Lecture 11 (part 3):
Linear Algebra Algorithms

Jack Dongarra, U of Tennessee

Slides are adapted from Jim Demmel, UCB’s Lecture on Linear Algebra Algorithms

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

```plaintext
... 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
tmp = A(j,i);
for k = i to n
    A(j,k) = A(j,k) - (tmp/A(i,i)) * A(i,k)
```

After \( i = 1 \)  
After \( i = 2 \)  
After \( i = 3 \)  
After \( i = n-1 \)
Refine GE Algorithm (1)

- Initial Version

```plaintext
... 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
    tmp = A(j,i);
    for k = i to n
        A(j,k) = A(j,k) - (tmp/A(i,i)) * A(i,k)
```

- Remove computation of constant tmp/A(i,i) from inner loop.

```plaintext
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

```plaintext
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

```plaintext
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

\[
\begin{align*}
\text{for } i = 1 \text{ to } n-1 \\
\text{for } j = i+1 \text{ to } n \\
m = \frac{A(j,i)}{A(i,i)} \\
\text{for } k = i+1 \text{ to } n \\
A(j,k) = A(j,k) - m \cdot A(i,k)
\end{align*}
\]

• Store multipliers \( m \) below diagonal in zeroed entries for later use

\[
\begin{align*}
\text{for } i = 1 \text{ to } n-1 \\
\text{for } j = i+1 \text{ to } n \\
A(j,i) = \frac{A(j,i)}{A(i,i)} \\
\text{for } k = i+1 \text{ to } n \\
A(j,k) = A(j,k) - A(j,i) \cdot A(i,k)
\end{align*}
\]

Refine GE Algorithm (4)

• Last version

\[
\begin{align*}
\text{for } i = 1 \text{ to } n-1 \\
\text{for } j = i+1 \text{ to } n \\
A(j,i) = \frac{A(j,i)}{A(i,i)} \\
\text{for } k = i+1 \text{ to } n \\
A(j,k) = A(j,k) - A(j,i) \cdot A(i,k)
\end{align*}
\]

• Split Loop

\[
\begin{align*}
\text{for } i = 1 \text{ to } n-1 \\
\text{for } j = i+1 \text{ to } n \\
A(j,i) = \frac{A(j,i)}{A(i,i)} \\
\text{for } j = i+1 \text{ to } n \\
\text{for } k = i+1 \text{ to } n \\
A(j,k) = A(j,k) - A(j,i) \cdot A(i,k)
\end{align*}
\]
Refine GE Algorithm (5)

- Last version

\[
\begin{align*}
\text{for } i = 1 \text{ to } n-1 \\
\quad &\text{for } j = i+1 \text{ to } n \\
\quad &\quad A(j,i) = A(j,i) / A(i,i) \\
\quad &\text{for } j = i+1 \text{ to } n \\
\quad &\quad A(i+1:n,i) = A(i+1:n,i) - A(i+1:n,i) \cdot A(i,i) \\
\end{align*}
\]

- Express using matrix operations (BLAS)

Work at step 1 of Gaussian Elimination

\[
\begin{align*}
\text{for } i = 1 \text{ to } n-1 \\
\quad &\text{for } j = i+1 \text{ to } n \\
\quad &\quad A(j,i) = A(j,i) / A(i,i) \\
\quad &\text{for } j = i+1 \text{ to } n \\
\quad &\quad \text{for } k = i+1 \text{ to } n \\
\quad &\quad \quad A(j,k) = A(j,k) - A(j,i) \cdot A(i,k)
\end{align*}
\]

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 \quad (cost = 2/3 n^3 \text{ flops})
  - Solve \( L^*y = b \) for \( y \), using substitution \( (\text{cost} = n^2 \text{ flops}) \)
  - Solve \( U^*x = y \) for \( x \), using substitution \( (\text{cost} = n^2 \text{ flops}) \)
- 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…)

```
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) - A(i+1:n, i) * A(i, i+1:n)
```

*IBM RS/6000 Power 3 (200 MHz, 800 Mflop/s Peak)*

**Pivoting in Gaussian Elimination**

- A = \[
\begin{bmatrix}
0 & 1 \\
1 & 0 \\
\end{bmatrix}
\] fails completely because can’t divide by $A(1,1)=0$

- But solving $Ax=b$ should be easy!

- When diagonal $A(i,i)$ is tiny (not just zero), algorithm may terminate but get completely wrong answer
  - Numerical instability
  - Roundoff error is cause

- Cure: Pivot (swap rows of A) so $A(i,i)$ large
Gaussian Elimination with Partial Pivoting (GEPP)

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

for $i = 1$ to $n-1$
  find and record $k$ where $|A(k,i)| = \max\{|i \leq j \leq n| \, |A(j,i)|\}$ ... 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 $k \neq i$
    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.
- This algorithm is numerically stable in practice
- For details see LAPACK code at [http://www.netlib.org/lapack/single/sgetf2.f](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 $n-1$

$A(i+1:n,i) = A(i+1:n,i) / A(i,i)$ ... BLAS 1 (scale a vector)
$A(i+1:n,i+1:n) = A(i+1:n,i+1:n) - A(i+1:n,i) * A(i,i+1:n)$ ... BLAS 2 (rank-1 update)

IBM RS/6000 Power 3 (200 MHz, 800 Mflop/s Peak)
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'
    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
```

---

![Diagram of Gaussian Elimination using BLAS 3](image-url)
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

LU Factorization
Pentium 4, 1.5 GHz, using SSE2

<table>
<thead>
<tr>
<th>Order</th>
<th>Mflop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>500</td>
</tr>
<tr>
<td>300</td>
<td>1000</td>
</tr>
<tr>
<td>500</td>
<td>1500</td>
</tr>
<tr>
<td>700</td>
<td>2000</td>
</tr>
<tr>
<td>900</td>
<td>2500</td>
</tr>
<tr>
<td>1200</td>
<td>3000</td>
</tr>
<tr>
<td>1600</td>
<td>3500</td>
</tr>
<tr>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>2400</td>
<td></td>
</tr>
<tr>
<td>2800</td>
<td></td>
</tr>
</tbody>
</table>

sLU  dLU  cLU  zLU
Efficiency of Blocked GEPP

- Speed (LAPACK/LU) / Speed (best effort)
- Speed (Matmul) / HW Peak
- Speed (LAPACK LU) / Speed (MatMul)

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

- LU Factorization

- Cholesky factorization

- Symmetric indefinite factorization

- Matrix inversion

- QR, QL, RQ, LQ factorizations

- Form Q or Q'TC

---

Derivation of Blocked Algorithms

Cholesky Factorization \( A = U^T U \)

\[
\begin{pmatrix}
A_{11} & a_j & A_{13} \\
A_{12}^T & a_{jj} & A_{23} \\
A_{31} & A_{32} & A_{33}
\end{pmatrix} =
\begin{pmatrix}
U_{11}^T & 0 & 0 \\
U_{12}^T & u_j & 0 \\
U_{13}^T & \mu_j & U_{33}^T
\end{pmatrix}
\begin{pmatrix}
U_{11} & u_j & U_{13} \\
0 & u_{jj} & \mu_j \\
0 & 0 & U_{33}
\end{pmatrix}
\]

Equating coefficient of the \( j^{th} \) column, we obtain

\[
a_j = U_{11}^T u_j
\]

\[
a_{jj} = u_j^T u_j + u_{jj}^2
\]

Hence, if \( U_{11} \) has already been computed, we can compute \( u_j \) and \( u_{jj} \) from the equations:

\[
U_{11}^T u_j = a_j
\]

\[
u_{jj}^2 = a_{jj} - u_j^T u_j
\]
**LINPACK Implementation**

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

```fortran
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
20 CONTINUE
S = A( J, J ) - S
C        ...EXIT
   IF( S.LE.ZERO ) GO TO 40
   A( J, J ) = SQRT( S )
30 CONTINUE
```

**LAPACK Implementation**

```fortran
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.
Derivation of Blocked Algorithms

\[
\begin{pmatrix}
A_{11} & A_{12} & A_{13} \\
A_{12}^T & A_{22} & A_{23} \\
A_{13}^T & A_{23}^T & A_{33}
\end{pmatrix}
= 
\begin{pmatrix}
U_{11}^T & 0 & 0 \\
U_{12}^T & U_{22}^T & 0 \\
U_{13}^T & U_{23}^T & U_{33}^T
\end{pmatrix}
\begin{pmatrix}
U_{11} & U_{12} & U_{13} \\
0 & U_{22} & U_{23} \\
0 & 0 & U_{33}
\end{pmatrix}
\]

Equating coefficient of second block of columns, we obtain

\[
A_{12} = U_{11}^T U_{12}
\]

\[
A_{22} = U_{12}^T U_{12} + U_{22}^T U_{22}
\]

Hence, if \( U_{11} \) has already been computed, we can compute \( U_{12} \) as the solution of the following equations by a call to the Level 3 BLAS routine STRSM:

\[
U_{11}^T U_{12} = A_{12}
\]

\[
U_{22}^T U_{22} = A_{22} - U_{12}^T U_{12}
\]

LAPACK Blocked Algorithms

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', 'J-1,-ONE, A( J, J ), LDA, INFO )
    IF( INFO.NE.0 ) GO TO 20
10 CONTINUE

• On Pentium 4, L3 BLAS squeezes a lot more out of 1 proc

<table>
<thead>
<tr>
<th></th>
<th>Rate of Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Pentium 4 1.7 GHz</td>
<td>N = 500</td>
</tr>
<tr>
<td>Linpack variant (L1B)</td>
<td>238 Mflop/s</td>
</tr>
<tr>
<td>Level 2 BLAS Variant</td>
<td>312 Mflop/s</td>
</tr>
<tr>
<td>Level 3 BLAS Variant</td>
<td>1262 Mflop/s</td>
</tr>
</tbody>
</table>
Overview of LAPACK and ScaLAPACK

• 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, 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
LAPACK

• Most of the parallelism in the BLAS.

• Advantages of using the BLAS for parallelism:
  - Clarity
  - Modularity
  - Performance
  - Portability

Review: BLAS 3 (Blocked) GEPP

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)
    - 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

- **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
  
  ```
  for $i = 1$ to $n-1$
      $A(i+1:n,i) = A(i+1:n,i) / A(i,i)$  \[\text{... BLAS 1 (scale a vector)}\]
      $A(i+1:n,i+1:n) = A(i+1:n, i+1:n)$ \[\text{... BLAS 2 (rank-1 update)}\]
      $A(i+1:n,i) \times A(i,i+1:n)$
  ```
- 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</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) 1D Column Blocked Layout</td>
<td>Bad load balance: P0 idle after first n/4 steps</td>
</tr>
<tr>
<td>2) 1D Column Cyclic Layout</td>
<td>Load balanced, but can't easily use BLAS2 or BLAS3</td>
</tr>
<tr>
<td>3) 1D Column Block Cyclic Layout</td>
<td>Can trade load balance and BLAS2/3 performance by choosing b, but factorization of block column is a bottleneck</td>
</tr>
<tr>
<td>4) Block Skewed Layout</td>
<td>Complicated addressing</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>The winner!</td>
</tr>
</tbody>
</table>

Review of Parallel MatMul

- Want Large Problem Size Per Processor
  - PDGEMM = PBLAS matrix multiply
    - 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 = # Procs * speed of DGEMM
    - Efficiency always at least 48%
    - For fixed N, as P increases, efficiency drops
    - For fixed P, as N increases, efficiency increases

Performance of PBLAS

<table>
<thead>
<tr>
<th>Machine</th>
<th>Procs</th>
<th>Block Size</th>
<th>Block N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clay T3E</td>
<td>4x2x2</td>
<td>32</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td>16x4x4</td>
<td>32</td>
<td>3000</td>
</tr>
<tr>
<td></td>
<td>64x8x8</td>
<td>32</td>
<td>4000</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>4</td>
<td>64</td>
<td>5000</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>64</td>
<td>5000</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>64</td>
<td>5000</td>
</tr>
<tr>
<td>X/Y SMP</td>
<td>4</td>
<td>32</td>
<td>5000</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>64</td>
<td>5000</td>
</tr>
<tr>
<td>Beijing NOW</td>
<td>4</td>
<td>32</td>
<td>5000</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>64</td>
<td>5000</td>
</tr>
</tbody>
</table>

Efficiency = Mflops(DGEMM)/(Proc*N Mflops)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Peak proc</th>
<th>DGEMM proc</th>
<th>DGEMM N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clay T3E</td>
<td>800</td>
<td>32</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>32</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>32</td>
<td>1000</td>
</tr>
<tr>
<td>IBM SP2</td>
<td>200</td>
<td>32</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>32</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>32</td>
<td>1000</td>
</tr>
<tr>
<td>X/Y SMP</td>
<td>100</td>
<td>32</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>32</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>32</td>
<td>1000</td>
</tr>
<tr>
<td>Beijing NOW</td>
<td>320</td>
<td>32</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>32</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>32</td>
<td>1000</td>
</tr>
</tbody>
</table>

Efficiency = Mflops(DGEMM)/(Proc*N Mflops)
Row and Column Block Cyclic Layout

- processors and matrix blocks are distributed in a 2d array
  - prow-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, CRI T3, Intel, TMC
    - Provides right level of notation.
  - BLAS
- LAPACK software expertise/quality
  - Software approach
  - Numerical methods
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

PBLAS

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

CALL DGEXXX ( M, N, A( IA, JA ), LDA,... )

CALL PDGEXXX( M, N, A, IA, JA, DESCA,... )
ScaLAPACK Overview

ScaLAPACK Performance Models (1)

ScaLAPACK Operation Counts

\[ \begin{align*}
T(N, P) &= \frac{C_t N^2}{P} t_f + \frac{C_m N e}{\sqrt{P}} t_m + \frac{C_v N}{N_b} t_v, \\
T_{opt}(N, P) &= N^2 F. \\
B(N, P) &= \left( 1 + \frac{1}{N_b} \frac{C_m t_m}{N^2} + \frac{C_v t_v}{P} \right)^{-1}.
\end{align*} \]

<table>
<thead>
<tr>
<th>Driver</th>
<th>Optx</th>
<th>( G_f )</th>
<th>( C_t )</th>
<th>( G_m )</th>
</tr>
</thead>
<tbody>
<tr>
<td>PxGESV</td>
<td>1 right hand side</td>
<td>2/3</td>
<td>3 + 1/4 \log_2 P</td>
<td>( N_b (5 + \log_2 P) )</td>
</tr>
<tr>
<td>PxPOSV</td>
<td>1 right hand side</td>
<td>1/3</td>
<td>2 + 1/3 \log_2 P</td>
<td>( 4 \log_2 P )</td>
</tr>
<tr>
<td>PxGESV</td>
<td>1 right hand side</td>
<td>4/3</td>
<td>3 + \log_2 P</td>
<td>( 2 (N_b \log_2 P + 1) )</td>
</tr>
<tr>
<td>PxSYEVX</td>
<td>eigenvalues only</td>
<td>4/3</td>
<td>5/3 \log_2 P</td>
<td>( 17/2 N_b + 1 )</td>
</tr>
<tr>
<td>PxSYEVX</td>
<td>eigenvalues and eigenvectors</td>
<td>10/3</td>
<td>5 \log_2 P</td>
<td>( 17/2 N_b + 3 )</td>
</tr>
<tr>
<td>PxSYEVX</td>
<td>eigenvalues only</td>
<td>4/3</td>
<td>5/3 \log_2 P</td>
<td>( 17/2 N_b + 1 )</td>
</tr>
<tr>
<td>PxSYEVX</td>
<td>eigenvalues and eigenvectors</td>
<td>38/3</td>
<td>5 \log_2 P</td>
<td>( 17/2 N_b + 3 )</td>
</tr>
<tr>
<td>PxHEVD</td>
<td>singular values only</td>
<td>38/3</td>
<td>10 \log_2 P</td>
<td>( 17N_b )</td>
</tr>
<tr>
<td>PxHEVD</td>
<td>singular values and left and right singular vectors</td>
<td>38/3</td>
<td>14 \log_2 P</td>
<td>( 17N_b )</td>
</tr>
<tr>
<td>PxLAEQR</td>
<td>eigenvalues only</td>
<td>5</td>
<td>( 9/3 (\sqrt{P} + \log_2 P) + 2 \log_2 P )</td>
<td>( 9(3 + \log_2 P) N )</td>
</tr>
<tr>
<td>PxLAEQR</td>
<td>full Schur form</td>
<td>18</td>
<td>( 9/3 (\sqrt{P} + \log_2 P) + 2 \log_2 P )</td>
<td>( 9(3 + \log_2 P) N )</td>
</tr>
</tbody>
</table>
ScaLAPACK Performance Models (2)

Compare Predictions and Measurements

<table>
<thead>
<tr>
<th>IBM SP2*</th>
<th>P</th>
<th>Values of N</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>5000</td>
<td>867</td>
<td>421</td>
<td>833</td>
<td>833</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>5000</td>
<td>497</td>
<td>722</td>
<td>183</td>
<td>183</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>10000</td>
<td>652</td>
<td>924</td>
<td>4429</td>
<td>4429</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>7600</td>
<td>3017</td>
<td>4536</td>
<td>4536</td>
<td>4536</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>10000</td>
<td>2116</td>
<td>2404</td>
<td>2149</td>
<td>2149</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>15000</td>
<td>1903</td>
<td>2404</td>
<td>2149</td>
<td>2149</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>18000</td>
<td>5793</td>
<td>5185</td>
<td>7067</td>
<td>7067</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>20000</td>
<td>5793</td>
<td>5185</td>
<td>7067</td>
<td>7067</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>20000</td>
<td>6325</td>
<td>7025</td>
<td>2812</td>
<td>2812</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>25000</td>
<td>5793</td>
<td>5185</td>
<td>7067</td>
<td>7067</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>50000</td>
<td>6325</td>
<td>7025</td>
<td>2812</td>
<td>2812</td>
</tr>
</tbody>
</table>

*One process spawned per node and one computational IBM POWER3 600 processor per node.

PDGESV = ScaLAPACK Parallel LU

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 \( (2/3)n^3 / (2/3)n^3 = 3 \) times faster, but communication makes it a little slower.
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](http://lawra.uni-c.dk/lawra/index.html)
**Gaussian Elimination via a Recursive Algorithm**

F. Gustavson and S. Toledo

LU Algorithm:
1: Split matrix into two rectangles (m x n/2)
   if only 1 column, scale by reciprocal of pivot & return
2: Apply LU Algorithm to the left part
3: Apply transformations to right part
   (triangular solve $A_{12} = L^{-1}A_{12}$ and
   matrix multiplication $A_{22} = A_{22} - A_{21} \cdot A_{12}$)
4: Apply LU Algorithm to right part

Most of the work in the matrix multiply
Matrices of size n/2, n/4, n/8, ...

Source: Jack Dongarra

**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 point-wise 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: $A = Q*R$
  - Can be nearly 100% BLAS 3
  - Susceptible to recursive algorithms

- 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)**

<table>
<thead>
<tr>
<th>Participants</th>
<th>Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kroet Annext (UC Berkeley)</td>
<td>Schmidt A (UC Berkeley)</td>
</tr>
<tr>
<td>Richard Barrett (U. Texas)</td>
<td>Michael Berry (U. Texas)</td>
</tr>
<tr>
<td>Jeff Bilmes (UC Berkeley)</td>
<td>Michael Dayd (ANL)</td>
</tr>
<tr>
<td>Susan Blackford (ORNL)</td>
<td>Chris Bischof (ANL)</td>
</tr>
<tr>
<td>Tony Chan (UCLA)</td>
<td>Susan Chabaud (UC Berkeley)</td>
</tr>
<tr>
<td>Jieping Choi (RICE)</td>
<td>Chao-Wye Chin (UC Berkeley)</td>
</tr>
<tr>
<td>Ed D'Amore (ORNL)</td>
<td>Andy Chong (LLNL)</td>
</tr>
<tr>
<td>Bozil Dhillon (UC Berkeley)</td>
<td>Jim Demmel (UC Berkeley)</td>
</tr>
<tr>
<td>Jack Dongarra (U. Texas, ORNL)</td>
<td>John J. Dongarra (ORNL)</td>
</tr>
<tr>
<td>James Demmel (NAG)</td>
<td>Jim Demmel (UC Berkeley)</td>
</tr>
<tr>
<td>Stanley Eisenstat (Yale)</td>
<td>Victor Eijkhout (UCLA)</td>
</tr>
<tr>
<td>John Gilbert (Stanford)</td>
<td>Victor Eijkhout (UCLA)</td>
</tr>
<tr>
<td>Steven Hanrahan (NASA)</td>
<td>Ming Gu (UC Berkeley, ERL)</td>
</tr>
<tr>
<td>Greg Henry (Intel)</td>
<td>Mike Kaufman (U. Illinois)</td>
</tr>
<tr>
<td>Steve Hudson (NASA)</td>
<td>Karm iram (UC Berkeley)</td>
</tr>
<tr>
<td>W. Kahan (UC Berkeley)</td>
<td>Yash Atase (UC Berkeley)</td>
</tr>
<tr>
<td>How Leigh (UC Berkeley)</td>
<td>Zhou Li (UC Berkeley)</td>
</tr>
<tr>
<td>Joseph Liu (Yale)</td>
<td>Bennett Pease (UC Berkeley)</td>
</tr>
<tr>
<td>Antoine Paillet (U. Texas)</td>
<td>Peter Poromaa (U. Uppsala)</td>
</tr>
<tr>
<td>Robert Perry (U. Texas)</td>
<td>Pedraa Reapmin (U. Illinois)</td>
</tr>
<tr>
<td>Jian Ren (UC Berkeley)</td>
<td>Howard Robinson (UC Berkeley)</td>
</tr>
<tr>
<td>Charles Rolfe (ORNL)</td>
<td>Jeff Rutland (UC Berkeley)</td>
</tr>
<tr>
<td>Feng Shi (Purdue)</td>
<td>Jeri Somers (Emerson)</td>
</tr>
<tr>
<td>Ken Stanley (UC Berkeley)</td>
<td>Xiaohai Sun (INRIA)</td>
</tr>
<tr>
<td>Bert Lent (Philips (U. Texas)</td>
<td>Attr Puesiri (NCI)</td>
</tr>
<tr>
<td>Robert van de Goede (U. Texas)</td>
<td>Mark van de Water (Enrauichi)</td>
</tr>
<tr>
<td>Paul Van Dooren (U. Liege)</td>
<td>Kees Voesen (U. Houston)</td>
</tr>
<tr>
<td>David Walker (ORNL)</td>
<td>Clint Whaley (U. Texas)</td>
</tr>
<tr>
<td>Kathy Yelick (UC Berkeley)</td>
<td>George Wills (U. Texas)</td>
</tr>
</tbody>
</table>

With the cooperation of
Cray, IBM, Convex, DEC, Fujitsu, NEC, NAG, ESSL.

Supported by ARPA, NSF, DOE