

CS 5220

GEMV, GEMM, and LU

David Bindel

2024-10-24

Matrix multiply

Simple $y = Ax$ involves two indices:

$$y_i = \sum_j A_{ij}x_j$$

Sums can go in any order!

Matrix vector product

Organize $y = Ax$ around rows or columns:

```
% 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



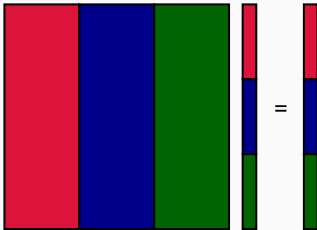
Broadcast x , compute rows independently.

Parallel matvec: 1D row-blocked



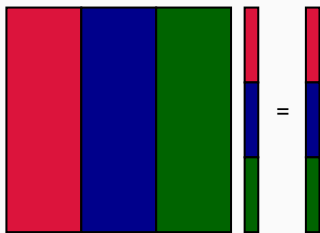
```
Allgather(xlocal, xall)  
ylocal = Alocal * xall
```

Parallel matvec: 1D col-blocked



Compute partial matvecs and reduce.

Parallel matvec: 1D col-blocked



```
z = Alocal * xlocal
```

```
for j = 1:p
```

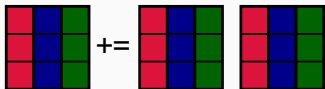
```
    Reduce(sum, z[i], ylocal at proc i)
```

```
end
```



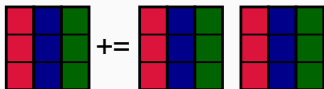

- Involves broadcast *and* reduction
- ... but with subsets of processors

- Basic operation: $C = C + AB$
- Computation: $2n^3$ flops
- Goal: $2n^3/p$ flops per processor, minimal communication
- Two main contenders: SUMMA and Cannon



- 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 ?

1D layout on ring



- 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: $(p - 1)\alpha + (1 - 1/p)n^2\beta$

Recall outer product organization:

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

Outer product algorithm

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 $(i, j) \implies$ 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)*B(k,j);
end
```

Parallel outer product (SUMMA)

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 $b < n/s$

SUMMA + “pass it around?”

Idea: Reindex products in block matrix multiply

$$\begin{aligned} C(i, j) &= \sum_{k=0}^{p-1} A(i, k) B(k, j) \\ &= \sum_{k=0}^{p-1} A(i, k + i + j \pmod{p}) B(k + i + j \pmod{p}, j) \end{aligned}$$

For a fixed k , a given block of A (or B) is needed for contribution to *exactly one* $C(i, j)$.

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

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
```

- 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

Recall

$$\text{Speedup} := t_{\text{serial}}/t_{\text{parallel}}$$

$$\text{Efficiency} := \text{Speedup}/p$$

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

Assuming no overlap of communication and computation.

LU



On board... or not.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 13 \\ 22 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ -7 & 0 & 1 \end{bmatrix} \left(\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 13 \\ 22 \end{bmatrix} \right)$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & -6 & -11 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ -3 \\ -6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix} \left(\begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & -6 & -11 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ -3 \\ -6 \end{bmatrix} \right)$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ -3 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix}$$

Overwrite A with L and U

```
for j = 1:n-1
    for i = j+1:n
        A(i,j) = A(i,j) / A(j,j);    % Compute multiplier
        for k = j+1:n
            A(i,k) -= A(i,j) * A(j,k); % Update row
        end
    end
end
end
```

Overwrite A with L and U

```
for j = 1:n-1
    A(j+1:n,j) = A(j+1:n,j)/A(j,j);           % Compute multipliers
    A(j+1:n,j+1:n) -= A(j+1:n,j) * A(j, j+1:n); % Trailing update
end
```

Stability is a problem! Compute $PA = LU$

```
p = 1:n;
for j = 1:n-1
    [~,jpiv] = max(abs(A(j+1:n,j)));           % Find pivot
    A([j, j+jpiv-1],:) = A([j+jpiv-1, j]);   % Swap pivot row
    p([j, j+jpiv-1],:) = p([j+jpiv-1, j]);   % Save pivot info
    A(j+1:n,j) = A(j+1:n,j)/A(j,j);           % Compute multipliers
    A(j+1:n,j+1:n) -= A(j+1:n,j) * A(j, j+1:n); % Trailing update
end
```

Think in a way that uses level 3 BLAS

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix}$$

Think in a way that uses level 3 BLAS

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{bmatrix}$$

Think in a way that uses level 3 BLAS

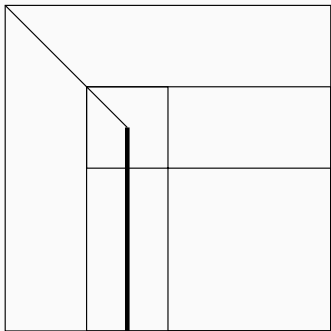
$$L_{11}U_{11} = A_{11}$$

$$U_{12} = L_{11}^{-1}A_{12}$$

$$L_{21} = A_{21}U_{11}^{-1}$$

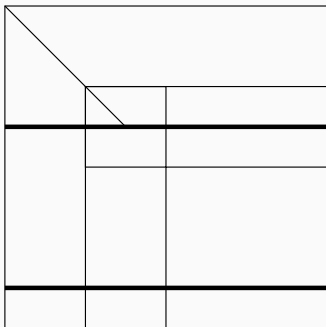
$$L_{22}U_{22} = A_{22} - L_{21}U_{12}$$

- Still haven't showed how to do pivoting!
- Easier to draw diagrams from here
- Take 6210 or 4220 if you want more on LU!

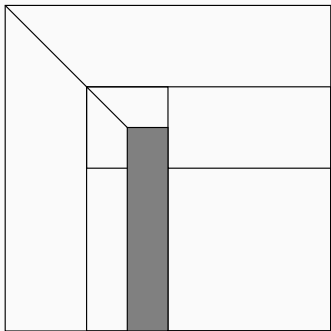


Find pivot

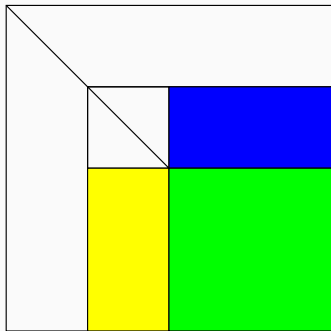
Blocked GEPP



Swap pivot row



Update within block column



Delayed update (at end of block)

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

Issues left over (block size?)...

What to do:

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

How should we share the matrix across ranks?

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

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

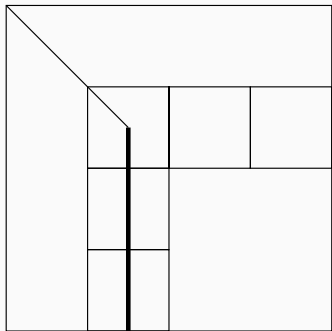
$$\begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 0 & 0 & 1 & 1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

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

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

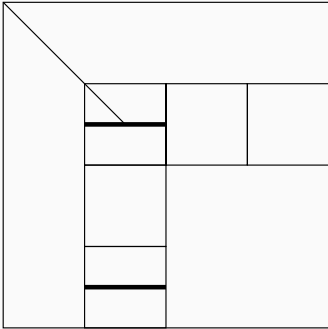
Possible matrix layouts

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

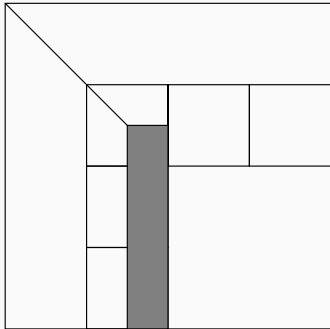


Find pivot (column broadcast)

Distributed GEPP

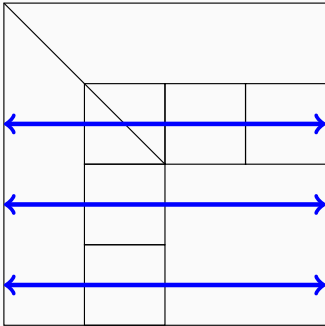


Swap pivot row within block column + broadcast pivot



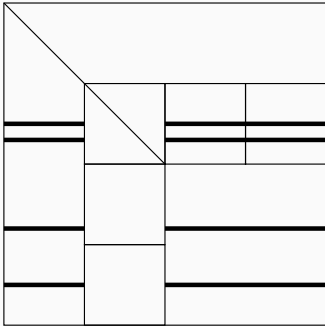
Update within block column

Distributed GEPP

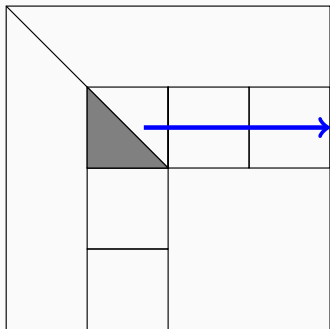


At end of block, broadcast swap info along rows

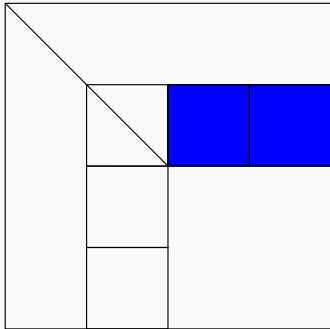
Distributed GEPP



Apply all row swaps to other columns

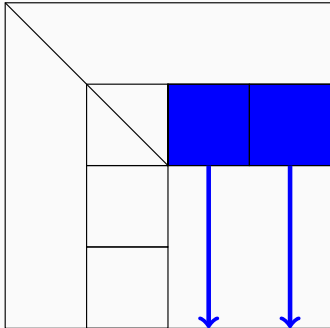


Broadcast block L_{II} right



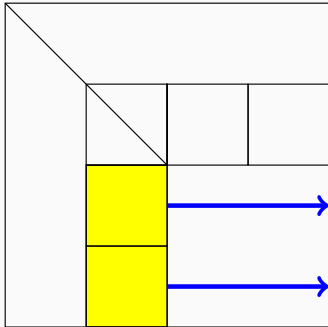
Update remainder of block row

Distributed GEPP

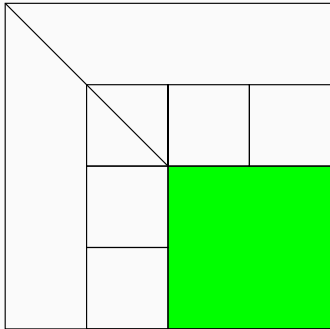


Broadcast rest of block row down

Distributed GEPP



Broadcast rest of block col right



Update of trailing submatrix

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
- Tournaments for stability without partial pivoting

If you don't care about dense LU?

Let's review some ideas in a different setting..

Goal: All pairs shortest path lengths.

Idea: Dynamic programming! Define

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

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

- Cache-efficient (blocked) *serial* implementation?
- Message-passing *parallel* implementation?

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

Next up: Sparse linear algebra and iterative solvers!