Program 3: K-Nearest Neighbor Classification (MPI + OpenMP)

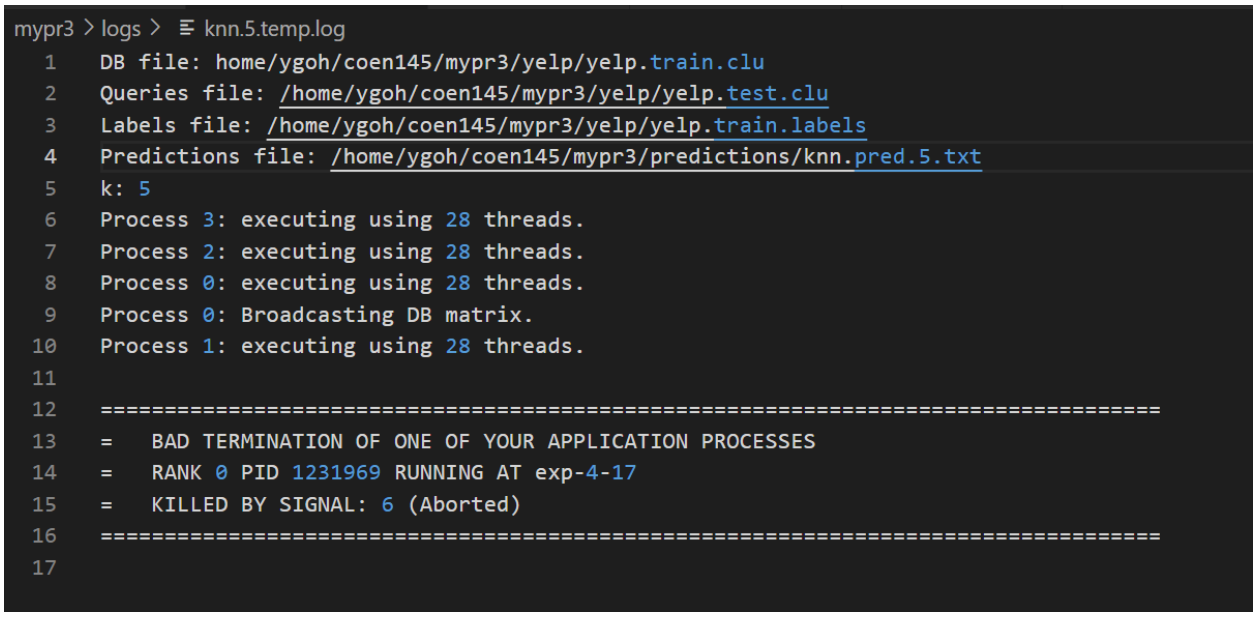
a. A short description of how you went about parallelizing the k-nearest neighbor classifier. One thing that you should be sure to include is how you decided to perform the sparse matrix multiplication and how that decision influenced the communication that needed to occur between the different MPI processes.

This function, 'my_find_top_k_neighbors()', is designed to find the top k neighbors for a given 'subQuery' within the 'dbs' matrix using dot products. It has been parallelized using OpenMP to improve performance by distributing the workload across multiple threads. The outer loop which iterates over each column in the 'dbs' matrix, has been parallelized using OpenMP directives. The '#pragma omp parallel for' directive indicates that the iterations of the outer loop can be executed in parallel. At first I thought about parallelizing all the for loops by adding the 'collapse' clause. However, this does not work because there is a branch dependency at the if statement in the third for loop. Inside the parallelized loop, each thread accumulates the dot product independently in a private variable (dotProduct). This avoids data races that could occur if multiple threads wrote to a shared variable simultaneously. By parallelizing the loop over the columns of the 'dbs' matrix, the function efficiently utilizes multiple threads to compute dot products, leading to improved performance for large matrices. Additionally, careful consideration has been given to avoid potential data races and ensure the correctness of the results.

Each process contains its own set of subqueries distributed by 'read_and_bcase_csr()'. Then I further break down the subqueries by iterating through the sets of subqueries serially in each process and find the k nearest neighbor of each iteration. In my program, it is not necessary for the processes to communicate with each other while computing for the k nearest neighbors and making the predictions, but after predictions have been computed, I utilized MPI_Gather() to accumulate the predicted labels of all queries.

b. Brief Analysis of Results

For this lab we used Expanse to test our program. As mentioned by the Professor, MPI and OpenMP hybrid codes can only run on particular versions of MPI that have proper support. After debugging common programming syntax errors, running the shell script returned a log file with errors detailed below:



After some brief debugging from tracing the output, I found that the program terminates when it tries to read and broadcast the sparse matrices from the .clu files to each process. Due to poor time management, I was unable to fix this bug and get satisfactory results. In the future, I will continue to work through the program till it is fully functional. For example, since the 'read_and_bcast_csr()' function was given to us and I did not edit the given stub code, there is most likely an issue in the program residing somewhere else. The text in lines 13 to 15 of the output "knn.5.temp.log" file indicates that the MPI application encountered a fatal error. In this case, the process with rank 0 and process ID 1231969 was terminated due to receiving a signal 6, which corresponds to the "Aborted" signal. Based on personal experience with past MPI programs, there is likely invalid MPI usage. Because in previous MPI programs, I was allocating memory with uninitialized variables that the process had not even received yet. So, a possible solution is to add print statements to log output information about the program's execution flow, variable values, and synchronization points. Another way I could go about debugging is to focus on a specific computation in a process. Specifically, computations that involve more MPI functions. For example, there may be errors occurring in the area where I am using 'MPI_Allreduce()', 'MPI_Bcast()'.

The impact of the number of nodes on the runtime of a hybrid OpenMP and MPI program depends on various factors, including the nature of the application, the workload distribution, communication patterns, and the efficiency of the parallelization strategies. For example, in my program I have large time consuming sequential parts because there are certain dependencies that prevent me from being able to parallelize them. A good example of a sequential part of my program is the part where I iterate through each row of the subquery. According to Amdahl's Law, the speedup of a program is limited by the sequential portion of the code. Thus, since I have a large sequential portion, adding more nodes may have diminishing returns. Since this is partially an MPI program, increasing the number of nodes may cause a bottleneck situation if the

communication intensity is high. Adding more nodes may not be beneficial because of increased communication overhead/times.