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 substitution.

```
for i = 1 to n-1
  for j = i+1 to n
```

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 each row $j$ below row $i$
  - add a multiple of row $i$ to row $j$

```
for i = 1 to n-1
  for j = i+1 to n
    tmp = A[j,i];
    for k = i to n
```

- Remove computation of constant $\text{tmp}/A(i,i)$ from inner loop.

```
for i = 1 to n-1
  m = A[i,i]/A[i,i]
  for k = i+1 to n
```

Refine GE Algorithm (2)

- Last version
  - 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
```

Refine GE Algorithm (3)

- Last version
  - Store multipliers $m$ below diagonal in zeroed entries for later use

```
for i = 1 to n-1
  for j = i+1 to n
    for k = i+1 to n
```

Refine GE Algorithm (4)

- Last version
  - Split Loop

```
for i = 1 to n-1
  for j = i+1 to n
    for k = i+1 to n
```

Store all $m$’s here before updating rest of matrix.
Refine GE Algorithm (5)

- Last version

- Express using matrix operations (BLAS)

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

- 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

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 all BLAS 1 or BLAS 2, but we know that BLAS 3 (matrix multiply) is fastest (earlier lectures…)

```plaintext
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)
```

What if some A(i,i) is zero? Or very small?

- Result may not exist, or be “unstable”, so need to pivot
- Current computation all BLAS 1 or BLAS 2, but we know that BLAS 3 (matrix multiply) is fastest (earlier lectures…)

Gaussian Elimination with Partial Pivoting (GEPP)

- Partial Pivoting: swap rows so that A(i,i) is largest in column

```plaintext
for i = 1 to n-1
  find and record k where |A(k,i)| = max_{j = i+1}^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 otherwise
  swap rows i and k of A
end if
```

- Lemma: This algorithm computes A = P*L*U, where P is a permutation matrix.
- This algorithm is numerically stable in practice
- For details see LAPACK code at http://www.netlib.org/lapack/single/sgetf2.f

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 all BLAS 1 or BLAS 2, but we know that BLAS 3 (matrix multiply) is fastest (earlier lectures…)

```
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)
```
Converting BLAS2 to BLAS3 in GEPP

- **Blocking**
  - Used to optimize matrix-multiplication
  - Harder here because of data dependencies in GEPP

- **BIG IDEA: Delayed Updates**
  - Save updates to "trailing matrix" from several consecutive BLAS2 updates
  - Apply many updates simultaneously in one BLAS3 operation

- Same idea works for much of dense linear algebra
  - Open questions remain

- **First Approach:** Need to choose a block size $b$
  - Algorithm will save and apply $b$ updates
  - $b$ must be small enough so that active submatrix consisting of $b$ columns of $A$ fits in cache
  - $b$ must be large enough to make BLAS3 fast

Blocked GEPP (www.netlib.org/lapack/single/sgetrf.f)

```
for ib = 1 to n-1 step b 
  ... Process matrix b 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) 
  ... apply delayed updates with single matrix-multiply 
  ... with inner dimension $b$
```

Gaussian Elimination

- **Standard Way**
  - Subtract a multiple of a row

- **LINPACK**
  - Apply sequence to a column

- **LAPACK**
  - Apply sequence to $nb$
  - Then apply $nb$ to rest of matrix

Efficiency of Blocked GEPP

```
Efficiency = 

- Speed(SLAPACK/SLU)/Speed(best effort)
- Speed(Matmul)/HW Peak
- Speed(LAPACK/SLU)/Speed(MatMul)
```

LU Factorization

```
Pentium 4, 1.5 GHz, using SSE2

Order | sLU | dLU | cLU | zLU
-----|-----|-----|-----|-----
100  | 100 | 100 | 100 | 100
200  | 200 | 200 | 200 | 200
300  | 300 | 300 | 300 | 300
400  | 400 | 400 | 400 | 400
500  | 500 | 500 | 500 | 500
600  | 600 | 600 | 600 | 600
700  | 700 | 700 | 700 | 700
800  | 800 | 800 | 800 | 800
900  | 900 | 900 | 900 | 900
1000 | 1000| 1000| 1000| 1000
3000 | 3000| 3000| 3000| 3000
```

History of Block Partitioned Algorithms

- **Early algorithms** involved use of small main memory using tapes as secondary storage.

- **Recent work** centers on use of vector registers, level 1 and 2 cache, main memory, and "out of core" memory.
### Blocked Partitioned Algorithms

- Orthogonal reduction to:
  - (upper) Hessenberg form
  - symmetric tridiagonal form
  - bidiagonal form
- Block QR iteration for nonsymmetric eigenvalue problems

### Derivation of Blocked Algorithms

**Cholesky Factorization**

A = UTU

Equating coefficient of the jth column, we obtain

\[
\begin{align*}
A_{j1} &= U_{jj}^TU_{j1} \\
A_{j2} &= U_{jj}^TU_{j2} + U_{j1}^TU_{12}
\end{align*}
\]

Hence, if \( U_{jj} \) has already been computed, we can compute \( u_j \) and \( u_{jj} \) from the equations:

\[
U_{jj}^TU_{j1} = a_j \\
u_{jj} = a_{jj} - u_j^2u_j
\]

### LINPACK Implementation

Here is the body of the LINPACK routine SPOFA which implements the method:

```plaintext
DO 30 J = 1, N
    INFO = J
    S = 0.0E0
    JM1 = J - 1
    IF( JM1.LT.1 ) GO TO 20
    DO 10 K = 1, JM1
        T = A( K, J ) - SDOT( K-1, A( 1, K ), 1,A( 1, J ), 1 )
        T = T / A( K, K )
        A( K, J ) = T
        S = S + T*T
    10 CONTINUE
    S = A( J, J ) - S
    C        ...EXIT
    IF( S.LE.0.0E0 ) GO TO 40
    A( J, J ) = SQRT( S )
    30 CONTINUE
```

### LAPACK Implementation

```plaintext
DO 10 J = 1, N
    CALL STRSV( 'Upper', 'Transpose', 'Non-Unit', J-1, A, LDA, A( 1, J ), 1 )
    S = A( J, J ) - SDOT( J-1, A( 1, J ), 1, A( 1, J ), 1 )
    IF( S.LE.ZERO ) GO TO 20
    A( J, J ) = SQRT( S )
    10 CONTINUE
```

This change by itself is sufficient to significantly improve the performance on a number of machines.

- From 238 to 312 Mflop/s for a matrix of order 500 on a Pentium 4-1.7 GHz.
- However on peak is 1,700 Mflop/s.
- Suggest further work needed.

### LAPACK Blocked Algorithms

```plaintext
DO 10 J = 1, N, NB
    CALL STRSM( 'Left', 'Upper', 'Transpose','Non-Unit', J-1, JB, ONE, A, LDA, $               A( 1, J ), LDA )
    CALL SSYRK( 'Upper', 'Transpose', JB, J-1,-ONE, A( 1, J ), LDA, ONE, $               A( J, J ), LDA )
    CALL SPOTF2( 'Upper', JB, A( J, J ), LDA, INFO )
    IF( INFO.NE.0 ) GO TO 20
10 CONTINUE
```

This improvement is significant.

### Derivation of Blocked Algorithms

**Equating coefficient of second block of columns, we obtain**

\[
A_{j1} = U_{jj}^TU_{j1} \\
A_{j2} = U_{jj}^TU_{j2} + U_{j1}^TU_{12}
\]

Hence, if \( U_{jj} \) has already been computed, we can compute \( u_j \) as the solution of the following equations by a call to the Level 3 BLAS routine STRSM:

\[
U_{jj}^TU_{j1} = a_j \\
U_{jj}^TU_{j2} = A_{j2} - U_{j1}^TU_{12}
\]
Overview of LAPACK and ScaLAPACK

- Standard library for dense/banded linear algebra
  - Linear systems: $A^t x = b$
  - Least squares problems: $\min_{x} ||A^t x - b||^2$
  - 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, Matlab ...
- Many algorithmic innovations remain
  - Projects available

LAPACK

- Linear Algebra library in Fortran 77
  - Solution of systems of equations
  - Solution of eigenvalue problems
- Combine algorithms from LINPACK and EISPACK into a single package
- Efficient on a wide range of computers
  - RISC, Vector, SMPs
- User interface similar to LINPACK
  - Single, Double, Complex, Double Complex
- Built on the Level 1, 2, and 3 BLAS

Review: BLAS 3 (Blocked) GEPP

```plaintext
for ib = 1 to n-1 step b  ... Process matrix b 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^T L^T U^T
  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
```

Parallelizing Gaussian Elimination

- Parallelization steps
  - Decomposition: identify enough parallel work, but not too much
  - Assignment: load balance work among threads
  - Orchestrate: communication and synchronization
  - Mapping: which processors execute which threads
- Decomposition
  - In BLAS 2 algorithm nearly each flop in inner loop can be done in parallel, so with $n^2$ processors, need $3n$ parallel steps
  ```plaintext
  for i = 1 to n-1
    A[i+1:n,i] = A[i+1:n,i] / A[i,i]         ... BLAS 1 (scale a vector)
  ```
  - This is too fine-grained, prefer calls to local matmuls instead
  - Need to use parallel matrix multiplication
- Assignment
  - Which processors are responsible for which submatrices?

Challenges in Developing Distributed Memory Libraries

- How to integrate software?
  - Until recently no standards
  - Many parallel languages
  - Various parallel programming models
  - Assumptions about the parallel environment
    - granularity
    - topology
    - overlapping of communication/computation
    - development tools
- Where is the data?
  - Who owns it?
  - Opt data distribution
- Who determines data layout?
  - Determined by user?
  - Determined by library developer?
  - Allow dynamic data dist.
  - Load balancing
Different Data Layouts for Parallel GE

<table>
<thead>
<tr>
<th>Layout Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) 1D Column Blocked Layout</td>
<td>Load balanced, but can't easily use BLAS2 or BLAS3</td>
</tr>
<tr>
<td>2) 1D Column Cyclic Layout</td>
<td>Complicated addressing</td>
</tr>
<tr>
<td>3) 1D Column Block Cyclic Layout</td>
<td>Bad load balance: P0 idle after first n/4 steps</td>
</tr>
<tr>
<td>4) Block Skewed Layout</td>
<td>The winner!</td>
</tr>
<tr>
<td>5) 2D Row and Column Blocked Layout</td>
<td>Bad load balance: P0 idle after first n/2 steps</td>
</tr>
<tr>
<td>6) 2D Row and Column Block Cyclic Layout</td>
<td>Block size b in the algorithm and the block sizes brow and bcol in the layout satisfy b=bcol.</td>
</tr>
</tbody>
</table>

Row and Column Block Cyclic Layout

- processors and matrix blocks are distributed in a 2d array
- row-by-pcol array of processors
- brow-by-bcol matrix blocks
- pcol-fold parallelism in any column, and calls to the BLAS2 and BLAS3 on matrices of size brow-by-bcol
- serial bottleneck is eased
- prow != pcol and brow != bcol possible, even desirable

Distributed 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=bcol.
- shaded regions indicate processors busy with computation or communication.
- unnecessary to have a barrier between each step of the algorithm, e.g., step 9, 10, and 11 can be pipelined

ScaLAPACK

- Library of software dealing with dense & banded routines
- Distributed Memory - Message Passing
- MIMD Computers and Networks of Workstations
- Clusters of SMPs

Programming Style

- SPMD Fortran 77 with object based design
- Built on various modules
  - PBLAS Interprocessor communication
  - BLACS
  - PVM, MPI, IBM SP, CRAY T3, Intel, TMC
  - Provides right level of notation.
  - BLAS
- LAPACK software expertise/quality
  - Software approach
  - Numerical methods

Review of Parallel MatMul

- Want Large Problem Size Per Processor
- PDGEMM = PBLAS matrix multiply

Observations:
- For fixed N, as P increases
  - Mflops increases, but less than 100% efficiency
- For fixed P, as N increases,
  - Mflops (efficiency) rises

DGEMM = BLAS routine for matrix multiply

Maximum speed for PDGEMM = # Proc * speed of DGEMM

Observations:
- Efficiency always at least 48%
- For fixed N, as P increases, efficiency drops
- For fixed P, as N increases, efficiency increases

Performance of PDGEMM

<table>
<thead>
<tr>
<th>Processor</th>
<th># Proc</th>
<th>Speed of DGEMM</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray T3E</td>
<td>8</td>
<td>273</td>
<td>77</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>16</td>
<td>273</td>
<td>77</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>32</td>
<td>273</td>
<td>77</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>64</td>
<td>273</td>
<td>77</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>128</td>
<td>273</td>
<td>77</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>256</td>
<td>273</td>
<td>77</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>512</td>
<td>273</td>
<td>77</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>1024</td>
<td>273</td>
<td>77</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>2048</td>
<td>273</td>
<td>77</td>
</tr>
<tr>
<td>Cray T3E</td>
<td>4096</td>
<td>273</td>
<td>77</td>
</tr>
</tbody>
</table>
**ScaLAPACK Performance Models (1)**

**PDGESV = ScaLAPACK Parallel LU**

Since it can run no faster than its:
- inner loop (POEM), we measure:
- Efficiency:
  Speed/PDGEMM(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
  - Avx3 about half of matrix multiply for large enough matrices.
  - From the flop counts we would expect it to be \((2N^2)(2N^2)\) = 3 times faster, but communication makes it a little slower.

---

**Overall Structure of Software**

- Object based - Array descriptor
  - Contains information required to establish mapping between a global array entry and its corresponding process and memory location.
  - Provides a flexible framework to easily specify additional data distributions or matrix types.
  - Currently dense, banded, & out-of-core
- Using the concept of context

**ScaLAPACK Overview**

ScaLAPACK SOFTWARE HIERARCHY

- **ScaLAPACK**
- **PBLAS**
- **LAPACK**
- **BLAS**
- Message Passing Primitives (MPI, PV M, etc.)

**ScaLAPACK Performance Models (2)**

**Compare Predictions and Measurements**

---

**PBLAS**

- Similar to the BLAS in functionality and naming.
- Built on the BLAS and BLACS
- Provide global view of matrix

\[
\text{CALL DGEXXX} \ (M, N, A(IA, JA), LDA, \ldots)\]

\[
\text{CALL PDGEXXX} \ (M, N, A, IA, JA, DESCX, \ldots)\]

---

**ScaLAPACK Performance Models (1)**

**ScalAPACK Operation Counts**

**PDGESV = ScaLAPACK Parallel LU**

Performance of scaLAPACK LU

---

**38**

**39**

**40**

**41**
Out of “Core” Algorithms

Out-of-core means matrix lives on disk; too big for main mem. Much harder to hide latency of disk. QR much easier than LU because no pivoting needed for QR.

Recursive Algorithms

- Still uses delayed updates, but organized differently (formulas on board)
- Can exploit recursive data layouts
  - 3x speedups on least squares for tall, thin matrices
- Theoretically optimal memory hierarchy performance
- See references at
  - http://lawra.uni-c.dk/lawra/index.html

Recursive Factorizations

- Just as accurate as conventional method
- Same number of operations
- Automatic variable-size blocking
  - Level 1 and 3 BLAS only!
- Extreme clarity and simplicity of expression
- Highly efficient
- The recursive formulation is just a rearrangement of the pointwise LINPACK algorithm
- The standard error analysis applies (assuming the matrix operations are computed the “conventional” way).

Recursive Algorithms – Limits

- Two kinds of dense matrix compositions
  - One Sided
    - Sequence of simple operations applied on left of matrix
      - Gaussian Elimination: \( A = L^*U \) or \( A = P^*L^*U \)
      - Symmetric Gaussian Elimination: \( A = L^*D^*L^T \)
      - Cholesky: \( A = L^*L^T \)
    - QR Decomposition for Least Squares: \( A = Q*R \)
  - Two Sided
    - Sequence of simple operations applied on both sides, alternating
      - Eigenvalue algorithms, SVD
      - At least ~25% BLAS 2
      - Seem impervious to recursive approach?
      - Some recent progress on SVD (25% vs 50% BLAS2)
LAPACK and ScaLAPACK Status

- "One-sided Problems" are scalable
  - In Gaussian elimination, $A$ factored into product of 2 matrices $A = LU$ by premultiplying $A$ by sequence of simpler matrices
  - Asymptotically 100% BLAS3
  - LU ("Linpack Benchmark")
  - Cholesky, QR

- "Two-sided Problems" are harder
  - $A$ factored into product of 3 matrices by pre and post multiplication
  - Half BLAS2, not all BLAS3
  - Eigenproblems, SVD
  - Nonsymmetric eigenproblem hardest

- Narrow band problems hardest (to do BLAS3 or parallelize)
  - Solving and eigenproblems

www.netlib.org/lapack,scalapack

Some contributors (incomplete list)

-www.netlib.org/lapack,scalapack