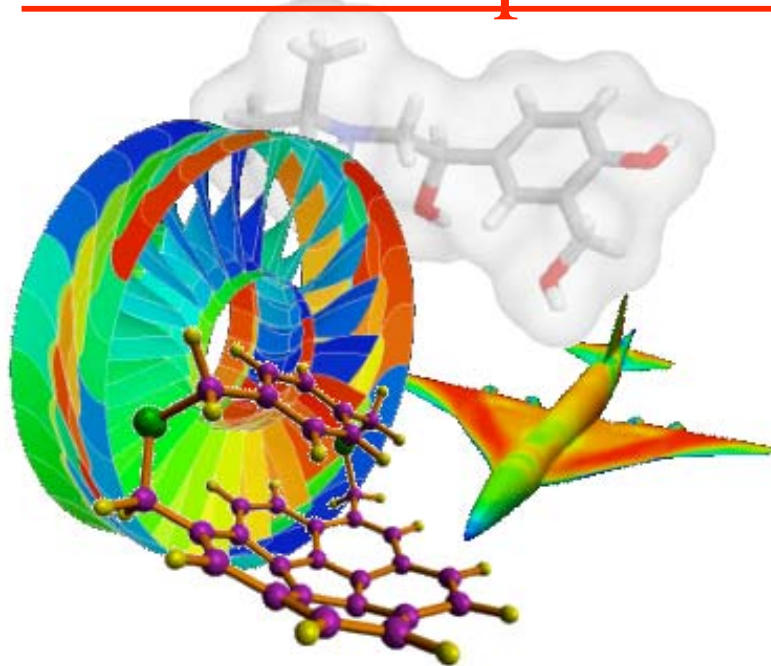


CME342 / AA220 / CS238 - Parallel Methods in Numerical Analysis

Matrix Computations: Direct Methods I



May 2, 2005

Lecture 14

Announcements

- Homework 2 due Wednesday
- Comments on HW1 late policy
- Overall weighting of homework vs. final project (65% / 35%)
- A portion of the notes comes from Prof. J. Demmel's **CS267** course at UC Berkeley

Outline of Next 3 Lectures

- **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

Outline of Next 3 Lectures

- 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

- Most 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

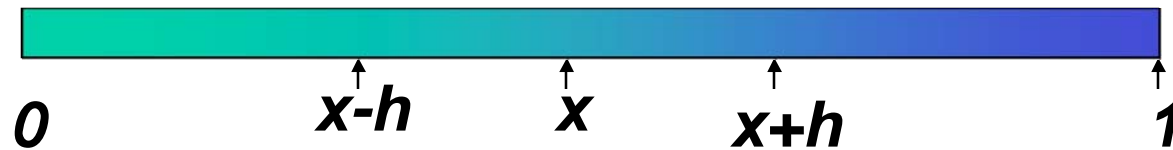
- It is true that these matrices are typically **banded**, and therefore, the cost of full factorization is not necessary. Bear with us.

Motivation: Continuous Variables, Continuous Parameters

Examples of such systems include

- Heat flow: $\text{Temperature}(\text{position}, \text{time})$
- Diffusion: $\text{Concentration}(\text{position}, \text{time})$
- Electrostatic or Gravitational Potential:
 $\text{Potential}(\text{position})$
- Fluid flow: $\text{Velocity, Pressure, Density}(\text{position}, \text{time})$
- Quantum mechanics: $\text{Wave-function}(\text{position}, \text{time})$
- Elasticity: $\text{Stress, Strain}(\text{position}, \text{time})$

Example: Deriving the Heat Equation



Consider a simple problem

- A bar of uniform material, insulated except at ends
- Let $u(x,t)$ be the temperature at position x at time t
- Heat travels from $x-h$ to $x+h$ at rate proportional to:

$$\frac{d u(x,t)}{dt} = C * \frac{(u(x-h,t)-u(x,t))/h - (u(x,t)- u(x+h,t))/h}{h}$$

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

$$\frac{d u(x,t)}{dt} = C * \frac{d^2 u(x,t)}{dx^2}$$

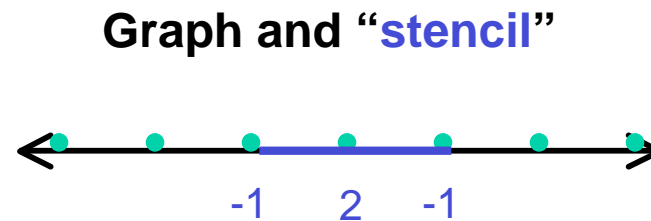
Implicit Solution

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

$$(I + (z/2)*T) * u[:,i+1] = (I - (z/2)*T) * u[:,i]$$

- Here I is the identity matrix and T is:

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



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

Algorithms for 2D Poisson Equation with N unknowns

Algorithm	Serial	PRAM	Memory	#Procs
• Dense LU	N^3	N	N^2	N^2
• Band LU	N^2	N	$N^{3/2}$	N
• Jacobi	N^2	N	N	N
• Explicit Inv.	N	$\log N$	N	N
• Conj.Grad.	$N^{3/2}$	$N^{1/2} * \log N$	N	N
• RB SOR	$N^{3/2}$	$N^{1/2}$	N	N
• Sparse LU	$N^{3/2}$	$N^{1/2}$	$N * \log N$	N
• FFT	$N * \log N$	$\log N$	N	N
• Multigrid	N	$\log^2 N$	N	N
• Lower bound	N	$\log N$	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)

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

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 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 and another one multiplied by a scalar
- 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 result 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 $O(mn)$ operations, where the matrices involved have size $m \times n$

BLAS Level 2 Routines

- Typical operations involve:

$$y = \alpha Ax + \beta y$$

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

$$y = Tx$$

$$y = T^T x$$

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

- 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 (look at “man blas” on junior)
- 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

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

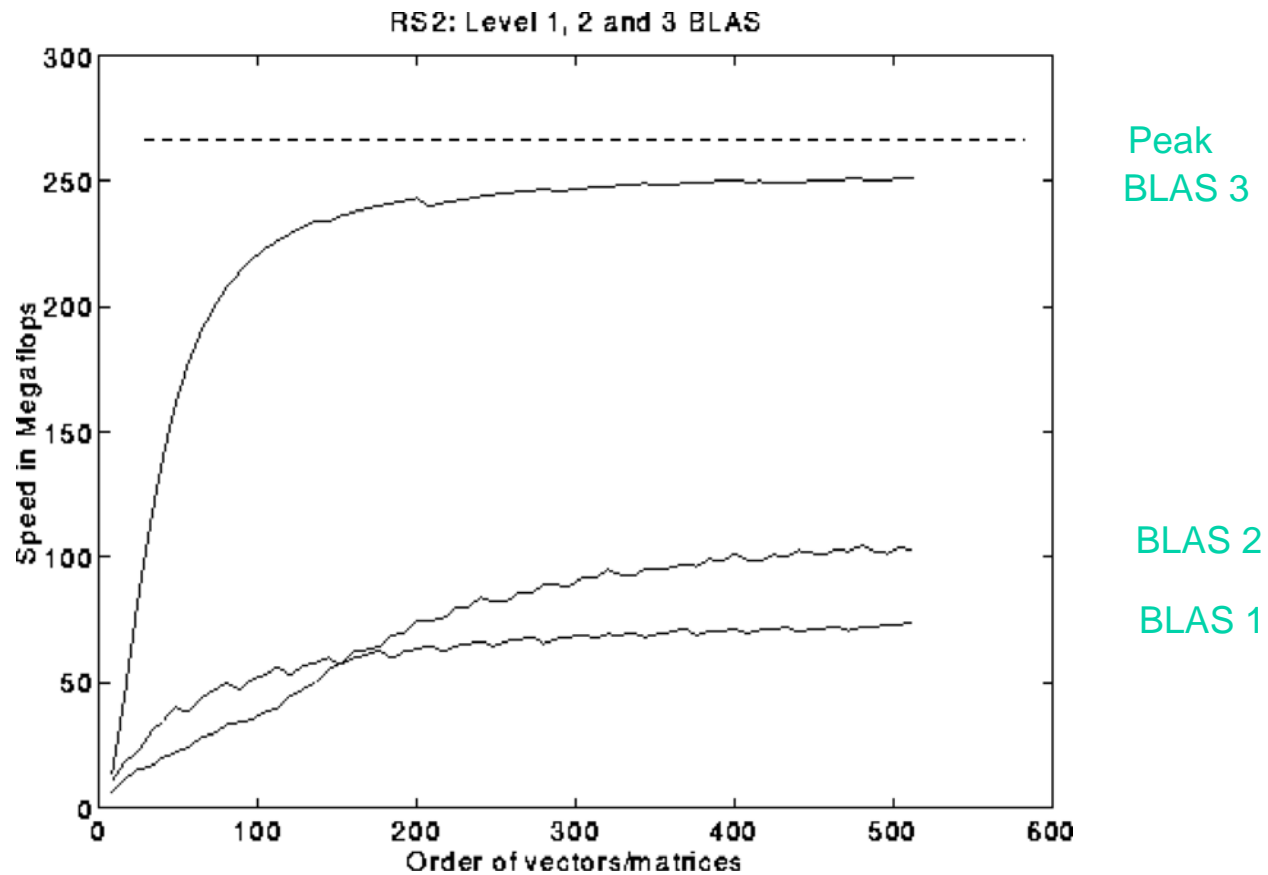
$$C = \alpha AA^T + \beta C$$

$$B = \alpha TB$$

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

- as well as rank-k updates and solutions of systems involving triangular matrices
- Better performance is achieved

BLAS Level 3 Routines



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

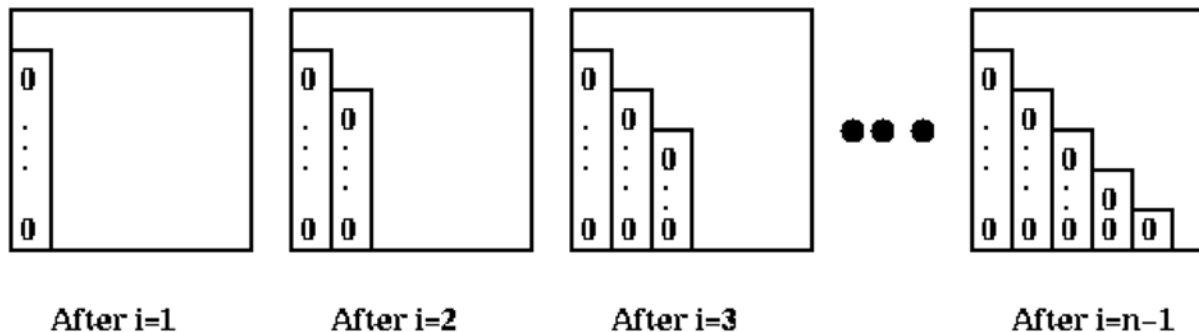
Task	Work	Parallelism	Parallel Sp
Fill	$O(n^2)$	embarrassing	low
→ Factor	$O(n^3)$	moderately diff.	very high
Solve	$O(n^2)$	moderately diff.	high
Field Calc.	$O(n)$	embarrassing	high

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 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
$$A(j,k) = A(j,k) - (A(j,i)/A(i,i)) * A(i,k)$$

Structure of Matrix during simple version of Gaussian Elimination



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)

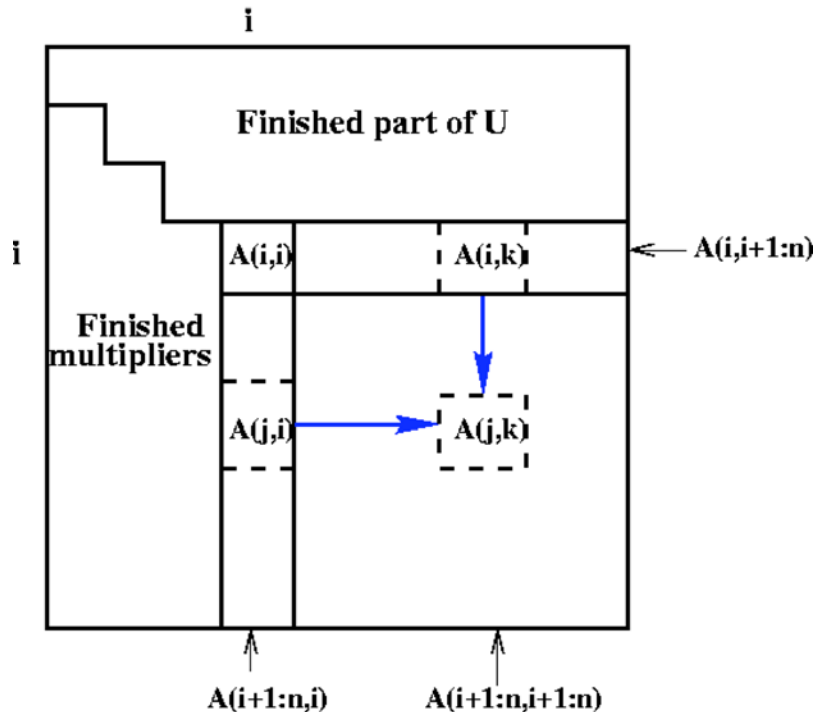
- Last version

```

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 matrix operations (BLAS)

Work at step i of Gaussian Elimination



```

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)
```

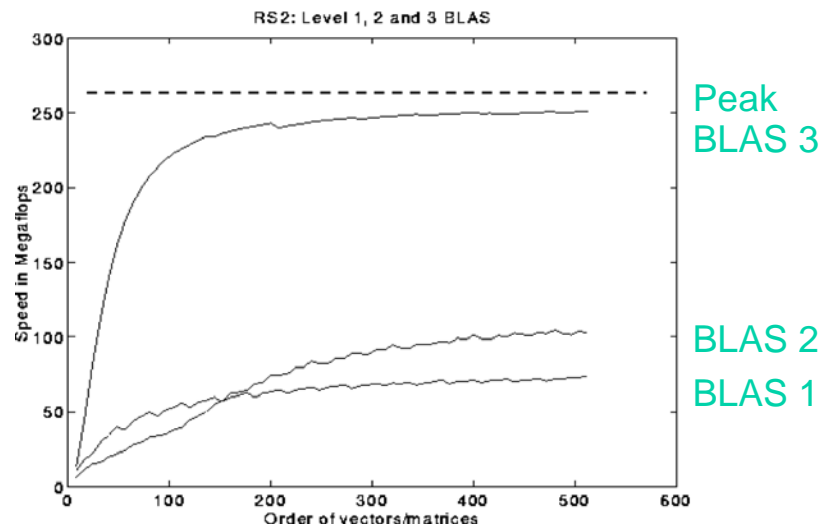
```
  A(i+1:n,i+1:n) = A(i+1:n , i+1:n ) - A(i+1:n , i) * A(i , i+1:n)
```

- 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

```
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)
```



Pivoting in Gaussian Elimination

- $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ fails completely, even though A is “easy”

- Illustrate problems in 3-decimal digit arithmetic:

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

- Result of LU decomposition is

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

$$U = \begin{bmatrix} 1e-4 & 1 \\ 0 & fl(1-1e4*1) \end{bmatrix} = \begin{bmatrix} 1e-4 & 1 \\ 0 & -1e4 \end{bmatrix} \quad \dots \text{ Error in 4th decimal place}$$

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

- 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 $|L(i,j)| = |A(j,i)/A(i,i)| \leq 1$

```
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
- Since each entry of $|L(i,j)| \leq 1$, this algorithm is considered numerically stable
- For details see LAPACK code at www.netlib.org/lapack/single/sgetf2 and Dongarra's book