Spring Semester, 2019
Subject: Introduction to Parallel Scientific Computing (CSE504)

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY HYDERABAD

Tutorial I
February 20, 2019

Due: 5.03.19 | **Instructor: Dr. Pawan Kumar** | Maximum Marks: 45

INSTRUCTIONS:

The codes can be written in either C, C++, MATLAB, Octave (Open source clone of MATLAB), FORTRAN, or Python. However, it is also recommended to write these codes in C/C++. Usually, if time permits, it is a good practice to write two versions of the same algorithm, first in MATLAB/Octave or Python, and then convert the same to C/C++. Write useful comments, and do proper indentation.

For testing, write a code that generates random matrix of desired size. In MATLAB, this can be done with `A = rand(m,n)`.

The questions prefixed with * are "somewhat" challenging, they may not be considered for grading, however, they may be seen as a sign of interest or motivation for working in this field in future. You may use MATLAB or other resources in ABACUS HPC system to write and run your codes.**

Please consult TAs if you have any doubts. In case you are busy, you may submit this assignment after deadline with a penalty of 10 percent.

Have fun with matrix algorithms!

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.
   2. A and B are **banded** matrices. Use the banded storage scheme discussed in class.
   3. **(Write this in only C/C++)** A and B are **sparse** matrices (a matrix that has significantly more number of zeros compared to number of non-zeros). 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 arrays `val` and corresponding row and col indices 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 1st non-zero entry of `i`-th row is found.

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

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

By convention, last entry of `row_pointers` is number of nonzeros in the matrix plus 1.

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

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

(a) 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.

(b) 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.]

(c) Adapt your code to optimally compute the LU factorization of a Hessenberg matrix.

(d) ***(**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 $Ax = b$.

3. **(QR Factorization)** Given $A \in \mathbb{R}^{m\times n}$. [2+2+2+3+3+6]

(a) Given a vector $x \in \mathbb{R}^m$, write a code that computes $v \in \mathbb{R}^m$ with $v(1) = 1$, and $\beta \in \mathbb{R}$ such that the Householder matrix $P = I_m - \beta vv^T$ is orthogonal and $Px = \|x\|_2 e_1$. Name this function `house.*`.

(b) Write a code `myqr.*` that computes Householder QR. The Q and R factors must be overwritten in A. Then use this factorization to solve the linear system $Ax = b$ by using the fact that

$$Ax = b \implies QRx = b \implies Rx = Q^T b.$$

Since R is upper triangular, back substitution easily gives the desired solution x.

(c) Adapt your code to optimally compute the LU factorization of a Hessenberg matrix.

(d) Write a code `recursive_qr.*` for recursive implementation of QR algorithm as discussed in class.

(e) Let $m > n$, then the linear system $Ax = b$ may not have a solution. In this case, we may seek a minimum norm solution, for example, a least squares solution. Use the QR algorithm above to write a code that solves the following least squares problem

$$\min_{x\in\mathbb{R}^n} \|b - Ax\|_2.$$

Note that since $m > n$, in R, the rows from $n+1$ to m will be zero. This means that the these rows are redundant and can be removed from R to obtain $\hat{R}$, similarly, the columns $n+1$ to m can be removed from Q. to obtain $\hat{Q}$. Then we have

$$A = \hat{Q}\hat{R}.$$

The above minimization problem reduces (how?) to

$$\min_{x\in\mathbb{R}^n} \|\hat{Q}^T b - \hat{R}x\|_2 + C, \quad \text{C is constant (what is C?).}$$

Hence, the solution to the least squares problem is obtained by doing backward sweep (since $\hat{R}$ is upper triangular and square) to solve

$$\hat{R}\hat{x} = \hat{b}, \quad \hat{b} = \hat{Q}^T b, \quad x \in \mathbb{R}^n.$$

So, $x = [\hat{x} \,|\, 0]^T \in \mathbb{R}^n$ is the solution.

(f) ***(**Write this in C/C++**) Repeat item 2 above, when the given matrix A is stored in CSR format. Determine the flop count in this case.
