Lecture 19 – Parallel Direct Matrix Methods

- **Motivation for parallel solution of linear algebra problems using direct methods**
- **Brief discussion of existing sequential methods for most relevant operations:**
  - Gaussian Elimination / Matrix Factorization
  - Matrix-vector and matrix-matrix multiplication
  - Eigenvalue / eigenvector calculation
- **Parallel algorithms for these matrix operations with complexity estimates**
- **Existing parallel linear algebra subroutines and libraries (PBLAS, ScaLAPACK, ATLAS, etc)**
- **Similar discussions for these operations performed on *sparse* matrices**

Motivation: Dense Linear Algebra

- **Many problems in computational physics can be reduced to the form**

$$Ax = b$$

- **This is true whether the original problems are *linear* or *non-linear* (with appropriate linearization), whether the problems are 1D, 2D, 3D, and whether an approximate factorization has been performed or not**

Motivation: Dense Linear Algebra

- **In such cases, we may be solving for a subset of the problem every time we solve the equation A x = b, since A can be written as**

$$Ax = (A_I A_J A_K)x = b$$

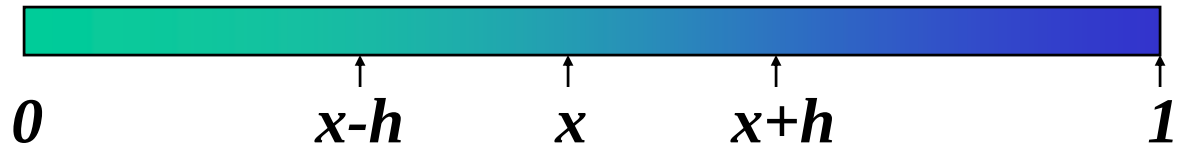
with some factorization error

- **It is true that these matrices are typically *banded*, and therefore, the cost of full factorization is not necessary.**

Motivation: Continuous Variables, Continuous Parameters

- **Examples of such systems include**
  - Heat flow: Temperature(position, time)
  - Diffusion: Concentration(position, time)
  - Electrostatic or Gravitational Potential: Potential(position)
  - Fluid flow: Velocity,Pressure,Density(position,time)
  - Quantum mechanics: Wave-function(position,time)
  - Elasticity: Stress,Strain(position,time)

Example: The Heat Conduction Equation



- **Consider the heat conduction equation again:**
  - A bar of uniform material, insulated except at ends
  - Let $T(x,t)$ be the temperature at position x at time t
  - Heat travels from x-h to x+h at rate proportional to:

$$\frac{\partial T(x,t)}{\partial t} = C\frac{\left[T(x-h,t)-T(x,t)\right]/h-\left[T(x,t)-T(x+h,t)\right]/h}{h}$$

  - As $h \to 0$, we get the heat equation:

$$\frac{\partial T(x,t)}{\partial t} = C\frac{\partial^2 T(x,t)}{\partial x^2}$$

Implicit Solution

- As with many (stiff) ODEs, need an implicit method
- This turns into solving the following equation

$$\frac{T_i^{n+1} - T_i^n}{\Delta t} = C\frac{\left[T_{i-1}^{n+1} - T_i^{n+1}\right]/h - \left[T_i^{n+1} - T_{i+1}^{n+1}\right]/h}{h}$$

$$-T_{i-1}^{n+1} + \left(\frac{h^2}{C\Delta t} + 2\right)T_i^{n+1} - T_{i+1}^{n+1} = \frac{h^2}{C\Delta t}T_i^n$$

$$\left[b\bar{\bar{I}} + \bar{\bar{A}}\right]\bar{T}^{n+1} = b\bar{T}^n$$

- Here, *I* is the identity matrix, $b = h^2/C\Delta t$, and *A* is:

$$A = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & -1 & 2 & -1 & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

Graph and "stencil"

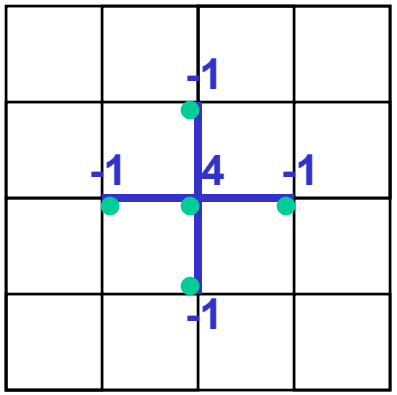
- I.e., essentially solving Poisson's equation in 1D

2D Implicit Method

- Similar to the 1D case, but the matrix *A* is now

$$A = \left(\begin{array}{ccc:ccc:ccc} 4 & -1 & & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ \hdashline -1 & & & 4 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ \hdashline & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{array}\right)$$

Graph and "stencil"



- Multiplying by this matrix (as in the explicit case) is simply nearest neighbor computation on 2D grid
- To solve this system, there are several techniques

Algorithms for 2D Poisson Equation with N Unknowns

Algorithm	Serial Ops.	Memory	#Procs
• Dense LU	N^3	N^2	N^2
• Band LU	N^2	$N^{3/2}$	N
• Jacobi	N^2	N	N
• Explicit Inv.	N	N	N
• Conj.Grad.	$N^{3/2}$	N	N
• RB SOR	$N^{3/2}$	N	N
• Sparse LU	$N^{3/2}$	N*log N	N
• FFT	N*log N	N	N
• Multigrid	N	N	N
• Lower bound	N	N	N

Building Blocks in Linear Algebra

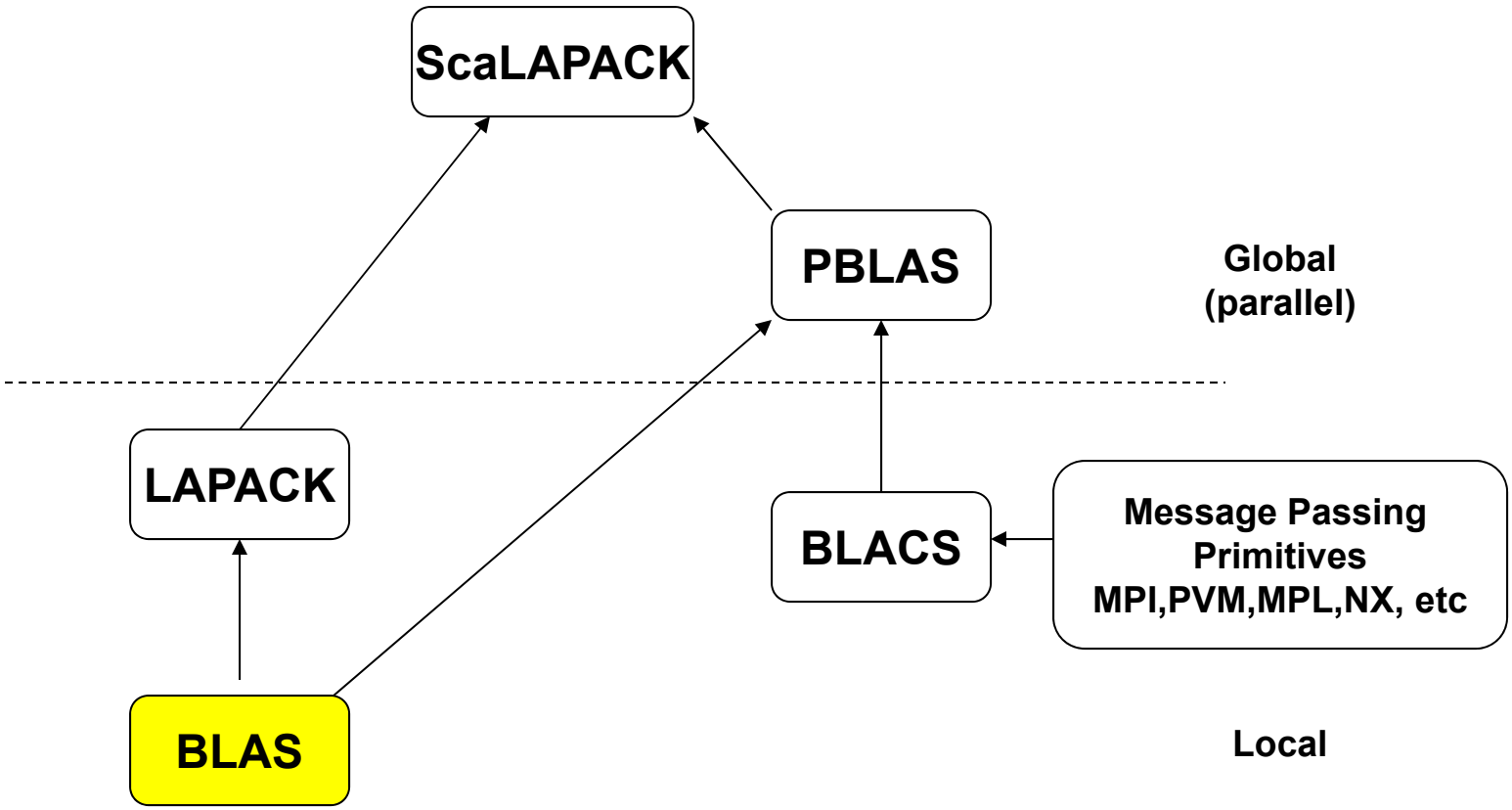
- **BLAS (Basic Linear Algebra Subprograms) created / defined in 1979 by Lawson et. al**
- **BLAS intends to modularize problems in linear algebra by identifying typical operations present in complex algorithms in linear algebra, and defining a standard interface to them**
- **This way, hardware vendors can optimize their own version of BLAS and allow users' programs to run efficiently with simple recompilation**
- **Optimized BLAS implementations are usually hand-tuned (and coded in assembly language)**
- **The BLAS library compiled with PGI is located on hpc1 at /opt/pgi/linux86-64/15.4/lib/libblas.a**

http://www.netlib.org/blas

Building Blocks in Linear Algebra

- **BLAS routines have to be simple enough that high levels of optimization can be obtained**
- **BLAS routines have to be general enough so that complex algorithms can be constructed as sequences of calls to these basic routines**
- **Others (LINPACK, LAPACK, EISPACK, etc.) have followed suit and have tried to do a similar job for a variety of linear algebra problems**

Software Library Hierarchy



Building Blocks in Linear Algebra

- **BLAS advantages:**
  - Robustness: BLAS routines are programmed with robustness in mind. Various exit conditions can be diagnosed from the routines themselves, overflow is predicted, and general pivoting algorithms are implemented
  - Portability: the calling API is fixed; hardware vendors optimize behind-the-scenes
  - Readability: since the names of BLAS routines are fairly common, one knows exactly what a program is doing by reading the routine name and source code; auto-documentation

BLAS Level 1 Routines

- **Perform low level functions (typically operations between vectors like dot products, sums, etc.)**

- **4 or 5 letter names preceded by s, d, c, z to indicate precision type. For example, DAXPY is a double precision addition of a vector with another one multiplied by a scalar (Ax+Y)**

- **Typical operations are O(n), where n is the length of the vectors being operated on**

- **Large ratio of floating point operations to memory loads and stores prevents high Mflop rating of these routines in most computers**

BLAS Level 1 Routines

- **Typical operations**

$$y \leftarrow \alpha x + y$$

$$x \leftarrow \alpha x$$

$$dot \leftarrow x^T y$$

$$asum \leftarrow \|re(x)\| + \|im(x)\|$$

$$nrm2 \leftarrow \|x\|_2$$

$$amax \leftarrow 1^{st}\, k \ni |re(x_k)| + |img(x_k)| = \|x\|_\infty$$

BLAS Level 1 Routines

- **One of the most basic operations, a matrix-vector multiply, can actually be done with a sequence of n SAXPY operations**
  - but the resulting vector is stored to memory and retrieved from it at every step,
  - but it could have remained in memory for the actual computation.

- **BLAS Level 2 routines add functionality to help out in this situation**

BLAS Level 2 Routines

- **Level 1 BLAS routines do not have enough granularity to achieve high performance: reuse of registers must occur because of high cost of memory accesses and limitations in current chip architectures**

- **Optimization at least at the level of matrix-vector operations is necessary. Level 1 disallows this by hiding details from the compiler.**

- **Level 2 BLAS includes these kinds of operations which typically involve *O(m n)* operations, where the matrices involved have size *m x n***

BLAS Level 2 Routines

- **Typical operations involve:**

$$y = \alpha A x + \beta y$$

$$y = \alpha A^T x + \beta y$$

$$y = Tx$$

$$y = T^T x$$

$$x = T^{-1} x$$

- **as well as rank-1 and rank-2 updates to a matrix (optimization)**

BLAS Level 2 Routines

- **Additional operations for banded, Hermitian, triangular, etc. matrices are also available**
- **Efficiency of implementations can be increased in this way, but there are drawbacks for cache-based architectures which still want to reuse memory as much as possible.**
- **Level 3 BLAS addresses this problem**

BLAS Level 3 Routines

- **Sometimes it is preferable to decompose matrices into blocks to perform various operations on a matrix-matrix basis**

- **Data reuse is enhanced in this way**

- **Typically obtain $O(n^3)$ operations with $O(n^2)$ data references (similar to granularity surface-to-volume effect discussed earlier)**

- **Two opportunities for parallelism:**
  - operations on distinct blocks may be done in parallel
  - operations within a block may have loop-level parallelism

BLAS Level 3 Routines

- **Typical operations involve matrix-matrix products**

$$C = \alpha AB + \beta Y$$

$$C = \alpha AA^T + \beta C$$

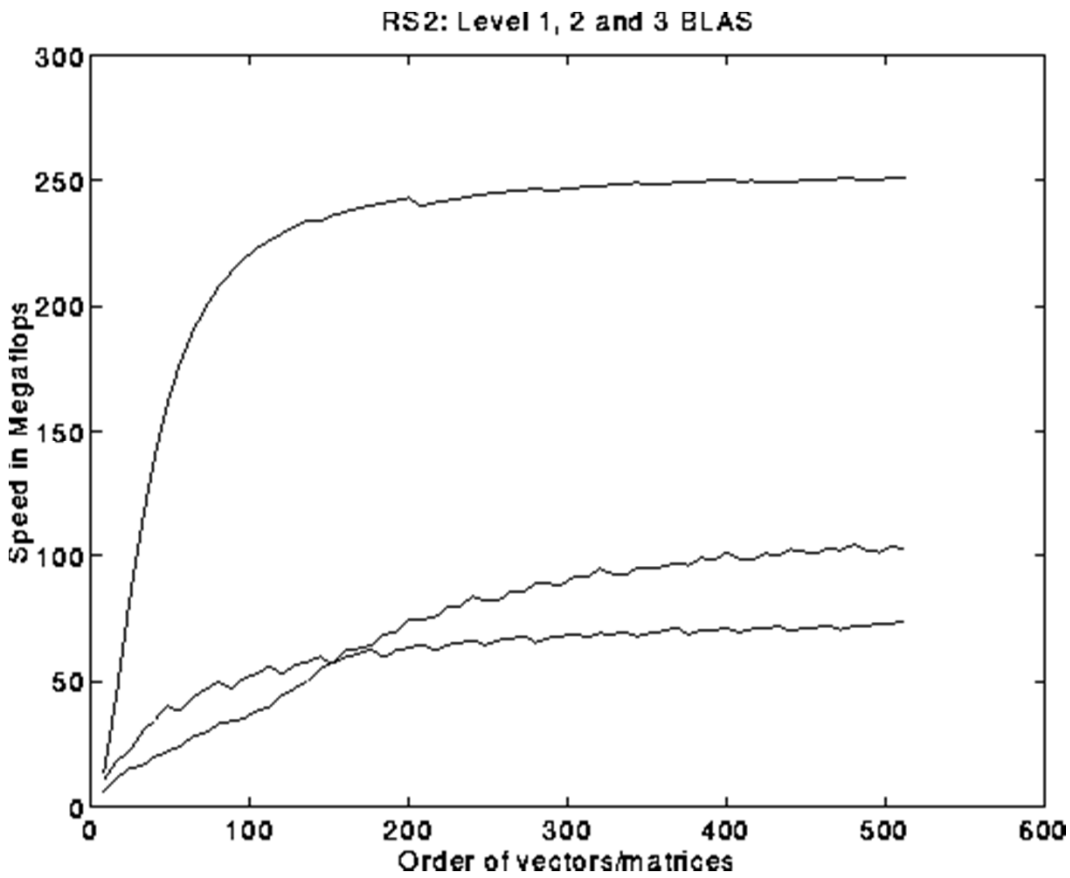
$$B = \alpha TB$$

$$B = \alpha T^{-1} B$$

as well as rank-k updates and solutions of systems involving triangular matrices

- **Better performance is achieved**

BLAS Level 3 Routines



Peak
BLAS 3

BLAS 2

BLAS 1

From Dongarra et al.

Matrix Problem Solution, Ax=b

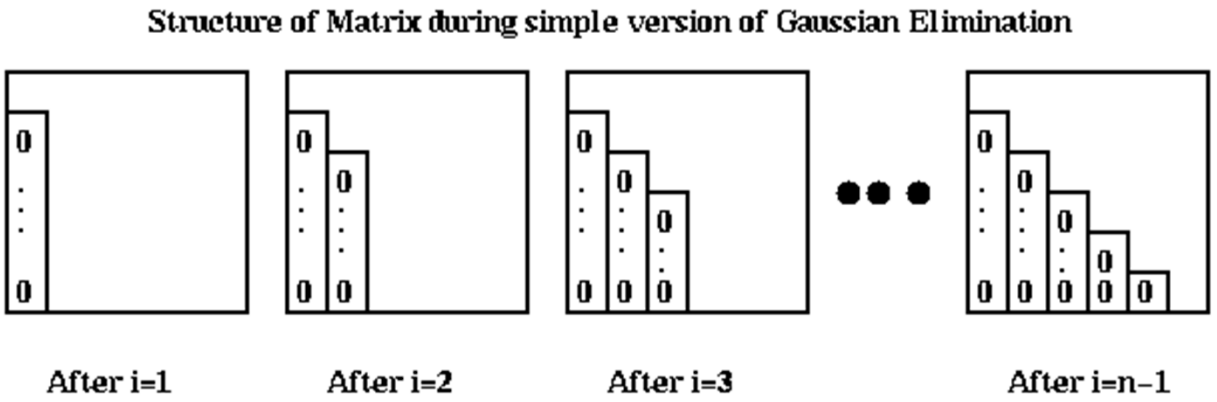
- **The main steps in the solution process are**
  - Fill: computing the matrix elements of A
  - Factor: factoring the dense matrix A
  - Solve: solving for one or more right hand sides, b

Task	Work	Parallelism	Parallel SpeedUp
Fill	**$O(n^2)$**	**embarrassing**	**low**
→ **Factor**	**$O(n^3)$**	**moderately diff.**	**very high**
Solve	**$O(n^2)$**	**moderately diff.**	**high**
Field Calc.	**O(n)**	**embarrassing**	**high**

Review of Gaussian Elimination (GE) for Solving Ax=b

- **Add multiples of each row to later rows to make A upper triangular**
- **Solve resulting triangular system Ux = c by back substitution**

```
… for each column i
… zero it out below the diagonal by adding multiples of row i to later rows
for i = 1 to n-1
    … for each row j below row i
    for j = i+1 to n
        … add a multiple of row i to row j
        for k = i to n
            A(j,k) = A(j,k) - (A(j,i)/A(i,i)) * A(i,k)
```



Refine GE Algorithm (1)

- **Initial Version**

```
… for each column i
… zero it out below the diagonal by adding multiples of row i to later rows
for i = 1 to n-1
    … for each row j below row i
    for j = i+1 to n
        … add a multiple of row i to row j
        for k = i to n
            A(j,k) = A(j,k) - (A(j,i)/A(i,i)) * A(i,k)
```

- **Remove computation of constant A(j,i)/A(i,i) from inner loop**

```
for i = 1 to n-1
    for j = i+1 to n
        m = A(j,i)/A(i,i)
        for k = i to n
            A(j,k) = A(j,k) - m * A(i,k)
```

Refine GE Algorithm (2)

- **Last version**

```
for i = 1 to n-1
    for j = i+1 to n
        m = A(j,i)/A(i,i)
        for k = i to n
            A(j,k) = A(j,k) - m * A(i,k)
```

- **Don't compute what we already know: zeros below diagonal in column i**

```
for i = 1 to n-1
    for j = i+1 to n
        m = A(j,i)/A(i,i)
        for k = i+1 to n
            A(j,k) = A(j,k) - m * A(i,k)
```

Refine GE Algorithm (3)

- **Last version**

```
for i = 1 to n-1
    for j = i+1 to n
        m = A(j,i)/A(i,i)
        for k = i+1 to n
            A(j,k) = A(j,k) - m * A(i,k)
```

- **Store multipliers m below diagonal in zeroed entries for later use**

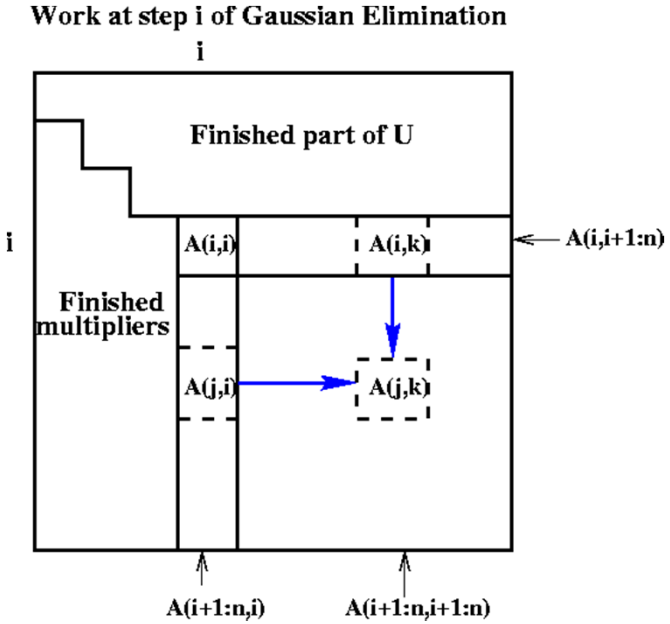
```
for i = 1 to n-1
    for j = i+1 to n
        A(j,i) = A(j,i)/A(i,i)
        for k = i+1 to n
            A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

Refine GE Algorithm (4)

- **Last version**

```
for i = 1 to n-1
    for j = i+1 to n
        A(j,i) = A(j,i)/A(i,i)
        for k = i+1 to n
            A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

- **Express using array (matrix) operations (BLAS)**




```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n )
            - A(i+1:n , i) * A(i , i+1:n)
```

What GE Really Computes

```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)                          ← M
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n ) - A(i+1:n , i) * A(i , i+1:n)    ← U
```

- **Call the strictly lower triangular matrix of multipliers M, and let L = I+M**
- **Call the upper triangle of the final matrix U**
- ***Lemma (LU Factorization):* If the above algorithm terminates (does not divide by zero) then A = L*U**
- **Solving A*x=b using GE**
  - Factorize A = L*U using GE (cost = 2/3 n^3 flops)
  - Solve L*y = b for y, using substitution (cost = n^2 flops)
  - Solve U*x = y for x, using substitution (cost = n^2 flops)
- **Thus A*x = (L*U)*x = L*(U*x) = L*y = b as desired**

Substitution to Compute y

- **L*y = b**
- **So y = L^{-1} b**
- **Since L is a lower diagonal matrix with 1's on the diagonal, L^{-1} is easy to find**

$$L = \begin{bmatrix} 1 & 0 & 0 & . & 0 \\ l_{21} & 1 & 0 & . & . \\ . & . & 1. & . & . \\ . & . & . & 1. & 0 \\ l_{n1} & l_{n2} & . & l_{nn-1} & 1 \end{bmatrix}$$

by multiplying row 1 of L and b by l_{21} and subtracting from row 2, etc.

- **This leads to L^{-1} which can be used to find y**

Substitution to Compute x

- **Likewise, U*x = y**
- **So x = U^{-1} y**
- **Since U is an upper diagonal matrix, U^{-1} is easy to find**

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & . & u_{1n} \\ 0 & u_{22} & u_{23} & . & u_{2n} \\ . & . & u_{33}. & . & . \\ . & . & . & . & u_{n-1n} \\ 0 & 0 & . & 0 & u_{nn} \end{bmatrix}$$

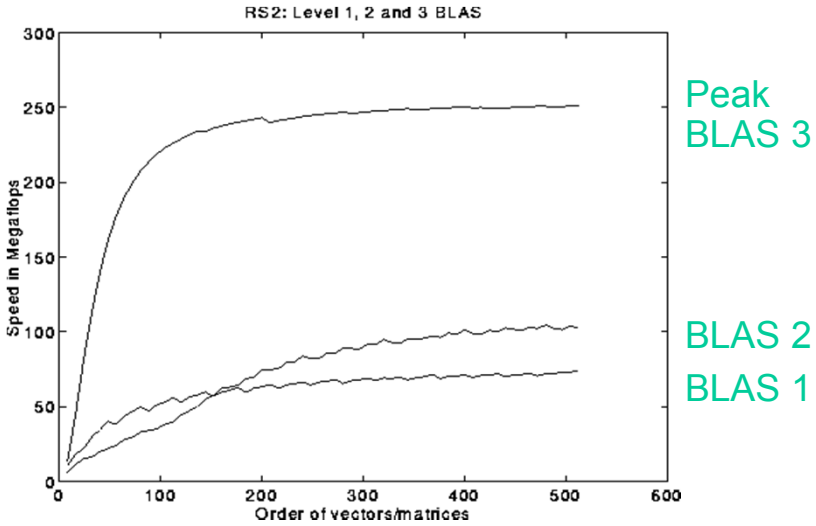
by multiplying row n of U and y by u_{n-1n}/u_{nn} and subtracting from row n-1, etc.

- **This leads to U^{-1} which can be used to find x**

Problems with Basic GE Algorithm

- **What if some A(i,i) is zero? Or very small?**
  - Result may not exist, or be "unstable", so need to pivot
- **Current computation consists of all BLAS 1 or BLAS 2, but we know that BLAS 3 (matrix multiply) is fastest**

```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)          … BLAS 1 (scale a vector)
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n )  … BLAS 2 (rank-1 update)
                     - A(i+1:n , i) * A(i , i+1:n)
```



Example: Problem with Gaussian Elimination of Ax=b

$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ fails completely, even though A is "easy"

Illustrate problems in 3-decimal digit arithmetic:

$A = \begin{pmatrix} 1e\text{-}4 & 1 \\ 1 & 1 \end{pmatrix}$ and $b = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ correct answer to 3 places is $x = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$

Result of LU decomposition is

$L = \begin{pmatrix} 1 & 0 \\ fl(1/1e\text{-}4) & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1e4 & 1 \end{pmatrix}$ … No roundoff error yet

$U = \begin{pmatrix} 1e\text{-}4 & 1 \\ 0 & fl(1\text{-}1e4*1) \end{pmatrix} = \begin{pmatrix} 1e\text{-}4 & 1 \\ 0 & -1e4 \end{pmatrix}$ … Error in 4th decimal place due to 3-decimal arithmetic

Check if $A = L*U = \begin{pmatrix} 1e\text{-}4 & 1 \\ 1 & 0 \end{pmatrix}$ … (2,2) entry entirely wrong

Algorithm "forgets" (2,2) entry, gets same L and U for all |A(2,2)|<5

Numerical instability

Computed solution x totally inaccurate

Cure: Pivot (swap rows of A) so entries of L and U bounded

Gaussian Elimination with Partial Pivoting (GEPP)

Partial Pivoting: swap rows so that each multiplier |L(i,j)| = |A(j,i)/A(i,i)| <= 1

```
for i = 1 to n-1
    find and record k where |A(k,i)| = max{i <= j <= n} |A(j,i)|
          … i.e. largest entry in rest of column i
    if |A(k,i)| = 0
        exit with a warning that A is singular, or nearly so
    elseif  k /= i
        swap rows i and k of A
    end if
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)        … each quotient lies in [-1,1]
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n ) - A(i+1:n , i) * A(i , i+1:n)
```

Lemma: This algorithm computes A = P*L*U, where P is a permutation matrix

Since each entry of |L(i,j)| <= 1, this algorithm is considered numerically stable

For details see LAPACK code at www.netlib.org/lapack/single/sgetf2.f

Summary of BLAS Routines

- **You have seen an example of how a typical matrix operation (an important one) can be reduced to using lower level BLAS routines that would have been optimized for a given hardware platform**

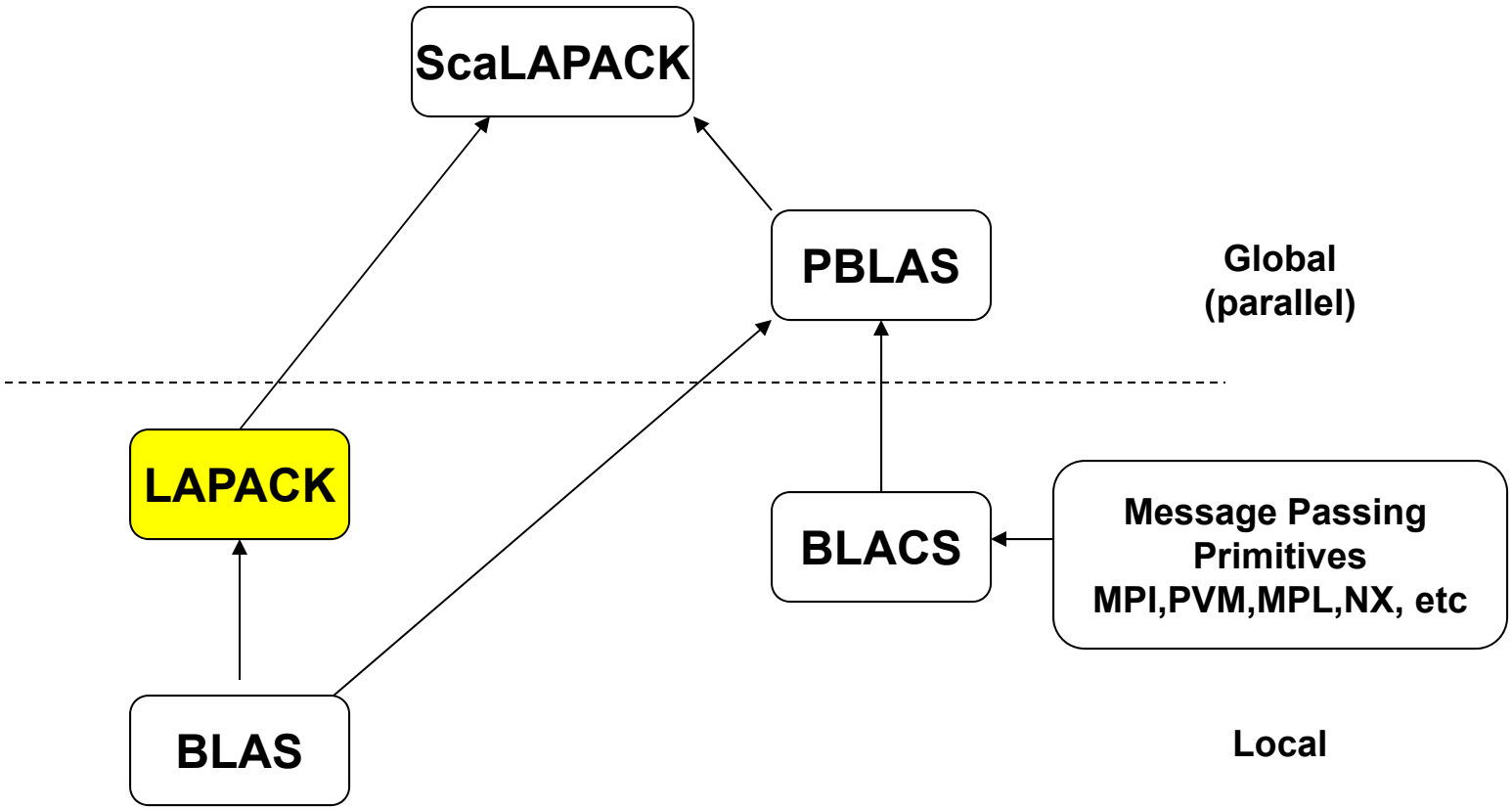
- **Any other matrix operation that you may think of (matrix-matrix multiply, Cholesky factorization, Householder method, etc) can be constructed from BLAS subprograms in a similar fashion and in fact have been constructed in the package called LAPACK**
  - Lapack libraries are also on hpc1 at: /opt/pgi/linux86-64/15.4/lib/liblapack.a

- **Note that only Level 1 and 2 BLAS routines have been used in LU decomposition.**

Overview of LAPACK and ScaLAPACK

- **LAPACK: Standard library for dense/banded linear algebra**
  - Linear systems: A*x=b
  - Least squares problems: $\min_x \| A*x-b \|_2$
  - Eigenvalue problems: $Ax = \lambda x$, $Ax = \lambda Bx$
  - Singular value decomposition (SVD): $A = U\Sigma V^T$
- **Algorithms reorganized to use BLAS3 as much as possible**
- **Basis of math libraries on many computers**
- **Many algorithmic innovations remain**

Software Library Hierarchy

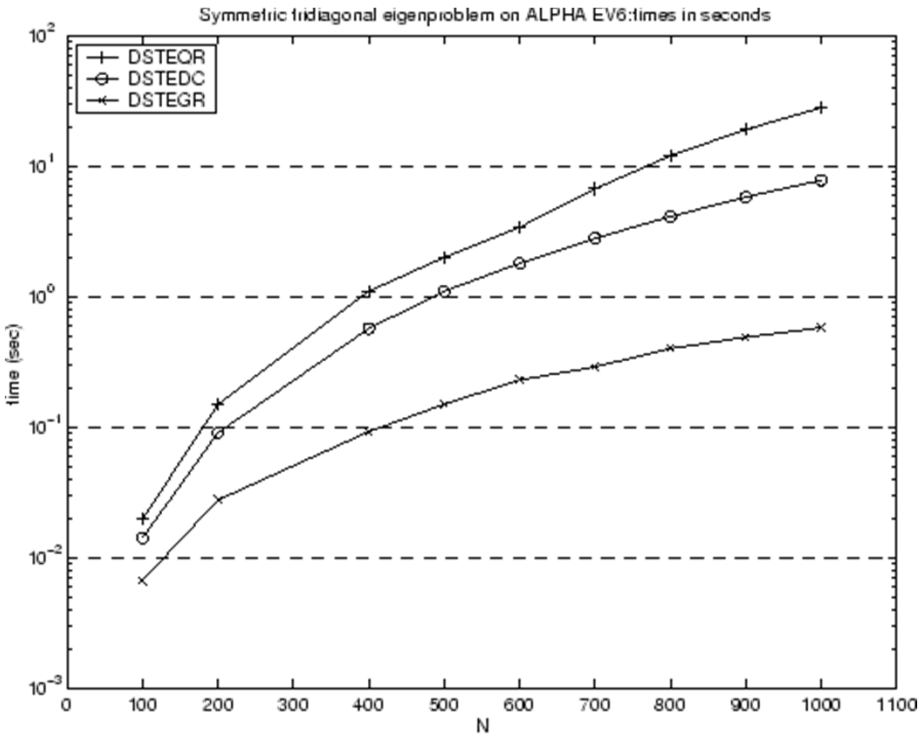


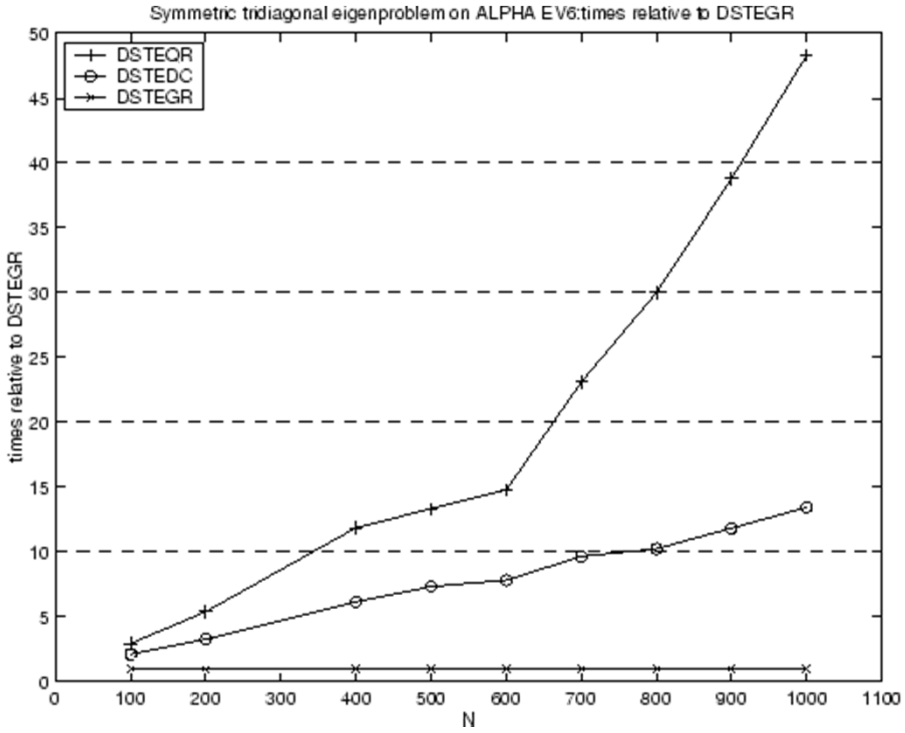
LAPACK

- **“LAPACK was originally targeted to achieve good performance on *single-processor vector machines* and on *shared memory multiprocessor machines* with a modest number of powerful processors.**
- **Since the start of the project, another class of machines has emerged for which LAPACK software is equally well-suited--the *high-performance ``super-scalar'' workstations*.**
- **LAPACK is intended to be used across the whole spectrum of modern computers, but when considering performance, the emphasis is on machines at the more powerful end of the spectrum.”**

Benchmarks: http://www.netlib.org/lapack/lug/node60.html

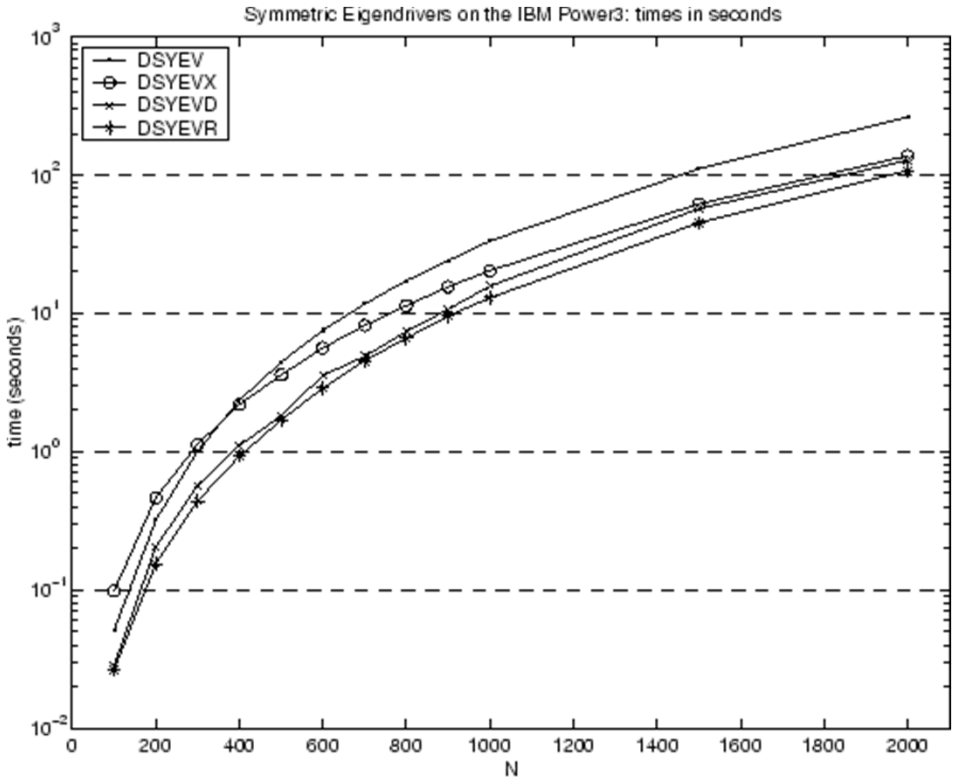
LAPACK Benchmarks – Eigenvalue and Eigenvectors of Symmetric Tridiagonal Matrix (Alpha)

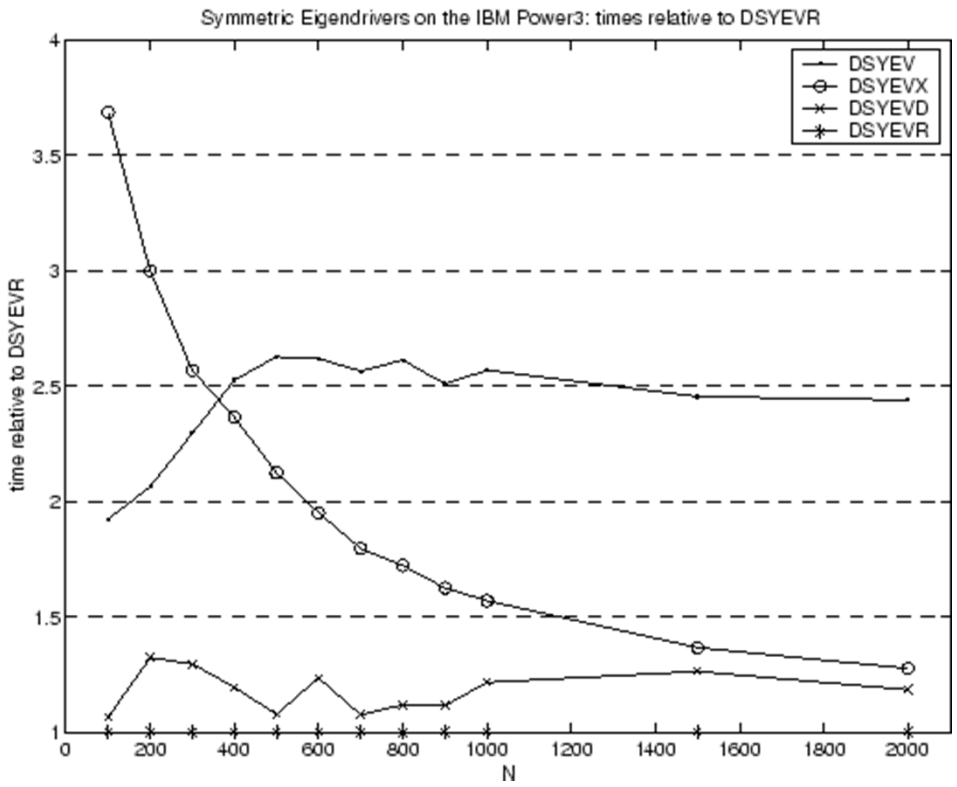




Timings of routines for computing all eigenvalues and eigenvectors of a symmetric tridiagonal matrix. The left graph shows times in seconds on a Compaq AlphaServer DS-20. The right graph shows times relative to the fastest routine DSTEGR, which appears as a horizontal line at 1.

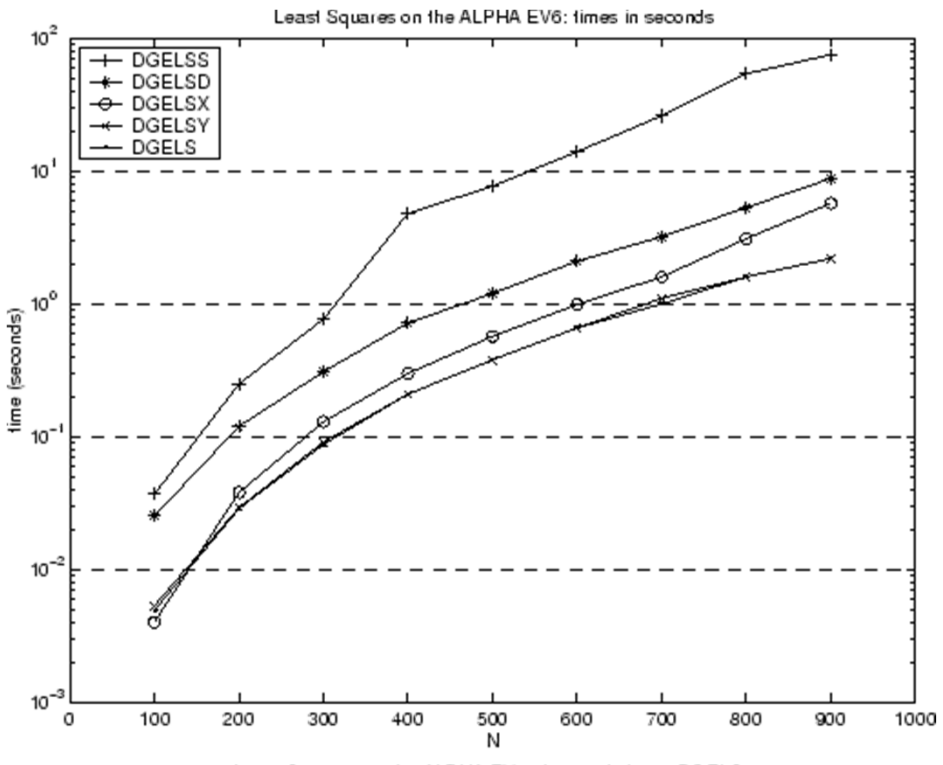
LAPACK Benchmarks – Eigenvalue and Eigenvectors of Symmetric Tridiagonal Matrix (IBM P3)

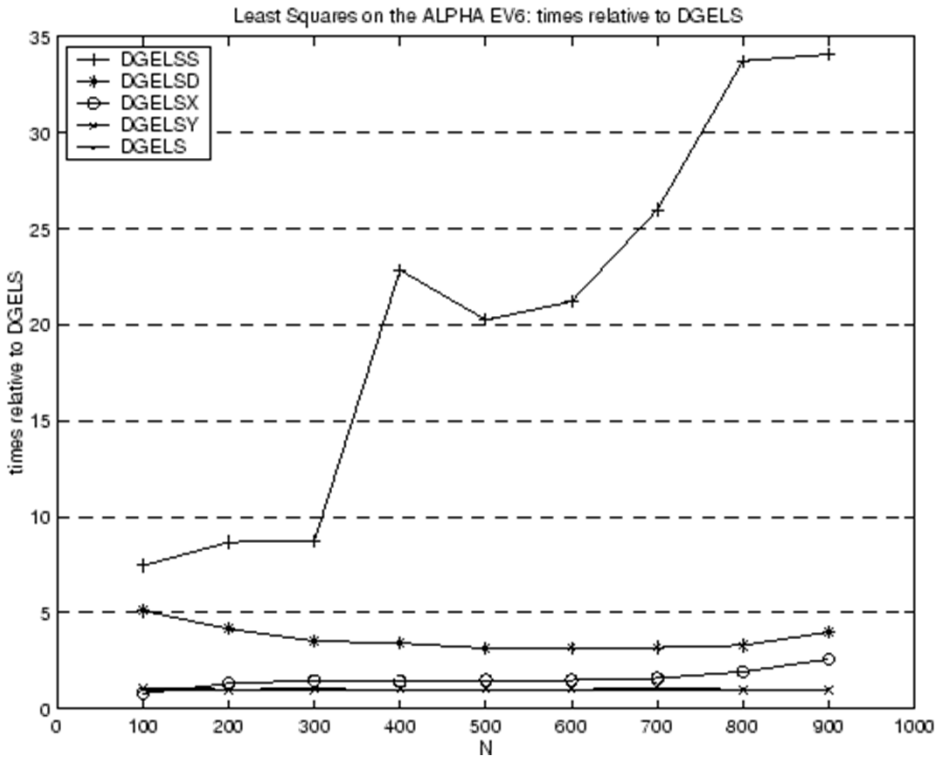




Timings of driver routines for computing all eigenvalues and eigenvectors of a dense symmetric matrix. The left graph shows times in seconds on an IBM Power3. The right graph shows times relative to the fastest routine DSYEVR, which appears as a horizontal line at 1.

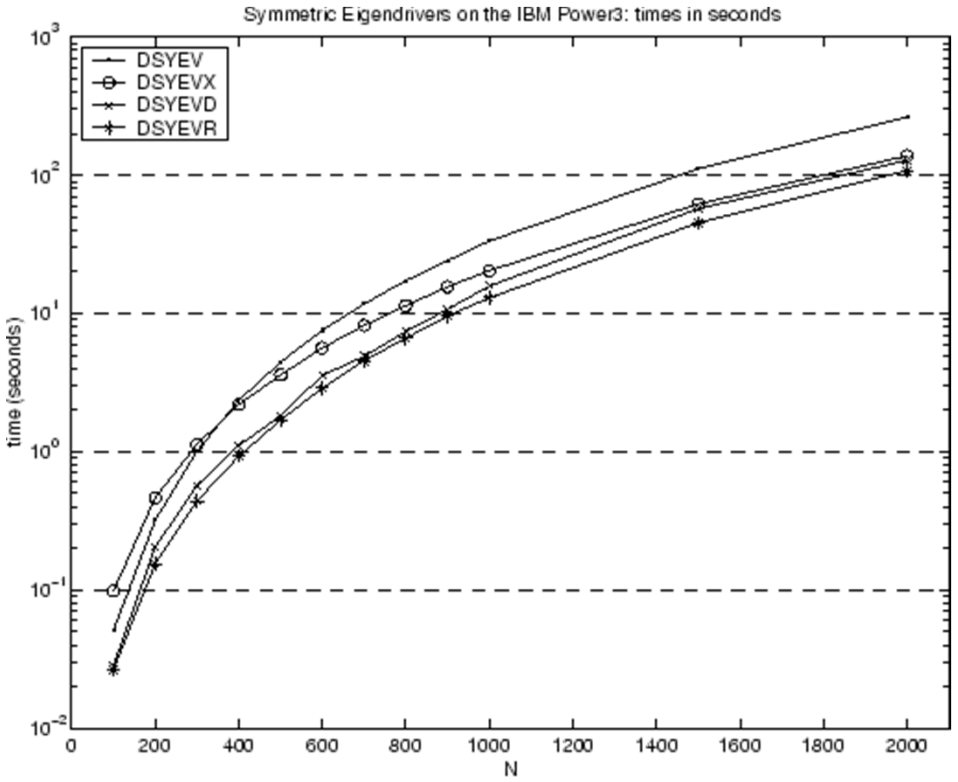
LAPACK Benchmarks – Least Squares (Alpha)

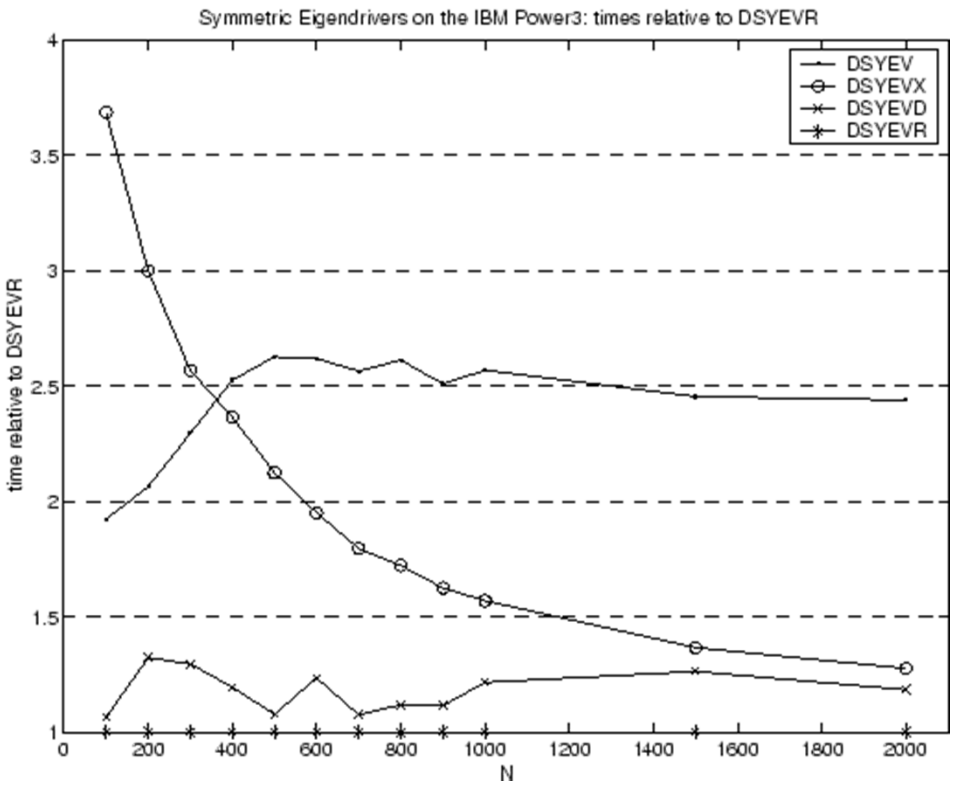




Timings of driver routines for the least squares problem. The left graph shows times in seconds on a Compaq AlphaServer DS-20. The right graph shows times relative to the fastest routine DGELS, which appears as a horizontal line at 1.

LAPACK Benchmarks – Least Squares (IBM P3)



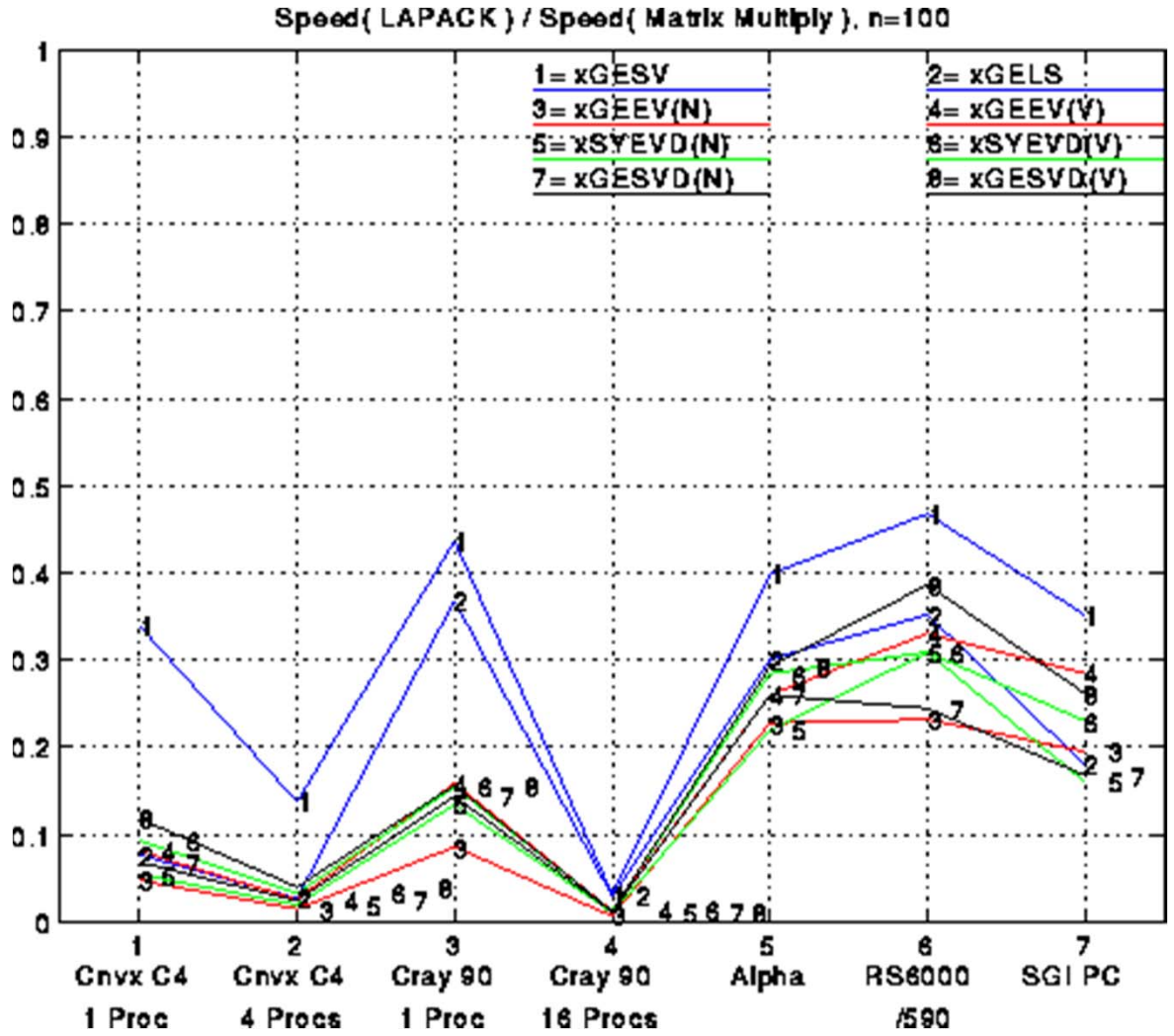


Timings of driver routines for computing all eigenvalues and eigenvectors of a dense symmetric matrix. The left graph shows times in seconds on an IBM Power3. The right graph shows times relative to the fastest routine DSYEVR, which appears as a horizontal line at 1.

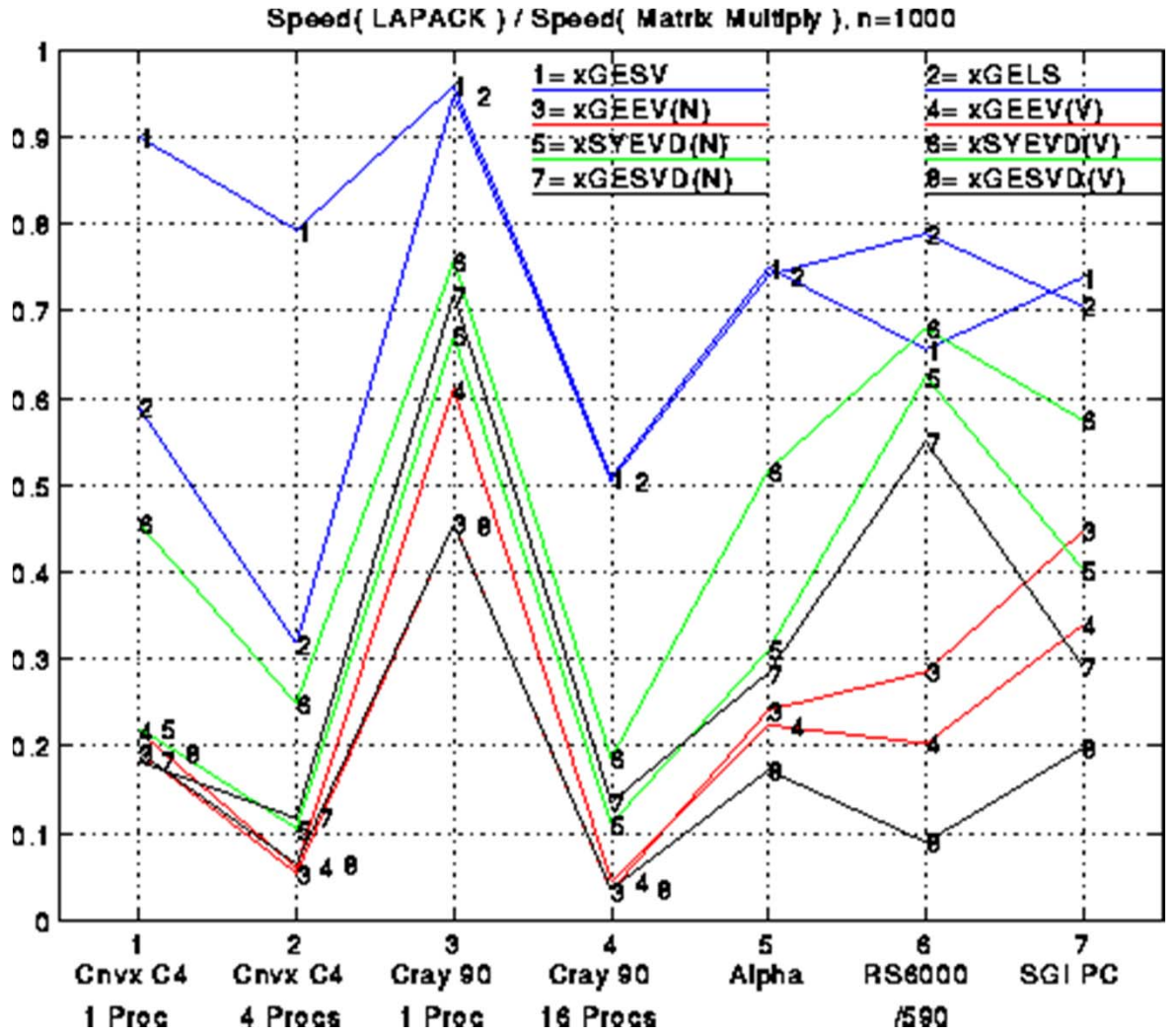
LAPACK Benchmarks – Matrix-Vector and Matrix-Matrix Multiply

	DGEMV				DGEMM			
	Values of ***n=m=k***							
	100		**1000**		**100**		**1000**	
	Time	**Mflops**	**Time**	**Mflops**	**Time**	**Mflops**	**Time**	**Mflops**
Dec Alpha Miata	.0151	66	27.778	36	.0018	543	1.712	584
Compaq AlphaServer DS-20	.0027	376	8.929	112	.0019	522	2.000	500
IBM Power 3	.0032	304	2.857	350	.0018	567	1.385	722
IBM PowerPC	.0435	23	40.000	25	.0063	160	4.717	212
Intel Pentium II	.0075	134	16.969	59	.0031	320	3.003	333
Intel Pentium III	.0071	141	14.925	67	.0030	333	2.500	400
SGI O2K (1 proc)	.0046	216	4.762	210	.0018	563	1.801	555
SGI O2K (4 proc)	5.000	0.2	2.375	421	.0250	40	0.517	1936
Sun Ultra 2 (1 proc)	.0081	124	17.544	57	.0033	302	3.484	287
Sun Enterprise 450 (1 proc)	.0037	267	11.628	86	.0021	474	1.898	42 527

Performance of LAPACK (n=100)



Performance of LAPACK (n=1000)



LAPACK

- **Need to devise algorithms that can make use of Level 3 BLAS (matrix-matrix) routines, for several reasons:**
  - Level 3 routines are known to run much more efficiently due to larger ratio of computation to memory references/communication
  - Parallel algorithms on distributed memory machines will require that we decompose the original matrix into blocks which reside in each processor
  - Parallel algorithms will require that we minimize the surface-to-volume ratio of our decompositions, and blocking becomes the natural approach.

Converting BLAS2 to BLAS3 in Gaussian Elimination with Partial Pivoting (GEPP)

- **Blocking**
  - Used to optimize matrix-multiplication
  - Harder here because of data dependencies in GEPP
- **Delayed Updates**
  - Save updates to "trailing matrix" from several consecutive BLAS2 updates
  - Apply many saved updates simultaneously in one BLAS3 operation
- **Same idea works for much of dense linear algebra**
  - Open questions remain
- **Need to choose a block size b**
  - Algorithm will save and apply b updates
  - b must be small enough so that active sub-matrix consisting of b columns of A fits in cache
  - b must be large enough to make BLAS3 fast

Parallel Algorithms for Dense Matrices

- **All discussion that follows is applicable to *dense or full matrices only*. Square matrices discussed, but arguments are valid for rectangular matrices as well**

- **Typical parallelization steps**
  - Decomposition: identify parallel work and partitioning
  - Mapping: which processors execute which portion of the work
  - Assignment: load balance work among threads/processors
  - Organization: communication and synchronization

Parallel Algorithms for Dense Matrices

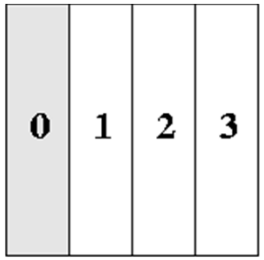
- **The thread/processor that owns a given portion of a matrix is responsible for doing all of the computation that involves that portion of the matrix**

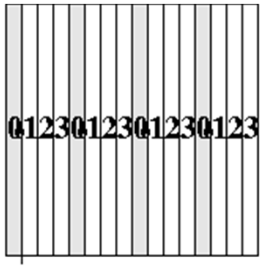
- **This is the sensible thing to do since communication is minimized**

- **The question is: how should we subdivide a matrix so that parallel efficiency is maximized? There are various options.**

Different Data Layouts for Parallel GE (on 4 procs)

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

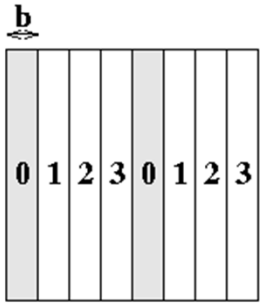


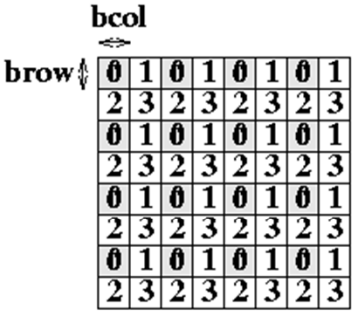


Load balanced, but can't easily use BLAS2 or BLAS3

1) Column Blocked Layout 2) Column Cyclic Layout

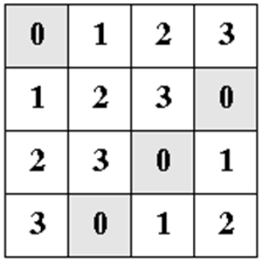
Can trade load balance and BLAS2/3 performance by choosing b, but factorization of block column is a bottleneck





The winner!

3) Column Block Cyclic Layout 4) Row and Column Block Cyclic Layout



Complicated addressing

5) Block Skewed Layout

Parallel Matrix Transpose - Block Partition

- **Start with simple example: obtain the transpose of a matrix, *A* defined as:**

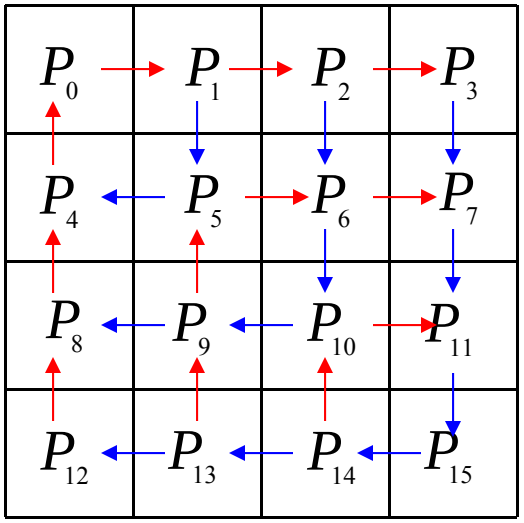
$$A^T[i, j] = A[j, i] \quad , 0 \le i, j \le n$$

- **All elements below diagonal move above the diagonal and vice-versa.**
- **Assume it takes unit time to exchange a pair of matrix elements. Sequential time of transposing an *n x n* matrix is given by**

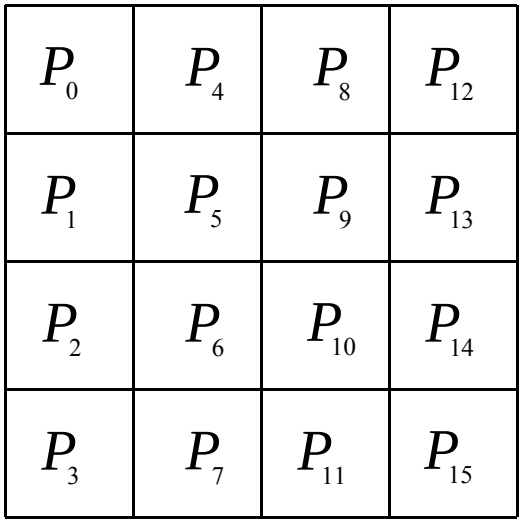
$$(n^2 - n)/2$$

- **Consider parallel architectures organized in both 2D mesh and hypercube structures**

Parallel Matrix Transpose - 2D Mesh



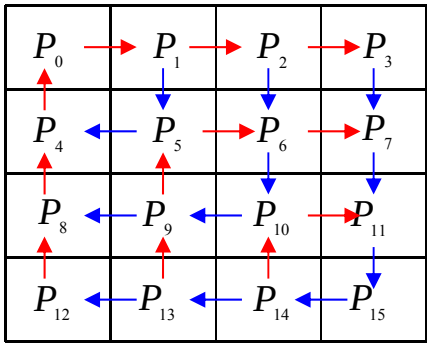
Initial Matrix



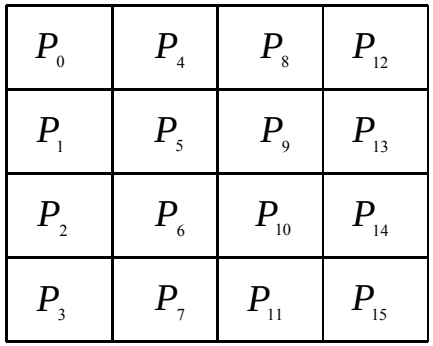
Final Matrix

- **Elements/blocks on lower-left part of matrix move up to the diagonal, and then right to their final location. Each step taken requires communication**
- **Elements/blocks on upper-right part of matrix move down to the diagonal, and then left to their final location. Each step taken requires communication**

Parallel Matrix Transpose - 2D Mesh



Initial Matrix



Final Matrix

- **If each of the *p* processors contains a single number, after all of these communication steps, the matrix has been transposed.**
- **However, if each processor contains a sub-block of the matrix (the more typical situation), after all blocks have been communicated to their final locations, they need to be locally transposed. Each sub-block will contain $(n/\sqrt{p})\times(n/\sqrt{p})$ elements and the cost of communication will be higher than before.**
- **Cost of communication is dominated by elements/blocks that reside in the top-right and bottom-left corners. They have to take an approximate number of hops equal to $2\sqrt{p}$**

Parallel Matrix Transpose - 2D Mesh

- **Each block contains n^2 / p elements, so it takes**

$$2(t_s + t_w n^2 / p)\sqrt{p}$$

where t_s is start-up (latency) and t_w is the per-word transfer time

for all blocks to move to their final destinations. After that, the local elements need to be transposed, which can be done in an amount of time equal to $n^2/(2p)$ for a total wall clock time equal to

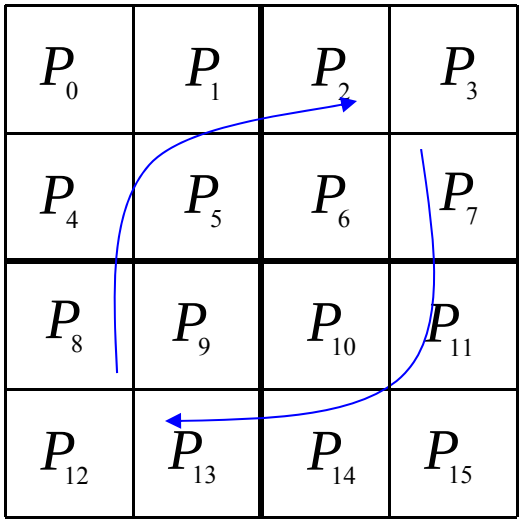
$$T_P = \frac{n^2}{2p} + 2t_s\sqrt{p} + 2t_w\frac{n^2}{\sqrt{p}}$$

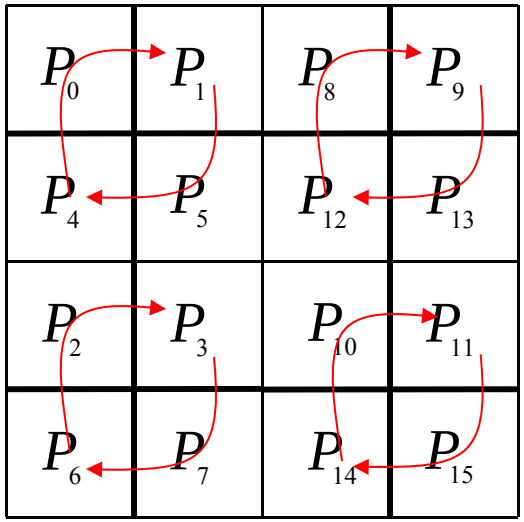
- **Summing over all *p* processors, the total time consumed by the parallel algorithm is of order**

$$T_{TOTAL} = \Theta(n^2\sqrt{p})$$

which is a higher cost than the sequential complexity (order n^2). This algorithm, on a 2D mesh is not cost optimal. The same is true regardless of whether store-and-forward or cut-through routing schemes are used

Parallel Matrix Transpose - Hypercube





- **This algorithm is called recursive subdivision and maps naturally onto a hypercube**
- **After the blocks have all been transposed, the elements inside each block (local to a processor) still need to be transposed**
- **Wall clock time** **Total time**

$$T_P = \frac{n^2}{2p} + (t_s + t_w \frac{n^2}{\sqrt{p}}) \log p \qquad T_{TOTAL} = \Theta(n^2 \log p)$$

P_0	P_4	P_8	P_{12}
P_1	P_5	P_9	P_{13}
P_2	P_6	P_{10}	P_{14}
P_3	P_7	P_{11}	P_{15}

Parallel Matrix Transpose - Hypercube

- **With *cut-through routing*, the timing improves slightly to**

$$T_P = (t_s + t_w \frac{n^2}{p} + 2t_h)\log p \Rightarrow T_{TOTAL} = \Theta(n^2 \log p)$$

which is still sub-optimal

- **Using *striped partitioning* (a.k.a. column blocked layout) and cut-through routing on a hypercube, the total time becomes cost-optimal**

$$T_P = \frac{n^2}{2p} + t_s(p-1) + t_w \frac{n^2}{p} + \frac{1}{2}t_h p \log p \Rightarrow T_{TOTAL} = \Theta(n^2)$$

- **Note that this type of partitioning may be cost-optimal for the transpose operation. But not necessarily for other matrix operations, such as LU factorization and matrix-matrix multiply.**

Parallel Matrix Manipulation Routines

- This example illustrates some of the issues with matrix manipulation (transpose, matrix-vector, matrix-matrix, matrix inversion, eigenvalues, etc.) in parallel
- Matrix decomposition and matrix manipulation algorithm is very important
- Experts have worked on these problems for a number of years for different parallel computer systems → ScaLAPACK

Homework 9

- **Read Chapters 8 and 13 in Introduction to Parallel Computing by Grama et al.**