K-Nearest Neighbor Classification (MPI+OpenMP)

Introduction

The purpose of this assignment is to become familiar with MPI and OpenMP by implementing sparse matrix-matrix multiplication in a hybrid distributed computing environment and using the implemented routine to solve a k-nearest neighbor classification problem.

K-Nearest Neighbor Classification

Given a set of *m* objects described by *n* features, we represent each object as a point in the Euclidean space by a vector of size *n* containing numeric value representations of those features. The matrix **D** of size *m* x *n* thus contains the vectors for all the objects in our database. For each object in **D**, we also have an additional feature, called the class or label, that tells us something important about the objects. Additionally, we have another set of *q* objects, described by the same types of features, except with an unknown class. The query objects can be stored in a matrix **Q**. The goal of the classification task is to assign a predicted class label to each of the objects in **Q**.

For a given query **q**, the k-nearest neighbor (KNN) classifier works by finding the *k* nearest neighbors of **q** in the database **D** and assigning to it the predicted class as a function of the class labels of the found nearest neighbors. The classifier has a number of meta-parameters, such as *k*, the number of neighbors we should choose, but also the choice of proximity function that describes how neighborly objects are to each other (e.g., Euclidean distance or cosine similarity), and the choice of label aggregation strategy for the found neighbors. For the purpose of this program, we will use cosine similarity as the proximity function and weighted average class assignment (the code for which will be given). Students will need to tune the *k* parameter to achieve the best results they can for the given problem.

Cosine similarity is defined as the cosine of the angle between two vectors in Euclidean space. If the angle is small, the vectors are very close to each other and thus similar. For two vectors **q** and **c**, it is computed as the dot-product of the two vectors divided by the product of their vector lengths (L2-norms). However, if the vector lengths are all changed to be 1, the cosine similarity remains the same (since the angles of the vectors did not change) but the cosine similarity reduces to simply the dot-product of the two vectors. We can make vectors unit-norm (change their lengths to 1) by dividing the vector by their L2-norm, a process called normalization (code will be provided for this). Therefore, after normalizing the vectors in **Q** and **D**, the process of finding the nearest neighbors for all vectors in **Q** reduces to the problem of multiplying **Q** and the transpose of **D** ($\mathbf{C} = \mathbf{QD}^T$), and reducing the list of identified results in **C** to the *k* with the largest values.

Distributed sparse matrix multiplication ($\mathbf{C} = \mathbf{QD}^T$)

We are assuming a hybrid execution environment with *p* nodes, each with at least *t* cores. To allow the solution to scale to the largest problem size, we will assume that the matrices **Q** and **D** may not fit on one node and will thus split them across all *p* nodes. Given the irregular nature of the sparse matrices, we will split each such that **each process receives roughly the same number of non-zero values of each**

matrix. Since there is only one value for each database object, the class labels for all the database objects can be sent to all MPI processes.

After distributing the matrices, each node is responsible for finding the KNN for the objects it has been given in its section of **Q** and predicting their class label. After communicating with other processes to receive the data it needs to complete the task, the MPI process should use OpenMP threads to parallelize the task of finding the nearest neighbors. Finally, all predicted class labels for the queries should be communicated to the root process and it will store them in a file, one per line.

Efficiency (runtime speed) is just as important as effectiveness (getting the highest accuracy) when solving this problem. You should implement the best strategy you can think of to solve the problem. The top-2 solutions with the highest efficiency and accuracy within 2% of the best accuracy achieved by Prof. Anastasiu's solution will receive a 1% final grade boost as extra credit. Runtime will be measured as the total execution time of the program when executed using 4 nodes in the `compute partition`, each using 28 threads, on Expanse.

Problem

You are given a training set containing 1,881,627 reviews (yelp.train.clu) and a test set containing 100,000 reviews (yelp.test.clu) for businesses on Yelp in 5 categories. Given the labels associated with the training set descriptions (yelp.train.labels), you must predict the labels for the test set. The test set labels are held out. Your task is to find the predictions for each of the test set objects. The dataset can be found on WAVE, in the /WAVE/projects/COEN-145-Fa23/datasets/yelp/ directory.

Testing

Test your code on Expanse or WAVE using 1–4 nodes and 28 threads for each node. Submit the predictions for your chosen value for *k* to https://clp.engr.scu.edu/ to find out the performance of your method.

What you need to turn in

1. The source code of your program.
2. Log files showing the execution of your code on Expanse or WAVE.
3. A short report that includes the following:
    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.
    b. Timing results for your parallel executions. These results should be reported in a strong scaling chart reporting the speedup at different numbers of nodes vs the execution on a single node.
    c. A brief analysis of your results. Some things to consider might be:
        i. How does the number of nodes affect the runtime? Why do you think that is?
        ii. How well does your program scale, i.e., if you keep increasing the number of nodes, do you think the performance will keep increasing at the same rate?

Submission specifications

- A makefile must be provided to compile and generate the executable file.
- The executable file should be named 'knn'.
- All files (code + report) MUST be in a single directory and the directory's name MUST be your university student ID. Your submission directory MUST include at least the following files (other auxiliary files may also be included):

```
<Student ID>/knn.cpp
<Student ID>/Makefile
<Student ID>/report.pdf
```

- Submission MUST be in .tar.gz
- The following sequence of commands should work on your submission file:

```
tar xzvf <Student ID>.tar.gz
cd <Student ID>
make
ls -ld knn
```

This ensures that your submission is packaged correctly, your directory is named correctly, your makefile works correctly, and your output executable files are named correctly. If any of these does not work, modify it so that you do not lose points. I can answer questions about correctly formatting your submission BEFORE the assignment is due. Do not expect questions to be answered the night it is due.

Evaluation criteria

The goal for this assignment is for you to become familiar with the OpenMP API and develop an efficient parallel program. As such, the following things will be evaluated:

1. follows the assignment directions,
2. solve the problem correctly,
3. do so in parallel, following the prescribed data decomposition,
4. achieve speedup (5 / 10 points will be reserved for this criterion).