UNIVERSITÀ
DI PAVIA

Open MPI implementation of K-Nearest Neighbors classifier

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

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

Date: February 16, 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

CONTENTS

1 What is K-Nearest Neighbors 1

2 Sample datasets generation 2

3 Serial implementation 2

4 Available parallelism 3

5 Open-mpi implementation 3

6 Performance analysis 4

7 Conclusions 5

8 Contributions 5

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).

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.

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

Clearly it should have some sense considering very far (high K) points.

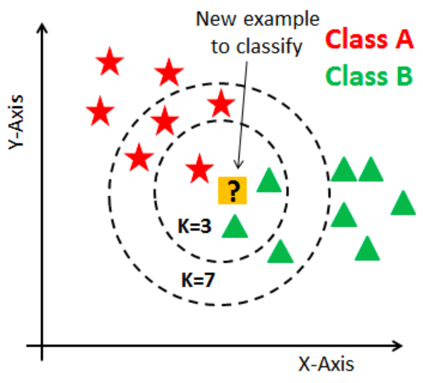

Fig. 1: Simple K-Nearest Neighbors problem [9]

2. Sample datasets generation

Even though the implementations that will be discussed in details in the following sections are capable of perform classification even on real case datasets (as will be shown on the well-known Iris dataset [8]), the objectives of this analysis are more biased on execution performances with respect to Machine Learning in a strict sense (the objective is not to find the best configuration for a KNN classifier in a specific field of study).

So, even a general purpose dataset (with not a real meaning of the features representing a specific characteristic for that sample of a given class) can be considered as good as benchmark to test how the code behaves in terms of performances by considering different aspects such as: the number of samples (dataset dimension), the number of classes, the number of features for each class (still related to the dataset dimension), and whether to apply Min-Max scaling.

In order to cover this aspect, a specific Python script has been implemented [6].

3. Serial implementation

In the section, it will be discussed how the serial version of the KNN algorithm has been implemented from scratch, using C++ [5].

The code can be described as a set of consequent steps, in order to perform classification on a general dataset. The final objective is, given some parameters [1] (a more detailed description can be found in the project's README file [3]), to obtain train and test accuracy of a KNN model.

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.

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

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

- Consider each test sample and compute the Euclidean distance (1) with respect to 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 (classification accuracy consequently increases), otherwise not
- Divide the number of correctly classified test points by the total number of test points, multiply for 100, which represents the test accuracy

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

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

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

The discussed serial algorithm has been applied also on the already cited Iris dataset (considering a train-test split of 80%-20% and fine-tuning the K parameter - using a random search approach), in order to show that it can work properly also with datasets used daily by the data scientist community. For this simple classification problem, both train and test accuracies are almost 97% (neither over-fitting nor under-fitting, since the discrepancy between the two is negligible).

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

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

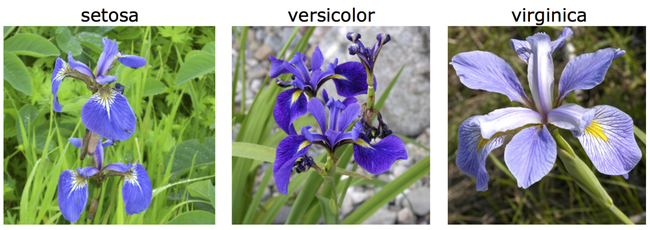

Fig. 2: Visual representation of the different iris classes. It is well-known that the two more distinguishable features between all those iris types are sepal and petal [7]

Cost associated with each task of the serial code, in terms of number of instructions For what concerns the serial implementation an a priori estimate on the number of instructions needed to conclude a specific tasks has been conducted, in order to better understand which were the functions that needed more (if possible) parallelization, since they were more computationally expensive.

The results of that analysis are shown in the following table, which has been obtained by looking directly at the code of the serial version (they can also be checked more precisely by using a specific profiler tool such as Valgrind or Gprof).

Task	Cost (# of instructions)
Euclidean distance	$NFeatures + 2$
Sorting	$nTrainSamples * log(nTrainSamples)$
KNN prediction	$nTrainSamples * EuclDist + Sort + 3 * nClasses + K$
Array calculus	$nTestSamples * KNNPrediction + 2$
Read train samples	$1 + nTrainSamples * (2 + nFeatures)$
Read test samples	$1 + nTestSamples * (2 + nFeatures)$
Initialization	20

Table 2: Estimate of the number of instruction executed, for each function of the serial implemented version

An aspect that can be pointed out is that typically, in the Machine Learning field, the number of samples belonging into the train set are substantially more with respect to the one appertaining to the test set.

This choice comes from a very linear reasoning: because the test set's job is only evaluating the trained model's performance on unseen data (while the model learning happens on train data). Learning the model is a far more complex task than evaluating the model performance. If the training set is small, the model tends to overfit and may not generalize well to new data patterns that are not seen in training data. So model performance typically improves as training dataset size increases (until a certain point, when the gains get saturated).

So, the cost as the number of instructions associated with each task that regards the test set is lower if compared with the cost of the same tasks working with the train set.

4. 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. 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 next section).

Amdalh's law In order to understand the plausible speedup which can be obtained parallelizing the serial version, the Amhdal's law has been used. Its formula is the following one:

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

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

By looking at the previous above table it can be noticed that the number of iterations for each task, among other things, is strongly correlated with the dataset dimension (the number of samples to be considered both in train and test).

Since the speedup depends on the portion of serializable (and non) fraction of code and those two depend on the number of samples, it can be noticed that the speedup does not change only with the number of cores (which can equally spread the work), but also on the number of samples.

The serializable code fraction diminishes with the increasing of the dataset dimension (tends to zero, as shown on the next graph), so consequently the speedup increases linearly with the core number.

In reality, differently from what discussed from the "theoretical" view point, what is expect is that the dataset dimension impacts negatively (to be valued how much) the execution time, since the data transmission between master and slaves depends (grows proportionally) on the amount of train and test of samples.

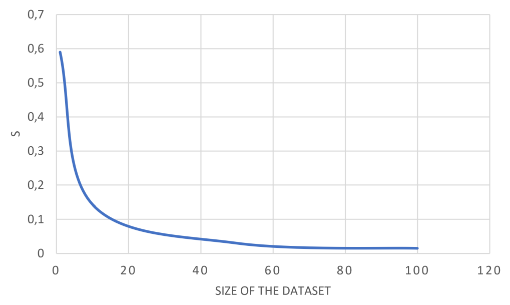

Fig. 3: Theoretical fraction of code that cannot be parallelized, in relation to the size of the dataset

5. Open-MPI implementation

Parallelize or distribute a task, such that the same operation can be executed on different nodes, is considered a must in lot of field, let's just think for example about the Big Data analysis (have a look at Apache Spark if interested in that topic). How can be possible that a single CPU can in a computationally feasible time compute a task? Exploiting the parallelism of a single machine can drastically reduce the execution time of a task, if the parallelization procedure has been well-designed, but it suffers from the issue of the lack of possible memory allocation.

For that specific purpose, using distributed systems allowed to exploit not only vertical scalability (so to use all the computational power provided by a single machine, that as imaginable is limited), but also the possibility to use the whole computational power of multiple machine coping together to achieve a specific goal.

In order to do that, there are basically two major ways: shared memory systems or distributed memory system. The two mentioned options have both advantages and disadvantages.

Designing those solutions is not as easy as it seems to be, for multiple factors such as load balancing and data partitioning. In this specific case, using Open MPI (Message Passing Interface) allows implementing, test and run a solution on a distributed memory system. Through messages sending between a master node and its slaves, it is possible to handle the spread of data between multiple machines, where each one of them has its own memory and its own computational power to use.

The code that will be discussed in this section, has been implemented by considering the serial version and distributing it using Open MPI, still with C++ [4].

Even in this particular case, the implementation can be described as a set of consequent steps, in order to perform classification on a general dataset. It takes as input the already discussed parameters shown in the serial implementation, and it prints as output the train and test accuracy of the selected KNN model, on the specified datasets. Furthermore, to facilitate the performance analysis, the executions times and the specific configurations of parameters have been stored in specific files.

Two very important additional parameters are the number of CPUs on which run the code, but also MPI gives the possibility to specify the host-file, that will be useful to perform tests with different clusters of machines.

To obtain the test accuracy (almost the same line of reasoning follows for train accuracy, except some very little differences), the following steps are performed:

- The master node reads both the train and the test datasets, then scatters to all the slaves the test dataset, and broadcasts the train samples (already done since used also to assess the train accuracy)
- Each slave works on a fraction of the test dataset to assess its own accuracy. This accuracy assessment of each node is basically the same one as already summarized in the serial implementation
- In case the fraction of data is not divisible by the number of slaves the master node works also on the remaining fraction of samples. Furthermore, the accuracy calculated by each slave weights differently (since just one machine has computed also the accuracy on the reminder)
- Finally a reduce function is performed by the master to obtain the overall test accuracy, gathering all the slaves accuracies (3)

This version is obviously consistent with the serial one (maintaining fixed the dataset and all the other parameters), but also by increasing the number of CPUs on which this code is executed. Example on how the script can be executed, once compiled, on a relative big dataset (containing 12K samples, 100 features for each sample and 6 possible classes):

```
mpirun --hostfile hostfile
-n 16 knn-distributed.o
../Datasets/train.txt 10000
../Datasets/test.txt 2000
10 100 6
```

Overall accuracy (distributed case) The used overall accuracy is the one depicted below.

$$Acc_{tot} = \frac{1}{N}\left(\sum_{k=1}^{C} Acc_k * \frac{N}{C} + (Acc_{rem} * N \bmod C)\right) \quad (3)$$

Where N is the total number of samples, C is the total number of CPUs available, Acc_k is the accuracy obtained by the k-th CPU, whereas Acc_{rem} is the accuracy obtained by a node on the reminder (whether $\frac{N}{C} \neq 0$).

6. Performance analysis

Once treated in detail, the discussion of the distributed implementation with Open MPI is necessary to consider how well this implementation is in terms of performance, more specifically for what concerns execution time.

The testing and debugging has previously done on local machines. Once different tests passed without bug occurrence, the code has been moved to the latter described Google Cloud Platform (GCP) clusters. GCP is a suite of cloud computing services offered by Google that provides a range of infrastructure and platform services, including computing, storage, networking, big data, machine learning, and security, making it a popular choice for businesses of all sizes.

For the KNN distributed calculations and performance tests through the GCP, three different clusters have been created:

- **Light cluster**: six e2-small (2vCPU, 2GBRAM), intra-regional
- **Fat cluster**: three e2-highcpu-8 (8vCPU, 8GBRAM), intra-regional
- **Infra-regional (fat) cluster**: three e2-highcpu-8 (8vCPU, 8GBRAM), each one located in a different region (Central Europe, East Asia and West America)

In order to automatize the tests, a bash script [1] has been created. It considers an already generated dataset, with the following requirements: at least 25K overall samples, 50 features and 4 classes. The user can select whether to run the tests on the light cluster, or fat cluster (intra or infra regional) and then all the tests are being performed by varying the number of samples considered and the number of cores on which the code is executed. The other parameters (e.g. the number of features, K, ...) have clearly an influence on the overall execution time, but in order to keep simple and easily understandable the performance analysis discussion it has been chosen to vary only the two already mentioned parameters (number of cores and number of samples).

Each time a single test is executed, a new row is added into a *.csv* file, that contains all the execution details (number of samples, number of cores, execution time) for one specific cluster, and then we moved for simplicity the different execution details files, for the three already discussed clusters, in a specific folder [2].

The three execution details files have been used to conduce the performance analysis and generate all the graphs that will be soon discussed.

UNIVERSITÀ
DI PAVIA

7. Conclusions

8. Contributions

In the following table, it is described to which task each project's member have contributed.

Author	Major contribution
F. Amato	Report + GitHub repo's description
	Datasets generation
	Serial + parallel implementations
	Code testing (also GCP) + debugging
	Performance + scalability analysis
D. Ligari	Available parallelism + Ahmdal's law
	Code testing + debugging
	GCP clusters configuration + testing
	Performance + scalability analysis

References

[1] A. Francesco D. Ligari. *Bash script that automates the KNN distributed classifier tests on GCP clusters (light or fat)*. `https://github.com/Amatofrancesco99/KNN-OpenMPI/blob/main/Parallel/Tests/GCP/run-tests.sh`. 2023.

[2] A. Francesco D. Ligari. *Folder that contains multiple files regarding the execution details of the distributed KNN on a single GCP cluster, varying the number of samples and the number of CPUs*. `https://github.com/Amatofrancesco99/KNN-OpenMPI/tree/main/Parallel/Tests/Results`. 2023.

[3] A. Francesco D. Ligari. *KNN classifier project description*. `https://github.com/Amatofrancesco99/KNN-OpenMPI/blob/main/README.md`. 2023.

[4] A. Francesco D. Ligari. *KNN classifier, distributed implementation with OpenMPI*. `https://github.com/Amatofrancesco99/KNN-OpenMPI/blob/main/Parallel/main.cpp`. 2023.

[5] A. Francesco D. Ligari. *KNN classifier, serial implementation with C++*. `https://github.com/Amatofrancesco99/KNN-OpenMPI/blob/main/Serial/main.cpp`. 2023.

[6] A. Francesco D. Ligari. *Python script that generates train and test dataset for testing the KNN classifier*. `https://github.com/Amatofrancesco99/KNN-OpenMPI/blob/main/Datasets/generate.py`. 2023.

[7] Fukuit. *Get started with machine learning in Python*. `https://qiita.com/fukuit/items/b94e58191790bfce7f8b`. 2016.

[8] Manimala. *Iris Dataset*. `https://www.kaggle.com/datasets/vikrishnan/iris-dataset`. 2017.

[9] Rajvi Shah. *Introduction to k-Nearest Neighbors (kNN) Algorithm*. `https://ai.plainenglish.io/introduction-to-k-nearest-neighbors-knn-algorithm-e8617a448fa8`. 2021.