INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Spring Semester, 2020
Subject: Introduction to Parallel Scientific Computing (CS504)

HW I
January 25, 2020

Due: 05.02.20 **Instructor: Dr. Pawan Kumar** Maximum Marks: 27

INSTRUCTIONS:

The codes must be written in either C, C++, Python, or Matlab. If using Python, dont use existing libraries (such as numpy or scipy) for dense or sparse matrices. However, you may use it to check the correctness of your algorithm. Questions with * are difficult, and can be skipped. Submit this assignment on Moodle with your roll number as file name.**

1. (**Matrix Multiplication**) Given two matrices A and B. Write a code for matrix-matrix multiplication when [2+2+3+3+2]

   1. A and B are **dense** matrices. A and B are random matrices.
   2. A and B are **banded** matrices. Use the banded storage scheme discussed in class. You may write a code that converts dense banded matrix to banded format, then does banded matrix multiply.
   3. A and B are **sparse** matrices (a matrix that has significantly more number of zeros compared to number of non-zeros).

      Hint: To do sparse matrix times sparse matrix multiplication, you may do the following:

      $$C = AB = [Ab_1, Ab_2, \cdots, Ab_n],$$

      that is, ith column of output matrix C is obtained by Ab_i. Hence, you may first write sparse matrix times dense vector routine. You need to create dense vector for b_i. then after doing $c_i = Ab_i$, you will obtain dense c_i, you may sparsify c_i, and store these as sparse (CSR/COO) data format for C. If A and B are very sparse, then C is expected to be very sparse, hence, we want to keep the output matrix C sparse too!

      For this case, consider the following storage schemes.

      (a) **Coordinate Storage Format (COO)**: In this format, the matrix entries are stored in three arrays, namely, `row_indices`, `col_indices`, and `val` :

         i. The array `row_indices` contains the row indices of non-zero entries. It is of length `nz`. Here `nz` refers to the total number of non-zeros.

         ii. The array `col_indices` contains the column indices. It is of length `nz`.

         iii. Array `val` contains the matrix entries at the corresponding row and column. It is also of length `nz`.

For example, for the following matrix

$$\begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}, \tag{1}$$

the arrays `row_indices`, `col_indices`, and `val` are given as follows

$$\begin{aligned} \texttt{row\_indices} &= \begin{bmatrix} 5 & 3 & 3 & 2 & 1 & 1 & 4 & 2 & 3 & 2 & 3 & 4 \end{bmatrix}, \\ \texttt{col\_indices} &= \begin{bmatrix} 5 & 5 & 3 & 4 & 1 & 4 & 4 & 1 & 1 & 2 & 4 & 3 \end{bmatrix}, \\ \texttt{val} &= \begin{bmatrix} 12 & 9 & 7 & 5 & 1 & 2 & 11 & 3 & 6 & 4 & 8 & 10 \end{bmatrix}. \end{aligned}$$

Note that the entries of the array `val` are not written in ordered way, for example, the first entry is 12, which is the $(5,5)th$ entry of the matrix.

(b) **Compressed Sparse Row (CSR) Format**: Here again, the matrix data is stored in three arrays, namely, `val`, `col_indices`, and `row_pointers` :

i. All the matrix entries are stored in arrays `val` row by row. Along each row, they are stored from smallest column number to the largest. The length of `val` is `nz`.

ii. An integer array `col_indices` contains the column indices of the elements stored in the array `val` above.

iii. An integer array `row_pointers` contains the pointers to the beginning of each row in the arrays `val` and `col_indices`. Thus, the content of `row_pointers(i)` is the position in arrays `val` and `col_indices` where the `i`-th row starts.

For the matrix (1) above, the CSR format is given as follows

$$\begin{aligned} \texttt{val} &= \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{bmatrix}, \\ \texttt{col\_indices} &= \begin{bmatrix} 1 & 4 & 1 & 2 & 4 & 1 & 3 & 4 & 5 & 3 & 4 & 5 \end{bmatrix}, \\ \texttt{row\_pointers} &= \begin{bmatrix} 1 & 3 & 6 & 10 & 12 & 13 \end{bmatrix}. \end{aligned}$$

4. Determine the storage complexity and flops for all three items above.

2. **(LU Factorization)** Given a dense matrix $A \in \mathbb{R}^{n \times n}$. [3+2+2+8]

1. Write a code to compute the LU factorization of A. The factors L and U must be overwritten in A. Name this function `mylu.*`. Write a code that implements forward substitution (`foreward.*`) with a lower triangular matrix L. Similarly, write a code that implements backward substitution (`back.*`) with an upper triangular matrix U.

2. Use algorithms above to write a code, for example, `lu_solve.*` that takes the matrix A and a right hand side vector b as inputs, and it outputs the solution to the equation $Ax = b$ by first doing forward substitution, i.e., by solving $Lt = b$, then performing the backward substitution, i.e., by solving $Ux = t$. Check your solution by computing $\|b - Ax\|_2$, it should be close to zero (in double precision arithmetic). [Note that LU factors are stored in A, so keep another copy of A to check your solution.]

3. Adapt your code to optimally compute the LU factorization of a Hessenberg matrix.

4. ***(**Write this in C/C++**) Write a LU factorization routine when the matrix A is stored in CSR format. Name this function `lu_sparse.*`. The L and U factors must also be stored in CSR format. Then write a routine `forward_sparse.*` to do forward substitution for a sparse lower triangular matrix stored in CSR format, similarly, write a backward substitution routine `backward_sparse.*` for a sparse upper triangular matrix stored in CSR format. As before, to solve $Ax = b$, you need to first solve $Lt = b$, which is a forward substitution, and then solve $Ux = t$, which is a backward substitution to obtain the solution x to the given linear system $Ax = b$. What difficulties you faced? Did you wish doing some symbolic analysis to know the sparsity pattern? What ideas you suggest? [Hint: Elimination tree!]
