Parallel algorithms for Dense Matrix Problems

Chris Kauffman

Last Updated:
Thu Feb 16 01:40:55 PM CST 2023

Logistics

Assignments

- A1 grades *almost done*
- Mini-Exam 1 grading have commenced
- Likely to have all grading done by next Mon
- A2 is delayed: apologies

Reading

Grama Ch 8 on Dense Matrix Algorithms

- Naive Matrix Multiply
- Cannon's Algorithm
- LU Decomposition

Today

- Dense vs Sparse Matrices
- Matrix algorithms

Recall Matrix Transpose

- Common operation on matrices is a **transpose** notated A^T
- Interchanges rows/columns of A: $a_{ij} \to a_{ji}$
- Diagonal elements stay the same
- Algorithms that perform operations on A can often be performed on A^T without re-arranging A - how?
  Hint: consider summing rows of A vs summing rows of A^T

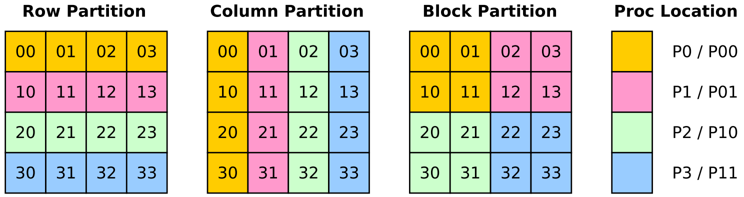
Original matrix A

0	5	10	15
20	25	30	35
40	45	50	55
60	65	70	75

transpose(A)

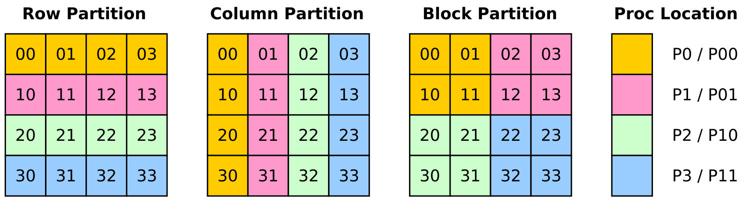
0	20	40	60
5	25	45	65
10	30	50	70
15	35	55	75

Exercise: Matrix Partitioning Across Processors



- Recall several ways to partition matrices across processors
- Diagram shows these
  - Entry `ij` may be an individual element OR...
  - Entry `ij` may be a **Block**: ex. Block (2,3) is the 100x100 submatrix rows 200-299 and cols 300-399
- Assume **square** matrices : #rows = #cols
- For Mat-Mult $C = A \times B$, what is...
  - Ideal partitioning for A and B in matrix multiply?
  - Ideal partitioning for $C = A^T \times B$
  - Ideal partitioning for $C = A \times B^T$

Answers: Matrix Partitioning Across Processors



- $C = A \times B$
  - Ideally A is row-partitioned, B is column partitioned
  - Then block-partitioned C could be computed w/o communication
  - e.g. Proc 0 owns A[0,:] and B[:,0] so can compute C[0,0]
- $C = A^T \times B$
  - Ideally A and B column-partitioned
- $C = A \times B^T$
  - Ideally A and B row-partitioned
- Block-partitioning often used: not ideal for any version but less communication required when both A and A^T will b used

Naive Parallel Dense Multiplication: Overview

Block Partitioning Appears Frequently

- Specific applications may be able to select a favorable partitioning (e.g. Row Partition for repeated mat-vec mult)
- Many applications use both A and A^T so employ block-partitioned matrices: middle-way approach which does not favor rows or columns
- Parallel Libraries often use block partitions by default

Matrix Multiply with Blocks

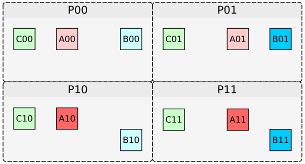
- To compute Matrix-Matrix multiply, procs must (eventually) multiply full rows by full columns to compute an output block
- Naive method: each Proc stores full rows/columns needed for it to independently compute output block which it stores

Naive Parallel Dense Multiplication: Demo

- Distributed Parallel Matrix-Matrix Multiply
- Block Partition of Matrices A, B, C among processors
- Diagram shows 4 processors in a 2×2 grid

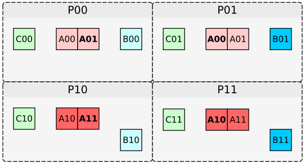
1. Initial data layout: each Proc holds a block of A, B, and C respectively. Processors are arragned in a logical grid that reflects their initial data.

In subsequent steps, received / computed data is bolded.



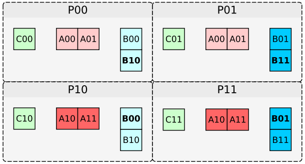
2. Each proc participates in an All-to-All sharing of data for the Row it is in.

This leaves each proc with entire rows of A.

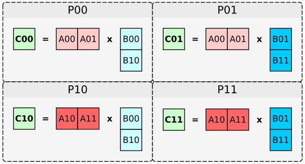


3. Each proc participates in an All-to-All sharing of data for the Column it is in.

This leaves each row with complete columns as well.



4. Each Proc now has a unique set of complete Rows and Columns and can independently compute a block of output matrix C through block multiplication.


P00
C00 = A00 A01 x B00 B10
P01
C01 = A00 A01 x B01 B11
P10
C10 = A10 A11 x B00 B10
P11
C11 = A10 A11 x B01 B11

Exercise: Analysis of Naive Dense Mult.

Assumptions

- Matrices A and B are size $N \times N$ so N^2 elements
- P processors in a $\sqrt{P} \times \sqrt{P}$ grid (P is a perfect square)
- Each Proc has block with N^2/P elements of A, B as a $(N/\sqrt{P}) \times (N/\sqrt{P})$ submatrix
- Simplified communication cost for All-to-All on a Ring with p #procs in ring, t_s comm startup time, t_w per word transfer rate, M message size:

$$t_{comm} = (p-1)(t_s + t_w M)$$

Questions

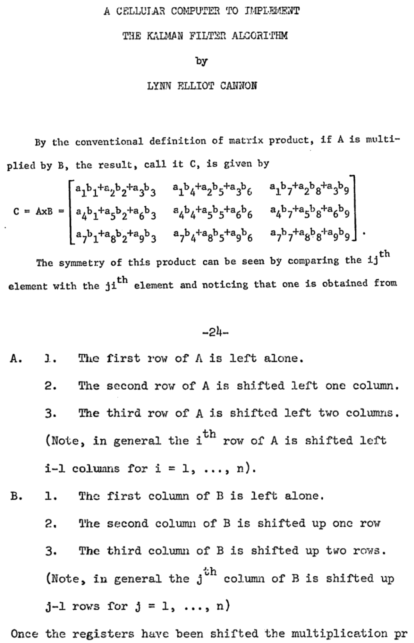
1. What is communication cost of this algorithm?
2. How much time does the final block matrix multiply take?
3. What is the memory requirement for each proc?
4. Downsides of this algorithm?

Answers: Analysis of Naive Dense Mult.

1. What is communication cost of this algorithm?
   - #Procs in rows/cols is $\sqrt{P} \sim$ ring size
   - $M = N^2/P$: message size is num elements on each proc
   - 2 All-to-All shares : 1 for rows, 1 for cols
   $$t_{comm} = 2(\sqrt{P} - 1) \times (t_s + t_w(N^2/P))$$
2. What is the memory requirement for each proc? E.g. how many submatrices of A,B are on each proc?
   - Full rows/cols on each proc
   - Requires $2\sqrt{P}$ submatrices for each Proc
3. How much time does the final block matrix multiply take?
   - Each proc has $\sqrt{P}$ submats of A,B to multiply
   - MatMult is for size s is $O(s^3)$; submat size $s = N/\sqrt{P}$
   $$t_{mult} = O((\sqrt{P}) \times ((N/\sqrt{P})^3)) = O(N^3/P)$$
4. Downsides of this algorithm?
   - Major: The need to store $\sqrt{P}$ sub matrices on all procs may be prohibitive: $2\sqrt{P} \times N^2/P$ space on each proc
   - Minor: Not much chance to overlap communication / computation in the algorithm

Cannon's Algorithm

- Proposed in Lynn Elliot Cannon's 1969 thesis
- Target was very small parallel machines implementing a Kalman Filter algorithm in hardware
- "Communication" happening between small Procs with data in registers
- Scales nicely to large distributed machines and overcomes the large memory requirement of the Naive Mat-Mult Algorithm



A CELLULAR COMPUTER TO IMPLEMENT

THE KALMAN FILTER ALGORITHM

by

LYNN ELLIOT CANNON

By the conventional definition of matrix product, if A is multiplied by B, the result, call it C, is given by

$$C = AxB = \begin{bmatrix} a_1b_1+a_2b_2+a_3b_3 & a_1b_4+a_2b_5+a_3b_6 & a_1b_7+a_2b_8+a_3b_9 \\ a_4b_1+a_5b_2+a_6b_3 & a_4b_4+a_5b_5+a_6b_6 & a_4b_7+a_5b_8+a_6b_9 \\ a_7b_1+a_8b_2+a_9b_3 & a_7b_4+a_8b_5+a_9b_6 & a_7b_7+a_8b_8+a_9b_9 \end{bmatrix}.$$

The symmetry of this product can be seen by comparing the ijth element with the jith element and noticing that one is obtained from

-24-

A. 1. The first row of A is left alone.

2. The second row of A is shifted left one column.

3. The third row of A is shifted left two columns.

(Note, in general the ith row of A is shifted left i-1 columns for i = 1, ..., n).

B. 1. The first column of B is left alone.

2. The second column of B is shifted up one row

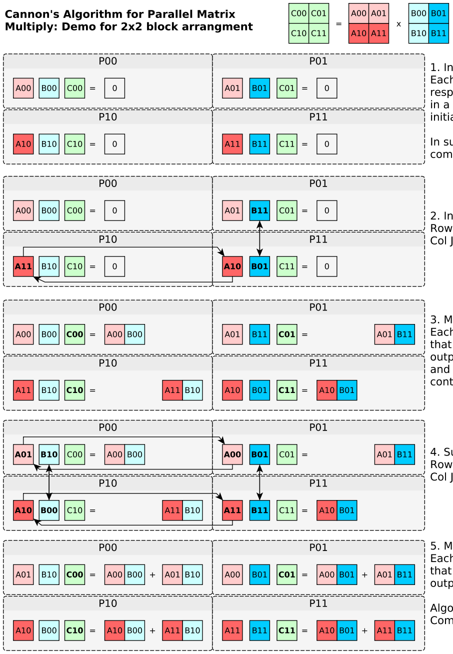
3. The third column of B is shifted up two rows.

(Note, in general the jth column of B is shifted up j-1 rows for j = 1, ..., n)

Once the registers have been shifted the multiplication pr

Demo

Cannon's Algorithm for Parallel Matrix Multiply: Demo for 2x2 block arrangment



1. Initial data layout.
Each Proc holds a block of A, B, and C respectively. Processors are arragned in a logical grid that reflects their initial data.

In subsequent steps, received / computed data is bolded.

2. Initial communication.
Row I: Rotate A blocks left by I procs
Col J: Rotate B blocks down by J procs

3. Multiply Step
Each Proc has a submatrix of A and B that allows partial computation of its output block of C. Stored blocks of A and B are multiplied and added onto contents of C.

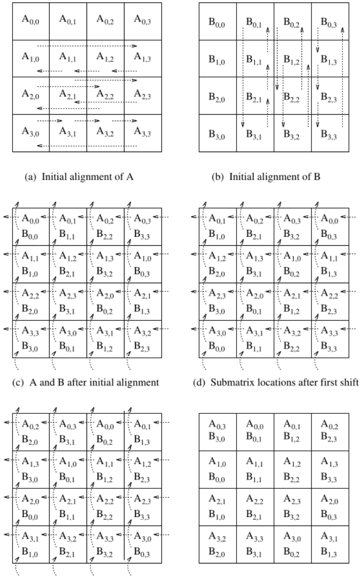
4. Subsequent communication.
Row I: Rotate A blocks left by 1.
Col J: Rotate B blocks down by 1.

5. Multiply Step (final).
Each Proc has a submatrix of A and B that allows partial computation of its output block of C

Algorithm terminates after sqrt(P) Communicate / Multiply steps.

Cannon's Algorithm Pseudocode

```
Cannon_MM(i, j, Q){
  PE(i,j) has blocks A1=A(i,j) and B1=B(i,j)
  Q is the Block Dimension : A is Q*Q blocks

  Allocate space A2, B2, Cij sized as A1

  doboth send A1 to   PE(i, j-i+Q % Q)
         recv A2 from PE(i, j+i+Q % Q)
  doboth send B1 to   PE(i-j+Q % Q, j)
         recv B2 from PE(i+j+Q % Q, j)

  for(k=1 to Q){
    copy A2 into A1, B2 into B1
    Cij += A1 * B1

    doboth send A1 to   PE(i, j-1+Q % Q)
           recv A2 from PE(i, j+1+Q % Q)
    doboth send B1 to   PE(i-1+Q % Q, j)
           recv B2 from PE(i+1+Q % Q, j)
    // optionally skip last comm
  }

  Cij now contains output block of C(i,j)
}
```


$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$
$A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$
$A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$
$A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
$A_{0,0}$ $B_{0,0}$ $A_{0,1}$ $B_{1,1}$ $A_{0,2}$ $B_{2,2}$ $A_{0,3}$ $B_{3,3}$
$A_{1,1}$ $B_{1,0}$ $A_{1,2}$ $B_{2,1}$ $A_{1,3}$ $B_{3,2}$ $A_{1,0}$ $B_{0,3}$
$A_{2,2}$ $B_{2,0}$ $A_{2,3}$ $B_{3,1}$ $A_{2,0}$ $B_{0,2}$ $A_{2,1}$ $B_{1,3}$
$A_{3,3}$ $B_{3,0}$ $A_{3,0}$ $B_{0,1}$ $A_{3,1}$ $B_{1,2}$ $A_{3,2}$ $B_{2,3}$
$A_{0,1}$ $B_{1,0}$ $A_{0,2}$ $B_{2,1}$ $A_{0,3}$ $B_{3,2}$ $A_{0,0}$ $B_{0,3}$
$A_{1,2}$ $B_{2,0}$ $A_{1,3}$ $B_{3,1}$ $A_{1,0}$ $B_{0,2}$ $A_{1,1}$ $B_{1,3}$
$A_{2,3}$ $B_{3,0}$ $A_{2,0}$ $B_{0,1}$ $A_{2,1}$ $B_{1,2}$ $A_{2,2}$ $B_{2,3}$
$A_{3,0}$ $B_{0,0}$ $A_{3,1}$ $B_{1,1}$ $A_{3,2}$ $B_{2,2}$ $A_{3,3}$ $B_{3,3}$
$A_{0,2}$ $B_{2,0}$ $A_{0,3}$ $B_{3,1}$ $A_{0,0}$ $B_{0,2}$ $A_{0,1}$ $B_{1,3}$
$A_{1,3}$ $B_{3,0}$ $A_{1,0}$ $B_{0,1}$ $A_{1,1}$ $B_{1,2}$ $A_{1,2}$ $B_{2,3}$
$A_{2,0}$ $B_{0,0}$ $A_{2,1}$ $B_{1,1}$ $A_{2,2}$ $B_{2,2}$ $A_{2,3}$ $B_{3,3}$
$A_{3,1}$ $B_{1,0}$ $A_{3,2}$ $B_{2,1}$ $A_{3,3}$ $B_{3,2}$ $A_{3,0}$ $B_{0,3}$
$A_{0,3}$ $B_{3,0}$ $A_{0,0}$ $B_{0,1}$ $A_{0,1}$ $B_{1,2}$ $A_{0,2}$ $B_{2,3}$
$A_{1,0}$ $B_{0,0}$ $A_{1,1}$ $B_{1,1}$ $A_{1,2}$ $B_{2,2}$ $A_{1,3}$ $B_{3,3}$
$A_{2,1}$ $B_{1,0}$ $A_{2,2}$ $B_{2,1}$ $A_{2,3}$ $B_{3,2}$ $A_{2,0}$ $B_{0,3}$
$A_{3,2}$ $B_{2,0}$ $A_{3,3}$ $B_{3,1}$ $A_{3,0}$ $B_{0,2}$ $A_{3,1}$ $B_{1,3}$

(a) Initial alignment of A

(b) Initial alignment of B

(c) A and B after initial alignment

(d) Submatrix locations after first shift

(e) Submatrix locations after second shift

(f) Submatrix locations after third shift

Figure 8.3 The communication steps in Cannon's algorithm on 16 processes.

Exercise: Analysis of Cannon's Algorithm

Assumptions

- Matrices A and B are size $N \times N$ so N^2 elements
- P processors with block partitioning: initially N^2/P elements of A, B on each proc (assume P is a perfect square)
- Simplified communication cost for send/recv on a Ring:

$$t_{comm} = t_s + t_w M$$

  with p #procs in ring, t_s comm startup time, t_w per word transfer rate, M message size.

Questions

1. What is communication cost of this Cannon's algorithm?
2. Is this any better/worse/same as the Naive algorithm?
3. What is the memory requirement for each proc?
4. Is this any better/worse/same as the Naive algorithm?

Answers: Analysis of Cannon's Algorithm

1. What is communication cost of this Cannon's algorithm?
   - In each step, each proc performs 2 send/recv ops
   - Each send/recv is a block of size N^2/P
   - Block Dim `Q` $= \sqrt{P}$ for square 2D Torus
   - Total $\sqrt{P}$ steps : can skip last comm step

$$t_{comm} = 2(\sqrt{P} - 1) \times (t_s + t_w(N^2/P))$$

2. Is this any better/worse/same as the Naive algorithm?
   - Same communication cost as Naive algorithm
3. What is the memory requirement for each proc?
   - $O(N^2/P)$: 5 blocks as stated in pseudcode,
   - 3 blocks for A_{ij}, B_{ij}, C_{ij}
   - 2 "workspaces" to allow send/recv of blocks:
   - Eliminate workspace blocks in a refinement
4. Is this any better/worse/same as the Naive algorithm?
   - Cannon's $O(N^2/P)$ vs Naive $O(\sqrt{P} \times N^2/P)$
   - Memory overhead is much better: constant number of blocks rather than the need to store entire rows/cols on single procs

Lessons from Cannon's Algorithm

- Illustrates "pipelining": blocks used to compute partial results then fed forward other processors
- Benefits greatly from a 2D Grid / Torus network which facilitates local communications that arise in the algorithm
- While not as ideal as row/col partitioning for A, B, realistic and relatively efficient
- Variants of central idea exist in some libraries such as Scalapack which has a parallel `xGEMM()` using many similar ideas
- Could really use some code support for
  - 2D Coordinates for processors rather than linear rank...
  - Sending/receiving in a ring...

MPI Tricks for Rings

Sendrecv in a Ring

`MPI_Sendrecv()` allows ring-link partnering

```
// sendrecv_ring.c
int left_part  = (myrank - 1 + npes) % npes;
int right_part = (myrank + 1 + npes) % npes;
MPI_Sendrecv(&mine,  1, MPI_INT, right_part, 1,
             &yours, 1, MPI_INT, left_part,  1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Sendrecv with Replacement

`MPI_Sendrecv_replace()` allows send/recv in the same buffer

```
// sendrecv_ring.c
int mydata = 10*myrank;
MPI_Sendrecv_replace(&mydata, 1, MPI_INT,
                     right_part, 1, left_part,  1,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

In Cannon's Alg, no longer need `A1 / A2`: can send/receive block of `A` with a single buffer.

MPI Tricks for Grids: `MPI_Cart_create()`

MPI has special support for Grid/Torus network configs; allows creation of a `MPI_Comm` that maps processors to a N-D grid

- 2D Torus for Cannon's Alg

```
// cartesian_comm.c
int dim_len = 2;  // Set up the Cartesian topology
int dims[2] = {sqrt(npes), sqrt(npes)}; // # rows/cols
int periods[2] = {1, 1};      // wrap-around rows/cols

// Create the Cartesian topology, with rank reordering
MPI_Comm comm_2d;
MPI_Cart_create(MPI_COMM_WORLD,         // original comm
                dim_len, dims, periods, // cartesian comm props
                1,           // re-order linear rank if beneficial
                &comm_2d);   // new communicator with 2D coords

// Get the rank and coordinates with respect to the new topology
int my2drank = -1;            // may be differ from world rank
MPI_Comm_rank(comm_2d, &my2drank);

int mycoords[2] = {-1, -1};   // (i,j) coords
MPI_Cart_coords(comm_2d, my2drank, 2, mycoords);

printf("Proc %2d (%s): my2drank %3d mycoords (%3d, %3d)\n",
        myrank,processor_name,
        my2drank,mycoords[0],mycoords[1]);
```

MPI Tricks for Shifting

Shifts are eased by the `MPI_Cart_shift()` function

- Calculates linear rank of source/dest procs for shift operations in a Cartesian grid of procs.
- Data exchange via `MPI_Sendrecv()` is then direct

```
// cartesian_comm.c
int mydata = (100*mycoords[0])+mycoords[1];
int rowsend=-1, rowrecv=-1;

MPI_Cart_shift(comm_2d, 0, rowshift, &rowrecv, &rowsend);

MPI_Sendrecv_replace(&mydata, 1, MPI_INT,
                     rowsend, 1, rowrecv,  1,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Cannon's Algorithm in MPI

- Grama Program 6.2 is Cannon's Matrix Multiply algorithm implemented via MPI
- Uses the tricks mentioned on the past 2 slides to ease implementation burden
- See `cannon_grama.c` for a source code version of it

Note: I haven't tested this code but everything from textbooks always works out the box, right?

Linear Equations

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m
\end{aligned}$$

Summarized in matrix form as

$$A\mathbf{x} = \mathbf{b}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Usually given A, b, must find x. An inordinate amount of CPU cycles are spent on this problem.

Solving Triangular Systems

Easier than a general system via **back substitution** process

```
       A         b                          A         b
[ 1  2  3  4 | 30 ]                 [ 1  2  3  0 | 14 ]  30-4*4
[ 0  5  6  7 | 56 ]                 [ 0  5  6  0 | 28 ]  56-7*4
[ 0  0  8  9 | 60 ]                 [ 0  0  8  0 | 24 ]  60-9*4
[ 0  0  0  1 |  4 ] x(3) == 4       [ 0  0  0  1 |  4 ] x(3) == 4

[ 1  2  3  0 | 14 ]                 [ 1  2  0  0 |  5 ]  14-3*3
[ 0  5  6  0 | 28 ]                 [ 0  5  0  0 | 10 ]  28-6*3
[ 0  0  1  0 |  3 ] x(2) == 3       [ 0  0  1  0 |  3 ] x(2) == 3
[ 0  0  0  1 |  4 ] x(3) == 4       [ 0  0  0  1 |  4 ] x(3) == 4

[ 1  2  0  0 |  5 ]                 [ 1  0  0  0 |  1 ]   5-2*2
[ 0  1  0  0 |  2 ] x(1) == 2       [ 0  1  0  0 |  2 ] x(1) == 2
[ 0  0  1  0 |  3 ] x(2) == 3       [ 0  0  1  0 |  3 ] x(2) == 3
[ 0  0  0  1 |  4 ] x(3) == 4       [ 0  0  0  1 |  4 ] x(3) == 4

[ 1  0  0  0 |  1 ] x(0) == 1
[ 0  1  0  0 |  2 ] x(1) == 2
[ 0  0  1  0 |  3 ] x(2) == 3
[ 0  0  0  1 |  4 ] x(3) == 4
```

Standard Code for Back Substitution

```
BACK_SUBSTITUTE(A,b,x){
  N = nrows(A)
  for( j=N-1 downto 0 ) {
    x[j] = b[j] / A[j,j]
    for[ i=j-1 downto 0 ] {
      b[i] = b[i] - A[i,j]*x[j]
      A[i,j] = 0  // OPTIONAL
    }
  }
  x[] now contains solutions
  b[] has been modified
  A[] has been modified if OPTIONAL is executed
}
```

Computational complexity for square matrix of size N?

Getting a Triangular Matrix via Gaussian Elimination

- Standard solution algorithm to find x in $Ax = b$
- Converts A to U which is upper triangular

```
           A             b
[  1    2    3   -4 | -14 ]
[  2    7   21   10 |  38 ] -2 row0
[  4   13   43    2 |  24 ] -4 row0
[ -2   -2    7   15 |  60 ] +2 row0

[  1    2    3   -4 | -14 ]
[  0    3   15   18 |  66 ]
[  0    5   31   18 |  80 ]    -5/3 row1
[  0    2   13    7 |  32 ]    -2/3 row1

[  1    2    3   -4 | -14 ]
[  0    3   15   18 |  66 ]
[  0    0    6  -12 | -30 ]
[  0    0    3   -5 | -12 ]      -1/2 row3

[  1    2    3   -4 | -14 ]   [  1    0    0    0]   L is formed from negative
[  0    3   15   18 |  66 ]   [  2    1    0    0]   coefficients found via
[  0    0    6  -12 | -30 ]   [  4  5/3    1    0]   Gaussian elimination with
[  0    0    0    1 |   3 ]   [ -2  2/3  1/2    1]   unit main diagonal.
           U             b'               L
```

LU: The Lower Upper Decomposition

- By tracking the coefficients used during the Gaussian elimination, one gets a matrix L which is lower triangular
- Modifications to A become an upper triangular matrix U
- One can verify that $A = LU$

```
octave> rats(L)
L =
    1        0        0       0
    2        1        0       0
    4      5/3        1       0
   -2      2/3      1/2       1
```

```
octave> U
 U =
   1     2     3    -4
   0     3    15    18
   0     0     6   -12
   0     0     0     1
```

```
octave> L * U
ans =
    1     2     3    -4
    2     7    21    10
    4    13    43     2
   -2    -2     7    15
```

```
octave> L * U - A
ans =
   0    0    0    0
   0    0    0    0
   0    0    0    0
   0    0    0    0
```

Exercise: LU Factorization Pseudocode

```
LU_FACTORS(A[] : square matrix){
  N = nrows(A)
  Allocate L as N*N identity mat
  Allocate U as copy of A

  for(d=0 to N-1){                          // leading row d
    for(i=d+1 to N-1){                      // remaining rows i
      scale = U[i,d] / U[d,d]               // scale for this row
      L[i,d] = scale                        // record scale in L
      for(j=d to N-1){                      // iterate over this row j
        U[i,j] = U[i,j] - scale*U[d,j]      // subtract off scaled leading row
      }
    }
  }
  return
    L: a lower triangle matrix with factors and unit diagonal
    U: an upper triangle matrix, obeys L*U = A
}
```

- Computational Complexity?
- Could anything go sideways numerically?

Answers: LU Factorization Pseudocode

```
 1  LU_FACTORS(A[] : square matrix){
 2    N = nrows(A)
 3    Allocate L as N*N identity mat
 4    Allocate U as copy of A
 5
 6    for(d=0 to N-1){                        // leading row d
 7      for(i=d+1 to N-1){                    // remaining rows i
 8        scale = U[i,d] / U[d,d]             // scale for this row
 9        L[i,d] = scale                      // record scale in L
10        for(j=d to N-1){                    // iterate over this row j
11          U[i,j] = U[i,j] - scale*U[d,j]    // subtract off scaled leading row
12        }
13      }
14    }
15    return
16      L: a lower triangle matrix with factors and unit diagonal
17      U: an upper triangle matrix, obeys L*U = A
18  }
```

- Computational Complexity?: $O(N^3)$ - 3 nested loops
- Could anything go sideways numerically? - Division by 0 at line 8
  - To fix this requires **pivoting**
  - Robust versions permute rows so the row with the largest `U[:,d]` element used at iteration `d`

Utility of LU Decomposition

General Process

1. Want x in $Ax = b$
2. Compute $LU = A$ via Gaussian elimination
3. Use forward-substitution to find y in $Ly = b$
4. Use back-substitution to find x in $Ux = y$

Solving in this fashion exploits the following identities

$$\begin{aligned} Ux &= y \quad \text{so} \quad & L^{-1}LUx &= L^{-1}Ly \\ LU &= A \quad \text{so} \quad & Ax &= Ly \\ Ly &= b \quad \text{so} \quad & Ax &= b \end{aligned}$$

vs Gaussian Elimination

- LU factorization costs little more than Gaussian Elim
- Saving the LU Factorization allows solving for a new b with only passes of back/forward substitution

  $$Ax_1 = b_1,\ Ax_2 = b_2,\ Ax_3 = b_3$$

  1 LU decomposition then 3 rounds of back/forward substitution
- LU Decomp is $O(N^3)$
- Back/Forward Sub is (N^2)

Variants

- To save space, overwrite `L,U` in `A`
  - Upper triangle of `A` becomes `U` including main diagonal
  - Lower triangle of `A` would have been 0's, store `L` there, implied 1 diagonal
- Grama's variant makes main diagonal of U all 1's: saves some ops in back/forward substitution
- We are ignoring the need to **pivot** and permute the matrix rows for numerical stability: doing so yields the LUP decomposition with permutation matrix P

Exercise: Now, about Parallelizing...

```
LU_FACTORS_INPLACE(A[] : square matrix){
  N = nrows(A)
  // Will overwrite A with its L*U factors, no allocation of L or U

  for(d=0 to N-1){                             // leading row d
    for(i=d+1 to N-1){                         // remaining rows i
      scale = A[i,d] / A[d,d]                  // scale for this row
      A[i,d] = scale                           // record scale in L
      for(j=d+1 to N-1){                       // iterate over this row j
        A[i,j] = A[i,j] - scale*A[d,j]         // subtract off scaled leading row
      }
    }
  }
  return; // A now has its L,U factors in its lower/upper triangles
}
```

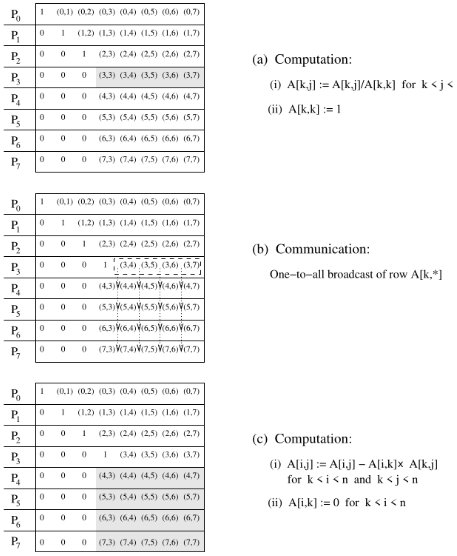
Assuming an in-place variant how would one go about parallelizing this?

- Decomposition / distribution of `A`?
- Communication at which steps?

Pitch some ideas

Answers: Now, about Parallelizing...

- Block decomposition means that some processors idle
- Row decomposition also leads to some idling, is described in Grama 8.3
- A **cyclic decomposition** leads to better balance
  - 100 x 100 matrix, 4 Procs, row cyclic
  - `P0: rows 4*i+0 = 0,4,8,12,...`
  - `P1: rows 4*i+1 = 1,5,9,13,...`
  - etc.
- Broadcast leading row from owning proc to all others



(a) Computation:
(i) A[k,j] := A[k,j]/A[k,k] for k < j <
(ii) A[k,k] := 1
(b) Communication:
One−to−all broadcast of row A[k,*]
(c) Computation:
(i) A[i,j] := A[i,j] − A[i,k]× A[k,j] for k < i < n and k < j < n
(ii) A[i,k] := 0 for k < i < n

Figure 8.6 Gaussian elimination steps during the iteration corresponding to $k = 3$ for an 8×8 matrix partitioned rowwise among eight processes.

Analysis of LU Decomposition

- Serial algorithm runs in $O(N^3)$
- Parallel approaches use
  1. N iterations of each row as the leading row
  2. Broadcast of leading row `d` to all P procs : N broadcasts
  3. Parallel modification of $N - d$ lower block of `A[]` to store L, U factors in it $O(N^2/P)$
- For a ring of P procs to broadcast length N row

  $$t_{broadcast} = \log_2(P) t_s + t_w N P$$

  leading to overall complexity of

  $$\begin{aligned} T &= N \times (N^2/P + \log_2(P) t_s + t_w N P) \\ &= N^3/P + N \log_2(P) t_s + t_w N^2 P \\ \text{is} \quad & O(N^3/P) \end{aligned}$$

- Main overhead is the need to broadcast a row at each step,
- **Pipelined** Broadcast improves on this: good implementations of `MPI_Bcast()` has a node pass on messages, begin computation again ASAP, not idle while broadcast completes