# CoE 163

Computing Architectures and Algorithms

## Sparse Matrices

# What are sparse matrices?

- Matrices with large number of zero entries
- If there are enough zeros, it might be worth using an algorithm that avoids:
  - Storing zero entries
  - Operating on zero entries

# How should our algorithm deal with sparse matrices?

- Many sparse methods available
- Choosing the best one often requires substantial knowledge about the matrix
- We shall focus on basic issues in sparse Gaussian elimination
- Supplementary references and exercises you can try on MATLAB are provided

# Examples of Sparse Matrices and Issues to Consider

# LU Factorization of an Arrow Matrix

- An arrow matrix forms an arrow with its nonzero entries
- Consider the example below:

$$
A = \begin{bmatrix} 1 & & & & .1 \\ & 1 & & & .1 \\ & & 1 & & .1 \\ & & & 1 & .1 \\ .1 & .1 & .1 & .1 & 1 \end{bmatrix}
$$

- [The zero entries have been left blank]
- What happens when we perform LU factorization? (try to solve it yourself manually—or use software)

# LU Factorization of an Arrow Matrix

- $A = \begin{bmatrix} 1 & & & & .1 \\ & 1 & & & .1 \\ & & 1 & & .1 \\ & & & 1 & .1 \\ .1 & .1 & .1 & .1 & 1 \end{bmatrix} = L \cdot U$

$$L \cdot U = \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ .1 & .1 & .1 & .1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & & & & .1 \\ & 1 & & & .1 \\ & & 1 & & .1 \\ & & & 1 & .1 \\ & & & & .96 \end{bmatrix}$$

# LU Factorization of an Arrow Matrix

$$A = L \cdot U = \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ .1 & .1 & .1 & .1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & & & & .1 \\ & 1 & & & .1 \\ & & 1 & & .1 \\ & & & 1 & .1 \\ & & & & .96 \end{bmatrix}$$

- None of the zero entries of $A$ were filled in
- $L$ and $U$ together can be stored in (overwrite) the nonzero entries of $A$
- Total *essential* arithmetic operations (exclude adding or multiplying by zero): only 12
  - 4 divisions for last row of L
  - 8 multiplications and additions to compute the bottom rightmost entry (0.96 in the example)

# LU Factorization of an Arrow Matrix

- LU factorization on a typical (not sparse) n-by-n matrix:
  - Need $n^2$ locations to store matrix
  - $\frac{2}{3}n^3$ floating point operations

- LU factorization on an n-by-n arrow matrix:
  - Only need $3n - 2$ locations to store matrix
  - $3n - 3$ floating point operations

- When $n$ is large, **space required** and **number of operations** is tiny compared to dense LU factorization

# What if we reversed the order of columns and rows?

- Suppose we instead had $A'$ shown below. What happens when we do LU factorization?

$$A' = \begin{bmatrix} 1 & .1 & .1 & .1 & .1 \\ .1 & 1 & & & \\ .1 & & 1 & & \\ .1 & & & 1 & \\ .1 & & & & 1 \end{bmatrix}$$

# L' and U' have filled completely!

$$A' = \begin{bmatrix} 1 & .1 & .1 & .1 & .1 \\ .1 & 1 & & & \\ .1 & & 1 & & \\ .1 & & & 1 & \\ .1 & & & & 1 \end{bmatrix} = L' \cdot U'$$

$$L' \cdot U' = \begin{bmatrix} 1 & & & & \\ .1 & 1 & & & \\ .1 & -.01 & 1 & & \\ .1 & -.01 & -.01 & 1 & \\ .1 & -.01 & -.01 & -.01 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & .1 & .1 & .1 & .1 \\ & .99 & -.01 & -.01 & -.01 \\ & & .99 & -.01 & -.01 \\ & & & .99 & -.01 \\ & & & & .99 \end{bmatrix}$$

# Need to use dense Gaussian elimination

$$L' \cdot U' = \begin{bmatrix} 1 & & & & \\ .1 & 1 & & & \\ .1 & -.01 & 1 & & \\ .1 & -.01 & -.01 & 1 & \\ .1 & -.01 & -.01 & -.01 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & .1 & .1 & .1 & .1 \\ & .99 & -.01 & -.01 & -.01 \\ & & .99 & -.01 & -.01 \\ & & & .99 & -.01 \\ & & & & .99 \end{bmatrix}$$

- Need $\boldsymbol{n^2}$ locations to store $L$' and $U$'
- Same amount of work as dense Gaussian elimination, $\boldsymbol{\frac{2}{3}n^3}$

Order of rows and columns is extremely important!

# How do we choose optimal permutations of rows and columns?

- We have seen that we can minimize storage if the correct order of rows/columns is used
- However, choosing the optimal order or permutations of rows and columns is extremely hard (NP-complete!)
  - All known algorithms to find optimal permutation grows exponentially with $n$
  - We need to settle for using heuristics (more practical/experimental way of problem-solving, but may not be most optimal)

# Rough strategy for optimizing Factorization of Sparse Matrices

1. Design/choose a data structure that holds only nonzero entries of A
2. Design/choose data structure to accommodate new entries of L and U that fill in during elimination
   - Option 1: Data grows dynamically
   - Option 2: Pre-compute size w/o actually performing elimination (computation must not be costly)
3. Use the data structure to perform only minimum number of floating point operations

# Learn by Benchmarking!

- Check out this MATLAB example: https://www.mathworks.com/help/matlab/math/sparse-matrix-reordering.html
- Follow the discussion and try to run the sample codes on MATLAB Online (your up.edu.ph webmail account should have access)


- If you need a tutorial on Cholesky decomposition, see the youtube link provided on UVLe

# Software for Sparse Matrix Operations

- MATLAB:
  - Operations on sparse matrices return sparse matrices; operations on full matrices return full matrices
  - MATLAB only stores nonzero entries of sparse matrices
  - Has built in functions for sparse matrix creation and manipulation

**Functions**

**Creation**

| | |
|---|---|
| `spalloc` | Allocate space for sparse matrix |
| `spdiags` | Extract nonzero diagonals and create sparse band and diagonal matrices |
| `speye` | Sparse identity matrix |
| `sprand` | Sparse uniformly distributed random matrix |
| `sprandn` | Sparse normally distributed random matrix |
| `sprandsym` | Sparse symmetric random matrix |
| `sparse` | Create sparse matrix |
| `spconvert` | Import from sparse matrix external format |

https://www.mathworks.com/help/matlab/sparse-matrices.html

# Software for Sparse Matrix Operations

- Other public domain and commercial sparse matrix software are available
- Active research area; difficult to recommend a single best algorithm

# Software for Sparse Matrix Operations

| Matrix type | Name | Algorithm | Status/ source |
|---|---|---|---|
| Serial algorithms | | | |
| nonsym. | SuperLU | LL, partial, BLAS-2.5 | Pub/NETLIB |
| nonsym. | UMFPACK [62, 63] | MF, Markowitz, BLAS-3 | Pub/NETLIB |
| | MA38 (same as UMFPACK) | | Com/HSL |
| nonsym. | MA48 [96] | Anal: RL, Markowitz<br>Fact: LL, partial, BLAS-1, SD | Com/HSL |
| nonsym. | SPARSE [167] | RL, Markowitz, scalar | Pub/NETLIB |
| sym-pattern | MUPS [5] | MF, threshold, BLAS-3 | Com/HSL |
| sym-pattern | MA42 [98] | Frontal, BLAS-3 | Com/HSL |
| sym. | MA27 [97]/MA47 [95] | MF, $LDL^T$, BLAS-1/BLAS-3 | Com/HSL |
| s.p.d. | Ng & Peyton [191] | LL, BLAS-3 | Pub/Author |
| Shared-memory algorithms | | | |
| nonsym. | SuperLU | LL, partial, BLAS-2.5 | Pub/UCB |
| nonsym. | PARASPAR [270, 271] | RL, Markowitz, BLAS-1, SD | Res/Author |
| sym-pattern | MUPS [6] | MF, threshold, BLAS-3 | Res/Author |
| nonsym. | George & Ng [115] | RL, partial, BLAS-1 | Res/Author |
| s.p.d. | Gupta et al. [133] | LL, BLAS-3 | Com/SGI<br>Pub/Author |
| s.p.d. | SPLASH [155] | RL, 2-D block, BLAS-3 | Pub/Stanford |
| Distributed-memory algorithms | | | |
| sym. | van der Stappen [245] | RL, Markowitz, scalar | Res/Author |
| sym-pattern | Lucas et al. [180] | MF, no pivoting, BLAS-1 | Res/Author |
| s.p.d. | Rothberg & Schreiber [207] | RL, 2-D block, BLAS-3 | Res/Author |
| s.p.d. | Gupta & Kumar [132] | MF, 2-D block, BLAS-3 | Res/Author |
| s.p.d. | CAPSS [143] | MF, full parallel, BLAS-1 (require coordinates) | Pub/NETLIB |

*Abbreviations used in the table:*
nonsym. = nonsymmetric.
sym-pattern = symmetric nonzero structure, nonsymmetric values.
sym. = symmetric and may be indefinite.
s.p.d. = symmetric and positive definite.
MF, LL, and RL = multifrontal, left-looking, and right-looking.
SD = switches to a dense code on a sufficiently dense trailing submatrix.
Pub = publicly available; authors may help use the code.
Res = published in literature but may not be available from the authors.
Com = commercial.
HSL = Harwell Subroutine Library:
http://www.rl.ac.uk/departments/ccd/numerical/hsl/hsl.html.
UCB = http://www.cs.berkeley.edu/~xiaoye/superlu.html.
Stanford = http://www-flash.stanford.edu/apps/SPLASH/.

Software to solve sparse linear systems, taken from *Applied Numerical Linear Algebra* by James W. Demmel

# Key Takeaways

- Sparse matrices provide computational advantages:
  - **Memory management:** minimize required memory locations by storing only nonzero elements
  - **Computational efficiency:** only perform *essential* operations, i.e. do not perform multiplications and additions by 0, for example
- Choosing the best algorithm/software is not straightforward
  - Need substantial knowledge of the matrix to know the optimal data structure and algorithm
  - No “one-size-fits-all” solution