

# PARALLEL PROGRAMMING PROJECT REPORT

K-NEAREST NEIGHBOURS USING MPI AND CUDA

**Soumya Sinha 160905346**

**Divij Agarwal 160905376**

**Neel Dani 160905396**

Department of Computer Science and Engineering

Manipal Institute of Technology

April 11, 2019

# Contents

<b>1</b>	<b>Abstract and Motivation</b>	<b>1</b>
<b>2</b>	<b>Objective</b>	<b>1</b>
<b>3</b>	<b>Introduction</b>	<b>1</b>
<b>4</b>	<b>Methodology</b>	<b>1</b>
4.1	Serial code . . . . .	1
4.2	Pre-processing the data . . . . .	6
4.2.1	Pre-processing using Python . . . . .	6
4.3	Methodology for MPI . . . . .	7
4.4	Methodology for CUDA . . . . .	7
<b>5</b>	<b>MPI Code</b>	<b>8</b>
5.1	Helper code for MPI . . . . .	12
5.2	Mergesort for MPI . . . . .	13
<b>6</b>	<b>CUDA Code</b>	<b>15</b>
<b>7</b>	<b>Result</b>	<b>21</b>
7.1	Output for MPI . . . . .	21
7.2	Output for CUDA . . . . .	21
<b>8</b>	<b>Limitations and possible improvements</b>	<b>21</b>
<b>9</b>	<b>Conclusion</b>	<b>22</b>
<b>10</b>	<b>References</b>	<b>22</b>

# 1 Abstract and Motivation

The k-NN algorithm is considered as one of the simplest machine learning algorithms. However, it is computationally expensive especially when the size of the training set becomes large which would cause the classification task to become very slow. Several attempts have been made to parallelize k-NN on the GPU by taking advantage of the natural parallel architecture of a GPU. Even though the problem is well-studied in serial environment, there is incentive in implementing it in a parallel fashion as it can provide considerable speed-up for large datasets.

## 2 Objective

To implement the k-NN algorithm in a parallel fashion using MPI and CUDA, and observe if any speed-up is attained.

## 3 Introduction

The k-nearest neighbors algorithm (k-NN) is a non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space. In k-NN classification, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). k-NN is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until classification. The neighbors are taken from a set of objects for which the class (for k-NN classification) or the object property value (for k-NN regression) is known. This can be thought of as the training set for the algorithm, though no explicit training step is required. This method has seen wide application in many domains, such as in recommendation systems, semantic searching, and anomaly detection.

## 4 Methodology

The gist of the kNN method is, for each classification query, to:

1. Compute a distance value between the item to be classified and every item in the training data-set.
2. Pick the k closest data points (the items with the k lowest distances).
3. Conduct a majority vote among those data points the dominating classification in that pool is decided as the final classification.

There are two important decisions that must be made before making classifications. One is the value of k that will be used and the other is the distance metric that will be used. Here, we have used Euclidean distance.

### 4.1 Serial code

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
```

```

#include <math.h>

#define LABELS 10

typedef struct
{
    float distance;
    int type;
} s_distance;

int sort_d(s_distance *arr, int elements);
float **parse_file(const char *filename, int *lines, int *features);
float euclidean_distance(float *u, float *v, int size);
void zero_array(int *arr, int size);
void print_matrix(int **confusion_matrix);

int main(int argc, char const* argv[])
{
    int training_lines, training_features;
    int testing_lines, testing_features;
    int k = atoi(argv[3]);

    float **training = parse_file(argv[1], &training_lines,
    &training_features);
    float **testing = parse_file(argv[2], &testing_lines,
    &testing_features);

    int i, j;

    s_distance **testing_distances =
    (s_distance **) malloc(sizeof(s_distance*) * testing_lines);

    #pragma omp parallel for shared(testing_distances) private(i, j)
    for (i = 0; i < testing_lines; i++)
    {
        testing_distances[i] = (s_distance *) malloc(sizeof(s_distance)
        * training_lines);
        for (j = 0; j < training_lines; j++)
        {
            testing_distances[i][j].distance = euclidean_distance
            (testing[i], training[j], training_features);
            testing_distances[i][j].type = training[j][training_features];
        }
    }

    #pragma omp barrier

    #pragma omp parallel for shared(testing_distances) private(i)

```

```

    for (i = 0; i < testing_lines; i++)
    {
        if (!sort_d(testing_distances[i], training_lines))
        {
            perror("SORT FAILED");
            exit(1);
        }
    }

#pragma omp barrier

    int **confusion_matrix = (int **) malloc(sizeof(int *) * training_lines);
    int *counts = (int *) malloc(sizeof(int) * training_lines);
    int errors = 0;

    for (i = 0; i < LABELS; i++)
    {
        confusion_matrix[i] = (int *) malloc(sizeof(int) * LABELS);
        zero_array(confusion_matrix[i], LABELS);
    }

    for (i = 0; i < testing_lines; i++)
    {
        zero_array(counts, training_lines);

        int max = 0;
        for (j = 0; j < k; j++)
        {
            counts[testing_distances[i][j].type]++;
            if (counts[testing_distances[i][j].type] > counts[max])
                max = testing_distances[i][j].type;
        }

        int should = (int) testing[i][training_features];

        if (max != should)
        {
            errors++;
        }
        confusion_matrix[should][max]++;
    }

    printf("hit: %d (%.2f%%); ", testing_lines - errors, 100 -
        ((float) errors / testing_lines) * 100);
    printf("miss: %d (%.2f%%)\n", errors,
        ((float) errors / testing_lines) * 100);
    return 0;
}

```

```

void zero_array(int *arr, int size)
{
    int i;

    #pragma omp parallel for private(i) shared(arr)
    for (i = 0; i < size; i++)
    {
        arr[i] = 0;
    }
}

float **parse_file(const char *filename, int *lines, int *features)
{
    FILE *file = fopen(filename, "r");

    fscanf(file, "%d %d", lines, features);

    float **db = (float **) malloc(sizeof(float *) * *lines);

    int i, j;

    for (i = 0; i < *lines; i++)
    {
        db[i] = (float *) malloc(sizeof(float) * *features);

        for (j = 0; j <= *features; j++)
        {
            fscanf(file, "%f", db[i] + j);
        }
    }
    return db;
}

void print_matrix(int **confusion_matrix)
{
    int i, j;

    printf ("\tSHOULD X RESULT\n");

    for (i = 0; i < LABELS; i++)
    {
        for (j = 0; j < LABELS; j++)
        {
            printf ("%02d ", confusion_matrix[i][j]);
        }
        printf ("\n");
    }
}

```

```

}

float euclidean_distance(float *u, float *v, int size)
{
    int i;
    float sum = 0;
    float aux;

    for (i = 0; i < size; i++)
    {
        aux = v[i] - u[i];
        sum += aux * aux;
    }

    return sqrt(sum);
}

int sort_d(s_distance *arr, int elements)
{
    #define MAX_LEVELS 1000

    int beg[MAX_LEVELS], end[MAX_LEVELS], i = 0, L, R;
    s_distance piv;

    beg[0] = 0;
    end[0] = elements;

    while (i >= 0)
    {
        L = beg[i];
        R = end[i] - 1;

        if (L < R)
        {
            piv = arr[L];
            if (i == MAX_LEVELS-1)
                return 0;

            while (L < R)
            {
                while (arr[R].distance >= piv.distance && L < R)
                    R--;
                if (L < R)
                    arr[L++] = arr[R];

                while (arr[L].distance <= piv.distance && L < R)
                    L++;
                if (L < R)

```

```

        arr[R--] = arr[L];
    }

    arr[L] = piv;
    beg[i + 1] = L + 1;
    end[i + 1] = end[i];
    end[i++] = L;
}

else
{
    i--;
}

    return 1;
}

```

## 4.2 Pre-processing the data

The dataset used is the "Iris" dataset, which is very widely used in pattern recognition literature. The dataset contains 3 classes of 50 instances each, where each class refers to a type of iris plant. Attribute information:

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm
5. class: Iris Setosa, Iris Versicolour, Iris Virginica

By implementing the k-NN algorithm for this dataset, we are aiming to predict the class of Iris plant based on the first 4 characteristics given in the query.

The data has been pre-processed using python scripts. .

### 4.2.1 Pre-processing using Python

The script splits the dataset into test and training data. It also encodes the classes, which have been given as strings in the data as numbers.

```

import numpy as np
import pandas as pd
from sklearn.cross_validation import train_test_split
from sklearn.preprocessing import LabelEncoder
df = pd.read_csv("Iris.csv");
data = df.values
features = data[:, 1:-1]
labels = data[:, -1]
print(features.shape)
print(labels.shape)

```



```

enc = LabelEncoder()
labels = enc.fit_transform(labels)
#print(features)
print(labels)
X_train, X_test, y_train, y_test = train_test_split(features, labels,
test_size=0.1)
y_train = y_train.reshape(y_train.shape[0], 1)
y_test = y_test.reshape(y_test.shape[0], 1)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
np.savetxt("X_train.csv", X_train, delimiter = ',')
np.savetxt("X_test.csv", X_test, delimiter = ',')
np.savetxt("y_train.csv", y_train, fmt = "%s", delimiter = ',')
np.savetxt("y_test.csv", y_test, fmt = "%s", delimiter = ',')

```

### 4.3 Methodology for MPI

1. The test and training data, and the corresponding classes are loaded from CSVs and read into arrays.
2. The training data is split equally among all available processes.
3. The test query is sent to all processes and the Euclidean distance is calculated in parallel and stored in an array.
4. The array is then sorted using a parallel implementation of mergesort, and simultaneously, the array containing the corresponding classes is sorted.
5. Based on the decided value of k, the top k items from the sorted array and classes are chosen to make the classification. The probability of the classification is also calculated.
6. This is repeated for all test data. At the end, the classification for each test query is obtained with the associated probability.
7. The time taken to obtain the classifications is calculated and displayed.

### 4.4 Methodology for CUDA

1. The test and training data, and the corresponding classes are loaded from CSVs and read into arrays.
2. For the kernels, 1D grid of 1D blocks are used. The number of threads created in each block is fixed, and based on that the number of blocks required are determined.
3. The first kernel is used to calculate the Euclidean distance from the test point to every training point. In the kernel, one thread is used to access each row in the array for training data and then calculate the distance to the test point and store it in an array.
4. The second kernel is used to sort the distance using selection sort, and arrange the classes appropriately. Here also, one thread is used to access one element in the distance matrix and class matrix.

5. Based on the decided value of  $k$ , the top  $k$  items from the sorted array and classes are chosen to make the classification. The probability of the classification is also calculated.
6. This is repeated for all test data. At the end, the classification for each test query is obtained with the associated probability.

## 5 MPI Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <string.h>
#include "helper.h"
#include "mergeSort.h"

#define NTRAIN 135
#define NTEST 15
#define NFEATURES 4
#define NCLASSES 3

#define SEPAL_LENGTH 0
#define SEPAL_WIDTH 1
#define PETAL_LENGTH 2
#define PETAL_WIDTH 3

char class[NCLASSES][25] = {"Iris-setosa", "Iris-versicolor", "Iris-virginica"};

void mpiInitialise(int *size, int *rank)
{
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, rank);
    MPI_Comm_size(MPI_COMM_WORLD, size);
}

float *initFeatures(char path[])
{
    int index = 0;
    FILE *f = NULL;
    float *mat = NULL;

    mat = getFloatMat(NTRAIN, NFEATURES);

    f = fopen(path, "r");
    checkFile(f);

    while (fscanf(f, "%f%c", &mat[index]) == 1)
        index++;
}
```

```

        fclose(f);
        return mat;
    }

float *initLabels(char path[])
{
    int index = 0;
    FILE *f = NULL;
    float *mat = NULL;

    mat = getFloatMat(NTRAIN, 1);

    f = fopen(path, "r");
    checkFile(f);

    while (fscanf(f, "%f%c", &mat[index]) == 1)
        index++;

    fclose(f);
    return mat;
}

int predict(float *distance, float *labels, int k, int topn)
{
    float* neighborCount = getFloatMat(NCLASSES, 1);
    float* probability = getFloatMat(NCLASSES, 1);

    int i;

    for(i=0; i<k; i++)
        neighborCount[(int)labels[i]]++;

    for(i=0; i<NCLASSES; i++)
        probability[i] = neighborCount[i]*1.0/(float)k*1.0;

    int predicted_class = (int)getMax(neighborCount, NCLASSES);

    for(i=0; i<topn; i++)
        printf("%s\t%f\n\n", class[i], probability[i]);

    free(neighborCount);
    free(probability);

    return predicted_class;
}

void calcDistance(int ndata_per_process, float *pdistance,

```

```

float *pdata, float *x)
{
    int index = 0, i, j;
    for(i=0; i<ndata_per_process; i=i+NFEATURES)
    {
        pdistance[index] = 0.0;

        for(j=0; j<NFEATURES; j++)
            pdistance[index] = pdistance[index] +
                (pdata[i+j]-x[j])*(pdata[i+j]-x[j]);

        index++;
    }
}

void fit(float *X_train, float *y_train, float *X_test, float *y_test,
int rank, int size)
{
    int i, j;
    int ndata_per_process, nrows_per_process;
    float *pdata, *distance, *pdistance;
    float *plabels;
    float *labels;

    if (NTRAIN % size != 0)
    {
        printf("Not divisible\n");
        exit(0);
    }

    nrows_per_process = NTRAIN/size;
    ndata_per_process = nrows_per_process*NFEATURES;

    pdata = getFloatMat(ndata_per_process, 1);
    pdistance = getFloatMat(nrows_per_process, 1);
    distance = getFloatMat(NTRAIN, 1);

    plabels = getFloatMat(nrows_per_process, 1);
    labels = getFloatMat(NTRAIN, 1);

    MPI_Scatter(X_train, ndata_per_process, MPI_FLOAT, pdata,
    ndata_per_process,
    MPI_FLOAT, 0, MPI_COMM_WORLD);

    float *x = getFloatMat(NFEATURES, 1);

    for (i=0; i<NTEST; i=i+1)
    {

```

```

        MPI_Scatter(y_train, nrows_per_process, MPI_FLOAT, plabels,
nrows_per_process,
MPI_FLOAT, 0, MPI_COMM_WORLD);

        for(j=0; j<NFEATURES; j++)
            x[j] = X_test[i*NFEATURES+j];

        calcDistance(ndata_per_process, pdistance, pdata, x);

        mergeSort(pdistance, 0, nrows_per_process - 1, plabels);

        MPI_Gather(pdistance, nrows_per_process, MPI_FLOAT, distance,
nrows_per_process, MPI_FLOAT, 0, MPI_COMM_WORLD);
        MPI_Gather(plabels, nrows_per_process, MPI_FLOAT, labels,
nrows_per_process, MPI_FLOAT, 0, MPI_COMM_WORLD);

        if (rank == 0)
        {
            mergeSort(distance, 0, NTRAIN - 1, labels);
            int predicted_class = predict(distance, labels, 5, 3);
            printf("%d) Predicted label: %d   True label: %d\n", i,
predicted_class, (int)y_test[i]);
        }
    }

    free(x);
    free(distance);
    free(pdistance);
}

void knn(char *X_train_path, char *y_train_path, char *X_test_path,
char *y_test_path)
{
    float *X_train;
    float *y_train;
    float *X_test;
    float *y_test;
    double t1, t2;
    int size, rank;

    mpiInitialise(&size, &rank);

    if (rank == 0)
    {
        X_train = initFeatures(X_train_path);
        y_train = initLabels(y_train_path);
    }
}

```

```

X_test = initFeatures(X_test_path);
y_test = initLabels(y_test_path);

if (rank == 0)
    t1 = MPI_Wtime();

fit(X_train, y_train, X_test, y_test, rank, size);

if (rank == 0)
    t2 = MPI_Wtime();

if (rank == 0)
{
    printf("Time for execution (%d Processors): %f\n",
        size, t2 - t1);
    free(X_train);
    free(y_train);
}

free(X_test);
free(y_test);
MPI_Finalize();
}

int main()
{
    knn("X_train.csv", "y_train.csv", "X_test.csv", "y_test.csv");
    return 0;
}

```

## 5.1 Helper code for MPI

```

#ifndef HELPER_H
#define HELPER_H

void checkFile(FILE *f)
{
    if (f == NULL)
    {
        printf("Error while reading file\n");
        exit(1);
    }
}

float *getFloatMat(int m, int n)
{
    float *mat = NULL;
    mat = (float*)calloc(m*n, sizeof(float));
}

```

```

        return mat;
    }

float getMax(float *x, int n)
{
    int i;
    float max = x[0];
    int maxIndex = 0;

    for(i=0; i<n; i++)
    {
        if (x[i] >= max)
        {
            max = x[i];
            maxIndex = i;
        }
    }

    return (float)maxIndex;
}

#endif

```

## 5.2 Mergesort for MPI

```

#ifndef M_SORT
#define M_SORT

void merge(float arr[], int l, int m, int r, float *y)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    float *L = getFloatMat(n1, 1);
    float *R = getFloatMat(n2, 1);
    float *Ly = getFloatMat(n1, 1);
    float *Ry = getFloatMat(n2, 1);

    for (i = 0; i < n1; i++)
    {
        L[i] = arr[l + i];
        Ly[i] = y[l + i];
    }

    for (j = 0; j < n2; j++)
    {
        R[j] = arr[m + 1 + j];
        Ry[j] = y[m + 1 + j];
    }
}

```

```

    }

    i = 0;
    j = 0;
    k = 1;

    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            y[k] = Ly[i];
            i++;
        }

        else
        {
            arr[k] = R[j];
            y[k] = Ry[j];
            j++;
        }
        k++;
    }

    while (i < n1)
    {
        arr[k] = L[i];
        y[k] = Ly[i];
        i++;
        k++;
    }

    while (j < n2)
    {
        arr[k] = R[j];
        y[k] = Ry[j];
        j++;
        k++;
    }

    free (L);
    free (R);

    free (Ly);
    free (Ry);
}

void mergeSort(float arr[], int l, int r, float *y)

```



```

{
    if (l < r)
    {
        int m = l+(r-l)/2;

        mergeSort(arr, l, m, y);
        mergeSort(arr, m+1, r, y);
        merge(arr, l, m, r, y);
    }
}

void printArray(float A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%f ", A[i]);
    printf("\n");
}

#endif

```

## 6 CUDA Code

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NTRAIN 135
#define NTEST 15
#define NFEATURES 4
#define NCLASSES 3
#define K 11
#define TOPN 3
#define THREADS_PER_BLOCK 5
#define X_TRAIN_PATH "X_train.csv"
#define Y_TRAIN_PATH "y_train.csv"
#define X_TEST_PATH "X_test.csv"
#define Y_TEST_PATH "y_test.csv"

char classes[NCLASSES][25] = {"Iris-setosa", "Iris-versicolor", "Iris-virginica"};

void checkFile(FILE *f)
{
    if (f == NULL)
    {
        printf("Error while reading file\n");
        exit(1);
    }
}

```

```

}

float *getFloatMat(int m, int n)
{
    float *mat = NULL;
    mat = (float*)malloc(m*n*sizeof(float));

    return mat;
}

float *initFeatures(char path[])
{
    int index = 0;
    FILE *f = NULL;
    float *mat = NULL;

    mat = getFloatMat(NTRAIN, NFEATURES);

    f = fopen(path, "r");
    checkFile(f);

    while (fscanf(f, "%f%c", &mat[index]) == 1)
        index++;

    fclose(f);
    return mat;
}

float getMax(float *x, int n)
{
    int i;
    float max = x[0];
    int maxIndex = 0;

    for(i=0; i<n; i++)
    {
        if (x[i] >= max)
        {
            max = x[i];
            maxIndex = i;
        }
    }
    return (float)maxIndex;
}

float *initLabels(char path[])
{
    int index = 0;

```

```

    FILE *f = NULL;
    float *mat = NULL;
    mat = getFloatMat(NTRAIN, 1);
    f = fopen(path, "r");
    checkFile(f);
    while (fscanf(f, "%f%c", &mat[index]) == 1)
        index++;

    fclose(f);
    return mat;
}

__global__ void calcDistance (float *X_train, float *X_test, float *distance)
{
    int blockid = blockIdx.x;
    int threadid = blockDim.x*blockid + threadIdx.x;

    if (threadid < NTRAIN)
    {
        int i;
        float dist = 0.0;

        for(i=0; i<NFEATURES; i++)
        {
            dist += (X_train[threadid*NFEATURES + i] - X_test[i])
                *(X_train[threadid*NFEATURES + i] - X_test[i]);
        }

        distance[threadid] = dist;
    }
}

__global__ void sortArray (float *distance, float *ytrain, float *sortedDistance,
float *sortedYtrain)
{
    int blockid = blockIdx.x;
    int threadid = blockDim.x*blockid + threadIdx.x;

    if (threadid < NTRAIN)
    {
        int i, position = 0;
        float element = distance[threadid];
        float label = ytrain[threadid];

        for(i=0; i<NTRAIN; i++)
        {
            if (distance[i] < element || (distance[i] == element
                && threadid < i))

```

```

        position++;
    }

    sortedDistance[position] = element;
    sortedYtrain[position] = label;
}

int predict(float *labels)
{
    float* neighborCount = getFloatMat(NCLASSES, 1);

    float* probability = getFloatMat(NCLASSES, 1);

    int i;
    for(i=0; i<NCLASSES; i++)
        neighborCount[i] = 0;

    for(i=0; i<K; i++)
        neighborCount[(int)labels[i]]++;

    for(i=0; i<NCLASSES; i++)
        probability[i] = neighborCount[i]*1.0/(float)K*1.0;

    int predicted_class = (int)getMax(neighborCount, NCLASSES);

    for(i=0; i<TOPN; i++)
        printf(" %s: %f ", classes[i], probability[i]);

    free(neighborCount);
    free(probability);

    return predicted_class;
}

float *fit(float *X_train, float *y_train, float *X_test)
{
    float *X_traind, *y_traind, *X_testd, *distanced, *distance;

    distance = getFloatMat(NTRAIN, 1);

    int X_train_size = sizeof(float)*NFEATURES*NTRAIN;
    int y_train_size = sizeof(float)*NTRAIN;
    int X_test_size = sizeof(float)*NFEATURES;
    int distance_size = sizeof(float)*NTRAIN;

    cudaMalloc((void**)&X_traind, X_train_size);
    cudaMalloc((void**)&y_traind, y_train_size);

```

```

    cudaMalloc((void**)&X_testd, X_test_size);
    cudaMalloc((void**)&distanced, distance_size);

    cudaMemcpy(X_traind, X_train, X_train_size, cudaMemcpyHostToDevice);
    cudaMemcpy(y_train, y_train, y_train_size, cudaMemcpyHostToDevice);
    cudaMemcpy(X_testd, X_test, X_test_size, cudaMemcpyHostToDevice);

    calcDistance <<< NTRAIN/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>> (X_traind,
X_testd, distanced);

    cudaMemcpy(distance, distanced, distance_size, cudaMemcpyDeviceToHost);

    cudaFree(X_traind);
    cudaFree(X_testd);

    float *sortedDistance, *sortedDistanced, *sortedytrain, *sortedytraind;

    sortedDistance = getFloatMat(NTRAIN, 1);
    sortedytrain = getFloatMat(NTRAIN, 1);

    cudaMalloc((void**)&sortedDistanced, distance_size);
    cudaMalloc((void**)&sortedytraind, y_train_size);

    cudaMemcpy(distanced, distance, distance_size, cudaMemcpyHostToDevice);

    sortArray <<< NTRAIN/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>> (distanced,
y_traind,sortedDistanced, sortedytraind);

    cudaMemcpy(sortedDistance, sortedDistanced, distance_size,
cudaMemcpyDeviceToHost);
    cudaMemcpy(sortedytrain, sortedytraind, y_train_size,
cudaMemcpyDeviceToHost);

    cudaFree(y_traind);
    cudaFree(distanced);
    cudaFree(sortedDistanced);
    cudaFree(sortedytraind);

    free(distance);
    free(sortedDistance);

    return sortedytrain;
}

float *getRandomTestData(float *X_test, int *randId)
{
    srand ( time(NULL) );
    *randId = rand()%NTEST;

```

```

float *data = getFloatMat(NFEATURES, 1);

int i;
for(i=0; i<NFEATURES; i++)
    data[i] = X_test[(randId)*NFEATURES + i];

return data;
}

void readData(float **X_train, float **y_train, float **X_test,
float **y_test)
{
    *X_train = initFeatures(X_TRAIN_PATH);
    *y_train = initLabels(Y_TRAIN_PATH);

    *X_test = initFeatures(X_TEST_PATH);
    *y_test = initLabels(Y_TEST_PATH);
}

int knn(float *X_train, float *y_train, float *X_test)
{
    printf(" Fitting model ");
    float *labels = fit(X_train, y_train, X_test);
    int predicted_class = predict(labels);
    return predicted_class;
}

int main()
{
    float *X_train;
    float *y_train;
    float *X_test;
    float *y_test;

    readData(&X_train, &y_train, &X_test, &y_test);

    int randId;
    float *X_random_test = getRandomTestData(X_test, &randId);

    int predicted_class = knn(X_train, y_train, X_random_test);
    printf("Predicted label: %d True label: %d", predicted_class,
(int)y_test[randId]);
    free(X_train);
    free(y_train);
    free(X_test);
    free(y_test);
    return 0;
}

```

## 7 Result

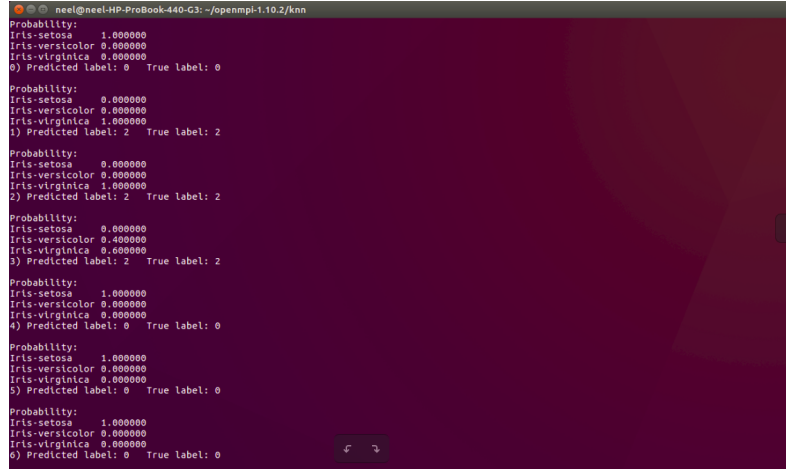
### 7.1 Output for MPI

Time complexity:  $O(\frac{m*n}{p}) + O(\text{communication overhead})$

where, m is the number of features

n is the size of the training set

and p is the number of processes used



### 7.2 Output for CUDA

Time complexity:  $O(n)$

where, n is the size of the training set

```
' Fitting model Time taken: 0.738400 Iris-setosa: 1.000000 Iris-versicolor: 0.000000 Iris-virginica: 0.000000 P
```

## 8 Limitations and possible improvements

- For the MPI implementation, the time taken is minimum when 3 processes are used. Due to the communication overhead, the processing time increases if the number of processes is further increased.
- Using a sorting algorithm with better time complexity will improve the time complexity of the k-NN implementation in both MPI and CUDA.
- As we worked on Google Colab, it wasn't possible to work with a very large dataset.
- Print statements and malloc increase the time taken for execution.

## 9 Conclusion

The parallel implementation of k-NN using CUDA and MPI provides considerable speed-up over the serial implementation. The time complexity for a serial implementation is  $O(m * n)$ , whereas for MPI it is  $O(\frac{m*n}{p}) + O(\text{communication overhead})$  and for CUDA it is  $O(n)$ .

## 10 References

- <http://archive.ics.uci.edu/ml/datasets/iris>
- <http://colab.research.google.com>
- <https://alitarhini.wordpress.com/2011/02/26/parallel-k-nearest-neighbor/>