Lecture plan

- Last time: Using the CRC, Floating Point
- This time: Dense Linear Algebra
  - BLAS
  - LAPACK
  - Parallel matrix multiplication
  - Parallel Gaussian Elimination

Linear Algebra Overview

Basic problems

- Linear systems: $Ax = b$
- Least squares: minimize $||Ax - b||_2^2$
- Eigenvalues: $Ax = \lambda x$

Basic paradigm: matrix factorization

- $A = LU$, $A = LL^T$
- $A = QR$
- $A = V \wedge V^{-1}$, $A = QTQ^T$
- $A = U\Sigma V^T$

Factorization: switch to a basis that makes problem easy

Sparse vs. Dense Linear Algebra

Dense == common structures, no complicated indexing

- General dense (all entries nonzero)
- Banded (zero below/above some diagonal)
- Symmetric/Hermitian
- Standard, robust algorithms (LAPACK)

Sparse == Not stored in dense form!

- Maybe few nonzeros (e.g. compressed sparse row formats)
- May be implicit (e.g. via finite differencing)
- May be "dense", but with compact representation (e.g. via FFT)
- Most algorithms are iterative; wider variety, more subtle

And... A little History

BLAS 1 (1973-1977)

- Standard library of 15 operators (mostly) on vectors
- Up to four versions of each: S/D/C/Z
- Example: DAXPY: Double precision (real), Computes Ax + y
- Goals
  - Raise level of programming abstraction
  - Robust implementation (e.g. avoid over/underflow)
  - *Portable interface*, efficient machine-specific implementation

And... A little History, cont'd

BLAS 2 (1984-1986)

- Standard library of 25 ops (mostly) on matrix/vector pairs
- Different data types and matrix types
- Example: DGEMV: Double precision, GEneral matrix, Matrix-Vector product
- Goals
  - BLAS1 insufficient
  - BLAS2 for better vectorization (when vector machines roamed the earth)
- BLAS2 $==$ $O(n^2)$ ops on $O(n^2)$ data

And... A little History, even more...

BLAS 3 (1987-1988)

- Standard library of 9 ops (mostly) on matrix/matrix
- Different data types and matrix types
- Example: DGEMM: Double precision, GEneral matrix, Matrix-Matrix product
- BLAS3 $==$ $O(n^3)$ ops on $O(n^2)$ data
- Goals: Efficient cache utilization
- 142 routines, 31K LOC

LAPACK

LAPACK (1989-present): `http://www.netlib.org/lapack`

- Supercedes earlier LINPACK and EISPACK
- High performance through BLAS
- Parallel to the extent BLAS are parallel (on SMP)
- Linear systems and least squares are nearly 100% BLAS 3
- Eigenproblems, SVD - only about 50% BLAS 3
- Careful error bounds on everything
- Lots of variants for different structures
- 1750 routines, 721K LOC (ouch!)

LAPACK name decoder

Fortran 77 (F77) : limited characters per name

- Data type (double/single/double complex/single complex)
- Matrix type (general/symmetric, banded/not banded)
- Operation type
- Example: DGETRF: Double precision, GEneral matrix, TRiangular Factorization
- Example: DSYEVX: Double precision, General SYmmetric matrix, EigenValue computation, eXpert driver

Structures

- General: general (GE), banded (GB), pair (GG), tridiag (GT)
- Symmetric: general (SY), banded (SB), packed (SP), tridiag (ST)
- Hermitian: general (HE), banded (HB), packed (HP)
- Positive definite (PO), packed (PP), tridiagonal (PT)
- Orthogonal (OR), orthogonal packed (OP)
- Unitary (UN), unitary packed (UP)
- Hessenberg (HS), Hessenberg pair (HG)
- Triangular (TR), packed (TP), banded (TB), pair (TG)
- Bidiagonal (BD)

LAPACK routines

- Linear systems (general, symmetric, SPD)
- Least squares (overdetermined, underdetermined, constrained, weighted)
- Symmetric eigenvalues and vectors
  - Standard: $Ax = \lambda x$
  - Generalized: $Ax = \lambda Bx$
- Nonsymmetric eigenproblems
  - Schur form: $A = QTQ^T$
  - Eigenvalues/vectors
  - Invariant subspaces
  - Generalized variants
  - SVD (standard/generalized)
- Different interfaces
  - Simple drivers
  - Expert drivers with error bounds, extra precision, etc
  - Low-level routines

Algorithm optimization

Running time of an algorithm is sum of 3 terms:

1. number of flops * time per flop
2. number of words moved / bandwidth
3. number of messages * latency

Time per flop $<<$ 1/ bandwidth $<<$ latency

Algorithm goals

- Minimize number of words moved
- Minimize number of messages sent: Need new data structures
- Minimize for multiple memory hierarchy levels:
  Cache-oblivious algorithms would be simplest
- Fewest flops when matrix fits in fastest memory:
  Cache-oblivious algorithms don't always attain this

Matrix vector product

Simple $y = Ax$ involves two indices: $y_i = \sum_j A_{ij} x_j$
Can organize around either one:

```
% Row-oriented
for i = 1:n
    y(i) = A(i,:)*x;
end
% Column-oriented
y = 0;
for j = 1:n
    y = y + A(:,j)*x(j);
end
```

Parallel matvec: 1D row-blocked

Example: 3 processors, 3 rows

Receive broadcast vector x_0, x_1, x_2 into local x_0, x_1, x_2; then

- On P0: $A_{00}x_0 + A_{01}x_1 + A_{02}x_2 = y_0$
- On P1: $A_{10}x_0 + A_{11}x_1 + A_{12}x_2 = y_1$
- On P2: $A_{20}x_0 + A_{21}x_1 + A_{22}x_2 = y_2$

Parallel matvec: 1D column-blocked

Example: 3 processors, 3 columns

Receive broadcast vector x_0, x_1, x_2 into local x_0, x_1, x_2; then

- On P0: $z_0 = \begin{bmatrix} A_{00} \\ A_{10} \\ A_{20} \end{bmatrix} x_0$
- On P1: $z_1 = \begin{bmatrix} A_{00} \\ A_{10} \\ A_{20} \end{bmatrix} x_1$
- On P2: $z_2 = \begin{bmatrix} A_{00} \\ A_{10} \\ A_{20} \end{bmatrix} x_2$

And perform reduction: $y = z_0 + z_1 + z_2$

Parallel matvec: 2D blocked

Involves broadcast and reduction ... but with subsets of processors

Broadcast x_0, x_1 to local copies x_0, x_1 at P0 and P2

Broadcast x_2, x_3 to local copies x_2, x_3 at P1 and P3

In parallel, compute:

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} z_0^{(0)} \\ z_1^{(0)} \end{bmatrix}$$

$$\begin{bmatrix} A_{20} & A_{21} \\ A_{30} & A_{31} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} z_0^{(1)} \\ z_0^{(1)} \end{bmatrix}$$

$$\begin{bmatrix} A_{02} & A_{03} \\ A_{12} & A_{13} \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}$$

$$\begin{bmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} z_2^{(3)} \\ z_3^{(3)} \end{bmatrix}$$

Reduce across rows:

$$\begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} z_0^{(0)} \\ z_1^{(0)} \end{bmatrix} \begin{bmatrix} z_0^{(1)} \\ z_1^{(1)} \end{bmatrix}, \quad \begin{bmatrix} y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} z_0^{(2)} \\ z_1^{(2)} \end{bmatrix} \begin{bmatrix} z_0^{(3)} \\ z_1^{(3)} \end{bmatrix}$$

Parallel matmul: Complexity

- Basic operation: $C = C + AB$
- Computation: $O(2n^3)$ Flops
- Goal: $O(2n^3/p)$ Flops per processor, minimal communication

Parallel matmul: 1D Layout

- Block MATLAB notation: `A(:, j)` means jth block column
- Processor j owns `A(:, j)`, `B(:, j)`, `C(:, j)`
- `C(:, j)` depends on all of A, but only `B(:, j)`

How do we communicate pieces of A?

1D layout on bus architecture

- Everyone computes local contributions first
- P0 sends `A(:, 0)` to each processor j in turn; processor j receives, computes `A(:, 0)B(0, j)`
- P1 sends `A(:, 1)` to each processor j in turn; processor j receives, computes `A(:, 1)B(1, j)`
- P2 sends `A(:, 2)` to each processor j in turn; processor j receives, computes `A(:, 2)B(2, j)`

1D layout on bus (no broadcast)

```
C(:,myproc) += A(:,myproc)*B(myproc,myproc)
for i = 0:p-1
    for j = 0:p-1
    if (i == j) continue;
    if (myproc == i)
        send A(:,i) to processor j
    if (myproc == j)
        receive A(:,i) from i
        C(:,myproc) += A(:,i)*B(i,myproc)
    end
end
end
```

Outer product algorithm

Serial: Recall outer product organization:

```
for k = 0:s-1
    C += A(:,k)*B(k,:);
end
```

Parallel: Assume $p = s^2$ processors, block $s \times s$ matrices.

For a 2×2 example:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{bmatrix} + \begin{bmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{bmatrix}$$

- Processor for each (i, j) $\to$ parallel work for each k !
- Note everyone in row i uses A(i, k) at once,
- and everyone in row j uses B(k , j) at once.

Parallel outer product (SUMMA)

```
for k = 0:s-1
    for each i in parallel
        broadcast A(i,k) to row
    for each j in parallel
        broadcast A(k,j) to col
    On processor (i,j), C(i,j) += A(i,k)*B(k,j);
end
```

Gaussian Elimination

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

Improving Gaussian Elimination

- Remove computation of constant `tmp/A(i,i)` from inner loop.
- Don't compute what we already know: zeros below diagonal in column i
- Store multipliers m below diagonal in zeroed entries for later use

But!

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

Distributed Gaussian Elimination

- Decompose into work chunks (matrix organization)
- Assign work to threads in a balanced way
- Orchestrate the communication and synchronization
- Map which processors execute which threads

1D column blocked: bad load balance

P0 is idle after n/3 steps.

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \end{bmatrix}$$

1D column cyclic: hard to use BLAS2/3

Load balanced but ...

$$\begin{bmatrix} 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \\ 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \\ 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \\ 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \\ 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \\ 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \\ 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \\ 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \\ 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \end{bmatrix}$$

1D column block cyclic: block column factorization a bottleneck

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Block skewed: indexing gets messy

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\ 2 & 2 & 2 & 0 & 0 & 0 & 1 & 1 & 1 \\ 2 & 2 & 2 & 0 & 0 & 0 & 1 & 1 & 1 \\ 2 & 2 & 2 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 & 2 & 2 & 0 & 0 & 0 \\ 1 & 1 & 1 & 2 & 2 & 2 & 0 & 0 & 0 \\ 1 & 1 & 1 & 2 & 2 & 2 & 0 & 0 & 0 \end{bmatrix}$$

2D Row and Column Blocked Layout

(Bad load balance: P0 idle after first n/2 steps)

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\ 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\ 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\ 2 & 2 & 3 & 2 & 3 & 3 & 3 & 3 \end{bmatrix}$$

2D block cyclic

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 2 & 2 & 3 & 3 & 2 & 2 & 3 & 3 \\ 2 & 2 & 3 & 3 & 2 & 2 & 3 & 3 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 2 & 2 & 3 & 3 & 2 & 2 & 3 & 3 \\ 2 & 2 & 3 & 3 & 2 & 2 & 3 & 3 \end{bmatrix}$$

Matrix layout summary

- 1D column blocked: bad load balance
- 1D column cyclic: hard to use BLAS2/3
- 1D column block cyclic: factoring column is a bottleneck
- Block skewed (a la Cannon): just complicated
- 2D row/column block: bad load balance
- 2D row/column block cyclic: win

Distributed Gaussian Elimination with 2D Cyclic Layout

```
for ib = 1 to n-1 step b
    end = min(ib + b-1, n)
    for i = ib to end
        find pivot row k, column broadcast
        swap rows k and i in block column, broadcast row k
        A(i+1:n,i) = A(i+1:n,i) / A(i,i)
        A(i+1:n, i+1:end) = A(i+1:n,i) * A(i,i+1:end)
    end for
broadcast all swap information left and right (sharing pivot)
apply all row swaps to other columns
broadcast LL right
A(ib:end, end+1:n) = LL  A(ib:end, end+1:n)
broadcast A(ib:end, end+1:n) down
broadcast A(end+1:n, ib:end) right
eliminate A(end+1:n, end+1:n)
```