CS 420, Parallel Programming, Spring 2023
MP4

Due Date: May 12th, 2023, at Midnight

1 Overview

The purpose of this assignment is for you to gain experience with **collective MPI** functions and Matrix-vector multiplications.

Note: This MP involves submitting tasks that run on different nodes. We have observed that sometimes, results are not collected across 8 nodes. If this is the case, please try re-submitting the job to the cluster a few times. If the results for 8 nodes is still unavailable, it is okay to ignore the data point for 8 nodes while plotting the required graphs

2 A Matrix-Vector Multiplication in MPI

2.1 Skeleton Programs

Pull the skeleton for MP4 from the `release` repository: `https://gitlab.engr.illinois.edu/sp23-cs420/turnin/<NETID>/mp4.git`

2.2 Matrix-Vector Multiplication

Matrix-vector multiplication is a specific form of matrix-matrix multiplication. The second matrix and the resulting matrix consists of one column, which simplifies the math involved since there is one less dimension to iterate over.

In a distributed context, the vectors also simplifies the communication involved. You will only need to split one array instead of two, and combine/reduce a vector instead of a 2D matrix. The computation is distributed over the various processes, with each process typically receiving entire rows. If each row of the matrix only goes to one process, then each element of the resulting vector is entirely on one process, and eliminates the need to reduce across processes. Figure 1 for an illustration of the flow.

2.3 Your Task

We have provided a skeleton program that handles much of the setup for this assignment, which can be pulled from the `_release` repository. Your task is to implement the communication for splitting the matrix `A` across the processes, duplicating the vector `x` across all the processes, and recombining the resulting vector `b`. You should do all of this inside the function `MPIMatrixMultiplication`.The skeleton creates `kNumPes` processes, and randomly generates both the matrix `A` and vector `x`.

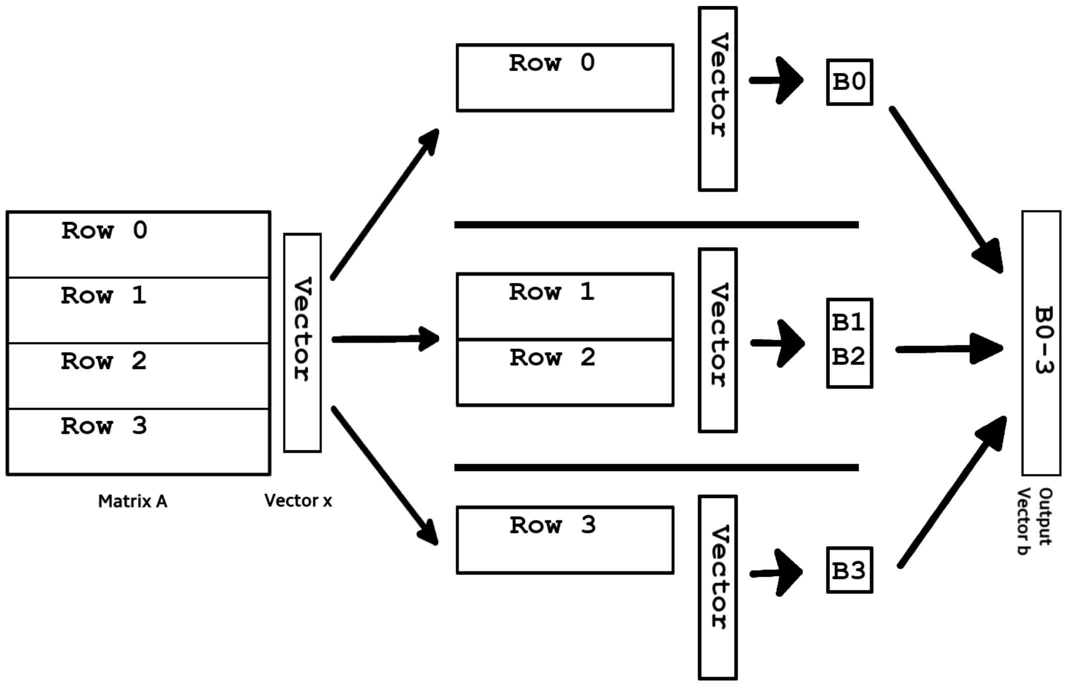

Figure 1: The flow of distributed matrix-vector multiplication.

Only the root process has both the generated matrix `A` and vector `x` mentioned above. The matrix `A` is `kHeight` rows tall and `kWidth` columns wide, and the vector `x` is `kWidth` rows tall and one column wide. There are no restrictions on the number of processes or the height or width of the matrix. The number of processes is not guaranteed to evenly divide the matrix. You are free to distribute the matrix and vector however you want, but each process should receive a roughly equal amount of work.

We have added necessary hints in the `mpimultiply.c` file. The function can be split into 3 sub tasks which have been marked as `TODO`.

- Split the input matrix `A` and distribute it to all the process. You must also send vector `x` to all processes (using collective communication)
- Compute the local result vector `b` at each process
- Recombine the result vector `b` at the root process (rank - 0)

Please make sure not to remove or modify the `timer_start` and `timer_end` functions. Those functions help measure the time that gets aggregated as your results.

?

Question 1: *Strong scaling* refers to how the execution time varies with the number of processes (PEs) for a fixed, overall problem size. **For a single-node run** and an 16384 by 16384 workload, plot *Execution Time vs. Number of processes* for process counts: 1, 2, 4, 8, and 16. Briefly comment on your program's *strong scaling*.

?

Question 2: *Weak scaling* refers to how the execution time varies with the number of processes for a fixed problem size per process. In other words, you increase both the overall workload and the number of processes correspondingly, keeping the workload per process constant. **For a multi-node run**, plot *Execution Time vs. Number of processes* for a fixed workload of 2048*4096, per process, for process counts: 2, 8 and 32. Briefly comment on your program's *weak scaling.* According to you what should be the ideal plot for weak scaling ? Plot the ideal curve in the same graph. Does your solution lie closer to the ideal curve ? If not briefly explain why you think your results are not close to the ideal values.

?

Question 3: For a 16384 by 16384 workload, how did running your program with **multiple nodes** affect its parallel performance? Plot Execution Time vs. Number of processes, for process counts: 1, 2, 4 and 8. The processes run on multiple nodes, that is, the 2 processes runs on 2 nodes, the 4 processes run on 4 nodes and the 8 processes run on 8 nodes. Briefly comment on your program's strong scaling which occurs across **multiple nodes**.

2.4 Skeleton Notes

This MP's skeleton program is rather complex, so it may be worth taking some time to understand it before diving into the assignment. Unlike the other skeleton's we have provided, this MP's verboseness is determined at compile-time using a macro variable, `VERBOSE_LEVEL`. You can specify this to cmake by adding `-DCMAKE_C_FLAGS="-DVERBOSE_LEVEL=X"` to `cmake ..`, where: $X = 0$ produces no output (use this for performance analysis), $X = 1$ prints information for correctness checking, and $X = 2$ prints additional information about the program. Since no communication is necessary in single process runs, these runs can be used as the basis for correctness checking.

The expected usage of the program is `./mpimult <height> <width> [seed]`, specifying the overall width and height of the workload and the seed for matrix/vector generation. The seed is an optional parameter and does not need to be included. If no seed is specified, then a random one will be used. This program **DOES NOT** assume that the workload will always be evenly divisible by the number of processes. You should handle the both the cases where it does and does not evenly divide the matrix.

2.5 Building on the Cluster

The following command is used to load MPI into the cluster's environment:

```
module load intel/18.0
```

Among others, this will add the program `mpiicc` to the `PATH`, used to compile MPI programs on the campus cluster. To build, you can run the provided script:

```
./scripts/compile_script.sh X
```

where the X is the desired verbosity and with the appropriate X value (0, 1, 2). The parameter is optional and can be left out for a default verbosity of 1. Alternatively, we have provided a `cmake` file for you to use to manually build your program. To build, enter your root directory and run:

```
mkdir build
cd build
cmake ..
cmake --build .
```

To specify verbosity, change

```
cmake ..
```

from the given commands to

```
cmake -DCMAKE_C_FLAGS="-DVERBOSE_LEVEL=X" ..
```

2.6 Running MPI Programs on the Cluster

The number of processes to be used is specified via `srun`; for example, the following command will run the program `mpimult` (with appropriate arguments) on 4 processes and one node:

```
srun --mpi=pmi2 --nodes 1 --ntasks 4 ./mpimult 64 64 5
```

It also uses a seed of 5, guaranteeing the same matrix every time. You should not use `mpirun` to execute any of your programs, as this runs on the login nodes. Please use the script `test.sh` found in the `tests` folder. This script will build and test your program. If you are in the root folder, you can run this script with

```
./tests/test.sh
```

Additionally, if a test case fails, we have included a very short description of what the test case is, that is - We denote the matrix dimensions along with the number of processes used for the test case.

Now you will be able to see the output generated by your program at `tests/outputs/output-tcno.txt`. This will enable you to compare your output with the expected result residing at `tests/solutions/sol-tcno.txt`

Additional debug logs can be added to `mpimultiply.c`. For example:

```
#if VERBOSE_LEVEL >= 2
printf("Rank of current process %d", kRank);
#endif
```

Once added please re-compile the program using

```
./scripts/compile_script.sh 2
```

You can then run the program on the cluster by following the instructions at the top of this section.

2.7 Benchmarking on the Cluster

If you are in the base folder, the following command will submit all of the necessary tests.

```
./scripts/submit_batch.sh
```

This will make two folders:

- `results`: containing all of the results from your program
- `batchFiles`: the output and error files from all of the jobs

Inside of the `results` folder, you can find a series of files:

- `strong_X`: result for the strong scaling with `X` process with fixed problem size
- `across_node_weak_X_Y`: result for the weak scaling with `X` process across Y nodes.
- `across_node_strong_X_Y`: result for strong scaling with `X` process across Y nodes

3 Submission Guidelines

3.1 Expectations For This Assignment

The graded portions of this assignment will be your mpimult program and answers to the short answer questions. Please submit the short answers to GradeScope.

3.1.1 Points Breakdown

- $3 * 15 pts.$ — Each Short Answer Question
- $55 pts.$ — Correctness of your MPIMult Program

3.2 Pushing Your Submission

Follow the git commands to commit and push your changes to the remote repo. Ensure that your files are visible on the GitHub web interface, your work will not be graded otherwise. Only the most recent commit before the deadline will be graded.

3.2.1 Files to change

- `mpimultiply.c`