Dense Linear Algebra

David Bindel

20 Oct 2015

Logistics

- Totient issues fixed? May still be some issues:
  - Login issues – working on it
  - Intermittent node non-responsiveness – working on it
- You should have finished mid-project report for water
  - Two pieces to performance: single-core and parallel
  - Single-core issues mostly related to vectorization
  - Parallelism and cache locality from tiling
  - Scaling studies, performance models are also good!
- Next assignment (All-Pairs Shortest Path) is up
  - Official release is Oct 22
- You should also be thinking of final projects
  - Talk to each other, use Piazza, etc
  - Next week is good for this

Where we are

- This week: *dense* linear algebra
    - Today: Matrix multiply as building block
    - Next time: Building parallel matrix multiply
- Next week: Bindel traveling
- Week after: *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 == $O(n^1)$ ops on $O(n^1)$ data
    - Used in LINPACK (and EISPACK?)

History

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

History

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!

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 × 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:

```
[L11,U11] = small_lu(A);    % Small block LU
U12 = L11\B;                % Triangular solve
L12 = C/U11;                % "
S   = D-L21*U12;            % Rank m update
[L22,U22] = lu(S);          % Finish factoring
```

Three level-3 BLAS calls!

- Two triangular solves
- One rank-k update

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

ScaLAPACK

ScaLAPACK (1995–present):
`http://www.netlib.org/scalapack`

- MPI implementations
- Only a small subset of LAPACK functionality

PLASMA and MAGMA

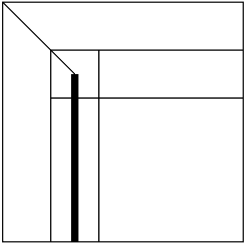
PLASMA and MAGMA (2008–present):

- 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

Reminder: Evolution of LU

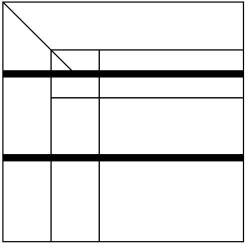
On board...

Blocked GEPP



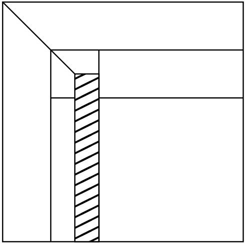
Find pivot

Blocked GEPP



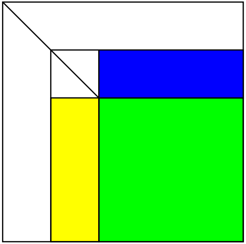
Swap pivot row

Blocked GEPP



Update within block

Blocked GEPP



Delayed update (at end of block)

Big idea

- *Delayed update* strategy lets us do LU fast
    - Could have also delayed application of pivots
- Same idea with other one-sided factorizations (QR)
- Can get decent multi-core speedup with parallel BLAS! ... assuming n sufficiently large.

There are still some issues left over (block size? pivoting?)...

Explicit parallelization of GE

What to do:

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

Possible matrix layouts

1D column blocked: bad load balance

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

Possible matrix layouts

1D column cyclic: hard to use BLAS2/3

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

Possible matrix layouts

1D column block cyclic: block column factorization a bottleneck

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

Possible matrix layouts

Block skewed: indexing gets messy

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

Possible matrix layouts

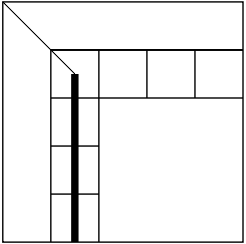
2D block cyclic:

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

Possible matrix layouts

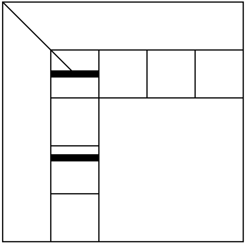
- 1D column blocked: bad load balance
- 1D column cyclic: hard to use BLAS2/3
- 1D column block cyclic: factoring column is a bottleneck
- Block skewed (a la Cannon – Thurs): just complicated
- 2D row/column block: bad load balance
- 2D row/column block cyclic: win!

Distributed GEPP



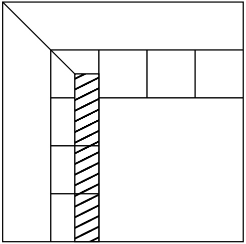
Find pivot (column broadcast)

Distributed GEPP



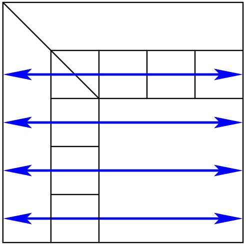
Swap pivot row within block column + broadcast pivot

Distributed GEPP



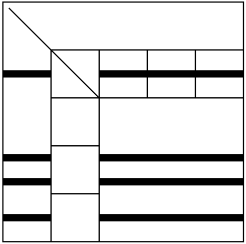
Update within block column

Distributed GEPP



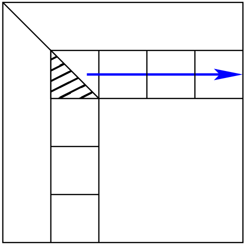
At end of block, broadcast swap info along rows

Distributed GEPP



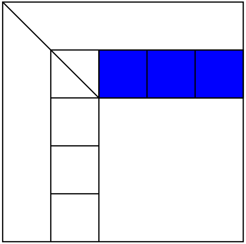
Apply all row swaps to other columns

Distributed GEPP



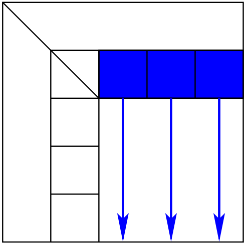
Broadcast block L_{II} right

Distributed GEPP



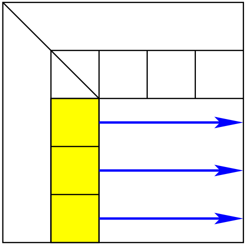
Update remainder of block row

Distributed GEPP



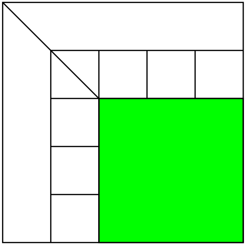
Broadcast rest of block row down

Distributed GEPP



Broadcast rest of block col right

Distributed GEPP



Update of trailing submatrix

Cost of ScaLAPACK GEPP

Communication costs:

- Lower bound: $O(n^2/\sqrt{P})$ words, $O(\sqrt{P})$ messages
- ScaLAPACK:
  - $O(n^2 \log P/\sqrt{P})$ words sent
  - $O(n \log p)$ messages
  - Problem: reduction to find pivot in each column
- Recent research on stable variants without partial pivoting

What if you don’t care about dense Gaussian elimination?
Let’s review some ideas in a different setting...

Floyd-Warshall

Goal: Find shortest path lengths between all node pairs.

Idea: Dynamic programming! Define

$$d_{ij}^{(k)} = \text{shortest path } i \text{ to } j \text{ with intermediates in } \{1, \ldots, k\}.$$

Then

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

and $d_{ij}^{(n)}$ is the desired shortest path length.

The same and different

Floyd's algorithm for all-pairs shortest paths:

```
for k=1:n
  for i = 1:n
    for j = 1:n
      D(i,j) = min(D(i,j), D(i,k)+D(k,j));
```

Unpivoted Gaussian elimination (overwriting *A*):

```
for k=1:n
  for i = k+1:n
    A(i,k) = A(i,k) / A(k,k);
    for j = k+1:n
      A(i,j) = A(i,j)-A(i,k)*A(k,j);
```

The same and different

- The same: $O(n^3)$ time, $O(n^2)$ space
- The same: can't move k loop (data dependencies)
  - ... at least, can't without care!
  - Different from matrix multiplication
- The same: $x_{ij}^{(k)} = f\left(x_{ij}^{(k-1)}, g\left(x_{ik}^{(k-1)}, x_{kj}^{(k-1)}\right)\right)$
  - Same basic dependency pattern in updates!
  - Similar algebraic relations satisfied
- Different: Update to full matrix vs trailing submatrix

How far can we get?

How would we

- Write a cache-efficient (blocked) *serial* implementation?
- Write a message-passing *parallel* implementation?

The full picture could make a fun class project...

Onward!

Next up: Sparse linear algebra and iterative solvers!