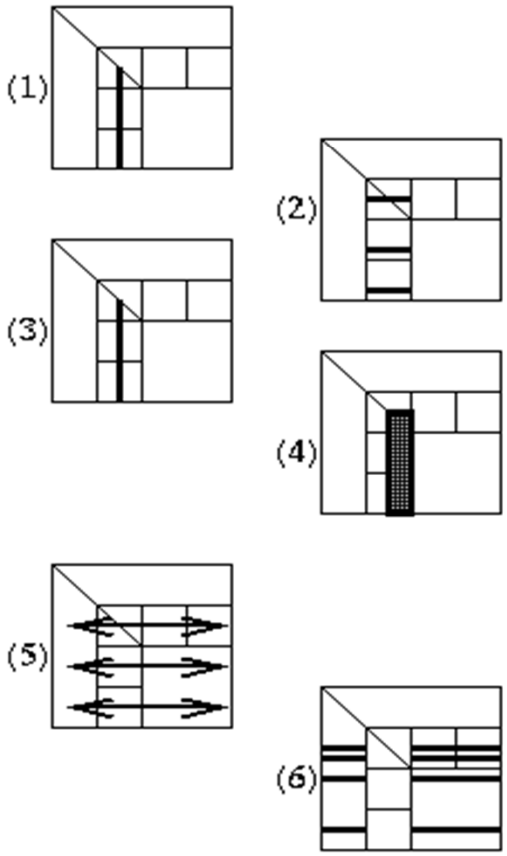
Distributed Parallel GE with a 2D Block Cyclic Layout

- **Block size, *b*, in the algorithm and the block sizes *brow* and *bcol* in the layout satisfy *b=brow=bcol*.**

- **In next few slides, shaded regions indicate busy processors or communication performed.**

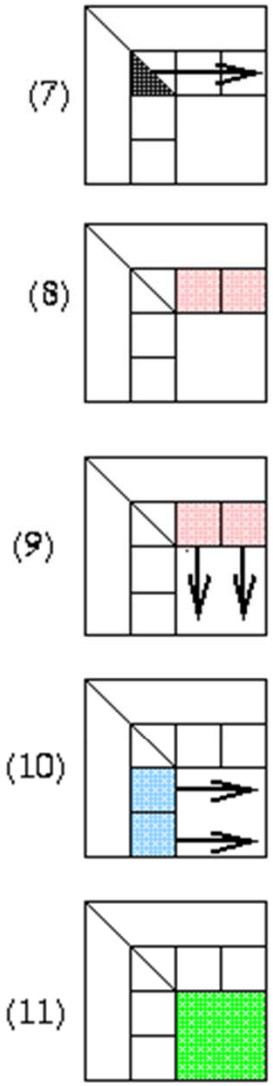
- **Unnecessary to have a barrier between each step of the algorithm, e.g. step 9, 10, and 11 can be pipelined**
  - See the next few slides

Distributed Parallel Gaussian Elimination with a 2D Block Cyclic Layout




```
for ib = 1 to n−1 step b
        end = min( ib+b−1, n )
        for i = ib to end
(1)     find pivot row k, column broadcast
(2)     swap rows k and i in block column, broadcast row k
(3)     A( i+1:n , i ) = A( i+1:n , i ) / A( i , i )
(4)     A( i+1:n , i+1:end) −= A( i+1:n , i ) * A( i , i+1:end )
        end for
(5)     broadcast all swap information right and left
(6)     apply all rows swaps to other columns
```

Distributed Parallel Gaussian Elimination with a 2D Block Cyclic Layout



(7) Broadcast LL right

(8) A(ib:end , end+1:n) = LL \ A(ib:end , end+1:n)

(9) Broadcast A(ib:end , end+1:n) down

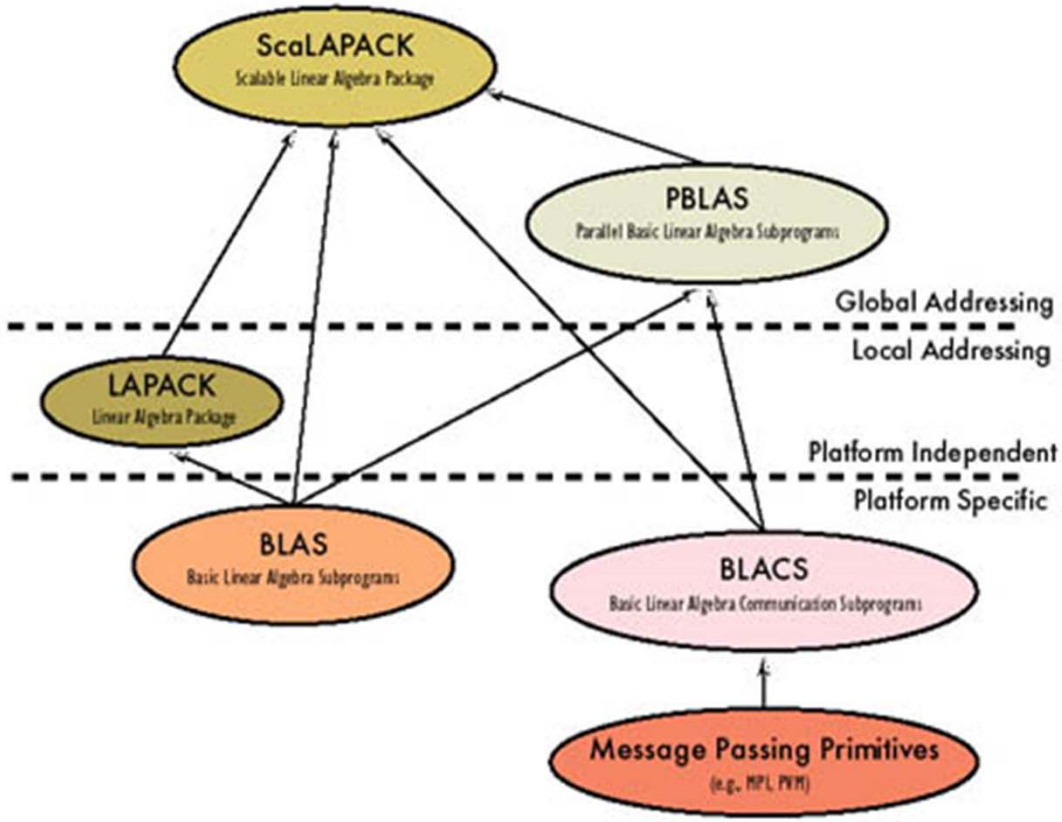
(10) Broadcast A(end+1:n , ib:end) right

(11) Eliminate A(end+1:n , end+1:n)

Matrix multiply of
green = green - blue * pink

ScaLAPACK

A Software Library for Linear Algebra Computations on Distributed-Memory Computers



AVAILABLE SOFTWARE:

Dense, Band, and Tridiagonal Linear Systems
- general
- symmetric positive definite

Full-Rank Linear Least Squares

Standard and Generalized Orthogonal Factorizations

Eigensolvers
- SEP: Symmetric Eigenproblem
- NEP: Nonsymmetric Eigenproblem
- GSEP: Generalized Symmetric Eigenproblem

SVD

Prototype Codes
- HPF interface to ScaLAPACK
- Matrix Sign Function for Eigenproblems
- Out-of-core solvers (LU, Cholesky, QR)
- Super LU
- PBLAS (algorithmic blocking and no alignment restrictions.)

DOCUMENTATION:

ScaLAPACK Users' Guide
http://www.netlib.org/scalapack/slug/scalapack_slug.html

Future Work
- Out-of-core Eigensolvers
- Divide and Conquer routines
- C++ and Java Interfaces

Commercial Use

ScaLAPACK has been incorporated into the following software packages:
- NAG Numerical Library
- IBM Parallel ESSL
- SGI Cray Scientific Software Library

and is being integrated into the VNI IMSL Numerical Library, as well as software libraries for Fujitsu, HP/Convex, Hitachi, and NEC.

http://www.netlib.org/scalapack/

The University of Tennessee

ornl Oak Ridge National Laboratory

Other Dense Matrix Factorizations

- **Details vary slightly, but overall procedure is quite similar**
- **LU, QR, and Cholesky factorizations can be effectively dealt with within this framework.**
- **You can also see the details of the ScaLAPACK source to understand the algorithm better.**

PDGESV = ScaLAPACK
parallel LU routine to solve Ax = b

Since it can run no faster than its inner loop (PDGEMM), we measure:
Efficiency =
Speed(PDGESV)/Speed(PDGEMM)

Observations:
- Efficiency well above 50% for large enough problems
- For fixed N, as P increases, efficiency decreases (just as for PDGEMM)
- For fixed P, as N increases efficiency increases (just as for PDGEMM)
- From bottom table, cost of solving Ax=b about half of matrix multiply for large enough matrices.
- From the flop counts we would expect it to be $(2*n^3)/(2/3*n^3) = 3$ times faster, but communication makes it a little slower.

Performance of ScaLAPACK LU

The higher the better →

Efficiency = MFlops(PDGESV)/MFlops(PDGEMM)

Machine	Procs	Block Size	N		
			2000	4000	10000
Cray T3E	4	32	.67	.82	
	16		.44	.65	.84
	64		.18	.47	.75
IBM SP2	4	50	.56		
	16		.29	.52	
	64		.15	.32	.66
Intel XP/S MP Paragon	4	32	.64		
	16		.37	.66	
	64		.16	.42	.75
Berkeley NOW	4	32	.76		
	32		.38	.62	.71
	64		.28	.54	.69

The lower the better →

Time(PDGESV)/Time(PDGEMM)

Machine	Procs	Block Size	N		
			2000	4000	10000
Cray T3E	4	32	.50	.40	
	16		.75	.51	.40
	64		1.86	.72	.45
IBM SP2	4	50	.60		
	16		1.16	.64	
	64		2.24	1.03	.51
Intel XP/S GP Paragon	4	32	.52		
	16		.89	.50	
	64		2.08	.79	.44
Berkeley NOW	4	32	.44		
	32		.88	.54	.47
	64		1.18	.62	.49

Targeted Machines and Applications

LAPACK and ScaLAPACK

	LAPACK	ScaLAPACK
Machines	Workstations, Vector, SMP	Distributed Memory, DSM
Based on	BLAS	BLAS, BLACS
Functionality	Linear Systems Least Squares Eigenproblems	Linear Systems Least Squares Eigenproblems (less than LAPACK)
Matrix types	Dense, band	Dense, band, out-of-core
Error Bounds	Complete	A few
Languages	F77 or C	F77 and C
Interfaces to	C++, F90	HPF
Manual?	Yes	Yes
Where?	www.netlib.org/ lapack	www.netlib.org/ scalapack

PDGELS = ScaLAPACK parallel QR factorization routine

Scales well, nearly full machine speed

Performance of ScaLAPACK QR (Least squares)

The higher the better

Efficiency = MFlops(PDGELS)/MFlops(PDGEMM)

Machine	Procs	Block Size	N		
			2000	4000	10000
Cray T3E	4	32	.54	.61	
	16		.46	.55	.60
	64		.26	.47	.54
IBM SP2	4	50	.51		
	16		.29	.51	
	64		.19	.36	.54
Intel XP/S GP Paragon	4	32	.61		
	16		.43	.63	
	64		.22	.48	.62
Berkeley NOW	4	32	.51	.77	
	32		.49	.66	.71
	64		.37	.60	.72

The lower the better

Time(PDGELS)/Time(PDGEMM)

Machine	Procs	Block Size	N		
			2000	4000	10000
Cray T3E	4	32	1.2	1.1	
	16		1.5	1.2	1.1
	64		2.6	1.4	1.2
IBM SP2	4	50	1.3		
	16		2.3	1.3	
	64		3.6	1.8	1.2
Intel XP/S GP Paragon	4	32	1.1		
	16		1.6	1.1	
	64		3.0	1.4	1.1
Berkeley NOW	4	32	1.3	.9	
	32		1.4	1.0	.9
	64		1.8	1.1	.9

Old Algorithms, plan to abandon

Performance of Symmetric Eigensolvers

The lower the better →

Time(PDSYEVX)/Time(PDGEMM)
(bisection + inverse iteration)

Machine	Procs	Block Size	N	
			2000	4000
Cray T3E	4	32	10	
	16		13	10
	64		29	14
IBB SP2	16	50	24	
	64		40	29
Intel XP/S GP	16	32	22	
Paragon	64		34	20
Berkeley NOW	16	32	20	
	32		24	52

The lower the better →

Time(PDSYEV)/Time(PDGEMM)
(QR iteration)

Machine	Procs	Block Size	N	
			2000	4000
Cray T3E	4	32	35	
	16		37	35
	64		57	41
IBM SP2	16	50	38	
	64		58	47
Intel XP/S GP	16	32	99	
Paragon	64		193	
Berkeley NOW	16	32	31	
	32		35	55

Singular Value Decomposition

Performance of SVD (Singular Value Decomposition)

The lower the better →

Time(PDGESVD)/Time(PDGEMM)				
Machine	Procs	Block Size	N	
			2000	4000
Cray T3E	4	32	67	
	16		66	64
	64		93	70
IBM SP2	4	50	97	
	16		60	
	64		81	
Berkeley NOW	4	32	72	
	16		38	16
	32		59	26

Hardest of all to parallelize

Performance of Nonsymmetric Eigensolver (QR iteration)

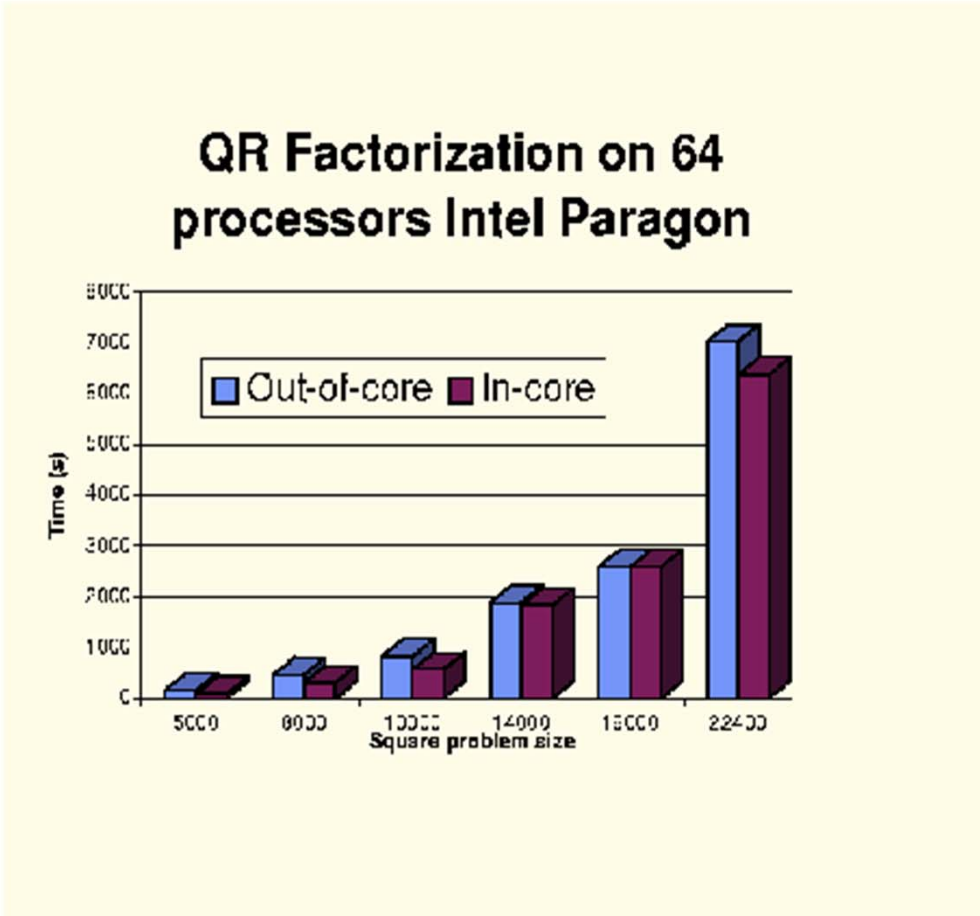
The lower the better →

Time(PDLAHQR)/Time(PDGEMM)				
Machine	Procs	Block Size	N	
			1000	1500
Intel XP/S MP Paragon	16	50	123	97

- **Out-of-core means matrix lives on disk; too big for main memory**
- **Much harder to hide latency of disk**
- **QR much easier than LU because no pivoting needed for QR**
- **Recommend using QR to solve Ax=b for very large matrices**

Out-of-Core Performance Results for Least Squares

- **Prototype code for Out-of-Core extension**
- **Linear solvers based on "Left-looking" variants of LU, QR, and Cholesky factorization**
- **Portable I/O interface for reading/writing ScaLA-PACK matrices**



Other ScaLAPACK Matrix Algorithms

- **ScaLAPACK implementations**
  - Sparse Gaussian Elimination
  - Direct Sparse Solvers
  - Eigenvalue / eigenvector computation
  - Singular Value Decomposition
  - etc…
- **Using the same block cyclic partitioning**
- **Look at links on web page for more info.**

  http://www.netlib.org/scalapack/

Alternative to Keeping Matrix Intact

- **An alternative to keeping the large matrix associated with the global domain intact and using parallel inversion techniques (ie LU, QR, etc) to solve, we could**
  - break up the global matrix into overlaid blocks
    - Each block uses additional (ghost) nodes to reach into adjacent blocks
  - perform a sub-iteration over the overlaid blocks
    - The sub-iteration is necessary to allow the flow of information between blocks
    - Introduces explicitness into algorithm for each sub-block generated.
    - However, allows for large problems without the need to store global matrices in memory
- **Our experience with this approach has been very good leading to good speed-ups and efficiency**

Parallel Multi-Disciplinary Applications

- **Simulations involving multiple disciplines has become more common**
  - Fluid/thermal/structural interaction
  - Design optimization
  - Other sciences
- **Many of these applications are performed on parallel computers**
- **There are different ways to interact between disciplines using parallel computers, some more elegant than others**
  - Tightly coupled using MPI Inter-communicators
  - Tightly coupled using MPI but without Inter-communicators
  - Loosely coupled using interface files

Loosely-Coupled Multi-Disciplinary Simulations

- **Simulations involving multiple disciplines and multiple codes can communicate indirectly using files that are written to and read from for each interaction**
- **CODE1←→ BC FILE ←→ CODE2**
- **Advantages:**
  - Portable to virtually every computer system
  - Not dependent on MPI implementation
- **Disadvantages:**
  - Simulations that require frequent interaction between disciplines can be slow due to high I/O overhead

Tightly Coupled

- **In a tightly-coupled scenario, the various discipline codes communicate directly with each other using MPI Sends and Recvs**
- **There are two techniques depending on the MPI implementation on your computer**
  - With inter-communicators
  - Without inter-communicators
- **MPI allows inter-code communicators to be created in MPI2 and in some versions of MPI1**
  - Really depends on the MPI implementation of mpirun
  - See section starting on page 214 of “Using MPI” by Gropp, Lusk, and Skjellum

Tightly Coupled with MPI Intercommunicators

- **Some implementations of MPI allow for multiple codes to be launched in a given mpirun command using an applications schema file to define the jobs**
- **For instance, LAM MPI allows:**

  mpirun appschema

  Where appschema is a file that contains

  -np 2 –machinefile nodefile code1

  -np 2 –machinefile nodefile code2
- **In this situation, 2 jobs (code1 and code2) are launched under the same MPI_Comm_World communicator which can be broken up into multiple communicators**
- **MPI2 implementations allow for this as well**

Intercommunicator Creation

- **Communicators for CODE1 and CODE2 can be created from the MPI_COMM_WORLD communicator using the MPI_Comm_Split utility:**
- **Example**

```
c
c     initialize mpi for code1
c
      call MPI_Init(IERROR)
      call MPI_Comm_Rank(MPI_COMM_WORLD,MYID,IERROR)
      call MPI_Comm_Size(MPI_COMM_WORLD,NPROCS,IERROR)
      call MPI_Comm_Group(MPI_COMM_WORLD,MPI_GROUP_WORLD,IERROR)
c
      mycode = 1
      call MPI_Comm_Split(MPI_COMM_WORLD,mycode,myid,
     &                    MPI_COMM_CODE1,ierror)
      call MPI_Comm_Rank(MPI_COMM_CODE1,myid,ierror)
      call MPI_Comm_Size(MPI_COMM_CODE1,nprocs,ierror)
      call MPI_Comm_Group(MPI_COMM_CODE1,mygroup,ierror)
```

MPI_Comm_Split

- **MPI_Comm_Split is a Collective Operation that must be seen by all processors that are to be in the new communicator**
- **Fortran:**

```
MPI_Comm_Split(old_communicator, color, key
&                          new_communicator, ierror)
integer old_communicator (handle of original MPI communicator)
integer color (code number or section number)
integer key (processor number to be included in new communicator)
integer new_communicator (handle of new communicator)
integer ierror
```

- **C:**

```
int MPI_Comm_split (MPI_Comm old comm, int color, int key,
MPI_Comm *newcomm)
```

Inter-communicator Creation

- In this case, only CODE1 has access to the processes included in the MPI_COMM_CODE1 communicator
- The processor numbers included in MPI_COMM_WORLD that have color = mycode will be included in MPI_COMM_CODE1 in the same order they were originally
- So, you can create a communicator for each code in the multi-disciplinary simulation

  MPI_COMM_CODE1

  MPI_COMM_CODE2

Intercommunicator Creation

- The processors that communicate to each other internally to CODE1 would then use MPI_COMM_CODE1
- The processors that communicate to each other internally to CODE2 would then use MPI_COMM_CODE2
- But what about the processors of CODE1 that need to talk to the processors of CODE2 or visa-versa?
- For this, we could construct an MPI Inter-communicator or use MPI_Comm_World

Inter-communicator Creation

- **MPI inter-communicators can be created using the MPI_Intercomm_Create utility**

Fortran:

MPI_Intercomm_Create(local_comm, local_leader,
peer_comm,remote_leader,
tag,newintercomm,ierror)

integer local_comm (handle/communicator of your code)

integer local leader (root processor of your code)

integer peer_comm (handle/communicator of other code)

integer remote_leader (root processor of other code)

integer tag (handle of communication between local leader of local comm and remote leader of peer comm)

integer newintercomm (new inter-communicator that can be used for messages between codes)

integer ierror

C:

int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, MPI_Comm peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)

Inter-communicator Creation

- **Example: CODE1 would call**

```
 call MPI_Intercomm_Create(MPI_COMM_CODE1,0,
&                          MPI_COMM_CODE2,0,
&                   tag,MPI_COMM_CODE12,IERROR)
```

- **Then MPI_COMM_CODE12 would be used for messages passed between CODE1 and CODE2**

- **See Example of Code1 and Code2 in the MPMD directory under the Codes directory on smartsite**

Other Inter-communicator Utilities

- **MPI_Comm_Remote_Size can be used to check on the remote (the other) codes size (number of processors)**

Fortran:

MPI_Comm_Remote_Size (comm,size,ierror)

integer comm (handle/communicator of remote code/group)

integer size (number of processors included in comm)

integer ierror

C:

int MPI_Comm_Remote_Size (MPI_Comm comm, int *size)

Tightly Coupled Without Inter-communicators

- For MPI implementations that do not allow for multiple codes to be executed with a single mpirun command, the original codes included in the multi-disciplinary application must be modified to now be subroutines to a master code
- The master code is then responsible for partitioning the processors to the codes
- New MPI communicators can be created using MPI_Comm_Split for each code to simplify bookkeeping of code processors (but not necessary)
- MPI inter-communicators could be created to simplify message passing between codes (but not necessary)

Example

```
      program master
c     this program creates new communicators for two different solvers and then
c       calls the solvers depending on the processor numbers.  4 processors are
c       used in this example
c
      implicit none
      include "mpif.h"
c
      integer :: iproc,iproc2
      integer :: nprocs,myid,mykey
      integer :: nprocs1,myid1,root1,nprocs2,myid2,root2
      integer :: status,ierror
      integer :: MPI_COMM_CODE1,MPI_COMM_CODE2
      integer :: MPI_COMM_CODE12,MPI_COMM_CODE21
      integer :: MPI_GROUP_WORLD
c
c     initialize mpi for master
c
      call MPI_Init(IERROR)
      call MPI_Comm_Rank(MPI_COMM_WORLD,MYID,IERROR)
      call MPI_Comm_Size(MPI_COMM_WORLD,NPROCS,IERROR)
      call MPI_Comm_Group(MPI_COMM_WORLD,MPI_GROUP_WORLD,IERROR)
      call MPI_Barrier(MPI_COMM_WORLD,ierror)
```

Example

```
      if(myid<=1) then
c
c       create the flowsolver communicator with 2 processors
c
        mykey = 1
        call MPI_Comm_Split(MPI_COMM_WORLD,mykey,myid,
     &                      MPI_COMM_CODE1,ierror)
      else
c
c       create the heatconduction communicator with remainder of processors
c
        mykey = 2
        call MPI_Comm_Split(MPI_COMM_WORLD,mykey,myid,
     &                      MPI_COMM_CODE2,ierror)
      endif
```

Example

```
c
      if(myid<=1) then
        call MPI_Comm_Rank(MPI_COMM_CODE1,myid1,ierror)
        call MPI_Comm_Size(MPI_COMM_CODE1,nprocs1,ierror)
        if(myid1==0) root1 = myid1
      else
        call MPI_Comm_Rank(MPI_COMM_CODE2,myid2,ierror)
        call MPI_Comm_Size(MPI_COMM_CODE2,nprocs2,ierror)
        if(myid2==0) root2 = myid2
       endif
c
      call MPI_Barrier(MPI_COMM_WORLD,ierror)
c
c     now partition out processors to each solver
c
      if(myid<=1) then
        call flowsolver
      else
        call heatconduction
      endif
```

During This Course, You Have Learned

- **How to Program Using Fortran 95/03**
  - Paying attention to compilers and single-processor performance
  - Issues related to data locality, cache, operation speed, cache and page misses
- **Parallel Programming Platforms**
  - Architecture, inter-connects, memory constructs, cache
  - Limitations to system performance
  - Analytical models to parallel platforms and programs
- **Multi-block simulations**
  - Data structures for multi-block simulations
  - How to program a multi-block simulation
- **Spatial domain decomposition**
- **Processor mapping**
  - Load balancing, existing libraries (Metis, Jostle, etc.)
- **Parallel interfaces**
  - MPI for distributed-memory parallel computing
    - Basic and advanced operations, MPI derived datatypes, MPI topological operators
  - OpenMP for shared-memory parallel computing on CPUs
  - CUDA for shared-memory parallel computing on GPUs
  - How to program a multi-block distributed/shared parallel simulation
- **Basic linear algebra solvers for solving systems involving matrices**
  - Existing libraries

Many More Things to Learn

- **You now have the basic information to *take advantage of parallel computers* in your engineering analyses**

- **Yet there are many more advanced techniques and algorithms to learn:**
  - Parallel “direct” solution methods
  - Parallel Krylov solvers for iterative solutions to linear systems
    - Ritz-Galerkin, GMRES, Petrov-Galerkin, Conjugate Gradient (CG), Bi-CG
  - Parallel preconditioning systems
  - Parallel linear eigenvalue and eigenvector solution techniques
  - Parallel multi-grid and Schwartz techniques
  - Data-structures for unstructured-grids
  - Parallel computing for unstructured- and hybrid-grid systems
  - Parallel post-processing and visualization