CX 4220/CSE 6220 High Performance Computing (Spring 2025)
Programming Assignment 3
Due: April 14, 2025

1 Problem Statement

In this assignment you will implement Sparse General Matrix-Matrix Multiplication (SpGEMM) with a 2D $\sqrt{p} \times \sqrt{p}$ processor grid using MPI. This will involve first distributing nonzero entries of input matrices among the processors and writing a 2D SpGEMM implementation that reaches required performance bounds. In addition, we will provide the code for an implementation of the All-Pairs Shortest Path (APSP) problem that will use your SpGEMM implementation. You will be required to provide the proper semiring operations in order to correctly solve APSP.

2 Problem Explanation

Sparse General Matrix-Matrix Multiplication (SpGEMM) is a variant of matrix multiplication developed for sparse matrices, where a large portion of entries are zero. SpGEMM can be contrasted with dense matrix-matrix multiplication, where matrices are assumed to have entries that are mostly nonzero. As SpGEMM concerns itself only with non-zero entries, it drastically reduces computational complexity and memory usage. SpGEMM relies on special data representations of the matrix that store and manipulate only the nonzero entries. SpGEMM's challenging part is how one devises a way to highlight and multiply only the relevant non-zero entries efficiently while the result remains in sparse format.

You will be expected to implement an SpGEMM algorithm with a 2D distribution. One way to do this is by using the Sparse SUMMA algorithm for your implementation, which is an extension of the dense parallel SUMMA algorithm for sparse matrices.

The operation being performed is $C = A * B$ for input matrices A and B. The matrices A, B will be provided to you in a COO representation, which has a structure of: `vector<pair<pair<`*row*`,`*col*`>,` *value*`>>` where only non zero entries are stored. On completion of the multiplication, we expect the matrix C to contain the results in COO structure. Note that while the input and the output formats are in COO, you can convert these into other data structures within your function in order to improve SpGEMM performance, as long as the final output is in COO.

We will be testing the SpGEMM implementation on an $A * A^T$ matrix multiplication. While the graph algorithm, which will be explained next, works with square $n \times n$ matrices, we expect your SpGEMM to handle non-square matrices. **Your SpGEMM code should work for any** $m \times p \ * \ p \times n$ **matrices.**

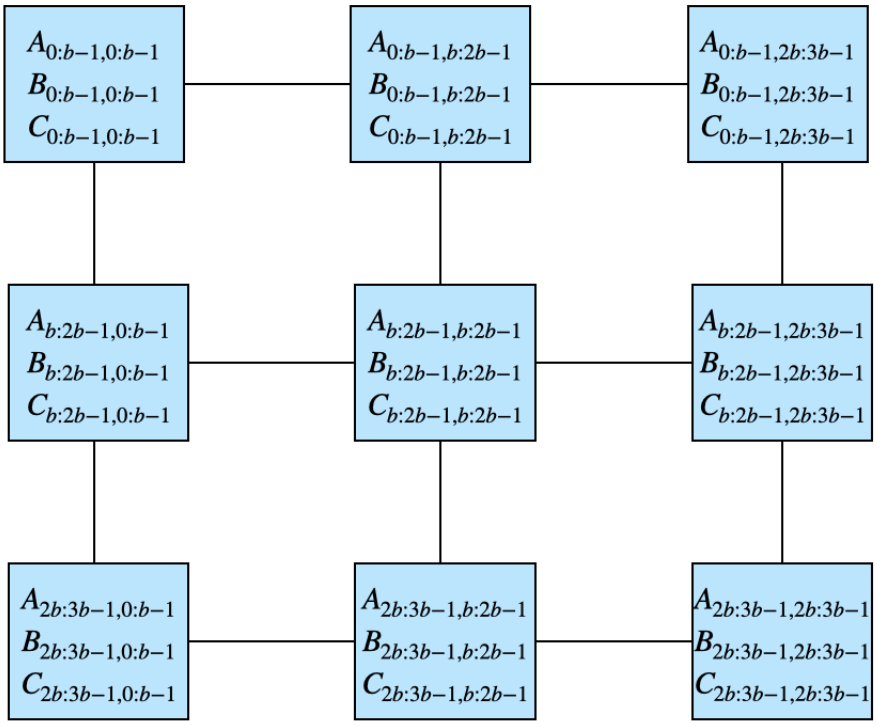

Figure 1: Example initial configuration of A, B, C matrices. Let b be the block size.

```
On each process P(i, j):
let block_size = n / proc_rows (assuming proc_rows = proc_cols)

for(int k = 0; k < proc_rows; ++k) {
  Bcast block column k of A (A[i*block_size:(i+1)*block_size - 1,
              k*block_size:(k+1)*block_size- 1]) within proc row i
  Bcast block row k of B (B[k*block_size:(k+1)*block_size - 1,
              j*block_size:(j+1)*block_size - 1]) within proc column j
  Do C += block of A * block of B
```

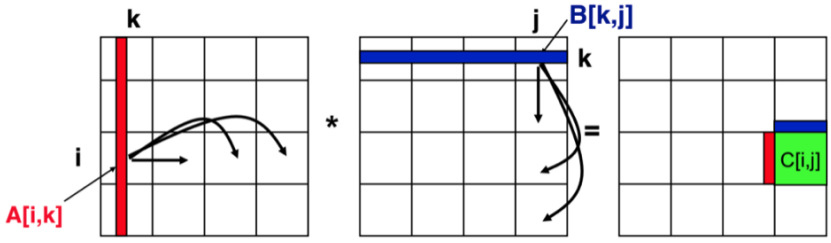

Figure 2: Pseudocode for blocked SUMMA on a square processor grid and associated illustration.

2.1 All-Pairs Shortest Path (APSP)

The **All-Pairs Shortest Path (APSP)** problem is an important problem in theoretical computer science. The goal of the All-Pairs Shortest Path (APSP) problem is to find the shortest path in a weighted graph between every single pair of vertices. A graph is weighted if every edge between vertices is assigned some weight. In our case we will assume that these weights are nonnegative integer values. A path, which is an ordered list of vertices where edges exist between each consecutive vertex, is the shortest path in a weighted graph if it is the path with the least total weight, computed as the sum of all edge weights on the path.

We will provide an implementation of APSP, which will use your SpGEMM internally. However, you will be required to choose the correct **semiring** operations to ensure that APSP is computed properly. A semiring allows us to redefine the operations that are normally performed for a matrix multiplication. Typical matrix multiplication multiplies entries and adds them to form new entries. Using a semiring, we can interchange the multiply ($\times$) and add ($+$) operations with other binary operations. Please reference the lectures on dense matrix algorithms and the CLRS book (chapter 23) for further information.

3 Code Framework

3.1 Download Files

Download the PA3 files from this link **(or)**
Use `git clone https://github.com/gtcse6220-s25/pa3.git`
Download the performance input files from Canvas and make sure to unzip them in the root of the project (there should be a directory `perf-testfiles`).

1. **Makefile**
2. **spgemm.cpp** - Develop your code for sparse matrix-matrix multiplication in this file. This is your **submission file**. You need to submit the **spgemm.cpp** file.
3. **apsp.cpp** - Develop code to perform the APSP algorithm using your version of SpGEMM. This your **submission file**. You need to submit the **apsp.cpp** files.
4. **distribute.cpp** - Develop code to distribute the nonzeros in a 2D processor grid. This is your **submission file**. You need to submit **distribute.cpp**.
5. **pa3.cpp** - The driver file for your code. You can make changes in the driver file for your own testing purpose, but make sure that the submitted files run with the original 'pa3.cpp' file
6. **functions.h** - This file contains the function definitions and you should not modify the file.
7. **Input files** - You will be given a set of files to test your code with. Each file is designed for a specific subproblem, that is to either run with SpGEMM alone or with APSP. The `testfiles`

directory will contain small inputs to be used for quickly checking correctness. There will be a zip file on canvas for the larger inputs to be used to check performance. See Section 3.4 for more information on the files and the file format.

3.1.1 Instructions

1. After downloading the code files, go through each of these and try to understand the structure.
2. Read over Section 3.3 to to understand the functions being implemented and their requirements.
3. Write your implementation of the functions in `distribute.cpp` and `spgemm.cpp`.
4. In `apsp.cpp`, you only need to change the function/lambda arguments that are passed to SpGEMM in order to ensure that APSP runs correctly (reference the CLRS book for more information).
5. When compiling the code, always make sure to run `module load openmpi` followed by **make**. An executable **pa3** will be created if there are no compilation errors.
6. Request the resources (interactive node) as discussed in the review session (using `salloc`) and use `module load openmpi` to load the MPI library (which will solve the include/command not found errors, if any).
7. You can use `srun -n <num_processors> ./pa3 <type> <input_file>` to run the program.

   `<type>` **options :**
   `spgemm` - to test the SpGEMM code
   `apsp` - for APSP using SpGEMM

8. Use the `autograder.sh` script to test correctness and performance once you feel that your implementation is complete.

3.2 Submission Files

On submission, please only submit the following files in a zip archive:

- **distribute.cpp**
- **spgemm.cpp**
- **apsp.cpp**

These files must compile with the other original project files. **Do not** add files or modify the other project files.

3.3 Function Definitions

1. **distribute.cpp - 2D Distribution Code**

```
void distribute_matrix_2d(int m, int n,
    std::vector<std::pair<std::pair<int, int>, int>> &full_matrix,
    std::vector<std::pair<std::pair<int, int>, int>> &local_matrix,
    int root, MPI_Comm comm_2d);
```

In this function, you will need to distribute the nonzeros of the matrix from the root rank to all processes. This function gives you a communicator that has an associated 2D cartesian topology. Lecture 16 will talk about this more in depth, but this will allow you to use functions such as `MPI_Cart_rank()` and `MPI_Cart_coord()` to determine coordinates of processors within a 2D grid. Remember, you will be running the program on squared number of processors (1, 4, 9, ...), which allows for a simple $\sqrt{p} \times \sqrt{p}$ processor grid. The goal here is to evenly distribute nonzeros to each process such as they each will have to do roughly an equivalent amount of work.

Specifically the argument for this function are as follows:

- `m` - number of rows in the matrix
- `n` - number of columns in the matrix
- `full_matrix` - the complete matrix (only on the root processor)
- `local_matrix` - the local matrix to be filled in on each processor
- `root` - the rank of the root processor (will be 0)
- `comm_2d` - the communicator with a 2D Cartesian topology

2. **spgemm.cpp - Sparse Matrix-Matrix Multiplication (SpGEMM)**

```
void spgemm_2d(int m, int p, int n,
    std::vector<std::pair<std::pair<int,int>, int>> &A,
    std::vector<std::pair<std::pair<int,int>, int>> &B,
    std::vector<std::pair<std::pair<int,int>, int>> &C,
    std::function<int(int,int)>plus, std::function<int(int,int)>times,
    MPI_Comm row_comm, MPI_Comm col_comm);
```

Here, A, B and C being passed to the function will have the values in the form:

`vector<pair<pair<`*row*`,`*col*`>,`*value*`>>`

The other arguments are:

- `m`, `p` and `n` are the integers which represent the dimensions of the matrices. dim(A) = `m` × `p`, dim(B) = `p` × `n` and dim(C) = `m` × `n`.
- A (2D partitioned matrix passed from pa3.cpp)
- B (2D partitioned matrix passed from pa3.cpp)
- **C (expected by pa3.cpp which should be a 2D partitioned matrix)**. You must ensure that it contains the required data towards the end of the function execution.
- `plus`, and `times`, are functions that are to be used to implement the **semiring** matrix multiplication
- `row_comm` and `col_comm` are the communicators to use for all communication. Remember, processors will be decomposed into a logical grid. `row_comm` allows for communication with all processors in the row, and `col_comm` for all processors in the column. **Do not** use `MPI_COMM_WORLD` for communication.

NOTE: Data passed to the spgemm() function has 2D partitioning, as performed by your `distribute_matrix_2d` function above.

3. **apsp.cpp** - APSP using SpGEMM

```
void apsp(int n, std::vector<std::pair<std::pair<int, int>, int>> &graph,
    std::vector<std::pair<std::pair<int, int>, int>> &result,
    MPI_Comm row_comm, MPI_Comm col_Comm)
```

Input to the APSP function will include the following:

- `n` - the number of nodes in the graph
- `graph` - the graph coordinates — this will be 2D distributed using your `distribute_matrix_2d` function
- `result` - the shortest path distances between nodes in the graph

This code is already provided for you. You will need to change only two lines. The APSP function internally makes multiple calls to your SpGEMM function and you need to make sure that the parameters `plus` and `times` are correct to ensure that the algorithm works.

3.4 Input Data

We will be testing your SpGEMM and APSP implementations using sparse matrices from the SuiteSparse Matrix Collection: `https://sparse.tamu.edu/`. We will give you an initial set of input files to test for correctness and performance. You can attempt to generate your own test output if you would like from other sparse matrices, but this is not required.

Input to the program will consist of three different sections:

```
# test_type is either spgemm or apsp
test_type
# First matrix A
0 1 2           # A[0, 1] = 2
2 1 10           # A[2, 1] = 10
.
.
.
33 22 2         # A[33, 22] = 2
----
# Correct graph
0 1 2
.
.
.
33 22 1
```

3.5 Deliverables

Note: Please make **one submission** per team on Gradescope for the code as well as the report.

1. **spgemm.cpp** - Make sure this file runs with the original 'pa3.cpp' file.
2. **apsp.cpp** - Make sure this file runs with the original 'pa3.cpp' file.
3. **distribute.cpp** - Make sure this file runs with the original 'pa3.cpp' file.
4. **Report** - Make sure to list names of all your teammates at the very beginning of your report.

3.6 Grading

- NOTE: We will be running your submissions on a Intel(R) Xeon(R) Gold 6226 CPU @ 2.70GHz (24 cores) node with OpenMPI Version 4.1.5
- The runtime will be on tested on two different inputs (see `PA3 Inputs` under Files on Canvas):
  - `GL7d14.dat` for SpGEMM
  - `G12.dat` for APSP

  Correctness will also be tested on other matrices that are not given.
- Total: **140 points**
- Code - **100 points**

  - Sparse matrix multiplication - **60 points**
    - Correctness - **20 points**
    - Runtime on **36 processors** - **20 points**
      - Runtime **under 10 sec** - 20 points
      - Runtime **under 20 sec** - 10 partial points
    - Runtime on **16 processors** - **20 points**
      - Runtime **under 20 sec** - 20 points
      - Runtime **under 30 sec** - 10 partial points
  - APSP using SpGEMM - **40 points**
    - Correctness - **10 points**
    - Runtime on **36 processors** - **15 points**
      - Runtime **under 30 sec** - 15 points
      - Runtime **under 40 sec** - 10 partial points
    - Runtime on **16 processors** - **15 points**
      - Runtime **under 65 sec** - 15 points
      - Runtime **under 80 sec** - 10 partial points

- Report - **35 points**
  - Explain your implementation of the 2D SUMMA SpGeMM Algorithm - **10 points**
  - Explain how APSP algorithm can be done using matrix multiplication - **5 points**
  - Create performance/scalabiltiy graphs comparing performance on 1, 4, 9, 16, 25, 36 processors for the SpGEMM and APSP algorithms - **20 points**
- Team contract - **5 points**

4 Resources

- What is a `Makefile` and how does it work?: `https://opensource.com/article/18/8/what-how-makefile`
- PACE ICE cluster guide: `https://gatech.service-now.com/home?id=kb_article_view&sysparm_article=KB0042102`.
- **All-Pairs Shortest Paths problem**
  Chapter 23, Introduction to Algorithms (Cormen, Leiserson, Rivest, and Stein).
- **Sparse SUMMA Algorithm**
  Aydin Buluç and John R. Gilbert SIAM Journal on Scientific Computing 2012 34:4, C170-C191. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. DOI: `https://doi.org/10.1137/110848244`

- Datasets:

    1. Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. ACM Transactions on Mathematical Software 38, 1, Article 1 (December 2011), 25 pages. DOI: `https://doi.org/10.1145/2049662.2049663`
    2. Pajek/Tina_DisCog `https://sparse.tamu.edu/Pajek/Tina_DisCog`
    3. SNAP/soc-Slashdot0811 `https://sparse.tamu.edu/SNAP/soc-Slashdot0811`
    4. Gleich/wb-cs-stanford `https://sparse.tamu.edu/Gleich/wb-cs-stanford`