Monday, 01/16/12:

Goals for today:

BLAS = basic Linear algebra subroutines

HW trapezoidal rule

HW power method

BLAS = Fortran functions and subroutines

= effort in 1970s to write high-quality, validated, free-of-charge code for basic linear algebra tasks, to be downloaded and incorporated into research and commercial code.

Written by the very best and most famous researchers.

BLAS 1 = level 1 BLAS = one for loop

Assume vectors and matrices have dimension n or $n \times n$, etc.

dot product of vectors

Euclidean norm

vector addition, scaling of a vector

saxpy = scalar alpha x plus y = $\alpha x + y$

$\Leftrightarrow$ $\alpha x_i + y_i$ for each i $\longrightarrow$ in CPU

2 operations (αx_i and $+ y_i$) in one clock cycle

certain matrix operations that touch only n elements, like

scaling of a row or column

BLAS 2 = level 2 BLAS = 2 (nested) for loops

matrix-vector product

scaling of matrix

outer product = $X * y'$ = col x times row y^T $\Leftrightarrow$ $A_{ij} = x_i y_j$

BLAS 3 = level 3 BLAS = 3 (nested) for loops

matrix-matrix multiplication

LU factorization → LINPACK, LAPACK

packages with higher-level lin. alg.:

LINPACK = lin. alg. like LU, back substitution
EISPACK = eigenvalue algorithms
} 1970s, 1980s

1990s these were updated and became LAPACK.

Key: LAPACK is programmed with variable block sizes, depending on size of cache on the actual computer used.

Really use BLAS by linking to a library like ACML

= AMD Core Math Library, sold along with the Portland Group compiler

libacml.so, header file acml.h → See HW and posted code

Documentation: – books from SIAM = Society for Industrial and Applied Math

– webpage www.netlib.org to see actual code and its comments.

At this page, browse for blas, lapack, etc.

We looked at ddot.f = double-precision dot product in BLAS

Notice that you can supply increments other than 1 in the call.

Most typical case would have increments of 1, so dot product of vectors x and y would be

d = ddot(n, x, 1, y, 1)

Main efficiency for CPUs with a cache lies in loop unrolling where we pre-load several consecutive components of x and y and act on them simultaneously:

roughly: $d = \sum_{i=1:5:n} (x_i y_i + x_{i+1} y_{i+1} + x_{i+2} y_{i+2} + x_{i+3} y_{i+3} + x_{i+4} y_{i+4})$

How to use increments other than 1 and HO on power:

$y = Ax$ ($\Leftrightarrow$ classical thinking $y_i = \sum_{j=1}^{n} A_{ij} x_j$

($\Leftrightarrow$ y_i = dot product of i-th of A with vector x

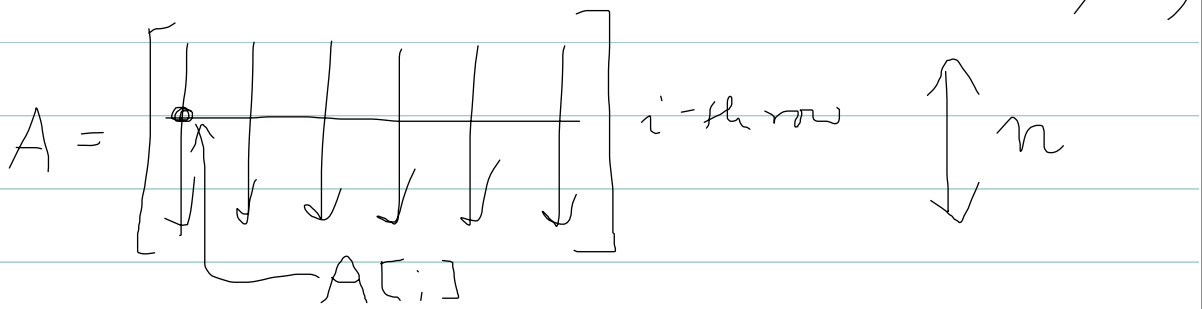
really bad: aux. vector for the ith row $ai = A(i,:)$

then $y_i = dot(ai, x) = ddot(n, ai, 1, x, 1)$

better = no aux. vector:

for $i = 1:n$

$y_i = \sum_{j=1}^{n} A_{ij} x_j = ddot(n, A[i], n, x, 1)$



this showed idea. Precise Code

$d = ddot(n, \&(A[i]), n, x, 1);$

If you follow idea above for power method / matrix-vector product in parallel, you will need one parallel_dot call for each row of A, that is, n MPI calls via MPI_Allreduce in parallel_dot that is worse than only 1 or 2 MPI calls ?

Find better solution by drawing data:

$$y = Ax = \left[\begin{array}{c|c|c} & & \\ l\_A & l\_A & l\_A \\ & & \end{array}\right] \left[\begin{array}{c} l\_x \\ \hline l\_x \\ \hline l\_x \end{array}\right]$$

$$= \underbrace{l\_A\, l\_x}_{\text{on P0}} + \underbrace{l\_A\, l\_x}_{\text{on P1}} + \underbrace{l\_A\, l\_x}_{\text{on P2}}$$

There are matrix-vector products on each process $\Rightarrow$ we better code for them !

This product:

bad: for i = 1 : n

for j = 1 : n

since this accesses A(i,j) out of order and you get cache misses

good: for j = 1 : n

for i = 1 : n

So that A's elements are accessed consecutively.

Even better as in BLAS:

idea of block matrices

$$y = Ax = \begin{bmatrix} \boxed{} & \boxed{} \\ \boxed{} & \boxed{} \end{bmatrix} \begin{bmatrix} \\ \end{bmatrix}$$

(out of time: HW on trapezoidal rule.)