COSC 420 - High-Performance Computing
Lab 3

Dr. Joe Anderson

Due: 31 October

1 Preliminaries

In scientific and industrial applications, it is exceedingly common to encounter data models that can be represented by a linear system of equations

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,m}x_m &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,m}x_m &= b_2 \\ &\vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,m}x_m &= b_n \end{aligned}$$

which can, of course, be captured by the more concise matrix formula $Ax = b$, where $A \in \mathbb{R}^{n\times m}$, $x \in \mathbb{R}^m$, and $b \in \mathbb{R}^n$ (we will typically interpret vectors as "column-vectors" in such equations, as above; at other times, we may need the notation x^T to denote that the vector is being used in a row-wise fashion).

One method of solving such equations is the Gauss-Jordan elimination algorithm, which proceeds as follows:

Algorithm 1 Gauss-Jordan Elimination

function GaussJordan($A \in \mathbb{R}^{n\times m}$, $b \in \mathbb{R}^n$)
  for Pivot row $k = 1, 2, \ldots, n$ **do**
    Compute the vector of scalings $\ell_i^k = A_{i,k}/A_{k,k}\ \forall i = 1, \ldots, n$.
    Broadcast the vector ℓ and perform the following on n nodes:
    for Each row $r \neq k$ **do**
      for Each column $c = 1, 2, \ldots m$ **do**
        $A_{r,c} \leftarrow A_{r,c} - \ell_r^k \cdot A_{k,c}$
      end for
      for Each column c of b **do**
        $b_{r,c} \leftarrow b_{r,c} - \ell_r^k \cdot b_{k,c}$
      end for
    end for
  end for
  Scale each row of A and b by the diagonal elements of A, so that the diagonal entries of A are now all 1, and off-diagonal entries are 0
  The vector b left over is the solution vector for x that solves the linear system. Verify this by using matrix-vector multiplication! That is, once you solve for x, use your matrix multiplication routine to check that Ax results in the b vector.
end function

The Gauss-Jordan elimination algorithm can also be used to compute the inverse of a matrix. That is, for matrix A, find the matrix A^{-1} so that $AA^T = I_n$, the n-dimensional identity matrix (the $n \times n$ matrix with diagonal entries equal to 1 and off-diagonal entries equal to 0). To do this, replace the vector b with I_n and perform the same algorithm! You will now essentially solve the linear matrix equation $AB = I_n$.

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. 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

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. First, take steps to improve the performance of your matrix processing library from last lab. Use benchmarking (timing, counting operations, etc.) to demonstrate the change in performance (FLOPS, data processed per second, etc.) versus the old version of your code. Record which programming techniques you used to achieve this speedup, whether you use cache blocking, different parallelism methods, etc.
3. Add to your library methods to solve linear systems of equations, and to compute the inverse of a matrix, by using Gauss-Jordan elimination. For inversion, you may restrict the input to be only square matrices. Utilize MPI to parallelize the algorithm, as noted in the pseudocode above.
4. Allow your code to run on file input, using the `MPI_File` family of functions to efficiently distribute the matrix data for the required operations. Then save the result (and possibly any intermediate data) to a (shared) file.
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 more basic timing method, you can use the features of `time.h`, e.g.:

```c
clock_t begin = clock();
/* here, do your time-consuming job */
clock_t end = clock();
// clock() returns a clock_t and
// CLOCKS_PER_SEC is used to convert it back to seconds
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?

4 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.

5 Bonus

1. (10 pts) Use MPI file views and custom datatypes to aid the distribution of your matrix data before and after processing.