COSC 420 - High-Performance Computing
Lab 2

Dr. Joe Anderson

Due: 24 September

1 Preliminaries

Recall that in the mathematical definition of a matrix, we say that A is a real $n \times m$ matrix if

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & a_{2,3} & \dots & a_{2,m} \\ \vdots & & \ddots & & \vdots \\ a_{n,1} & a_{n,2} & a_{n,3} & \dots & a_{n,m} \end{pmatrix}$$

where each $a_{i,j} \in \mathbb{R}$ for $i = 1, 2, 3, \dots, n$ and $j = 1, 2, 3, \dots, m$. In this notation, the matrix has n rows and m columns.

If A and B are both of dimension $n \times m$, then the sum is the $n \times m$ matrix C with entries given by

$$C_{i,j} = a_{i,j} + b_{i,j}$$

for $i = 1, 2, \dots, n$ and $j = 1, 2, \dots, m$.

If A is an $n \times m$ matrix and B is an $m \times k$ matrix, then the product AB is matrix C, where

$$C_{i,j} = \sum_{l=1}^{m} a_{i,l} b_{l,j}$$

for $i = 1, 2, \dots, n$, and $j = 1, 2, \dots, k$; so C is a new $n \times k$ matrix. Thus the (i, j) entry of C is, in another manner of speaking, the ith row of A, dot product with the jth column of B.

The transpose of $n \times m$ matrix A is a $m \times n$ matrix denoted as A^T and defined such that $(A^T)_{ij} = A_{ji}$.

Your task below is to design and implement a data structure in `C` to model a matrix of real numbers, then to define the operations of 1) addition, 2) subtraction, and 3) multiplication, and 4) transpose.

To program with matrices in C: Recall that in `c++`, one may declare multi-dimensional arrays. For example: `int arr[3][5]` declares an array of three five-element integer arrays, containing fifteen total integers. One way to visualize this is as follows:

```
[
  [1, 2, 3, 4, 5],  // arr[0]
  [6, 7, 8, 9, 10], // arr[1]
  [11,12,13,14,15]  // arr[2]
]
```

To access the second element in the third array, one would use the syntax `arr[2][1]`.

However, in the C language, it becomes cumbersome and less efficient to use the two-dimensional style, due to the lack of constructors, destructors, etc. Instead, a more conventional technique is to allocate a

single-dimension array large enough to hold all $n \cdot m$ elements. Then, to access the (i, j)th element, one needs to work a bit harder; the code really needs then to access element $i \cdot n + j$. To make this a bit simpler, a recommended helper is to use a macro (instead of a function, for efficiency).

```
// set up a C macro to calculate the location of element (i,j)
#define INDEX(n,m,i,j) m*i + j

// .. some code to get or calculate the dimensions n and m

// We will now need to store n*m*sizeof(int) bytes
int* A = malloc(n*m*sizeof(int));

// populate the arrayx with random numbers, using matrix-style indexing
for(int i=0; i<n; i++){
  for(int j=0; j<m; j++){
    A[INDEX(n,m,i,j)] = rand(); // the compiler substitutes inside the [] with "n*i + j"
  }
}

// the matrix is (much) easier to de-allocate as well
free(A);
```

2 Objectives

In this lab you will focus on the following objectives:

1. Develop familiarity with `C` programming language
2. Develop familiarity with parallel computing tools `MPICH` and `OpenMPI`
3. Explore empirical tests for program efficiency in terms of parallel computation and resources

3 Tasks

1. You may work in groups of one or two to complete this lab. Be sure to document in comments and your `README` who the group members are.
2. Write a short program to practice using `MPI_Scatter` and `MPI_Reduce` where:
   (a) A root node generates two random vectors in high dimension (thousands)
   (b) "Blocks" of those vectors are then scattered to various nodes, each of which performs a partial inner product
   (c) The partial inner products are then combined by the root and reported to the user
3. Write a library to perform the basic matrix operations of addition, subtraction, multiplication, and transpose.
   (a) Distribute the tasks of addition, by splitting the matrices into "blocks"
   (b) For multiplication, each member of the result matrix can be calculated independently, given enough parts of the argument matrices. Consider ways to do this distribution efficiently, keeping in mind the cost of replicating the matrices. Be sure to document your procedure and record its performance metrics.

(c) Use `MPI_Scatter` and its variants to distribute matrix data (possibly re-using the code from before).

4. Note that for debugging output, it may be helpful to have each node write its output to a file, unique to that process. This will be how we will gather logs when submitting to larger clusters. For example:

```
// using the fopen utility to get a file handle and send it data through a buffer
// use the stdio.h libraries to send output through buffers
// NB: the fflush(FILE* fh) function can force a buffer flush through the file

FILE *handle;
char* fname[256]; // should be big enough :)
sprintf(fname, "outfile_%d.txt", myRank);

handle = fopen(fname, "rw");

fprintf(handle, "Begin output from processor %d", myRank);
puts("Can also use puts!", handle);

fclose(handle); // release the file
```

5. Test the program on some large matrices (at least tens of thousands of rows/columns, perhaps more).
   (a) Run each input several times and use the `time` command to measure the time to complete each
   (b) Record the averages of each, report them in a clean tabular format
   (c) Be sure to use tools such as `valgrind` and `gdb` to find and fix bugs with your code. Make sure there are not memory leaks, invalid access, or usage of undefined variables!

6. Include a `README` file to document your code, any interesting design choices you made, and answer the following questions:
   (a) What is the theoretical time complexity of your algorithms (best and worst case), in terms of the input size?
   (b) According to the data, does adding more nodes perfectly divide the time taken by the program?
   (c) What are some real-world software examples that would need the above routines? Why? Would they benefit greatly from using your distributed code?
   (d) How could the code be improved in terms of usability, efficiency, and robustness?

4 Submission

All submitted labs must compile with `mpicc` and run on the COSC Linux environment. Include a `Makefile` to build your code. Include the output from test cases to demonstrate the correctness and completeness of your program (the `script`) utility can be of help here, as well as sending output directly to a file. Upload your project files to MyClasses in a single `.zip` file.