COMP 429/529- Parallel Programming: Assignment 3

Due: 11.59 pm June 5, 2020

Notes: You may discuss the problems with your peers but the submitted work must be your own work. Please submit your report and source code through blackboard. This assignment is worth 20% of your total grade.

Iterative Sparse Matrix-Vector Multiplication (iSpMV)

In this assignment you will implement the sparse matrix-vector multiplication discussed in class (Lecture 23) using MPI. Along with this project description, studying Lectures 22 and 23 might be very helpful for the assignment. Since implementing/debugging parallel codes under message passing can be more time consuming than under threads, give yourself sufficient time to complete the assignment. In addition, this assignment requires you to conduct various performance studies. Please also allow yourself enough time for performance measurements. The performance results will be collected on the KUACC cluster. For instructions about how to compile and run MPI jobs on KUACC, please read the Environment section carefully.

Background

SpMV plays an important role in a variety of scientific and engineering applications. Iterative version of SpMV (iSpMV) is a key operation in many graph-based data mining and machine learning algorithms. With the emerge of big data, the matrices can be so large that the SpMV has to be executed in parallel. In fact, iSpMV is the backbone of Page Ranking algorithm used by Google to rank pages for the search engine. This type of kernels involves multiplication of a matrix having sparsely-located non-zero elements with a vector. An entry in the matrix indicates a link from a page to another. For example, if the matrix entry (i, j) is non-zero, that means there is a link from webpage i to webpage j. Since there is no link from every page to every other page, matrix contains a lot of zero entries.

An SpMV is typically performed in the following form, $Ax = y$, such that A is a sparse matrix of size MxN, x is a vector of size N, and y is a vector of size M. In the iterative version of SpMV, input vector is also the output vector, so we are solving $Ax = x$ in a time loop.

There are different types of storage formats available for sparse matrices. Since storing zeros wastes a lot of memory space, we only store the non-zero elements and use metadata to identify the location of non-zero entires in the matrix. The most common format to store sparse matrices is compressed storage row (CSR), which is shown in below figure. The *val* array contains matrix entries, the *ind* array refers to the column indices, and the *ptr* array refers to the rows.

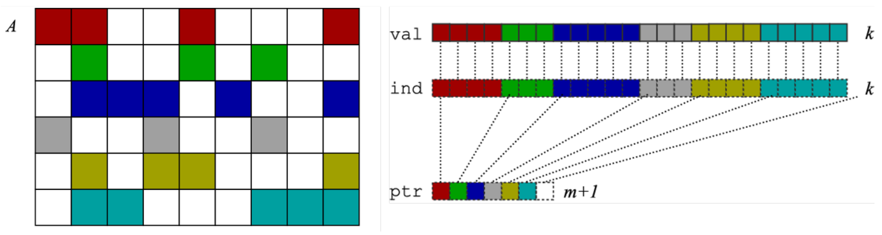

Figure 1: CSR format - an example sparse matrix

Serial Code

We are providing you with a working serial program that implements iSpMV. The provided code reads from a file containing sparse matrix from the Florida sparse matrix collection (source: https://sparse.tamu.edu/). It then multiplies the read matrix with a vector iteratively in a time steps loop.

Running The Code

- The serial code that we have provided includes a Makefile that is needed to compile the serial code. To compile and run the code, execute the following commands.

```
To compile, type
      make
To clean the objects and executables
      make clean
Example run:
       ./build/spmv <input-file>.mtx <number-of-time-steps>
```

- You could use the campus clusters or your local machines to develop and test your implementations. If you want to work locally, you need to install an MPI library.

Parallel Code

In this assignment, you are expected to parallelize the given serial SpMV code in three different tasks, one of them being optional. In the first task, you have to implement row-wise parallelization for the matrix-vector multiplication (Lecture 23 page 33). The second task, you are asked to introduce multithreaded parallelism with OpenMP on top of MPI for the row-wise partitioning. The third task is optional and you are expected to implement load-balanced partitioning to the same problem (Lecture 23 page 31).

Part I

Row-wise Partitioning and Parallelization with MPI: In this part of the assignment, you have to parallelize the given SpMV serial code by using only MPI. In parallelizing the code, the sparse matrix is to be partitioned by dividing it by rows. This parallelization strategy is illustrated in Figure 2.

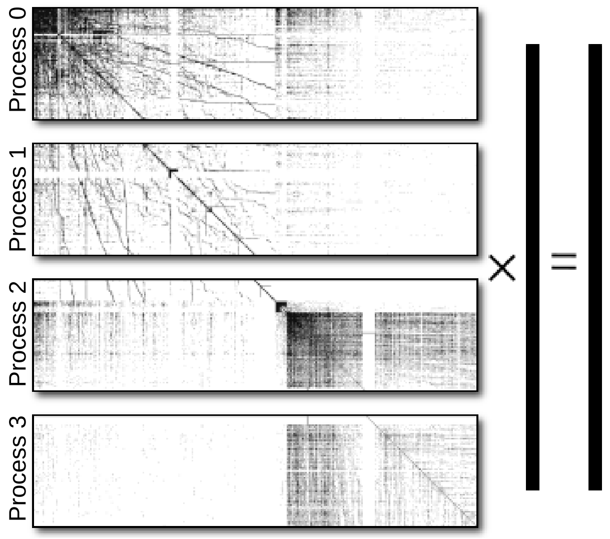

Figure 2: Row-wise parallelization of SpMV with MPI

As shown in the figure, a sparse matrix of size $M * N$ is partitioned across P processes, so that each process has a matrix chunk with size $(M/P) * N$, while the multiplier vector of size N is the same for all processes. Each of these processes performs matrix-vector multiplication on its matrix chunk. After multiplying all rows in the matrix chunk with the vector, each process will have its own vector of results with size (M/P). To generate the final output, these vectors of results are collected from all processes and appended to produce the final vector of size M in process 0. In the iterative version of iSpMV, the output vector should be distributed to all processors to update their vector x for the next iteration.

Part II

Part I with MPI+OpenMP: In this part of the assignment, you have to introduce multithreaded parallelism to each process in the row-wise parallelization code from the first task. You are expected to implement the multithreaded parallelism using OpenMP. Parallelize across the rows assigned to a single process.

Part III - Optional

Load Balancing - Bonus: The row-wise partitioning partitions the rows equally among processes. However, depending on the sparsity pattern of the matrix, the number of non-zeros that each process will compute may not be balanced. In other words, equal number of rows per process does not guarantee that each process will be responsible for the same amount of workload. Instead, in this part, you will still partition row-wise but considering the number of non-zeros per process. That means each process may be assigned with different number of rows to compute but same or similar amount of non-zeros. For simplicity, do not divide a row between two processes but assign a row to a single process only. You do not need to implement OpenMP version of this.

Experimental Study

- You will notice that the serial program we provided requires you to input the number of time steps as the second command line parameter.
- Conduct a strong scaling study on a single node for input file Cube_Coup_dt6.mtx and Flan_1565.mtx with 20 time steps. Observe the running time as you successively double the number of processes, starting at P=1 core and ending at 16 cores, while keeping number of time steps fixed. Compare single processor performance of your parallel MPI code against the performance of the original serial code. Perform this study on Part 1 and Part 3 (if you choose to implement it). You can download the input matrices from the following links:

  https://sparse.tamu.edu/MM/Janna/Flan_1565.tar.gz

  https://sparse.tamu.edu/MM/Janna/Cube_Coup_dt6.tar.gz
- Repeat single node performance study on MPI+OpenMP parallel code (the code of the second scheme) by using 1, 2, 4, 8, and 16 OpenMP threads per process (thus 16, 8, 4, 2, and 1 MPI processes). In a figure, compare the performance numbers with the strong scaling with MPI-only. Which combination of MPI+OpenMP is the fastest?

  Note: For mixing OpenMP and MPI, you will request the same number of total processors and then vary OpenMP threads and MPI processes to span over those processors. For instance, when you request 16 processors, you can use them to spawn 16 processes, 1 OpenMP thread, 8 MPI processes and 2 OpenMP threads and so on.

Grading

Part I: 70 pts (50 pts implementation + 20 pts performance study)
Part II: 20 pts (10 pts implementation + 10 pts performance study)
Part III: 20 pts (optional - bonus)
Report: 10 pts.
You may lose points both for correctness (e.g. deadlock) and performance bugs (e.g. unnecessary synch. point) in your implementation.

Submission

- Document your work in a well-written report which discusses your findings.
- Your report should present a clear evaluation of the design of your code, including bottlenecks of the implementation, and describe any special coding or tuned parameters.

- We have provided a timer to allow you to measure the running time of your serial code. For the MPI code, you could use MPI_Wtime() function to collect the execution time of your program (source: https://www.mcs.anl.gov/research/projects/mpi/tutorial/gropp/node139.html). Plot both execution time and speedup of your parallel code based on the execution time recorded by this function.
- Submit both the report and source code electronically through blackboard.
- Please create a parent folder named after your username(s). Your parent folder should include a report in pdf and three subdirectories: one for each task. Include all the necessary files to compile your code. Be sure to delete all objects and executable files before creating a zip file.
- GOOD LUCK.

Environment

The environment information will be made available soon.

References

https://en.wikipedia.org/wiki/Sparse_matrix-vector_multiplication