K-NN algorithm implementation in Message Passage Interface

The goal of this project was to create a k nearest neighbors algorithm of each point in corpus X, and the implement it in MPI (Message Passage Interface).

My Sequential Implementation

The algorithm that I created works as follow:

- For each element in the corpus X creates 2 arrays that contain the distance to the neighbors and the index of the neighbors.

- For the first k element of the dataset, it inserts the element one by one and sorts them by pushing the rest of the elements to the right. This way when the first k elements are inserted into the array, then we only need to check the last element to see if the distance is smaller than the biggest distance in the k-NN.

- If the distance is indeed smaller, we check where we need insert the current distance and index to the k-NN array and then push all the other elements to the right.

- We continue for all elements in query data points.

- We save the distances and indices in a structure that we created

MPI Implementation

The parallel implementation follows the same strategy as the previous, with the only difference that each process p_i will have a standard "piece" of X, X_i and a piece of the total points, Y_i.

The query points X_i will be the same to each process throughout the whole runtime of the algorithm, while the data points Y_i change through a ring process.

The algorithm works as follow:

- Before the algorithm starts working, process 0 will separate and send data to each other process and keep the remaining for itself.

- The second step is the same as the sequential. We create the arrays for distances and indices, fill the first k neighbors and then check whether the next element is less than the last element in the distances array. We continue for all elements in each Y_i

- After the all the elements are check in the current Y_i, we need to rotate the data points Y_i in a ring. So, each process sends its Y to the next, and the last send to the first

- Because the first k neighbors in each process have been filled, we don't need to repeat the first step and we just check the last element of the distances array.

- We repeat the two previous steps until all points X have been checked against all points Y (total rotations will be: number of processes -1).

- We send all data for the indices and distances back to process 0 and save them at the structure.

MPI send and Receive data

Most of the times the number of elements in our data won't be divisible exactly by the number of processes. To solve this problem, I made all the arrays have total elements equivalent to: total_element/num_of_Processes but with Euclidean division. All remainder points were going to process 0.

Of course, that created the problem that each process would be able to know how much data it was supposed to receive. For this I created two variables: size, previous_size.

- Size: the size after the ring tranfer_i.
- Previous_size: the size before the ring transfer.

So, each process send its current size to the next process, saves the current size to previous_size and receives the new size.

Now the number of data that each process will receive is size and the number it will send is Previous_size.

The data will be moved between processes with a ring strategy, as mentioned above.

The ring will work as follow:

- It will be separated to two parts: First the processes divisible by 2 and not.
- Those divisible will first send data and then receive, and those not the other way

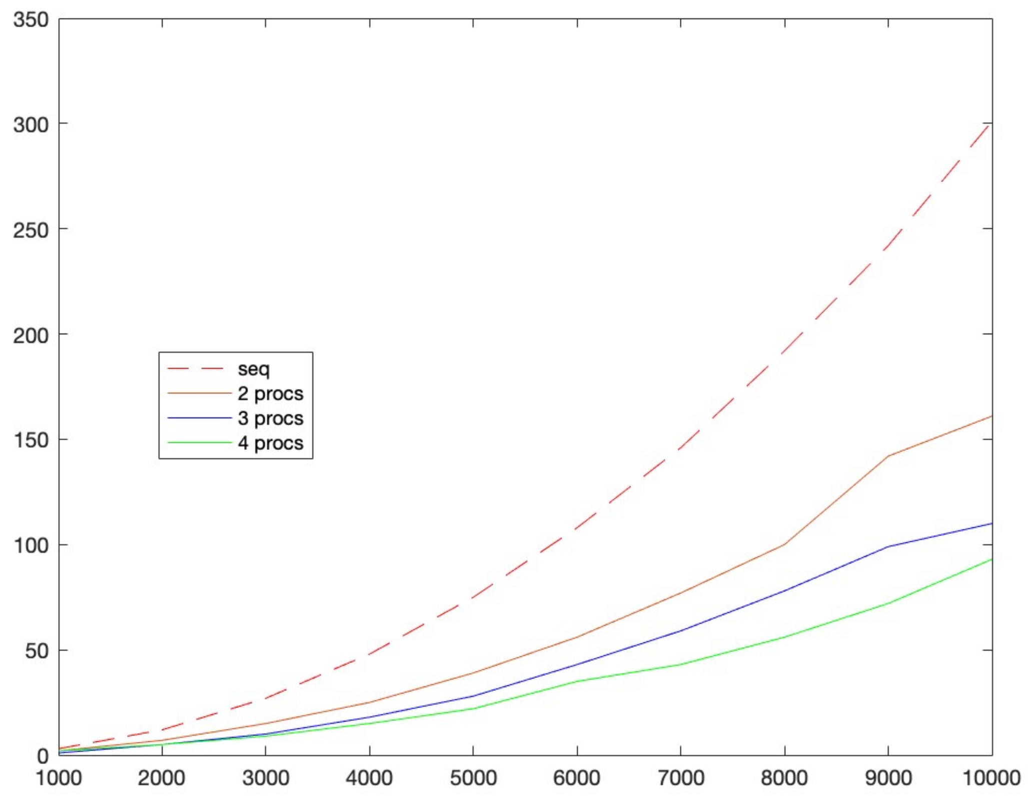
This way we won't have data conflicts.

Results

The following graph is runtimes of the algorithm running the MNIST dataset for different sizes and dimension of 784. I used the following guide to load the dataset https://github.com/takafumihoriuchi/MNIST_for_C

I run the algorithm with different number of processes to check for scalability.

The graph below shows the time in seconds(y-axis) to number of elements(x-axis)



As we can clearly see the more the processes the faster the runtime of the algorithm, with runtimes close to the division of sequential with the number of processes used. This clearly shows the scalability of the interface.

Notes

In order to run the algorithm, you need to install the required MPI requirements.
Compile with *mpicc* and run with *mpirun.*

The algorithm run locally. When run locally the maximum number of processes you can get is the number of your system processors. Because the algorithm was run locally it’s runtimes may be different from runtimes in true dynamic memory systems.

In the algorithm I create a regular 3 dimensioned grid 10x10x10 for testing for correctness.

Everything can only be changed through the code. (dimensions, number of elements etc.)

The algorithm is created to take different query points and data points. (X and Y respectively)

All my source code can be found in my GitHub: https://github.com/themis1234/K-NN-mpi

Arvanitis Themistoklis.
10118