UNIVERSITÀ
DI PAVIA

Open MPI implementation of K-Nearest Neighbors

F. Amato[1] and D. Ligari[1]

[1] *University of Pavia, Department of Computer Engineering (Data Science), Pavia, Italy*

Date: January 23, 2023

Abstract— The objective of this project is providing an efficient implementation of the K-Nearest Neighbors (Machine Learning) algorithm, that exploits multiple CPUs architectures, using Open MPI (on C++). Starting from a serial implementation, the code has been modified to work in parallel to obtain an improvement in terms of performances, and so a significant speedup (compared with the standard serial version). An a priori study of available parallelism has been conducted, to understand why and which parts of the serial code could have been parallelized. To better understand how this implementation scales, two dimensions have been varied: the datasets sizes, and processors number. In so doing, the code has been runned onto Google Cloud Platform virtual instances, to exploit the whole computational power of a cluster of machines cooperating one with each other.

Keywords— K-Nearest Neighbors • Machine Learning • Parallelization • Open MPI • Performances • Google Cloud Platform

1. What is K-Nearest Neighbors

Nowadays Machine Learning (ML), a branch of Artificial Intelligence (AI) that automates the construction of analytical models based on the idea that systems can learn from data without the need for human intervention, is becoming used in many different application fields, especially to build efficient models to help humans in taking decisions, understanding interesting patterns, and so on.

The problems that Machine Learning can efficiently solve belong in two different macro-groups: supervised learning (classification, regression) and unsupervised learning (clustering, dimensionality reduction). The former group uses labeled datasets to train algorithms to find which are the most impactful features for a given phenomenon outcome (categorical or continuous values). The latter, instead, uses unlabeled datasets to analyze and discover hidden patterns or data groupings.

The K-Nearest Neighbors algorithm (KNN) is falling into the supervised learning methods, and is based on proximity to make classifications or predictions of an individual data point. It works particularly well when the assumption that similar points can be found near one another is true.

The report threaten version can solve classification problems, but it can be easily extended (modifying properly it) to solve also regression.

Its basic working principle is to assign the class label to a given point on the basis of a majority vote, considering its K nearest neighbors. Those K points are obtained by performing a selected distance function (e.g: Euclidean, Manhattan, Minkowski or Hamming).

Contact data: F. Amato, francesco.amato01@universitadipavia.it

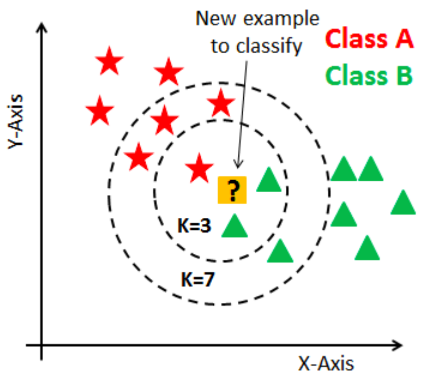

Fig. 1: Simple K-Nearest Neighbors problem

Obviously, each distance function provides different results. In this implementation, the Euclidean distance is the standard (and only) one.

In order to display its behavior, a simple example, where both the classes and feature number are just two, has been chosen [1]. A question that might come to your mind is:

- What is the class of the unclassified point?

Based on what stated previously, if the number of considered neighbors (K) is equal to three the most occurrent class is the green one, otherwise considering six it would have been red. The described working flow is the same considering a more complex example.

Even the choice of the value of the parameter K is a critical aspect, since it can lead to over-fitting (low K) or under-fitting (high K). It should be properly tuned, since it will largely depend on the input data, as data with more outliers or noise will likely perform better with higher values of K. Clearly it should have some sense considering very far (high K) points.

Parameter	Description
trainFile	Path of train file
nTrainSamples	Number of train points to use
testFile	Path of test file
nTestSamples	Number of test points to use
K	Number of neighbors
nFeatures	Number of features for each sample
nClasses	Number of classes

Table 1: Parameters of the serial version

2. Serial implementation

In the section, it will be discussed how the serial version of the KNN algorithm has been implemented, in C++.

The code can be seen on GitHub at:

- KNN-OpenMPI/Serial

This version can be described with a set of consequent steps. The final objective is, given the parameters described in the above table (a more detailed description can be found in the project's README file), to obtain train and test accuracy of a KNN model.

Features standardization and more precise classification performance metrics should have been used (e.g., to understand whether the selected model is more biased on recognizing a class with respect to another) but this is not the actual purpose of the project.

In order to obtain the test accuracy (the same follows for train accuracy), the following steps are performed one after the other:

- Consider each test sample and compute the Euclidean distance (1) from all the train samples
- Consider the most occurrent class of K-Nearest Neighbors points of each test sample and assign it to the predicted test sample class
- Compare the actual class of the test sample with the predicted one, if the values are equal the sample has been correctly classified
- Divide the number of correctly classified points by the total number of test points, multiply for 100 and that is the test accuracy

If the discussed serial algorithm is applied on the Iris dataset (considering a train-test split of 0.8/0.2) and fine-tuning the K parameter, using a grid-search approach, both train and test accuracies are around 97%.

Example on how the script can be executed, once compiled:

```
./knn-serial ../Datasets/train.txt
120 ../Datasets/test.txt 30 10 4 3
```

N-dimensional Euclidean distance formula The used distance formula is the depicted below.

$$d\left(p,q\right) = \sqrt{\sum_{i=1}^{n}\left(q_i - p_i\right)^2} \tag{1}$$

Where p and q are respectively a train and test sample, whereas n is the number of features that the two considered samples have.

3. Available parallelism

In this section it will be briefly discussed the expected improvement in terms of performances of parallelizing the serial solution described previously.

The serial version code has been analyzed discarding the less relevant (neither critical nor crucial) parts, such as: the checks on the parameters that can be passed to the `main()` function; the import of the libraries, and the both global variables and the sample (`struct`) definition.

In so doing, using the Amdalh's law (2), the *theoretical* obtained speedup, increasing the number of CPUs, has a trend as shown in the figure below ($\lim_{N\to\infty} Speedup(N) := \frac{1}{S} = 4.125$).

The only fraction of code that cannot be efficiently, nor simply, be parallelized is the reading of the two datasets (train and test). All the other functions can exploit multiple CPUs architectures (how they have been parallelized is explained in the following section). Basically the percentage of code that can be parallelized is almost the 76%.

Amdalh's law The formula used to obtain the theoretical speedup is depicted below.

$$Speedup(N) = \frac{1}{S + \frac{P}{N}} \tag{2}$$

Where N is the number of CPUs used, S is the the fraction of the code that cannot be parallelized (serial execution), while P the fraction that can be parallelized (recall that $P + S = 1$).

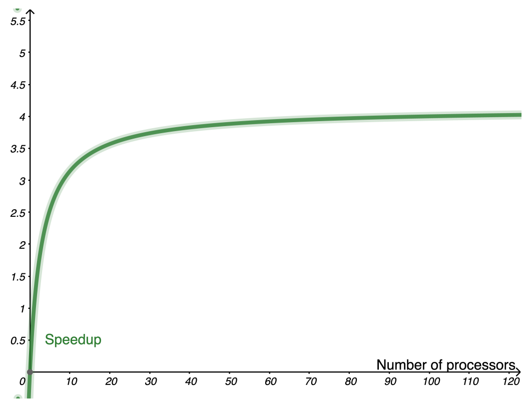

Fig. 2: Theoretical speedup considering Amdalh's law

4. Open-MPI implementation

5. Performance analysis

6. Conclusions

The conclusions should present a review of the key points of the article with special emphasis on the analysis and discussion of the results that were made in the previous sections and in the applications or extensions of them. You should not reproduce the summary in this section or repeat paragraphs already included in another sections of the work.