

CS575: LU Decomposition

Wim Bohm
Colorado State University

Except as otherwise noted, the content of this presentation is licensed under the Creative Commons Attribution 2.5 license.

Gaussian elimination vs LUD

- In Gaussian elimination we transform $Ax=b$ into $Ux=y$
i.e., we transform **both** A & b .
- So if we get a new system $Ax=c$, we need to do the whole transformation again.
- LU decomposition treats A independently;
 $A = LU$
L **unit** lower triangular matrix
U upper triangular matrix

Forward / backward substitution

- $Ax=b \rightarrow LUx=b$
- Define $Ux=y$, first solve $Ly=b$ then $Ux=y$
- Solving $Ly=b$ forward substitution

$$\begin{array}{l} 1 \ 0 \ 0 \ y_1 = b_1 \\ 2 \ 1 \ 0 \ y_2 = b_2 \quad 1 \ 0 \ y_2 = b_2 - 2b_1 \\ 3 \ 4 \ 1 \ y_3 = b_3 \quad 4 \ 1 \ y_3 = b_3 - 3b_1 \end{array}$$

Now fwd substitute y_2
- Solving $Ux=y$ backward substitution
 - same mechanism, studied for Gaussian elimination
- Both forward and backward easily pipelined

LU decomposition

- Recursive approach
 - $n=1$: done! $L=I_1, U=A$
 - $n>1$: $A = \begin{array}{c|ccc} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{array} = \begin{bmatrix} a_{11} & w^T \\ v & A' \end{bmatrix}$
v and w: $(n-1)$ -sized vectors
 - $A = \begin{bmatrix} a_{11} & w^T \\ v & A' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{bmatrix}$
- You can check this by matrix multiplication
Do it by hand for $n=2$ and $n=3$
We have created the first step of the LU decomposition.

n=2

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ a_{21}/a_{11} & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ 0 & p \end{bmatrix}$$

p = ?

n=2

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ a_{21}/a_{11} & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ 0 & p \end{bmatrix}$$

p = ?

$$a_{22} = (a_{21} * a_{12})/a_{11} + p$$

n=2

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ a_{21}/a_{11} & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ 0 & p \end{bmatrix}$$

p = ?

$$a_{22} = (a_{21} * a_{12})/a_{11} + p$$

$$p = a_{22} - (a_{21} * a_{12})/a_{11}$$

Do it yourself for n=3

- (You get $A^{-1} \cdot v w^T / a_{11} = \begin{matrix} p & q \\ r & s \end{matrix}$)
- Find expressions for p, q, r, and s

Recursive Leap

- Solve the rest recursively, ie $A' - vw^T / a_{11} = L' U'$
Element wise for this means: $A_{ij} := A_{ik} * A_{kj}$

$$\text{then } A = \begin{bmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & L' U' \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 \\ v/a_{11} & L' \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & U' \end{bmatrix}$$

- In place: L and U stored and computed in A
 - 0-s and 1-s of L and U are implicit (not stored)

The code

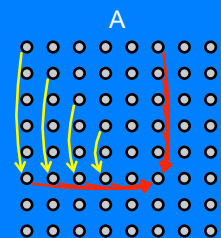
```
for k = 1 to n {
  for i = k+1 to n
    Aik /= Akk
  for i = k+1 to n
    for j = k+1 to n
      Aij := Aik * Akj
}
```

Pivoting

- Why?
 - A_{kk} can be close to 0 causing big error in $1/A_{kk}$, so we should find the absolute largest A_{ik} and swap rows i and k
- How?
 - We can register this in a permutation matrix or better an n sized array $\pi[i]$: position of row i.
- No pivoting:
 - LU code is a straightforward translation of our recurrence.

LUD: data dependence

```
for k = 1 to n {
  for i = k+1 to n
    Aik /= Akk
  for i = k+1 to n
    for j = k+1 to n
      Aij := Aik * Akj
}
```



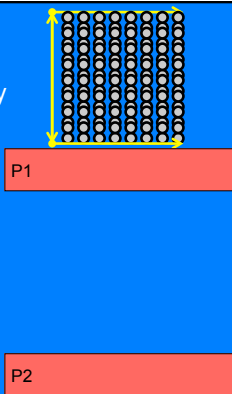
Pipelining

- Iteration $k+1$ can start after iteration k has finished row $k+1$
 - neither row k nor column k are read or written anymore
- This naturally leads to a streaming algorithm
 - every iteration becomes a process
 - results from iteration/process k flow to process $k+1$
 - process k uses row k to update sub matrix $A[k+1:n, k+1:n]$
- Actually, only a fixed number P of processes run in parallel
 - after that, data is re-circulated

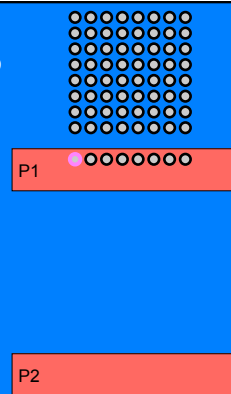
Pipelined LU

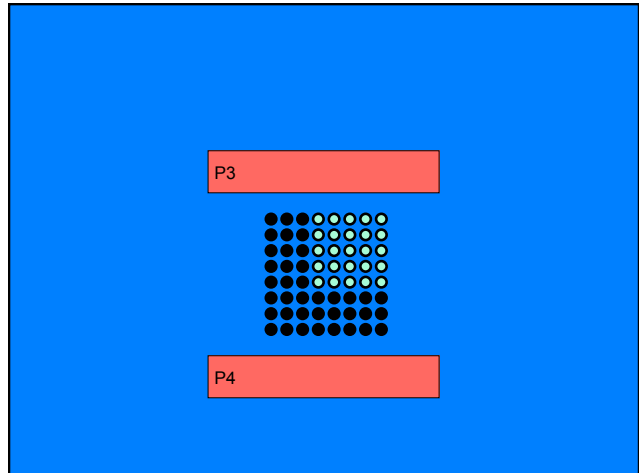
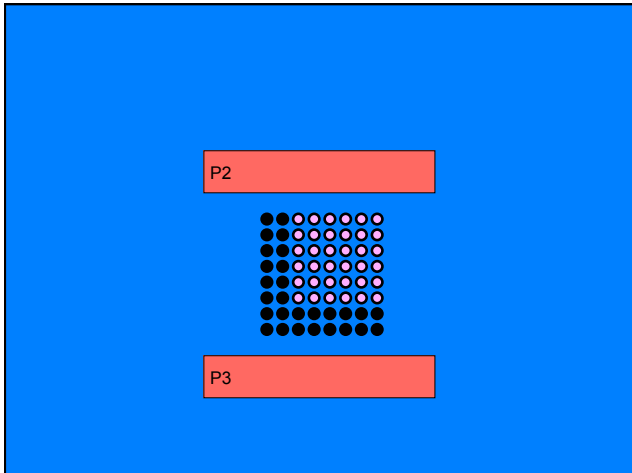
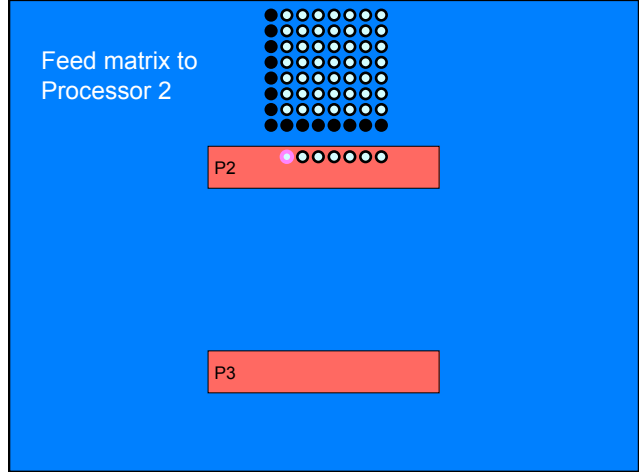
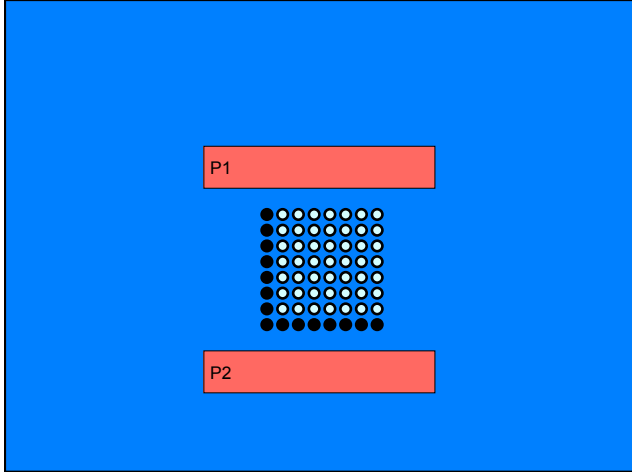
- Pipeline of P processes
 - parallel sections in OpenMP lingo
 - algorithm runs in $\text{ceil}(N/P)$ sweeps
- In one sweep P outer k iterations run in parallel
- After each sweep P rows and columns are done
 - Problem size n is reduced to $(n-P)^2$
 - Rest of the (updated) array is re-circulated
- Each process owns a row
 - Process p owns row $p + (i-1)P$ in sweep $i, i=1$ to $\text{ceil}(N/P)$
 - Array elements are updated while streaming through the pipe of processes
- Total work $O(n^3)$, speedup $\sim P$
- Forward / backward substitution are simple loops

First, Flip the matrix vertically

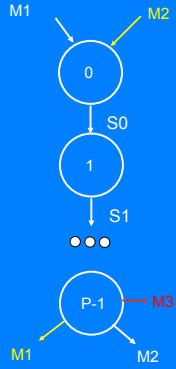


Feed matrix to Processor 1





Process Pipeline



Memories:

M1 -> M2 on even sweeps
M2 -> M1 on odd sweeps
M3 : siphoning off finished results

Processes are grouped in OpenMP style parallel sections with intermediate data in streams

Process 0 reads either from M1 or M2
writes to stream S0

Process 1 reads from stream Si-1
writes to Si

Process P-1
reads from stream Sp-1
writes finished data to M3
writes rest to M1 or M2

Inside a process



```

for(i = (s-1)*P; i < n; i++) {
  for(j = (s-1)*P; j < n; j++) {
    //ping-pong: read from M1 or M2
    if(k&0x1) w= M1 [i * n + j];
    else w= M2 [j * n + j];
    // if (i < me) leave data unchanged
    if (i ==me) { // store my row
      if (j == i) piv = w;
      myRow[j] = w;
    }
    else
    if (i > me) { // update this row with my row
      // if (j < me) leave data unchanged
      if (j == me) { w /= piv, mul = w;
        else if (j > me) w -= mul*myRow[j];
      }
    }
    put_stream( S0, w);
  }
}

```