Lecture 7: Linear Algebra Algorithms
Jack Dongarra, U of Tennessee

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

Outline
- Motivation, overview for Dense Linear Algebra
- Review Gaussian Elimination (GE) for solving Ax=b
- Optimizing GE for caches on sequential machines
  - using matrix-matrix multiplication (BLAS)
- LAPACK library overview and performance
- Data layouts on parallel machines
- Parallel Gaussian Elimination
- ScaLAPACK library overview
- Eigenvalue problems
- Open Problems

BLAS – Introduction
- Clarity: code is shorter and easier to read,
- Modularity: gives programmer larger building blocks,
- Performance: manufacturers will provide tuned machine-specific BLAS,
- Program portability: machine dependencies are confined to the BLAS

Memory Hierarchy
- Key to high performance in effective use of memory hierarchy
- True on all architectures

Level 1, 2 and 3 BLAS
- Level 1 BLAS
  Vector-Vector operations
- Level 2 BLAS
  Matrix-Vector operations
- Level 3 BLAS
  Matrix-Matrix operations

More on BLAS (Basic Linear Algebra Subroutines)
- Industry standard interface (evolving)
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s): vector operations: dot product, saxpy (y=α*x+y), etc
  - m=2*n, f=2*n, q ~1 or less
  - BLAS2 (mid 1980s): matrix-vector operations: matrix vector multiply, etc
  - m*n, f=2*n^2, q~2, less overhead
  - somewhat faster than BLAS1
  - BLAS3 (late 1980s): matrix-matrix operations: matrix matrix multiply, etc
  - m >= 4n^2, f=O(n^3), so q can possibly be as large as n, so BLAS3 is potentially much faster than BLAS2
  - Good algorithms used BLAS3 when possible (LAPACK)
Why Higher Level BLAS?

- Can only do arithmetic on data at the top of the hierarchy
- Higher level BLAS lets us do this

<table>
<thead>
<tr>
<th>BLAS</th>
<th>Memory Refs</th>
<th>Flops</th>
<th>Flops/Me Refs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>y = y + αx</td>
<td>3n</td>
<td>2n/3</td>
</tr>
<tr>
<td>Level 2</td>
<td>y = y + Ax</td>
<td>n²</td>
<td>2n²</td>
</tr>
<tr>
<td>Level 3</td>
<td>C = C + AB</td>
<td>4n²</td>
<td>n/2</td>
</tr>
</tbody>
</table>

BLAS for Performance

- Development of blocked algorithms important for performance

| Intel Pentium 4 w/SSE2 1.7 GHz |
|--------|--------|--------|
| Level 3 BLAS | Level 2 BLAS | Level 1 BLAS |

| IBM RS/6000-500 (66 MHz, 364 Mflop/s Peak) |
|--------|--------|--------|
| Level 3 BLAS | Level 2 BLAS | Level 1 BLAS |

| Alpha EV 5/6 500MHz (1Gflop/s peak) |
|--------|--------|--------|
| Level 3 BLAS | Level 2 BLAS | Level 1 BLAS |

Fast linear algebra kernels: BLAS

- Simple linear algebra kernels such as matrix-matrix multiply
- More complicated algorithms can be built from these basic kernels.
- The interfaces of these kernels have been standardized as the Basic Linear Algebra Subroutines (BLAS).
- Early agreement on standard interface (~1980)
- Led to portable libraries for vector and shared memory parallel machines.
- On distributed memory, there is a less-standard interface called the PBLAS
Level 2 BLAS

- Operate on a matrix and a vector;
  - return a matrix or a vector;
  - $O(n^2)$ operations
- **sgemv**: matrix-vector multiply
  - $y = y + A \cdot x$
  - where $A$ is m-by-n, $x$ is n-by-1 and $y$ is m-by-1.
- **sger**: rank-one update
  - $A = A + y \cdot x^T$, i.e., $A(i,j) = A(i,j) + y(i) \cdot x(j)$
  - where $A$ is m-by-n, $y$ is m-by-1, $x$ is n-by-1,
- **strsv**: triangular solve
  - solves $y = T \cdot x$ for $x$, where $T$ is triangular

Level 3 BLAS

- Operate on pairs or triples of matrices
  - returning a matrix;
  - complexity is $O(n^3)$
- **sgemm**: Matrix-matrix multiplication
  - $C = C + A \cdot B$
  - where $C$ is m-by-n, $A$ is m-by-k, and $B$ is k-by-n
- **strsm**: multiple triangular solve
  - solves $Y = T \cdot X$ for $X$
  - where $T$ is a triangular matrix, and $X$ is a rectangular matrix.

Review of the BLAS

- Building blocks for all linear algebra
- Parallel versions call serial versions on each processor
- So they must be fast!
- The larger is $q$, the faster the algorithm can go in the presence of memory hierarchy
- "axpy": $y = \alpha \cdot x + y$, where $\alpha$ scalar, $x$ and $y$ vectors

<table>
<thead>
<tr>
<th>BLAS level</th>
<th>Ex.</th>
<th># mem refs</th>
<th># flops</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&quot;Axpy&quot;; Dot prod</td>
<td>3n</td>
<td>2n$^2$</td>
<td>2/3</td>
</tr>
<tr>
<td>2</td>
<td>Matrix-vector mul</td>
<td>n$^2$</td>
<td>2n$^2$</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Matrix-matrix mul</td>
<td>4n$^2$</td>
<td>2n$^2$</td>
<td>n/2</td>
</tr>
</tbody>
</table>

Success Stories for ScaLAPACK

- New Science discovered through the solution of dense matrix systems
- ScaLAPACK is a library for dense and banded matrices
  - Nature article on the flat universe used ScaLAPACK
  - Other articles in Physics Review B that also use it
  - 1998 Gordon Bell Prize
  - www.nersc.gov/news/reports/newNERSC/Results050703.pdf
  - Joint effort between DOE, DARPA, and NSF

Motivation (1)

3 Basic Linear Algebra Problems
1. Linear Equations: Solve $Ax=b$ for $x$
2. Least Squares: Find $x$ that minimizes $||r||_2^2 = \sqrt{\sum r_i^2}$ where $r=Ax-b$
   - Statistics: Fitting data with simple functions
3a. Eigenvalues: Find $\lambda$ and $x$ where $Ax = \lambda x$
   - Vibration analysis, e.g., earthquakes, circuits
3b. Singular Value Decomposition: $A^T Ax = \sigma^2 x$
   - Data fitting, Information retrieval

Lots of variations depending on structure of $A$
- A symmetric, positive definite, banded, ...

Motivation (2)

- Why dense $A$, as opposed to sparse $A$?
  - Many large matrices are sparse, but ...
  - Dense algorithms easier to understand
  - Some applications yields large dense matrices
  - LINPACK Benchmark (www.top500.org)
    - "How fast is your computer?" = "How fast can you solve dense $Ax=b$?"
  - Large sparse matrix algorithms often yield smaller (but still large) dense problems
## Winner of TOPS 500 (LINPACK Benchmark)

<table>
<thead>
<tr>
<th>Year</th>
<th>Machine</th>
<th>Tflops</th>
<th>Factor Faster</th>
<th>Peak Tflops</th>
<th>N</th>
<th>N Proc</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004</td>
<td>Blue Gene / L, IBM</td>
<td>70.7</td>
<td>2.9</td>
<td>91.8</td>
<td>32768</td>
<td>3.3M</td>
</tr>
<tr>
<td>2002</td>
<td>Earth System Computer, NEC</td>
<td>35.6</td>
<td>4.9</td>
<td>48.2</td>
<td>51004</td>
<td>5.04M</td>
</tr>
<tr>
<td>2001</td>
<td>ASCI White, IBM SP Power 3</td>
<td>7.2</td>
<td>1.5</td>
<td>11.1</td>
<td>7424</td>
<td>5.3M</td>
</tr>
<tr>
<td>2000</td>
<td>ASCI White, IBM SP Power 3</td>
<td>4.9</td>
<td>2.1</td>
<td>11.1</td>
<td>7424</td>
<td>4.3M</td>
</tr>
<tr>
<td>1999</td>
<td>ASCI Red, Intel P6 M200 MHz</td>
<td>2.4</td>
<td>1.1</td>
<td>3.2</td>
<td>9932</td>
<td>2.6M</td>
</tr>
<tr>
<td>1998</td>
<td>ASCI Blue, IBM SP 604E</td>
<td>2.1</td>
<td>1.6</td>
<td>3.9</td>
<td>5808</td>
<td>4.3M</td>
</tr>
<tr>
<td>1997</td>
<td>ASCI Red, Intel P600, 200 MHz</td>
<td>1.3</td>
<td>3.8</td>
<td>1.8</td>
<td>9192</td>
<td>2.4M</td>
</tr>
<tr>
<td>1996</td>
<td>Maclachi/CP-ACS</td>
<td>.37</td>
<td>1.3</td>
<td>.4</td>
<td>2048</td>
<td>1.0M</td>
</tr>
<tr>
<td>1995</td>
<td>Intel Paragon XP/S MP</td>
<td>.38</td>
<td>1.1</td>
<td>.3</td>
<td>6188</td>
<td>1.3M</td>
</tr>
</tbody>
</table>

Source: Jack Dongarra (UK)

## Current Records for Solving Small Dense Systems

[www.netlib.org](http://www.netlib.org), click on Performance Database Server

<table>
<thead>
<tr>
<th>Megaflops</th>
</tr>
</thead>
</table>

### Machine

- **NEC SX II**
  - (8 proc, 2 GHz)
  - Peak: 75140
  - (1 proc, 2 GHz)
  - Peak: 149060

### Palm Pilot III
- 0.00169

---

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

- **Zero it out below the diagonal by adding multiples of row i to later rows**

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

## Refine GE Algorithm (1)

- **Initial Version**

  ```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)
  ```

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

## 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+1 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

## Refine GE Algorithm (3)

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

  Store multipliers m below diagonal in zeroed entries for later use

---

## Gaussian Elimination (GE) for solving 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 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**

  ```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)
  ```

  Store m here
Refine GE Algorithm (4)

• Last version

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

• Split Loop

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

Store all m’s here before updating rest of matrix

Refine GE Algorithm (5)

• Last version

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

• Express using matrix operations (BLAS)

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

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 \( \text{(cost = } 2/3 n^3 \text{ flops)} \)
- 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…)

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

Pivoting in Gaussian Elimination

- A = \[
\begin{pmatrix}
0 & 1 \\
1 & 0
\end{pmatrix}
\] fails completely because can’t divide by A(1,1)=0
- But solving A\*x=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

\[
\begin{align*}
\text{for } i = 1 \text{ to } n-1 \\
\text{find and record } k \text{ where } |A(k,i)| = \max_{i \leq j \leq n} |A(j,i)| \\
\text{... i.e. largest entry in rest of column } i \\
\text{if } |A(k,i)| = 0 \\
\text{exit with a warning that } A \text{ is singular, or nearly so} \\
\text{else } k = i \\
\text{swap rows } i \text{ and } k \text{ of } A \\
\text{end if}
\end{align*}
\]

• Lemma: This algorithm computes A = PL*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/singular/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...)

\[
\begin{align*}
\text{for } i = 1 \text{ to } n+1: & \\
A(i+1:n,i) = A(i+1:n,i+1) & \text{ BLAS 1 (scale a vector)} \\
A(i+1:n,i) = A(i+1:n,i) + bA(i+1:n,i+1) & \text{ BLAS 2 (rank-1 update)} \\
A(i+1:n,i) & \text{ BLAS 3 (scale a vector)}
\end{align*}
\]

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 \(ib = 0\) to \(n-1\) step \(ib\)
  - Process matrix b columns at a time
  - \(ib = ib + 1\)
  - Point to end of block of \(ib\) columns
  - Apply BLAS2 version of GEPP to: \(\text{get } A(ib:ib+ib-1, ib:end) + P' \cdot L' \cdot U'\)
  - Let \(L\) denote the strict lower triangular part of \(A(ib:end, ib+ib-1)\)
  - \(L = L' \cdot U' \cdot A(ib+ib-1, ib:end)\)
  - Update next b rows of \(U\)
  - \(A(ib+ib-1, ib:end) = A(ib+ib-1, ib+ib-1) \cdot A(ib+ib-1, end-ib)\)
  - Apply delayed updates with single matrix-multiply
  - with inner dimension \(b\)

Efficiency of Blocked GEPP

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

Performance of LAPACK (n=1000)
**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
  - This is too fine-grained, prefer calls to local matmults instead
  - Need to use parallel matrix multiplication
  - Assignment
  - Which processors are responsible for which submatrices?

**Review of Parallel MatMul**

- **Want Large Problem Size Per Processor**
  - `DGEMM = BLAS 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 `DGEMM`
    - # 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

**Review: BLAS 3 (Blocked) GEPP**

- for $ib = 1$ to $n-1$ steps $b$
  - Process matrix block columns at a time
  - apply BLAS2 version of GEPP to get $Aib,nb + ib,nb = P' * L' * U'$
  - $ib,nb$ denotes the $ib$th lower triangular part of $Aib,ib$ and $ib,nb$ + 1
  - $Aib,ib$ denotes $A(ib,ib)$ and $ib,nb$ + 1
  - $P' * L' * U'$

**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 ≠ pont` 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.

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
  - Eigensproblems, SVD
    - Nonsymmetric eigenproblem hardest
  - Narrow band problems hardest (to do BLAS3 or parallelize)
    - Solving and eigenproblems
- www.netlib.org/lapack,scalapack
ScaLAPACK Overview

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
  • About half of matrix multiply for large enough matrices.
  • From the flop counts we would expect it to be \((2n^3)/((2/3)n^3) = 3\) times faster, but communication makes it a little slower.

PDGESV = ScaLAPACK Parallel LU

QR (Least Squares)

Scales well, nearly full machine speed

Current algorithm:
• Faster than initial algorithm
• Occasional numerical instability
• New, faster and more stable algorithm planned

Initial algorithm:
• Numerically stable
• Easily parallelized
• Slow: will abandon

Scalable Symmetric Eigensolver and SVD

The "Holy Grail" (Parlett, Dhillon, Marques)
• Perfect Output complexity \(O(n \times \text{#vectors})\)
• Embarrassingly parallel
• Accurate

Have good ideas to speedup
Project available!

Performance of ScaLAPACK LU

Performance of ScaLAPACK QR (Least squares)

Performance of Symmetric Eigensolvers

Performance of SVD (Singular Value Decomposition)

Performance of Nonsymmetric Eigensolver (QR iteration)
Scalable Nonsymmetric Eigensolver

- \( A x = \lambda x \), Schur form \( A = QTQ^T \)
- Parallel HQR
  - Henry, Watkins, Dongarra, Van de Geijn
  - Now in ScaLAPACK
  - Not as scalable as LU: \( n \) times as many messages
  - Block-Hankel data layout better in theory, but not in ScaLAPACK
- Sign Function
  - Beavers, Denman, Lin, Znijewski, Bai, Demmel, Gu, Godunov, Bulgakov, Malyshev
  - \( A_{i+1} = (A_i + A_{i-1})/2 \rightarrow \) shifted projector onto \( \Re \lambda > 0 \)
  - Repeat on transformed \( A \) to divide-and-conquer spectrum
  - Only uses inversion, so scalable
  - Inverse free version exists (uses QRD)
  - Very high flop count compared to HQR, less stable

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

Gaussian Elimination via a Recursive Algorithm

F. Gustavson and S. Toledo

LU Algorithm:
1: Split matrix into two rectangles \((n \times n/2)\)
   - 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 \( A_{02} = L^{-1}A_{02} \) and matrix multiplication \( A_{02} = A_{02} - A_{01}A_{12} \))
4: Apply LU Algorithm to right part

Most of the work in the matrix multiply
Matrices of size \( n/2, n/4, n/8, \ldots \)

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: $A = L^*D*LT$
    - 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)

Next release of LAPACK and ScaLAPACK

- Class projects available
- New or improved LAPACK algorithms
  - Faster and/or more accurate routines for linear systems, least squares, eigenvalues, SVD
- Parallelizing algorithms for ScaLAPACK
  - Many LAPACK routines not parallelized yet
- Automatic performance tuning
  - Many tuning parameters in code

Some contributors (incomplete list)

<table>
<thead>
<tr>
<th>Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nick Austin (UC Berkeley)</td>
</tr>
<tr>
<td>Ruth Baker (UC Berkeley)</td>
</tr>
<tr>
<td>John Beattie (North Carolina State)</td>
</tr>
<tr>
<td>Tony Berger (UC Berkeley)</td>
</tr>
<tr>
<td>John Demmel (UC Berkeley)</td>
</tr>
<tr>
<td>Nick Dines (UC Berkeley)</td>
</tr>
<tr>
<td>Robert Falgout (U Colorado)</td>
</tr>
<tr>
<td>Howard Elman (Johns Hopkins)</td>
</tr>
<tr>
<td>John Fike (U Maryland)</td>
</tr>
<tr>
<td>Mark Garvin (U Minnesota)</td>
</tr>
<tr>
<td>Tony Heroux (U Tennessee)</td>
</tr>
<tr>
<td>David Keyes (ORNL)</td>
</tr>
<tr>
<td>Edges Keyes (U Mississippi)</td>
</tr>
<tr>
<td>Michael McCool (UC Berkeley)</td>
</tr>
<tr>
<td>John McInnes (U Washington)</td>
</tr>
<tr>
<td>John Nichols (U Wisconsin)</td>
</tr>
<tr>
<td>Andrew Quinlan (U Kentucky)</td>
</tr>
<tr>
<td>Bruce Roe (U Tennessee)</td>
</tr>
<tr>
<td>Shun Xin Wei (UC Berkeley)</td>
</tr>
<tr>
<td>Bishnu P. Lamichhane (IBB)</td>
</tr>
<tr>
<td>Ben Wait (U Colorado)</td>
</tr>
<tr>
<td>Ross Walker (U Kentucky)</td>
</tr>
<tr>
<td>Nick White (UC Berkeley)</td>
</tr>
</tbody>
</table>

Some contributors (incomplete list)

- Some contributors (incomplete list)

Some contributors (incomplete list)

- Some contributors (incomplete list)