HW 2 update

- I have speed-up over the (tuned) serial code!
  - ... provided $n > 2000$
  - ... and I don’t think I’ve seen $2 \times$
- What I expect from you
  - *Timing experiments with basic code!*
  - A faster working version
  - Plausible parallel versions
  - ... but not necessarily great speed-up
Where we are

- Mostly done with parallel programming models
  - I’ll talk about GAS languages (UPC) later
  - Someone will talk about CUDA!
- Lightning overview of parallel simulation
  - Recurring theme: linear algebra!
  - Sparse matvec and company appear often
- Today: some dense linear algebra
Numerical linear algebra in a nutshell

- Basic problems
  - Linear systems: $Ax = b$
  - Least squares: minimize $\|Ax - b\|_2^2$
  - Eigenvalues: $Ax = \lambda x$
- Basic paradigm: matrix factorization
  - $A = LU, A = LL^T$
  - $A = QR$
  - $A = V\Lambda V^{-1}, A = QTQ^T$
  - $A = U\Sigma V^T$
- Factorization $\equiv$ switch to basis that makes problem easy
Numerical linear algebra in a nutshell

Two flavors: dense and sparse

- **Dense == common structures, no complicated indexing**
  - General dense (all entries nonzero)
  - Banded (zero below/above some diagonal)
  - Symmetric/Hermitian
  - Standard, robust algorithms (LAPACK)

- **Sparse == stuff 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 repn (e.g. via FFT)
  - Most algorithms are iterative; wider variety, more subtle
  - Build on dense ideas
History

BLAS 1 (1973–1977)
- Standard library of 15 ops (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
- BLAS 1 == $O(n^1)$ ops on $O(n^1)$ data
- Used in LINPACK (and EISPACK?)
History


- 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)

- BLAS2 == $O(n^2)$ ops on $O(n^2)$ data
History


- 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!
BLAS goes on

- http://www.netlib.org/blas
- CBLAS interface standardized
- Lots of implementations (MKL, VecLib, ATLAS, Goto, ...)
- Still new developments (XBLAS, tuning for GPUs, ...)
Why BLAS?

Consider Gaussian elimination.

LU for $2 \times 2$:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ c/a & 1 \end{bmatrix} \begin{bmatrix} a & b \\ 0 & d - bc/a \end{bmatrix}$$

Block elimination

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} I & 0 \\ CA^{-1} & I \end{bmatrix} \begin{bmatrix} A & B \\ 0 & D - CA^{-1}B \end{bmatrix}$$

Block LU

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{12} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{12}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{bmatrix}$$
Why BLAS?

Block LU

\[
\begin{bmatrix}
  A & B \\
  C & D
\end{bmatrix} =
\begin{bmatrix}
  L_{11} & 0 \\
  L_{12} & L_{22}
\end{bmatrix}
\begin{bmatrix}
  U_{11} & U_{12} \\
  0 & U_{22}
\end{bmatrix} =
\begin{bmatrix}
  L_{11}U_{11} & L_{11}U_{12} \\
  L_{12}U_{11} & L_{21}U_{12} + L_{22}U_{22}
\end{bmatrix}
\]

Think of \( A \) as \( k \times k \), \( k \) moderate:

\[
\begin{align*}
\begin{bmatrix} L_{11}, U_{11} \end{bmatrix} &= \text{small}_\text{lu}(A); \quad \% \text{ Small block LU} \\
U_{12} &= L_{11} \backslash B; \quad \%
\begin{align*}
\begin{bmatrix} \text{small}_\text{lu}(A) \end{bmatrix} &= \text{small}_\text{lu}(A); \quad \% \text{ Small block LU} \\
U_{12} &= L_{11} \backslash B; \quad \%
\end{align*}
\end{align*}
\]

\[
\begin{align*}
U_{12} &= L_{11} \backslash B; \quad \% \text{ Triangular solve} \\
L_{12} &= C / U_{11}; \quad \% \ " \\
S &= D - L_{21} \ast U_{12}; \quad \% \text{ Rank m update} \\
\begin{bmatrix} L_{22}, U_{22} \end{bmatrix} &= \text{lu}(S); \quad \% \text{ Finish factoring}
\end{align*}
\]

Three level-3 BLAS calls!

- Two triangular solves
- One rank-\( k \) update
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
ScaLAPACK (1995–present):
http://www.netlib.org/scalapack
- MPI implementations
- Only a small subset of LAPACK functionality
Why is ScaLAPACK not all of LAPACK?

Consider what LAPACK contains...
Decoding LAPACK names

- **F77** $\implies$ limited characters per name
- **General scheme:**
  - 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 routine types

- 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
  - ... and ongoing discussions! (e.g. about C interfaces)
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

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

... or deal with index space in other ways!
Parallel matvec: 1D row-blocked

Receive broadcast $x_0, x_1, x_2$ 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 col-blocked

Independently compute

\[ z^{(0)} = \begin{bmatrix} A_{00} \\ A_{10} \\ A_{20} \end{bmatrix} x_0 \quad z^{(1)} = \begin{bmatrix} A_{00} \\ A_{10} \\ A_{20} \end{bmatrix} x_1 \quad z^{(2)} = \begin{bmatrix} A_{00} \\ A_{10} \\ A_{20} \end{bmatrix} x_2 \]

and perform reduction: \( y = z^{(0)} + z^{(1)} + z^{(2)} \).
Parallel matvec: 2D blocked

- Involves broadcast \textit{and} reduction
- ... but with subsets of processors
Parallel matvec: 2D blocked

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

$$\begin{bmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1
\end{bmatrix}
= 
\begin{bmatrix}
Z^{(0)}_0 \\
Z^{(0)}_1
\end{bmatrix}
\begin{bmatrix}
A_{02} & A_{03} \\
A_{12} & A_{13}
\end{bmatrix}
\begin{bmatrix}
x_2 \\
x_3
\end{bmatrix}
= 
\begin{bmatrix}
Z^{(1)}_0 \\
Z^{(1)}_1
\end{bmatrix}$$

$$\begin{bmatrix}
A_{20} & A_{21} \\
A_{30} & A_{31}
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1
\end{bmatrix}
= 
\begin{bmatrix}
Z^{(3)}_2 \\
Z^{(3)}_3
\end{bmatrix}
\begin{bmatrix}
A_{20} & A_{21} \\
A_{30} & A_{31}
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1
\end{bmatrix}
= 
\begin{bmatrix}
Z^{(3)}_2 \\
Z^{(3)}_3
\end{bmatrix}$$

Reduce across rows:

$$\begin{bmatrix}
y_0 \\
y_1
\end{bmatrix}
= 
\begin{bmatrix}
Z^{(0)}_0 \\
Z^{(0)}_1
\end{bmatrix}
\begin{bmatrix}
Z^{(1)}_0 \\
Z^{(1)}_1
\end{bmatrix}
+ 
\begin{bmatrix}
Z^{(2)}_0 \\
Z^{(2)}_1
\end{bmatrix}
\begin{bmatrix}
Z^{(3)}_0 \\
Z^{(3)}_1
\end{bmatrix}
$$

$$\begin{bmatrix}
y_2 \\
y_3
\end{bmatrix}
= 
\begin{bmatrix}
Z^{(2)}_2 \\
Z^{(2)}_3
\end{bmatrix}
\begin{bmatrix}
Z^{(3)}_2 \\
Z^{(3)}_3
\end{bmatrix}$$
Parallel matmul

- Basic operation: $C = C + AB$
- Computation: $2n^3$ flops
- Goal: $2n^3/p$ flops per processor, minimal communication
Block MATLAB notation: $A(\cdot, j)$ means $j$th block column

Processor $j$ owns $A(\cdot, j)$, $B(\cdot, j)$, $C(\cdot, j)$

$C(\cdot, j)$ depends on all of $A$, but only $B(\cdot, j)$

How do we communicate pieces of $A$?
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)
1D layout on bus (no broadcast)

\[ C(:,\text{myproc}) += A(:,\text{myproc}) \times B(\text{myproc},\text{myproc}) \]
for \( i = 0:p-1 \)
  for \( j = 0:p-1 \)
    if (\( i == j \)) continue;
    if (\( \text{myproc} == i \))
      send \( A(:,i) \) to processor \( j \)
    if (\( \text{myproc} == j \))
      receive \( A(:,i) \) from \( i \)
    \( C(:,\text{myproc}) += A(:,i) \times B(i,\text{myproc}) \)
  end
end
end

Performance model?
No overlapping communications, so in a simple $\alpha - \beta$ model:

- $p(p - 1)$ messages
- Each message involves $n^2/p$ data
- Communication cost: $p(p - 1)\alpha + (p - 1)n^2\beta$
Every process $j$ can send data to $j + 1$ simultaneously.

Pass slices of $A$ around the ring until everyone sees the whole matrix ($p - 1$ phases).
1D layout on ring

tmp = A(myproc)
C(myproc) += tmp*B(myproc,myproc)
for j = 1 to p-1
    sendrecv tmp to myproc+1 mod p,
    from myproc-1 mod p
    C(myproc) += tmp*B(myproc-j mod p, myproc)

Performance model?
In a simple $\alpha - \beta$ model, at each processor:

- $p - 1$ message sends (and simultaneous receives)
- Each message involves $n^2/p$ data
- Communication cost: $(p - 1)\alpha + (1 - 1/p)n^2\beta$