**Logistics**

- C4 issues fixed? Changed some interfaces:
  - **Old:** `mpisub -n 4` = four slots
  - **New:** `mpisub -n 4` = one slot with four processors
  - **New:** `mpisub -n 2x2` = two slots with two procs each
  - `ompsub` unaffected
  - See Piazza for the gruesome details

- You should be working on HW 3 (SPH)
  - Several tools installed – use them!
  - Bottlenecks mostly related to neighbor finding

- You should also be thinking of final projects
  - Talk to each other, use Piazza, etc
Using Amplifier XE

# In a shell script you run with ompsub
amplxe-cl -collect hotspots \
   -result-dir r001hs -- ./sph.x -F 10

# See hotspots
amplxe-cl -report hotspots -r r001hs

# Dig down into one function
amplxe-cl -report hotspots -source-object \
   function=compute_density -r r001hs
Where we are

- This week: *dense* linear algebra
- Next week: *sparse* 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 $\propto O(n^1)$ 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](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:

\[
\begin{align*}
[L_{11}, U_{11}] &= \text{small\_lu}(A); & \text{Small block LU} \\
U_{12} &= L_{11}ackslash B; & \text{Triangular solve} \\
L_{12} &= C/U_{11}; & \text{"} \\
S &= D - L_{21} \ast U_{12}; & \text{Rank m update} \\
[L_{22}, U_{22}] &= \text{lu}(S); & \text{Finish factoring}
\end{align*}
\]

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
PLASMA and MAGMA (2008–present):

- Parallel LA Software for Multicore Architectures
  - Target: Shared memory multiprocessors
  - Stacks on LAPACK/BLAS interfaces
  - Tile algorithms, tile data layout, dynamic scheduling
  - Other algorithmic ideas, too (randomization, etc)

- Matrix Algebra for GPU and Multicore Architectures
  - Target: CUDA, OpenCL, Xeon Phi
  - Still stacks (e.g. on CUDA BLAS)
  - Again: tile algorithms + data, dynamic scheduling
  - Mixed precision algorithms (+ iterative refinement)

- Dist memory: PaRSEC / DPLASMA
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

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

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

\[
\begin{bmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1
\end{bmatrix} =
\begin{bmatrix}
Z_0^{(0)} \\
Z_1^{(0)}
\end{bmatrix}
\begin{bmatrix}
A_{02} & A_{03} \\
A_{12} & A_{13}
\end{bmatrix}
\begin{bmatrix}
x_2 \\
x_3
\end{bmatrix} =
\begin{bmatrix}
Z_0^{(1)} \\
Z_1^{(1)}
\end{bmatrix}
\begin{bmatrix}
A_{20} & A_{21} \\
A_{30} & A_{31}
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1
\end{bmatrix} =
\begin{bmatrix}
Z_2^{(3)} \\
Z_3^{(3)}
\end{bmatrix}
\begin{bmatrix}
A_{20} & A_{21} \\
A_{30} & A_{31}
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1
\end{bmatrix} =
\begin{bmatrix}
Z_2^{(3)} \\
Z_3^{(3)}
\end{bmatrix}
\]

Reduce across rows:

\[
\begin{bmatrix}
y_0 \\
y_1
\end{bmatrix} =
\begin{bmatrix}
Z_0^{(0)} \\
Z_1^{(0)}
\end{bmatrix} +
\begin{bmatrix}
Z_0^{(1)} \\
Z_1^{(1)}
\end{bmatrix}
\begin{bmatrix}
y_2 \\
y_3
\end{bmatrix} =
\begin{bmatrix}
Z_2^{(2)} \\
Z_3^{(2)}
\end{bmatrix} +
\begin{bmatrix}
Z_2^{(3)} \\
Z_3^{(3)}
\end{bmatrix}
\]
Parallel matmul

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

Self A(:,1) A(:,2) A(:,0)

C + = A × B
1D layout on bus (no broadcast)

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

Performance model?
1D layout on bus (no broadcast)

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).
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: $(p - 1)\alpha + (1 - 1/p)n^2\beta$
Outer product algorithm

Serial: Recall outer product organization:

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

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

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

- Processor for each $(i,j) \implies$ 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 $(i,j)$, $C(i,j) += A(i,k) \times B(k,j)$;
end

If we have tree along each row/column, then

- $\log(s)$ messages per broadcast
- $\alpha + \beta n^2/s^2$ per message
- $2 \log(s)(\alpha s + \beta n^2/s)$ total communication
- Compare to 1D ring: $(p-1)\alpha + (1 - 1/p)n^2\beta$

Note: Same ideas work with block size $b < n/s$
Cannon’s algorithm

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

Idea: Reindex products in block matrix multiply

\[C(i, j) = \sum_{k=0}^{p-1} A(i, k)B(k, j)\]

\[= \sum_{k=0}^{p-1} A(i, k + i + j \mod p)B(k + i + j \mod p, j)\]

For a fixed \(k\), a given block of \(A\) (or \(B\)) is needed for contribution to exactly one \(C(i, j)\).
Cannon’s algorithm

% Move A(i,j) to A(i,i+j)
for i = 0 to s-1
    cycle A(i,:) left by i

% Move B(i,j) to B(i+j,j)
for j = 0 to s-1
    cycle B(:,j) up by j

for k = 0 to s-1
    in parallel;
    C(i,j) = C(i,j) + A(i,j)*B(i,j);
    cycle A(:,i) left by 1
    cycle B(:,j) up by 1
Cost of Cannon

- Assume 2D torus topology
- Initial cyclic shifts: $\leq s$ messages each ($\leq 2s$ total)
- For each phase: 2 messages each (2s total)
- Each message is size $n^2/s^2$
- Communication cost: $4s(\alpha + \beta n^2/s^2) = 4(\alpha s + \beta n^2/s)$
- This communication cost is optimal!
  ... but SUMMA is simpler, more flexible, almost as good
Speedup and efficiency

Recall

\[
\text{Speedup} := \frac{t_{\text{serial}}}{t_{\text{parallel}}}
\]
\[
\text{Efficiency} := \frac{\text{Speedup}}{p}
\]

Assuming no overlap of communication and computation, efficiencies are

- 1D layout: \( (1 + O\left(\frac{p}{n}\right))^{-1} \)
- SUMMA: \( (1 + O\left(\frac{\sqrt{p \log p}}{n}\right))^{-1} \)
- Cannon: \( (1 + O\left(\frac{\sqrt{p}}{n}\right))^{-1} \)