Lecture 6:
Linear Algebra Algorithms

Jack Dongarra, U of Tennessee

Slides are adapted from Jim Demmel, [illegible]

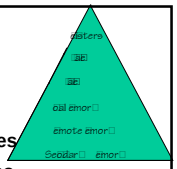
Algorithms and Architecture

- The key to performance is to understand the algorithm and architecture interaction.
- A significant improvement in performance can be obtained by matching algorithm to the architecture or vice versa.

Algorithm Issues

- Use of memory hierarchy
- Algorithm pre-fetching
- Loop unrolling
- Simulating higher precision arithmetic

Blocking



- TLB Blocking - minimize TLB misses
- Cache Blocking - minimize cache misses
- Register Blocking - minimize load/stores
- The general idea of blocking is to get the information to a high-speed storage and use it multiple times so as to amortize the cost of moving the data
- Cache blocking reduced traffic between memory and cache
- Register blocking reduces traffic between cache and CPU

Loop Unrolling

- Reduces data dependency delay
- Exploits multiple functional units and quad load/stores effectively.
- Minimizes load/stores
- Reduces loop overheads
- Gives more flexibility to compiler in scheduling
- Facilitates algorithm pre-fetching.
- What about vector computing?

Performance Numbers on RISC Processors

- Using Linpack Benchmark

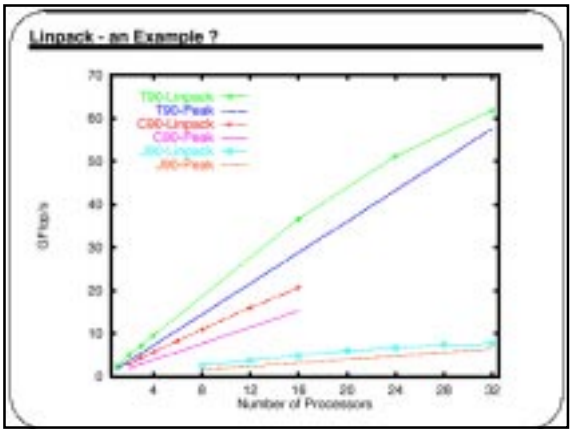
Machine	MHz	Linpack n=100 Mflop/s	Ax=b n=1000 Mflop/s	Peak Mflop/s
DEC Alpha	588	460 (40%)	847 (74%)	1150
IBM RS6K	375	409 (27%)	1236 (82%)	1500
HP PA	440	375 (21%)	1047 (59%)	1760
SUN Ultra	450	208 (23%)	607 (67%)	900
SGI Origin 2K	300	173 (29%)	553 (92%)	600
Intel PII Xeon	450	98 (22%)	295 (66%)	450
Cray T90	454	705 (39%)	1603 (89%)	1800
Cray C90	238	387 (41%)	902 (95%)	952
Cray Y-MP	166	161 (48%)	324 (97%)	333

What's Wrong With Speedup T_1/T_p ?

- Can lead to very false conclusions.
- Speedup in isolation without taking into account the speed of the processor is unrealistic and pointless.
- Speedup over what?
- T_1/T_p
  - There is usually no doubt about T_p
  - Often considerable dispute over the meaning of T_1
    - Serial code? Same algorithm?

Speedup

- Can be used to:
  - Study, in isolation, the scaling of one algorithm on one computer.
  - As a dimensionless variable in the theory of scaling.
- Should not be used to compare:
  - Different algorithms on the same computer
  - The same algorithm on different computers.
  - Different interconnection structures.



Strassen's Algorithm for Matrix Multiply

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Recall matrix multiply:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Strassen's Algorithm

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$P_2 = (A_{21} + A_{22})B_{11}$$
$$P_3 = A_{11}(B_{12} - B_{22})$$
$$P_4 = A_{22}(B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{12})B_{22}$$
$$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P_1 + P_4 - P_5 + P_7$$
$$C_{12} = P_3 + P_5$$
$$C_{21} = P_2 + P_5$$
$$C_{22} = P_1 + P_3 - P_2 + P_6$$

Strassen's Algorithm

- The count of arithmetic operations is:

	Floating Point Operations
Classical	$2n^3 - n^2$
Strassen	$7n^{2.807} - 6n^2$

One matrix multiply is replaced by 14 matrix additions

Strassen's Algorithm

° In reality the use of Strassen's Algorithm is limited by
- Additional memory required for storing the P matrices.
- More memory accesses are needed.

Outline

° Motivation for Dense Linear Algebra
- Ax=b: Computational Electromagnetics
- Ax = λx: Quantum Chemistry

° 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 matrix-matrix multiplication

° Parallel Gaussian Elimination

° ScaLAPACK library overview

° Eigenvalue problem

Parallelism in Sparse Matrix-vector multiplication

° y = A*x, where A is sparse and n x n

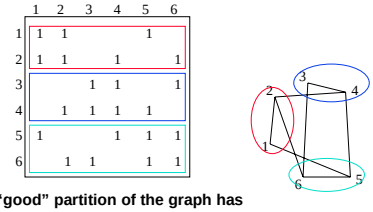
° Questions
- which processors store
  - y[i], x[i], and A[i,j]
- which processors compute
  - y[i] = sum (from 1 to n) A[i,j] * x[j]
    = (row i of A) . x … a sparse dot product

° Partitioning
- Partition index set {1,…,n} = N1 u N2 u … u Np
- For all i in Nk, Processor k stores y[i], x[i], and row i of A
- For all i in Nk, Processor k computes y[i] = (row i of A) . x
  - "owner computes" rule: Processor k compute the y[i]s it owns

° Goals of partitioning
- balance load (how is load measured?)
- balance storage (how much does each processor store?)
- minimize communication (how much is communicated?)

Graph Partitioning and Sparse Matrices

° Relationship between matrix and graph

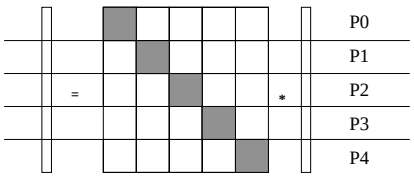


° A "good" partition of the graph has
- equal (weighted) number of nodes in each part (load and storage balance)
- minimum number of edges crossing between (minimize communication)

° Can reorder the rows/columns of the matrix by putting all the nodes in one partition together

More on Matrix Reordering via Graph Partitioning

° "Ideal" matrix structure for parallelism: (nearly) block diagonal
- p (number of processors) blocks
- few non-zeros outside these blocks, since these require communication



What about implicit methods and eigenproblems?

° Direct methods (Gaussian elimination)
- Called LU Decomposition, because we factor A = L*U
- Future lectures will consider both dense and sparse cases
- More complicated than sparse-matrix vector multiplication

° Iterative solvers
- Will discuss several of these in future
  - Jacobi, Successive overrelaxiation (SOR) , Conjugate Gradients (CG), Multigrid,...
- Most have sparse-matrix-vector multiplication in kernel

° Eigenproblems
- Future lectures will discuss dense and sparse cases
- Also depend on sparse-matrix-vector multiplication, direct methods

° Graph partitioning

Partial Differential Equations

PDEs

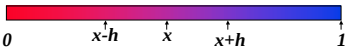
19

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: 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:

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

Explicit Solution of the Heat Equation

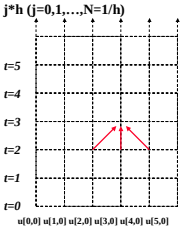
- For simplicity, assume $C=1$
- Discretize both time and position
- Use finite differences with u[j,i] as the heat at
  - time t= i*dt (i = 0,1,2,…) and position x = j*h (j=0,1,…,N=1/h)
  - initial conditions on u[j,0]
  - boundary conditions on u[0,i] and u[N,i]
- At each timestep i = 0,1,2,...

For j=0 to N

u[j,i+1]= z*u[j-1,i]+ (1-2*z)*u[j,i]+ z*u[j+1,i]

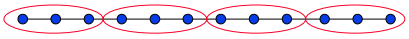
where z = dt/h²

- This corresponds to
  - matrix vector multiply (what is matrix?)
  - nearest neighbors on grid



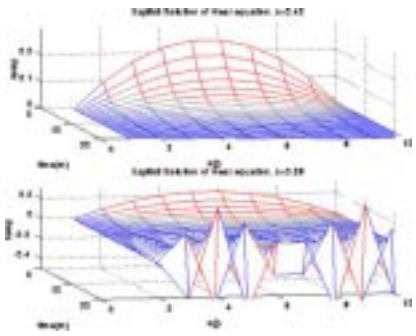
Parallelism in Explicit Method for PDEs

- Partitioning the space (x) into p largest chunks
  - good load balance (assuming large number of points relative to p)
  - minimized communication (only p chunks)



- Generalizes to
  - multiple dimensions
  - arbitrary graphs (= sparse matrices)
- Problem with explicit approach
  - numerical instability
  - solution blows up eventually if z = dt/h^2> .5
  - need to make the timesteps very small when h is small: dt < .5*h^2

Instability in solving the heat equation explicitly



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:

$T =$

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

Graph and "stencil"

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

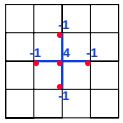
2D Implicit Method

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

$T =$

$$\begin{pmatrix} 4 & -1 & & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ -1 & & & 4 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{pmatrix}$$

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	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^2	log N	N^2	N^2
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	

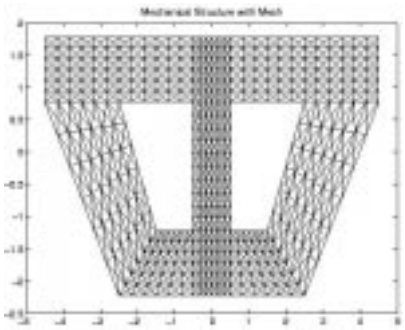
PRAM is an idealized parallel model with zero cost communication

(see next slide for explanation)

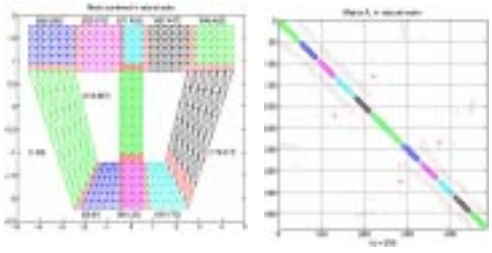
Short explanations of algorithms on previous slide

- Sorted in two orders (roughly):
  - from slowest to fastest on sequential machines
  - from most general (works on any matrix) to most specialized (works on matrices "like" T)
- Dense LU: Gaussian elimination; works on any N-by-N matrix
- Band LU: exploit fact that T is nonzero only on sqrt(N) diagonals nearest main diagonal, so faster
- Jacobi: essentially does matrix-vector multiply by T in inner loop of iterative algorithm
- Explicit Inverse: assume we want to solve many systems with T, so we can precompute and store inv(T) "for free", and just multiply by it
  - It's still expensive!
- Conjugate Gradients: uses matrix-vector multiplication, like Jacobi, but exploits mathematical properies of T that Jacobi does not
- Red-Black SOR (Successive Overrelaxation): Variation of Jacobi that exploits yet different mathematical properties of T
  - Used in Multigrid
- Sparse LU: Gaussian elimination exploiting particular zero structure of T
- FFT (Fast Fourier Transform): works only on matrices *very* like T
- Multigrid: also works on matrices like T, that come from elliptic PDEs
- Lower Bound: serial (time to print answer); parallel (time to combine N inputs)

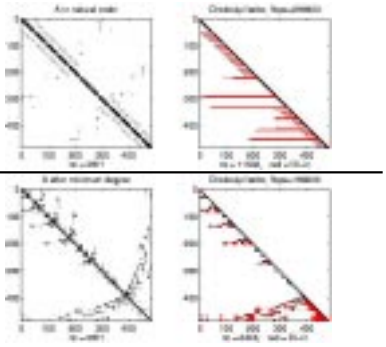
Composite mesh from a mechanical structure



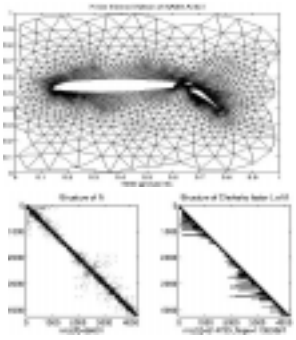
Converting the mesh to a matrix



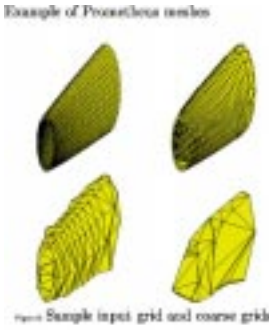
Effects of Ordering Rows and Columns on Gaussian Elimination



Irregular mesh: NASA Airfoil in 2D (direct solution)

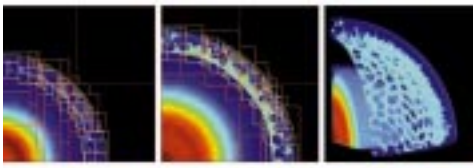


Irregular mesh: Tapered Tube (multigrid)



Sample input grid and coarse grid

Adaptive Mesh Refinement (AMR)



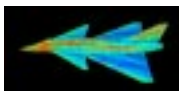
- Adaptive mesh around an explosion
- John Bell and Phil Colella at LBL (see class web page for URL)
- Goal of Titanium is to make these algorithms easier to implement in parallel

Computational Electromagnetics

- Developed during 1980s, driven by defense applications
- Determine the RCS (radar cross section) of airplane
- Reduce signature of plane (stealth technology)
- Other applications are antenna design, medical equipment
- Two fundamental numerical approaches:
  - MOM methods of moments (frequency domain), and
  - Finite differences (time domain)

Computational Electromagnetics

- Discretize surface into triangular facets using standard modeling tools
- Amplitude of currents on surface are unknowns



- Integral equation is discretized into a set of linear equations

image: NW Univ. Comp. Electromagnetics Laboratory http://nueml.ece.nwu.edu/

Computational Electromagnetics (MOM)

After discretization the integral equation has the form

$$A\ x = b$$

where

A is the (dense) impedance matrix,

x is the unknown vector of amplitudes, and

b is the excitation vector.

(see Cwik, Patterson, and Scott, Electromagnetic Scattering on the Intel Touchstone Delta, IEEE Supercomputing '92, pp 538 - 542)

Computational Electromagnetics (MOM)

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 excitations b
- Field Calc: computing the fields scattered from the object

Analysis of MOM for Parallel Implementation

Task	Work	Parallelism	Parallel Speed
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

Results for Parallel Implementation on Delta

Task	Time (hours)
Fill	9.20
Factor	8.25
Solve	2 .17
Field Calc.	0.12

The problem solved was for a matrix of size 48,672. (The world record in 1991.)

Current Records for Solving Dense Systems

Year	System Size	Machine	# Procs	Gflops	(Peak)
1950's	O(100)				
1995	128,600	Intel Paragon	6768	281	(338)
1996	215,000	Intel ASCI Red	7264	1068	(1453)
1998	148,000	Cray T3E	1488	1127	(1786)
1998	235,000	Intel ASCI Red	9152	1338	(1830)
1999	374,000	SGI ASCI Blue	5040	1608	(2520)
1999	362,880	Intel ASCI Red	9632	2379	(3207)

source: Alan Edelman http://www-math.mit.edu/~edelman/records.html
LINPACK Benchmark: http://www.netlib.org/performance/html/PDSreports.html

Computational Chemistry

- Seek energy levels of a molecule, crystal, etc.
  - Solve Schroedinger's Equation for energy levels = eigenvalues
  - Discretize to get $Ax = \lambda Bx$, solve for eigenvalues λ and eigenvectors x
  - A and B large, symmetric or Hermitian matrices (B positive definite)
  - May want some or all eigenvalues/eigenvectors
- MP-Quest (Sandia NL)
  - Si and sapphire crystals of up to 3072 atoms
  - Local Density Approximation to Schroedinger Equation
  - A and B up to n=40000, Hermitian
  - Need all eigenvalues and eigenvectors
  - Need to iterate up to 20 times (for self-consistency)
- Implemented on Intel ASCI Red
  - 9200 Pentium Pro 200 processors (4600 Duals, a CLUMP)
  - Overall application ran at 605 Gflops (out of 1800 Glops peak),
  - Eigensolver ran at 684 Gflops
  - www.cs.berkeley.edu/~stanley/gbell/index.html
  - Runner-up for Gordon Bell Prize at Supercomputing 98

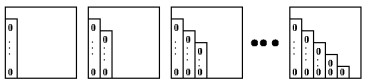
Review of Gaussian Elimination (GE) for solving Ax=b

° 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



After i=1 After i=2 After i=3 After i=n-1

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)

```
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

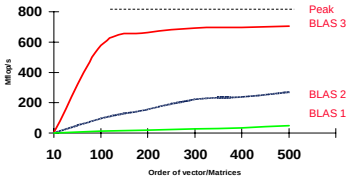
Problems with basic GE algorithm

° 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 (Lecture 2)

```
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)
```

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



Pivoting in Gaussian Elimination

° A = [0 1] fails completely, even though A is "easy"
[1 0]

° Illustrate problems in 3-decimal digit arithmetic:

A = [1e-4 1] and b = [1], correct answer to 3 places is x = [1]
[1 1] [2] [1]

° Result of LU decomposition is

L = [1 0] = [1 0] ... No roundoff error yet
[fl(1/1e-4) 1] [1e4 1]

U = [1e-4 1] = [1e-4 1] ... Error in 4th decimal place
[0 fl(1-1e4*1)] [0 -1e4]

Check if A = L*U = [1e-4 1] ... (2,2) entry entirely wrong
[1 0]

° 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

Converting BLAS2 to BLAS3 in 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

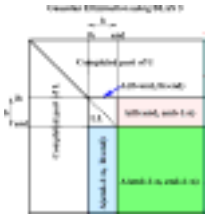
° Same idea works for much of dense linear algebra
- Open questions remain

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

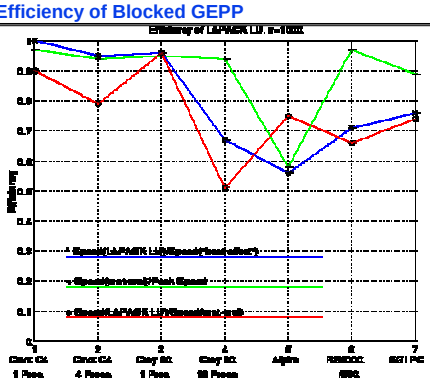
Blocked GEPP (www.netlib.org/lapack/single/sgetrf.f)

```
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 )
          - A(end+1:n , ib:end) * A(ib:end , end+1:n)
                    ... apply delayed updates with single matrix-multiply
                    ... with inner dimension b
```

(For a correctness proof, see on-lines notes.)



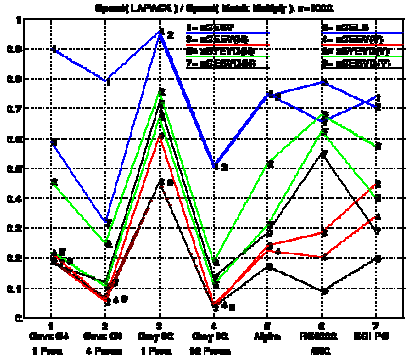
Efficiency of Blocked GEPP



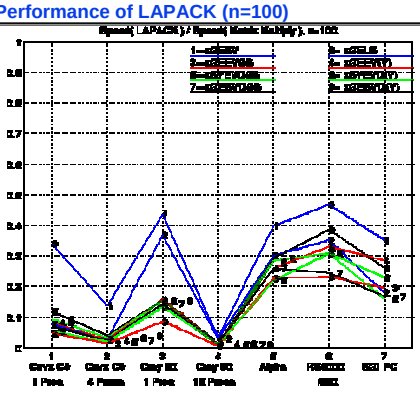
Overview of LAPACK

- 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
  - Projects available

Performance of LAPACK (n=1000)



Performance of LAPACK (n=100)



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

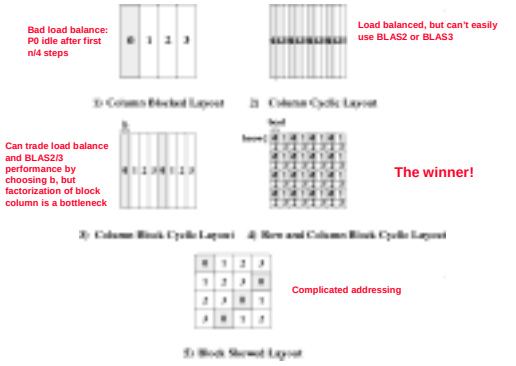
```
for i = 1 to n-1
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)          ... BLAS 1 (scale a vector)
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n )  ... BLAS 2 (rank-1 update)
        - A(i+1:n , i) * A(i , i+1:n)
```

  - This is too fine-grained, prefer calls to local matmuls instead

Assignment of parallel work in GE

- Think of assigning submatrices to threads, where each thread responsible for updating submatrix it owns
  - "owner computes" rule natural because of locality
- What should submatrices look like to achieve load balance?

Different Data Layouts for Parallel GE (on 4 procs)



Blocked Partitioned Algorithms

- LU Factorization
- Cholesky factorization
- Symmetric indefinite factorization
- Matrix inversion
- QR, QL, RQ, LQ factorizations
- Form Q or Q^TC
- Orthogonal reduction to:
  - (upper) Hessenberg form
  - symmetric tridiagonal form
  - bidiagonal form
- Block QR iteration for nonsymmetric eigenvalue problems

Memory Hierarchy and LAPACK

- ijk - implementations

```
for _  = 1:n;
   for _  = 1:n;
      for _  = 1:n;
         a_{i,j} ← a_{i,j} + b_{i,k} c_{k,j}
      end
   end
end
```

Effects order in which data referenced; some better at allowing data to keep in higher levels of memory hierarchy.

- Applies for matrix multiply, reductions to condensed form
  - May do slightly more
  - Up to 3 times faster

Derivation of Blocked Algorithms
Cholesky Factorization $A = U^TU$

$$\begin{pmatrix} A_{11} & a_j & A_{13} \\ a_j^T & a_{jj} & \alpha_j^T \\ A_{13}^T & \alpha_j & A_{33} \end{pmatrix} = \begin{pmatrix} U_{11}^T & 0 & 0 \\ u_j^T & u_{jj} & 0 \\ U_{13}^T & \mu_j & U_{33}^T \end{pmatrix} \begin{pmatrix} U_{11} & u_j & U_{13} \\ 0 & u_{jj} & \mu_j^T \\ 0 & 0 & U_{33} \end{pmatrix}$$

[illegible] of the [illegible] [illegible] [illegible]

$$a_j = U_{11}^T u_j$$

$$a_{jj} = u_j^T u_j + u_{jj}^2$$

[illegible] if [illegible] as already [illegible] computed, [illegible] compute [illegible] from the [illegible]

$$U_{11}^T u_j = a_j$$

$$u_{jj}^2 = a_{jj} - u_j^T u_j$$

LINPACK Implementation

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

```
      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.0.0E0 ) GO TO 40
         A( J, J ) = SQRT( S )
   30 CONTINUE
```

LAPACK Implementation

```
      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 72 to 251 Mflop/s for a matrix of order 500 on one processor of a CRAY Y-MP.
- However on 378 Mflop/s on 8 Procs. Of a CRAY Y-MP.
- Suggest further work needed.

Derivation of Blocked Algorithms

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

[illegible] of second [illegible] of the [illegible]

$$A_{12} = U_{11}^T U_{12}$$

$$A_{22} = U_{12}^T U_{12} + U_{22}^T U_{22}$$

[illegible] if [illegible] as already [illegible] computed, [illegible] compute [illegible] as the solution of the following [illegible] a call to [illegible] already [illegible]

$$U_{11}^T U_{12} = A_{12}$$

$$U_{22}^T U_{22} = A_{22} - U_{12}^T U_{12}$$

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', JB, A( J, J ), LDA, INFO )
   IF( INFO.NE.0 ) GO TO 20
10 CONTINUE
```

- [illegible] es a little more [illegible] of [illegible] pro[illegible] ma[illegible]es a lar[illegible] impro[illegible] eme[illegible] [illegible] pros [illegible]

[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]

LAPACK Contents

- Combines algorithms from LINPACK and EISPACK into a single package. User interface similar to LINPACK.
- Built on the L 1, 2 and 3 BLAS, for high performance (manufacturers optimize BLAS)
- LAPACK does not provide routines for structured problems or general sparse matrices (i.e sparse storage formats such as compressed-row, -column, -diagonal, skyline ...).

LAPACK Ongoing Work

- Add functionality
  - updating/downdating, divide and conquer least squares,bidiagonal bisection, bidiagonal inverse iteration, band SVD, Jacobi methods, ...
- Move to new generation of high performance machines
  - IBM SPs, CRAY T3E, SGI Origin, clusters of workstations
- New challenges
  - New languages: FORTRAN 90, HP FORTRAN, ...
  - (CMMD, MPL, NX ...)
    - many flavors of message passing, need standard (PVM, MPI): BLACS
- Highly varying ratio $\frac{\text{Computational speed}}{\text{Communication speed}}$
- Many ways to layout data,
- Fastest parallel algorithm sometimes less stable numerically.

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

- LU Factorization
- Cholesky factorization
- Symmetric indefinite factorization
- Matrix inversion
- QR, QL, RQ, LQ factorizations
- Form Q or Q^TC
- Orthogonal reduction to:
  - (upper) Hessenberg form
  - symmetric tridiagonal form
  - bidiagonal form
- Block QR iteration for nonsymmetric eigenvalue problems