COSC 420 - High-Performance Computing
Lab 4

Dr. Joe Anderson

Due: 7 November

1 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. Develop familiarity with linear algebra and solving linear systems of equations.
4. Explore empirical tests for program efficiency in terms of parallel computation and resources

2 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. You will add features to your matrix operation library from Lab 3. So begin by verifying that matrix operations operate correctly and efficiently.
3. Using `MPI`, implement a first-pass attempt to calculate the eigenvector corresponding to the largest eigenvector of a matrix. You may use the naïve power-method discussed in lecture:
   (a) Let $A \in \mathbb{R}^{d \times d}$ and let x be the all 1's vector in $\mathbb{R}^d$.
   (b) Perform update: $x \leftarrow Ax$
   (c) Normalize: $x \leftarrow x/\|x\|_2$. **Note:** $\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_d^2}$ is the Euclidean norm.
   (d) From this, x will converge quite quickly to the eigenvector corresponding to the largest eigenvalue. You may obtain an estimate of this eigenvalue by computing $\|Ax\|_2/\|x\|_2$.
      i. Make sure you test for convergence by choosing a small threshold (e.g. 10^{-16}) and stopping when the new x differs from the old one by less than that amount.
      ii. You will also want to set an upper-bound on the number of iterations to perform and report an error if it has not converged after that limit.
4. Implement two ways to parallelize the data-access of this algorithm:
   (a) One in which the root processor reads the entire A and x data from a file and scatters them appropriately to perform the matrix multiplication with your existing library. Then use other collectives as appropriate to perform the update and normalization in parallel.
   (b) One which uses `MPI_File` access methods to read/write the distributed "chunks" of A and x to files instead of using scatter/gather.

(c) Compare these two methods in terms of efficiency in your readme.

5. 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.

6. Test the program on some large matrices (at least tens of thousands of rows/columns, perhaps more).

   (a) Run each input several and measure the time to complete each.

      i. You can use the MPI "Walltime" function to measure the time taken to get between two points in a body of code:

```
double startTime, stopTime;
startTime = MPI_Wtime();
/* Do some work here */
stopTime = MPI_Wtime();
printf("MPI_Wtime measured: %1.2f seconds\n", stopTime-startTime);
fflush(stdout); // manually flush the buffer for safety
```

      ii. For a simple timing method on a single processor, you can use the features of `time.h`, e.g.:

```
clock_t begin = clock();
/* here, do your time-consuming job */
clock_t end = clock();
double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
```

      However, be aware that this *does not measure real-time*. It only measures the time that the OS spends on your process, which means it may be shorter than your real-world observation, for instance with the `time` shell command.

   (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!

7. 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?

3 Submission

All submitted labs must compile with `mpicc` and run on the COSC Linux environment. Include a `Makefile` to build your code. Upload your project files to MyClasses in a single `.zip` file. Finally, turn in (stapled) printouts of your source code, properly commented and formatted with your name, course number, and complete description of the code and constituent subroutines. Also turn in printouts reflecting several different runs of your program (you can copy/past from the terminal output window). Be sure to test different situations, show how the program handles erroneous input and different edge cases.