Comparing the execution of K-Nearest Neighbors algorithm on sequential and parallel execution

[1]A.N.H. Dhatreesh Sai, [2]A.Vishal Yadav, [3]B.H. Tilak, [4]Gautham Lankapalli

[1]1MS18CS002, [2]1MS18CS003, [3]1MS18CS034, [4]1MS18CS061

Abstract

In this paper we compare the classification of large amounts of data in the Skin Dataset running the K nearest neighbors' classifier (KNN) in serial and parallel. The large computing demand of this process is solved with a parallel computing implementation specially designed to work in Grid environments of multiprocessor computer farms. We check the running times of the algorithm for the dataset (consisting of 245057 instances for training and 100 instances for testing). The technologies that are being used are MPI. Finally, we compare the results obtained.

K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on Supervised Learning technique. It assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories. K-NN algorithm stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using K- NN algorithm.

K-NN algorithm can be used for Regression as well as for Classification but mostly it is used for the Classification problems. K-NN is a non-parametric algorithm, which means it does not make any assumption on underlying data.

It is also called a lazy learner algorithm because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.

KNN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similar to the new data.

Suppose, we have an image of a creature that looks similar to cat and dog, but we want to know either it is a cat or dog. So, for this identification, we can use the KNN algorithm, as it works on a similarity measure. Our KNN model will find the similar features of the new data set to the cats and dogs images and based on the most similar features it will put it in either cat or dog category.

The K-NN working can be explained on the basis of the below algorithm:

- Step-1: Select the number K of the neighbours
- Step-2: Calculate the Euclidean distance of K number of neighbours
- Step-3: Take the K nearest neighbours as per the calculated Euclidean distance.
- Step-4: Among these k neighbours, count the number of the data points in each category.
- Step-5: Assign the new data points to that category for which the number of the neighbour is maximum.
- Step-6: Our model is ready.

Below are some points to remember while selecting the value of K in the K-NN algorithm:

- There is no particular way to determine the best value for "K", so we need to try some values to find the best out of them. The most preferred value for K is 5.
- A very low value for K such as K=1 or K=2, can be noisy and lead to the effects of outliers in the model.
- Large values for K are good, but it may find some difficulties.

Advantages of KNN Algorithm:

- It is simple to implement.
- It is robust to the noisy training data
- It can be more effective if the training data is large.

Disadvantages of KNN Algorithm:

- Always needs to determine the value of K which may be complex some time.
- The computation cost is high because of calculating the distance between the data points for all the training samples.

Method

The following section describes our code for K-Nearest Neighbors which has been performed serially and parallelly.

Serial Implementation:

```
#include <iostream>
#include <vector>
#include <fstream>
#include <string>
#include <sstream>
#include <cmath>
#include <set>
#include <map>
#include <ctime>

using namespace std;

class Instance{
   private:
   double R;
   double G;
   double B;
   double isSkin;

   public:
   Instance(double R, double G, double B, int isSkin){
      this->R = R;
      this->G = G;
      this->B = B;
      this->isSkin = isSkin;
   }

   void setR(double R){
      this->R = R;
   }

   void setG(double G){
      this->G = G;
   }

   void setB(double B){
      this->B = B;
   }
```

```
    double getR(){
        return R;
    }

    double getG(){
        return G;
    }

    double getB(){
        return B;
    }

    int skin(){
        return isSkin;
    }

    double calculateDistance(double otherR, double otherG, double otherB){
        return sqrt((R - otherR) * (R - otherR) + (G - otherG) * (G - otherG) + (B - otherB) * (B - otherB));

    }

};

vector<string> split(string a,char e){
    vector<string> rez;
    string cur;
    for(int ctr1=0;ctr1<a.size();ctr1++){
        if(a[ctr1]!=e)
            cur.push_back(a[ctr1]);
        else
            rez.push_back(cur),cur.clear();
    }
    if(cur!="")
        rez.push_back(cur);
    return rez;
}


int main()
{
    string line;
    ifstream myfile("training.txt");
    vector<Instance> instances;

    //init
```

```
if (myfile.is_open())
{
   while (getline(myfile,line))
   {
      vector<string> parts = split(line, ' ');
      Instance instance(std::stod(parts[0]), std::stod(parts[1]), std::stod(parts[2]), std::stod(parts[3]));
      instances.push_back(instance);
   }
   myfile.close();
}

//find min and max
double minR = instances[0].getR();
double maxR = instances[0].getR();

double minG = instances[0].getG();
double maxG = instances[0].getG();

double minB = instances[0].getB();
double maxB = instances[0].getB();

for(int i = 0; i < instances.size(); i++){
   if(instances[i].getR() > maxR){
      maxR = instances[i].getR();
   }
   else if(instances[i].getR() < minR){
      minR = instances[i].getR();
   }

   if(instances[i].getG() > maxG){
      maxG = instances[i].getG();
   }
   else if(instances[i].getG() < minG){
      minG = instances[i].getG();
   }

   if(instances[i].getB() > maxB){
      maxB = instances[i].getB();
   }
   else if(instances[i].getB() < minB){
      minB = instances[i].getB();
   }
}
```

```
//standardization
for(int i = 0; i < instances.size(); i++){
   double curr = instances[i].getR();
   double res = (curr - minR) / (maxR - minR);
   instances[i].setR(res);

   curr = instances[i].getG();
   res = (curr - minG) / (maxG - minG);
   instances[i].setG(res);

   curr = instances[i].getB();
   res = (curr - minB) / (maxB - minB);
   instances[i].setB(res);

  // cout<<instances[i].getR()<<" "<<instances[i].getG()<<" "<<instances[i].getB()<<endl;
}

//setting k = sqrt(number of training instances)
int k = sqrt(instances.size());

//finding k-nn on input from test.txt
ifstream new_file("test.txt");
string new_line;

//start time
clock_t start_time = clock();

int test_instance = 0;

 if (new_file.is_open())
{
   while (getline(new_file,new_line))
   {

      test_instance++;

      vector<string> parts = split(new_line, ' ');

      set<double> distances;

      map<double, int> distanceToClass;


      double r = std::stod(parts[0]);
      double g = std::stod(parts[1]);
```

```
        double b = std::stod(parts[2]);

        r = (r - minR) / (maxR - minR);
        g = (g - minG) / (maxG - minG);
        b = (b - minB) / (maxB - minB);

        double minimum_distance = 500;
        for(int i = 0; i < instances.size(); i++){
           double d = instances[i].calculateDistance(r, g, b);
           distances.insert(d);
           distanceToClass.insert(std::pair<double, int> (d, instances[i].skin()));
        }

        int firstClassCounter = 0;
        int secondClassCounter = 0;

        set<double>::iterator it = distances.begin();

        while(firstClassCounter != k && secondClassCounter != k){
           if(distanceToClass.find(*it)->second == 1){
              firstClassCounter++;
           }
           else if(distanceToClass.find(*it)->second == 2){
              secondClassCounter++;
           }
           it++;
        }

        if(firstClassCounter == k){
           printf("Class for %d object is: 1\n", test_instance);
        }
        else if(secondClassCounter == k){
           printf("Class for %d object is: 2\n", test_instance);
        }

      }

   }

    new_file.close();
    clock_t end_time = clock();
    double elapsed_secs = double(end_time - start_time) / CLOCKS_PER_SEC;

    cout<<"Elapsed time: "<<elapsed_secs<<" seconds."<<endl;
}
```

Parallel implementation (1st Version):

```
#include <iostream>
#include <vector>
#include <fstream>
#include <string>
#include <sstream>
#include <cmath>
#include <set>
#include <map>
#include <ctime>
#include<mpi.h>
#include<set>

using namespace std;

class Instance{
   private:
   double R;
   double G;
   double B;
   double isSkin;

   public:
   Instance(double R, double G, double B, int isSkin){
      this->R = R;
      this->G = G;
      this->B = B;
      this->isSkin = isSkin;
   }

   void setR(double R){
      this->R = R;
   }

   void setG(double G){
      this->G = G;
   }

   void setB(double B){
      this->B = B;
   }
    double getR(){
      return R;
   }

    double getG(){
      return G;
   }

    double getB(){
      return B;
   }

    int skin(){
```

```
        return isSkin;
    }

    double calculateDistance(double otherR, double otherG, double otherB){
        return sqrt((R - otherR) * (R - otherR) + (G - otherG) * (G - otherG) + (B - otherB) * (B - otherB));

    }

};

class TestInstance{
  private:
    double R;
    double G;
    double B;


    public:
    TestInstance(double R, double G, double B){
        this->R = R;
        this->G = G;
        this->B = B;

    }

    void setR(double R){
        this->R = R;
    }

    void setG(double G){
        this->G = G;
    }

    void setB(double B){
        this->B = B;
    }
     double getR(){
        return R;
    }

     double getG(){
        return G;
    }

     double getB(){
        return B;
    }

};

vector<string> split(string a,char e){
    vector<string> rez;
    string cur;
    for(int ctr1=0;ctr1<a.size();ctr1++){
        if(a[ctr1]!=e)
            cur.push_back(a[ctr1]);
```

```
        else
            rez.push_back(cur),cur.clear();
    }
    if(cur!="")
        rez.push_back(cur);
    return rez;
}


        vector<Instance> instances;

        int k;

//returns the class value
int returnClassForObject(double r, double g, double b, set<double> distances, map<double, int>
distanceToClass){

        for(int i = 0; i < instances.size(); i++){
                double distance = instances[i].calculateDistance(r, g, b);
                distances.insert(distance);
                distanceToClass.insert(std::pair<double, int>(distance, instances[i].skin()));
        }

        int countFirstClass = 0;
        int countSecondClass = 0;

        set<double>::iterator it = distances.begin();

        while(countFirstClass != k && countSecondClass != k){
                if(distanceToClass.find(*it) -> second == 1){
                        countFirstClass++;
                }
                else if(distanceToClass.find(*it) -> second == 2){
                        countSecondClass++;
                }
                it++;
        }
        if(countFirstClass == k){
                return 1;
        }
        else if(countSecondClass == k){
                return 2;
        }

}



int main(int argc, char **argv)
{
    MPI_Init(NULL, NULL);


    int world_size;
    int rank;
```

```
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        MPI_Request requests[(world_size - 1) * 3];
MPI_Status statuses[(world_size - 1) * 3];

string line;
ifstream myfile("training.txt");


        //init
if (myfile.is_open())
{
   while (getline(myfile,line))
   {
      vector<string> parts = split(line, ' ');
      Instance instance(std::stod(parts[0]), std::stod(parts[1]), std::stod(parts[2]), std::stod(parts[3]));
      instances.push_back(instance);
   }
   myfile.close();
}

//find min and max
double minR = instances[0].getR();
double maxR = instances[0].getR();

double minG = instances[0].getG();
double maxG = instances[0].getG();

double minB = instances[0].getB();
double maxB = instances[0].getB();

for(int i = 0; i < instances.size(); i++){
   if(instances[i].getR() > maxR){
      maxR = instances[i].getR();
   }
   else if(instances[i].getR() < minR){
      minR = instances[i].getR();
   }

   if(instances[i].getG() > maxG){
      maxG = instances[i].getG();
   }
   else if(instances[i].getG() < minG){
      minG = instances[i].getG();
   }

   if(instances[i].getB() > maxB){
      maxB = instances[i].getB();
   }
   else if(instances[i].getB() < minB){
      minB = instances[i].getB();
   }
}
```

```
  //standardization
  for(int i = 0; i < instances.size(); i++){
     double curr = instances[i].getR();
     double res = (curr - minR) / (maxR - minR);
     instances[i].setR(res);

     curr = instances[i].getG();
     res = (curr - minG) / (maxG - minG);
     instances[i].setG(res);

     curr = instances[i].getB();
     res = (curr - minB) / (maxB - minB);
     instances[i].setB(res);


  }

        k = sqrt(instances.size());


  ifstream new_file("test.txt");
  string new_line;

  vector<TestInstance>test_instances;



        double start, end;

if(rank == 0){
   if (new_file.is_open())
  {
     while (getline(new_file,new_line))
     {

        vector<string> parts = split(new_line, ' ');

        double r = std::stod(parts[0]);
        double g = std::stod(parts[1]);
        double b = std::stod(parts[2]);

        r = (r - minR) / (maxR - minR);
        g = (g - minG) / (maxG - minG);
        b = (b - minB) / (maxB - minB);

        TestInstance new_instance(r, g, b);
        test_instances.push_back(new_instance);

     }

  }

        start = MPI_Wtime();

        int index = 1;
  for(int i = 1; i < test_instances.size(); i++){
```

```
                double r = test_instances[i].getR();
    MPI_Isend(&r, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, requests + index);
    index ++;
                double g = test_instances[i].getG();
    MPI_Isend(&g, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, requests + index);
    index ++;
                double b = test_instances[i].getB();
    MPI_Isend(&b, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, requests + index);
                index ++;
  }


        double r = test_instances[0].getR();
        double g = test_instances[0].getG();
        double b = test_instances[0].getB();
        map<double, int> distanceToClass;
        set<double> distances;
        int class_predicted = returnClassForObject(r, g, b, distances, distanceToClass);
        printf("Class for %d object is: %d\n", rank + 1, class_predicted);
        }
        else{
                double r;
    double g;
    double b;
    MPI_Irecv(&r, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, requests + rank + 1);
    MPI_Irecv(&g, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, requests + rank + 2);
    MPI_Irecv(&b, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, requests + rank + 3);

    MPI_Wait(requests + rank + 1, statuses + rank + 1);
    MPI_Wait(requests + rank + 2, statuses + rank + 2);
    MPI_Wait(requests + rank + 3, statuses + rank + 3);

          map<double, int> distanceToClass;
          set<double> distances;
          int class_predicted = returnClassForObject(r, g, b, distances, distanceToClass);
          printf("Class for %d object is: %d\n", rank + 1, class_predicted);
        }
        MPI_Barrier(MPI_COMM_WORLD);
        if(rank == 0){
                end = MPI_Wtime();
                printf("Elapsed time: %.2f seconds.\n", (end - start));
        }
        MPI_Finalize();

}
```

Parallel Implementation (2nd Version):

```
#include <iostream>
#include <vector>
#include <fstream>
#include <string>
#include <sstream>
#include <cmath>
#include <set>
```

```
#include <map>
#include <ctime>
#include<mpi.h>
#include<set>

using namespace std;

class Instance{
    private:
    double R;
    double G;
    double B;
    double isSkin;

    public:
    Instance(double R, double G, double B, int isSkin){
        this->R = R;
        this->G = G;
        this->B = B;
        this->isSkin = isSkin;
    }

    void setR(double R){
        this->R = R;
    }

    void setG(double G){
        this->G = G;
    }

    void setB(double B){
        this->B = B;
    }
     double getR(){
        return R;
    }

     double getG(){
        return G;
    }

     double getB(){
```

```
        return B;
    }

     int skin(){
        return isSkin;
    }

    double calculateDistance(double otherR, double otherG, double otherB){
        return sqrt((R - otherR) * (R - otherR) + (G - otherG) * (G - otherG) + (B - otherB) * (B - otherB));

    }

};

class TestInstance{
  private:
    double R;
    double G;
    double B;


    public:
    TestInstance(double R, double G, double B){
        this->R = R;
        this->G = G;
        this->B = B;

    }

    void setR(double R){
        this->R = R;
    }

    void setG(double G){
        this->G = G;
    }

    void setB(double B){
        this->B = B;
    }
     double getR(){
        return R;
```

```
    }

     double getG(){
        return G;
    }

     double getB(){
        return B;
    }

};

vector<string> split(string a,char e){
    vector<string> rez;
    string cur;
    for(int ctr1=0;ctr1<a.size();ctr1++){
        if(a[ctr1]!=e)
            cur.push_back(a[ctr1]);
        else
            rez.push_back(cur),cur.clear();
    }
    if(cur!="")
        rez.push_back(cur);
    return rez;
}


          vector<Instance> instances;

          int k;


          vector<map<double, int> > distanceToClass;
          vector<set<double> > distances;

int returnClassForObject(int index){

          int countFirstClass = 0;
          int countSecondClass = 0;

          set<double>::iterator it = distances[index].begin();
```

```
        while(countFirstClass != k && countSecondClass != k){
                if(distanceToClass[index].find(*it) -> second == 1){
                        countFirstClass++;
                }
                else if(distanceToClass[index].find(*it) -> second == 2){
                        countSecondClass++;
                }
                it++;
        }
        if(countFirstClass == k){
                return 1;
        }
        else if(countSecondClass == k){
                return 2;
        }
}
//adapt to the number of processors
int getStartRange(int id){
        switch(id){
                case 1: return 0;
                case 2: return 150000;
                //case 3: return 80000;
                //case 4: return 120000;
                //case 5: return 160000;
                //case 6: return 200000;
                default: return -1;
        }
}

int getEndRange(int id){
                switch(id){
                case 1: return 150000;
                case 2: return instances.size();
                //case 3: return 120000;
                //case 4: return 160000;
                //case 5: return 200000;
                //case 6: return instances.size();
                default: return -1;
        }
}
```

```
int main(int argc, char **argv)
{
   MPI_Init(NULL, NULL);


   int world_size;
   int rank;

   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   string line;
   ifstream myfile("training.txt");


        //init
   if (myfile.is_open())
   {
      while (getline(myfile,line))
      {
         vector<string> parts = split(line, ' ');
         Instance instance(std::stod(parts[0]), std::stod(parts[1]), std::stod(parts[2]), std::stod(parts[3]));
         instances.push_back(instance);
      }
      myfile.close();
   }

   //find min and max
   double minR = instances[0].getR();
   double maxR = instances[0].getR();

   double minG = instances[0].getG();
   double maxG = instances[0].getG();

   double minB = instances[0].getB();
   double maxB = instances[0].getB();

   for(int i = 0; i < instances.size(); i++){
      if(instances[i].getR() > maxR){
```

```
            maxR = instances[i].getR();
        }
        else if(instances[i].getR() < minR){
            minR = instances[i].getR();
        }

        if(instances[i].getG() > maxG){
            maxG = instances[i].getG();
        }
        else if(instances[i].getG() < minG){
            minG = instances[i].getG();
        }

        if(instances[i].getB() > maxB){
            maxB = instances[i].getB();
        }
        else if(instances[i].getB() < minB){
            minB = instances[i].getB();
        }
    }


    //standardization
    for(int i = 0; i < instances.size(); i++){
        double curr = instances[i].getR();
        double res = (curr - minR) / (maxR - minR);
        instances[i].setR(res);

        curr = instances[i].getG();
        res = (curr - minG) / (maxG - minG);
        instances[i].setG(res);

        curr = instances[i].getB();
        res = (curr - minB) / (maxB - minB);
        instances[i].setB(res);


    }

        //setting k = sqrt(number of training instances)
        k = sqrt(instances.size());
```

```
ifstream new_file("test.txt");
string new_line;

vector<TestInstance>test_instances;



    double start, end;



    for(int i = 0; i < instances.size(); i++){
            distanceToClass.push_back(map<double, int>());
            distances.push_back(set<double>());
    }


    //reading test instances from test.txt


 if (new_file.is_open())
{
   while (getline(new_file,new_line))
   {

      vector<string> parts = split(new_line, ' ');

      double r = std::stod(parts[0]);
      double g = std::stod(parts[1]);
      double b = std::stod(parts[2]);

      r = (r - minR) / (maxR - minR);
      g = (g - minG) / (maxG - minG);
      b = (b - minB) / (maxB - minB);

      TestInstance new_instance(r, g, b);
      test_instances.push_back(new_instance);

   }

}
```

```
for(int i = 0; i < test_instances.size(); i++){
        distances.push_back(set<double>());
        distanceToClass.push_back(map<double, int>());
}



start = MPI_Wtime();
MPI_Status status;


for(int i = 0; i < test_instances.size(); i++){

          double r = test_instances[i].getR();
          double g = test_instances[i].getG();
          double b = test_instances[i].getB();


          int startRange = getStartRange(rank);
          int endRange = getEndRange(rank);

        if(rank > 0){
                int length = endRange - startRange;

                double arrayToSend1[length];
                int arrayToSend2[length];

                int l = 0;
                int q = 0;
                for(int j = startRange; j < endRange; j++){
                        double distance = instances[j].calculateDistance(r, g, b);
                        arrayToSend1[l++] = distance;
                        arrayToSend2[q++] = instances[j].skin();
                }


                MPI_Send(&length, 1, MPI_INT, 0, rank, MPI_COMM_WORLD);
                MPI_Send(arrayToSend1, length, MPI_DOUBLE, 0, rank * 10,
MPI_COMM_WORLD);
                MPI_Send(arrayToSend2, length, MPI_INT, 0, rank * 100,
MPI_COMM_WORLD);
```

```
                }
                else{
                        for(int j = 1; j < world_size; j++){
                        int length;
                        MPI_Recv(&length, 1, MPI_INT, j, j, MPI_COMM_WORLD, &status);


                        double arrayToRecv[length];
                        int classValues[length];

                        MPI_Recv(arrayToRecv, length, MPI_DOUBLE, j, j * 10,
MPI_COMM_WORLD, &status);
                        MPI_Recv(classValues, length, MPI_INT, j, j * 100, MPI_COMM_WORLD,
&status);

                        for(int j = 0; j < length; j++){
                                distances[i].insert(arrayToRecv[j]);
                                distanceToClass[i].insert(std::pair<double, int>(arrayToRecv[j],
classValues[j]));

                        }
                        }

                }

        }

        if(rank == 0){

                for(int i = 0; i < test_instances.size(); i++){

                        int classForObject = returnClassForObject(i);
                        printf("Class for %d object: %d\n", i + 1, classForObject);
                }
                end = MPI_Wtime();
                printf("Elapsed time: %.2f seconds.\n", (end - start));
        }
        MPI_Finalize();

}
```

Function Descriptions: -

MPI methods used are:

1. MPI_Init - Initialize the MPI execution environment
   Syntax - int MPI_Init (int *argc, char ***argv)

2. MPI_send – Sends message from one process to another process
   Syntax - MPI_Send (void * data,
   int count,
   MPI_Datatype datatype,
   int destination,
   int tag,
   MPI_Comm communicator)

3. MPI_recv – Receives message from other process

Syntax- MPI_Recv (void * data,
int count,
MPI_Datatype datatype,
int source,
int tag,
MPI_Comm communicator, MPI_Status* status)

4. MPI_Comm_rank - Determines the rank of the calling process in the communicator

Syntax - int MPI_Comm_rank (MPI_Comm comm, int *rank)

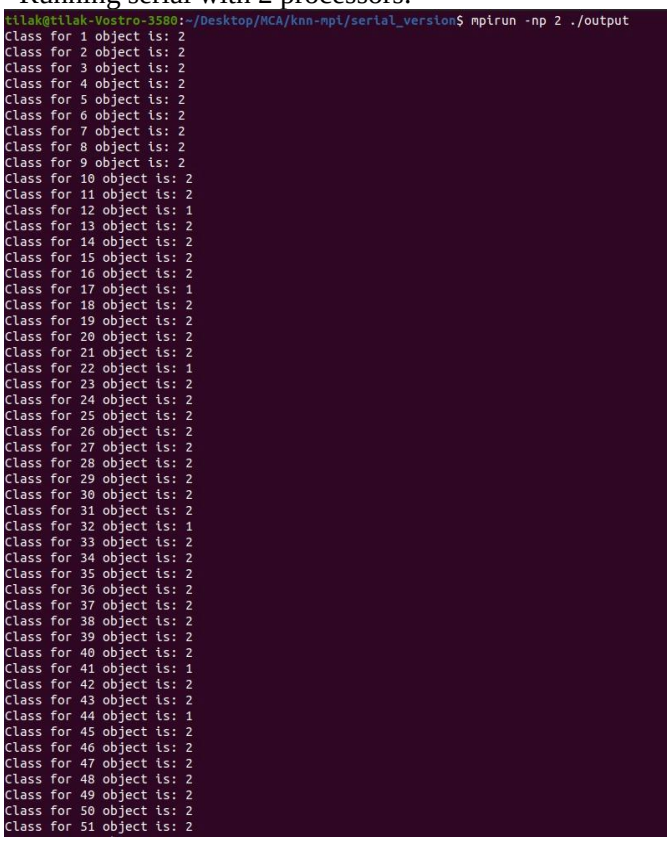
5. MPI_Comm_size – Determines the size of the group associated with a communicator

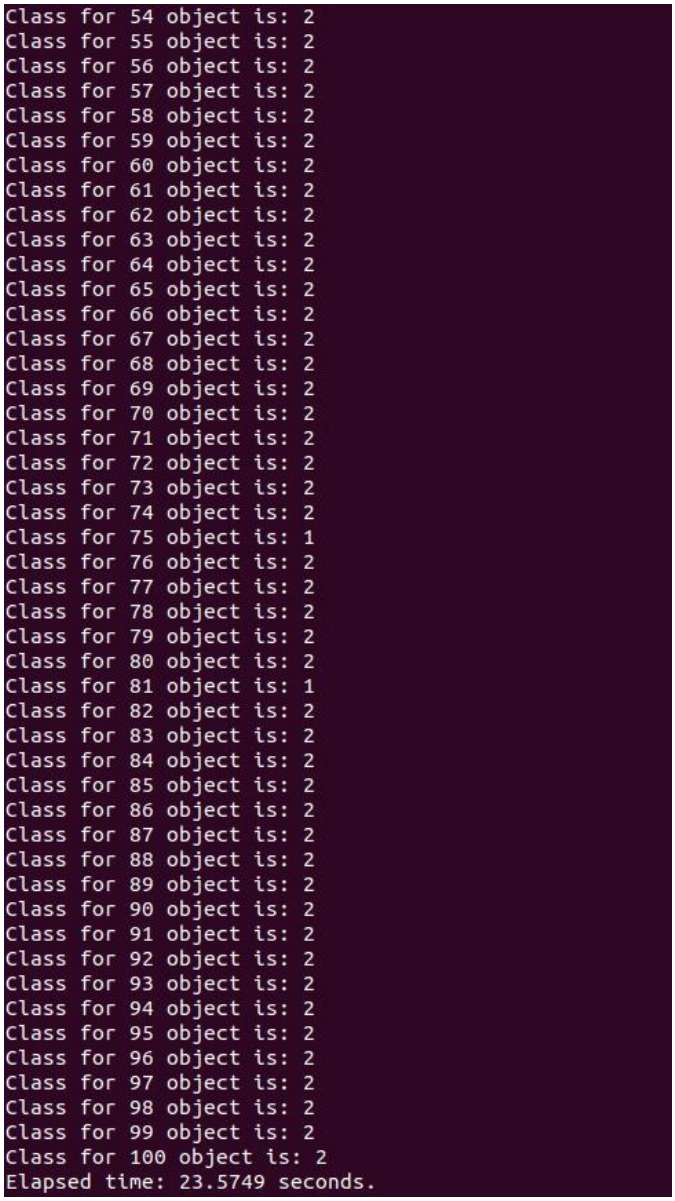
Syntax - int MPI_Comm_size (MPI_Comm comm, int *size)

Results

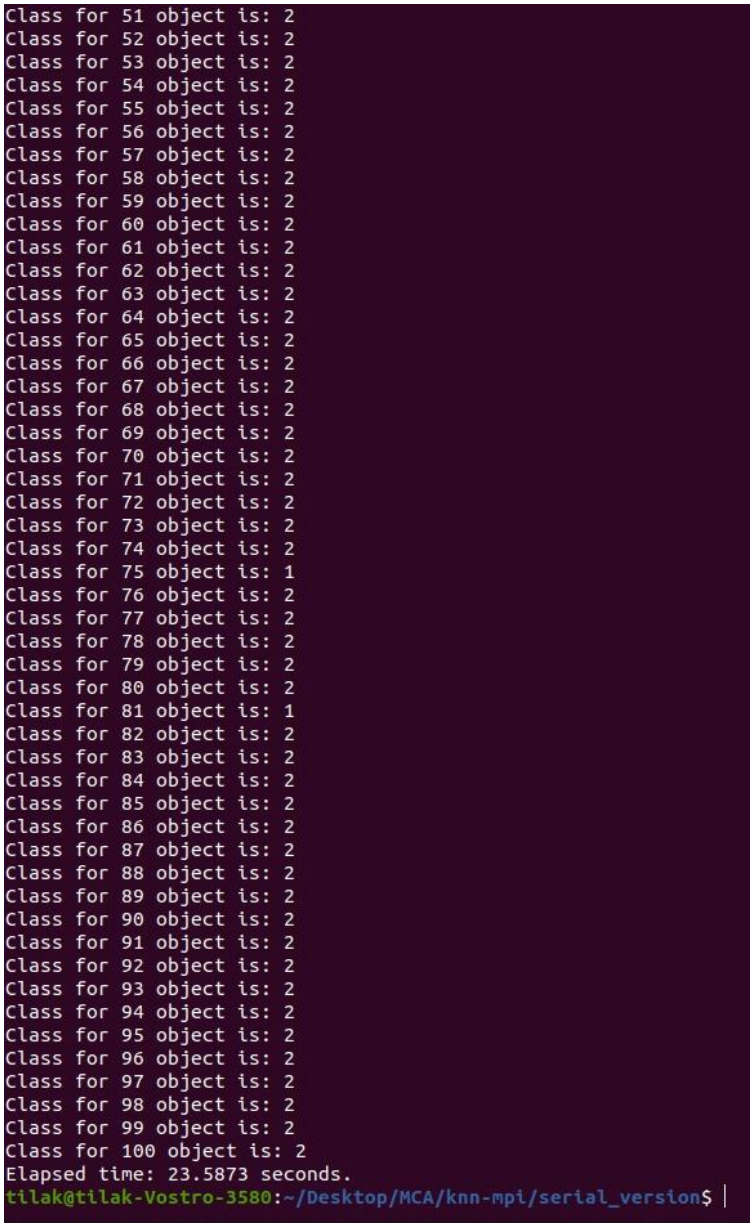
A sample set of 100 images were for testing. The execution in serial, where each processor runs the algorithm for each of the number of processors took a long time. The execution in parallel (1st version) where each instance in run on one processor took the shortest time, whereas the execution in parallel (2nd version) where the testing set is divided equally among the number of processors took a longer time than the first version. However, both parallel executions perform faster than the serial version.

Serial Execution:
Running serial with 2 processors:





Time elapsed on one processor



Time elapsed on second processor

Parallel execution (1st version):

```
tilak@tilak-Vostro-3580:~/Desktop/MCA/knn-mpi/mpi_version_1$ mpirun -np 100 ./output
Class for 4 object is: 2
Class for 17 object is: 1
Class for 14 object is: 2
Class for 22 object is: 1
Class for 18 object is: 2
Class for 3 object is: 2
Class for 27 object is: 2
Class for 19 object is: 2
Class for 20 object is: 2
Class for 16 object is: 2
Class for 24 object is: 2
Class for 33 object is: 2
Class for 5 object is: 2
Class for 8 object is: 2
Class for 11 object is: 2
Class for 9 object is: 2
Class for 31 object is: 2
Class for 10 object is: 2
Class for 7 object is: 2
Class for 37 object is: 2
Class for 13 object is: 2
Class for 15 object is: 2
Class for 34 object is: 2
Class for 28 object is: 2
Class for 6 object is: 2
Class for 21 object is: 2
Class for 36 object is: 2
Class for 1 object is: 2
Class for 35 object is: 2
Class for 23 object is: 2
Class for 25 object is: 2
Class for 26 object is: 2
Class for 2 object is: 2
Class for 12 object is: 1
Class for 32 object is: 1
Class for 30 object is: 2
Class for 29 object is: 2
Class for 68 object is: 2
Class for 54 object is: 2
Class for 65 object is: 2
Class for 72 object is: 2
Class for 45 object is: 2
```

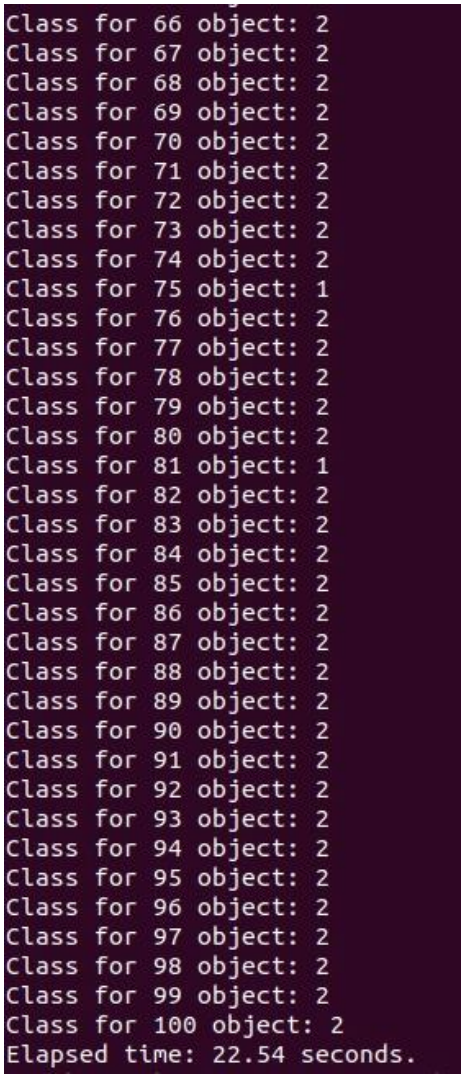
Execution is done with number of processors = number of instances (100)

```
Class for 90 object is: 2
Class for 43 object is: 2
Class for 63 object is: 2
Class for 74 object is: 2
Class for 77 object is: 2
Class for 87 object is: 2
Class for 73 object is: 2
Class for 71 object is: 2
Class for 67 object is: 2
Class for 58 object is: 2
Class for 62 object is: 2
Class for 100 object is: 2
Class for 83 object is: 2
Class for 99 object is: 2
Class for 88 object is: 2
Class for 53 object is: 2
Class for 40 object is: 2
Class for 86 object is: 2
Class for 79 object is: 2
Class for 41 object is: 2
Class for 91 object is: 2
Class for 82 object is: 2
Class for 46 object is: 2
Class for 78 object is: 2
Class for 48 object is: 2
Class for 52 object is: 2
Class for 70 object is: 2
Class for 61 object is: 2
Class for 66 object is: 2
Class for 59 object is: 2
Class for 44 object is: 2
Class for 49 object is: 2
Class for 50 object is: 2
Class for 69 object is: 2
Class for 94 object is: 2
Class for 81 object is: 2
Class for 75 object is: 2
Class for 51 object is: 2
Elapsed time: 9.33 seconds.
```

Time elapsed in 1st version

Parallel Execution (2nd version):

```
tilak@tilak-Vostro-3580:~/Desktop/MCA/knn-mpi/mpi_version_2$ mpirun -np 4 ./a.out
Class for 1 object: 2
Class for 2 object: 2
Class for 3 object: 2
Class for 4 object: 2
Class for 5 object: 2
Class for 6 object: 2
Class for 7 object: 2
Class for 8 object: 2
Class for 9 object: 2
Class for 10 object: 2
Class for 11 object: 2
Class for 12 object: 1
Class for 13 object: 2
Class for 14 object: 2
Class for 15 object: 2
Class for 16 object: 2
Class for 17 object: 1
Class for 18 object: 2
Class for 19 object: 2
Class for 20 object: 2
Class for 21 object: 2
Class for 22 object: 1
Class for 23 object: 2
Class for 24 object: 2
Class for 25 object: 2
Class for 26 object: 2
Class for 27 object: 2
Class for 28 object: 2
Class for 29 object: 2
Class for 30 object: 2
Class for 31 object: 2
Class for 32 object: 1
Class for 33 object: 2
Class for 34 object: 2
Class for 35 object: 2
Class for 36 object: 2
```



Time elapsed in 2nd version

No. Of Instances	Time taken for Serial Processing (s)	Time taken for Parallel Processing (1st version) (s)	Time taken for Parallel Processing (2nd version) (s)
100	23.57 (1st processor) 23.58 (2nd processor)	9.33	22.54

Time taken for Serial vs Parallel processing for 100 given instances

As we can see, serial processing takes a considerably longer time to finish execution when compared to parallel processing. The total time taken to classify 100 instances serially was 23.57 and 23,58 seconds for the 1st and 2nd processor respectively, whereas the total time taken for parallel compression was just 9.33 seconds in the first version and 22.54 seconds in the second version.

References

1. Aparício, G., Blanquer, I., & Hernández, V. (2006, June). A parallel implementation of the k nearest neighbours classifier in three levels: Threads, mpi processes and the grid. In *International Conference on High Performance Computing for Computational Science* (pp. 225-235). Springer, Berlin, Heidelberg.
2. https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning
3. https://towardsdatascience.com/laymans-introduction-to-knn-c793ed392bc2