Università di Pisa

Computer Science Master Degree

Implementing KNN using C++ threads and Fastflow

Author: Dario SALVATI
Mat: 619628

ACADEMIC YEAR 2020/2021

Abstract

K-Nearest Neighbors is a non-parametric classification method used mainly to label elements of a multi-dimensional space. The class assigned to each point is based on the plurality vote of its neighbors. In this report I'll discuss how I implemented the algorithm using C++ threads and also using the Fastflow framework, in order to minimize the execution time of the algorithm.

1 Introduction

The K-Nearest Neighbors algorithm is a very popular statistical method used for classifying data-points without the need to use complex computational models like SVMs or Neural Networks. Its simplicity is given from the fact that the only thing required to perform the classification is a dimensional representation of the data-points and their relative classes, given from the "training set". A new point is then classified using a plurality vote of its K nearest neighbors, where K is the only parameter required to perform the algorithm and it's usually learned.

However, the purpose of this project is not to learn the best parameter K, which in our case is stated by the user, nor classifying a new data-point.
We want to implement an algorithm that finds the closest K points for each input data-point, writing to an output file the points' coordinates and the relative distances. The implementation must also use parallel algorithmic skeletons in order to minimize the execution time of the algorithm, trying to maximize the speedup of the parallel implementation with respect to the sequential one.
A very inefficient pseudocode of the algorithm is:

Algorithm 1 Output to file the K-Nearest Neighbors of each data-point of the dataset

```
function KNN(dataset, k)
    for x ∈ dataset do
        for y ∈ dataset \ x do
            d ← DISTANCE(x, y)
            INSERT(k_nearest_neighbors, ⟨y; d⟩)
        end for
        PRINT_TO_FILE(file, x, FIRST_K_ELEMENTS(SORT(k_nearest_neighbors), k))
        CLEAR(k_nearest_neighbors)
    end for
end function
```

A way more efficient solution will be provided later in this report. In any case, as it's clearly visible from the pseudocode, the algorithm's time complexity is dominated by the nested `for loops`, that implies that the algorithm will run in quadratic time.

2 Implementation

2.1 Generating the space

The first step to perform the algorithm is to obtain a space containing the data-points. For this project it was required to deal with 2-dimensional points, hence a 2D space was generated using a Python script which, given the number of data-points to be generated and an upper and a lower bound for the values of the coordinates, generates N data-points from an uniform distribution. In this case, since the points lie on a 2D space, the measure of distance taken into consideration is the *Euclidean Distance*.
The points generated are saved to `data/inputs.txt`.

2.2 The sequential solution

Without diving yet too much into implementation details, we can divide the algorithm into three fundamental steps:

1. loading the dataset: we need to obtain a data structure containing the input data-points, that in this case are two dimensional points, hence representable as a pair of floats;

2. computing the closest K points for each point of the dataset: this is the most computational expensive step, since -as we saw in the pseudocode- it runs in quadratic time;

3. writing the results to the output file: once the collection of the closest K points -and relative distances- to the point we're considering is computed, it has to be written to the output file used to store all the results.

Without thinking about how we could parallelize the steps, let's first try to find an efficient solution to the problem. The most important consideration one should observe is that K is usually very much smaller than the cardinality of the input space. This is due to two main reasons: firstly, we want the input space as large as possible, since it represents our "training set"; secondly, the higher the value of K the less the relevance of the distance between the points is. A very high K implies that also distant points will contribute to the plurality vote for the classification of a given point, which is generally bad since the aim of the KNN algorithm is to classify data-points using their spacial representation.
Under this reasonable assumption, we should agree on the fact that we don't need to store all the points -and relative distances- while looking for the KNNs of a given point, like in the pseudocode presented in the introduction. Even better, we know that the data structure used to store the KNNs won't have a large size, which is ideal because that collection -of fixed size- will be used for insertion and will be eventually sorted.

In fact we can also avoid sorting the whole data structure, by simply keeping the data structure sorted while inserting new elements, which is exactly the point of a *priority queue*, which is the data structure of choice for this project's implementation.

Algorithm 2 Output to file the K-Nearest Neighbors of each data-point of the dataset

```
function KNN(dataset, k)
    for x ∈ dataset do
        for y ∈ dataset \ x do
            d ← DISTANCE(x, y)
            if SIZE(k_nearest_neighbors) == 0 then
                INSERT(k_nearest_neighbors, ⟨y; d⟩)
                continue
            end if
            if SIZE(k_nearest_neighbors) = k ∨ d > max(k_nearest_neighbors) then
                continue
            end if
            SORT_INSERT(k_nearest_neighbors, y)
            if SIZE(k_nearest_neighbors) then
                POP_BACK(k_nearest_neighbors)
            end if
        end for
        PRINT_TO_FILE(file, x, k_nearest_neighbors)
        CLEAR(k_nearest_neighbors)
    end for
end function
```

This implementation offers two main advantages: firstly, we we'll never sort the data structure, that would take $\mathcal{O}(k \log k)$; secondly we don't need to linearly scan the structure to know the maximum distance, hence it takes $\mathcal{O}(1)$ to check if we should insert or not the new point into the KNNs.

We can formalize the Completion Time T_c for this implementation as the sum of the three steps explained before:

$$T_c = T_{input} + N \times (T_{knn} + T_{output}) \tag{1}$$

where:

- T_{input} is the time required to load all the data-points from the input file;
- T_{knn} is the time required to compute the KNNs for a given input data-point;
- T_{output} is the time required to write the results to the output file;
- N is the number of points in the space.

2.3 Parallel Design

Now that we achieved an efficient sequential implementation, we can reason over the design of the parallel implementation.
Even if it's possible to mimic an input stream using the input file, the simpler and more obvious observation is that we're dealing with a *Data Parallel* problem. Furthermore, it's actually an **embarassingly data parallel** computation, since all the computations for each point of the space are completely independent from the others. One could argue that given two arbitrary points, there's no need to compute twice their distance and for this reason some kind of communication between the threads should occur. Even if it's logically true, this approach doesn't guarantee a speedup, on the contrary it would severely slow the computation for the amount of overhead given from the communication and the synchronization required to implement such logic.
Since we're dealing with an embarassingly data parallel problem that presents two nested loops, we should choose one of the following three approaches:

1. parallelize the outer cycle: this solution can be seen as dividing the space in nw areas with the same cardinality, where nw is the number of workers available at runtime. Hence, each worker will be in charge of a set of points and will compute the KNNs of such points sequentially;

2. parallelize the inner cycle: each point is accessed sequentially, however the computation of its KNNs is done in parallel, using nw workers;

3. parallelize both: parallelizing both cycles using nw workers in an efficient way.

The first implementation doesn't require any kind of synchronization to compute the KNNs. This is due the fact that the data structure that contains the space of datapoints is used as read only, to acquire each point assigned to the threads. At the same time each thread will have an unique priority queue used to store the KNNs computed for a given point, hence there's not any need of synchronization for the computation of the KNNs. The second implementation is in fact slower, since it requires synchronization over the priority queue used for storing the KNNs of a given point. Since multiple workers will compute distances between the point of reference and the other points, they'll need to access and write on the priority queue and that would add time expenses due to the synchronization overhead. The third implementation is in between the first two: if we consider S a parameter of the implementation as a factor that determines the ratio of distribution of workers between the inner and the outer cycle than this implementation can behave just like implementation 1 or implementation 2. With a proper S we could observe some improvements with respect to implementation 1 since we're trying to parallelize all the sequential portions of the algorithm, however we still need to consider the synchronization overhead that afflicts implementation 2.
At the same time we can consider that each worker will compute the KNNs of a given

point in approximately the same amount of time required from the other threads, since the number of operations is the same. For this reason a dynamic load balancing policy is not required, thus the implementation uses a static, block distribution policy that performs better in the case of uniformly distributed execution times.
The formalization of the Completion Time T_c for this parallel implementation is:

$$T_c = T_{input} + T_{overhead}(nw) + \frac{N}{nw} \times (T_{knn} + T_{output}) \tag{2}$$

where:

- T_{input} is the time required to sequentially load all the data-points from the input file;
- $T_{overhead}(nw)$ is the time needed to allocate and start the threads and synchronize;
- T_{knn} is the time required to compute the KNNs for a given input data-point;
- T_{output} is the time required to write the results of a point to the output file;
- N is the number of points in the space;
- nw is the number of workers.

2.4 C++ Threads Implementation

Using the standard C++ thread library, two implementations have been made, both conceptually based of Algorithm 2.
The first implementation, called `parallel.cpp` is the implementation that parallelizes only the outer cycle of the algorithm, as explained in the parallel design subsection. In the code, the `k_nearest_neighbors` data structure has been implemented as a `std::vector`, however the `SortInsert` function makes the container in fact a priority queue. I preferred to not use a standard C++ `std::priority_queue` because I found that the standard implementation was slower, perhaps because it introduces too much overhead -not from a parallel point of view- from managing the underlying container and `Compare` method. The number of workers and K points are given from the command line and the space of points is acquired from `data/inputs.txt` -or from a command line argument- and it's not synchronized because, after the acquisition of the points, it's used as a read-only data-structure. The last thing to notice is that a `std::mutex` is used to synchronize the output file, since the workers will access it to write the results of the computations.
The second implementation, called `double_par.cpp` is the implementation that parallelizes both cycles, as explained in the parallel design subsection. It's obviously based on the `parallel.cpp`, with the difference that each thread that manages the parallelization of the outer cycle also starts a number of threads that are in charge to compute the nearest neighbours of a given point. The data structure that contains the results is shared

by the threads, hence the need of a `std::mutex` for synchronizing the access to it. The program takes an additional parameter S which can be seen as a *"scaling factor"* which establishes the ratio between the number of workers that will manage the outer cycle and the number of workers that will manage the inner cycle.

2.5 Fastflow Implementation

The Fastflow implementation is very similar to the parallel C++ threads implementation presented in the last subsection, given the fact that in our case the framework has been used only for parallelizing the outer cycle of Algorithm 2, using the `ParallelFor` abstraction.

2.6 Benchmarking

Three bash scripts have been made to benchmark the three implementations, using the outputs of the executables. Each implementation used the `lib/utimer.hpp` library that prints to the standard output a message containing the completion time of the executable. The scripts `scripts/*_benchmark.sh` run the executables N times -where N is specified from the user- and print out the mean execution time computed.
Even if in the Fastflow implementation it was possible to use the `ffTime` function, I decided to use `lib/utimer.hpp` to keep the measurements as homogeneous as possible.

3 Results

3.1 Expected Results

Before diving into experimental results, let's try to formally figure out the expected results we should obtain from the standard C++ threads and Fastflow implementation with respect to the sequential one. We already stated the Completion time of the sequential implementation (eq. 1) and the one of the parallel implementation (eq. 2). We can now formalize some derived measures to predict the behaviour of the implementation. We can formalize the *speedup s(n)* as:

$$s(n) = \frac{T_{input}+N\times(T_{knn}+T_{output})}{T_{input}+T_{overhead}(nw)+\frac{N}{nw}\times(T_{knn}+T_{output})} \approx \frac{nw}{T_{overhead}(nw)} \tag{3}$$

hence the speedup is ruled by the number of workers nw and the amount of overhead they introduce. Ideally the speedup will tend to nw. The same reasoning can be done for the *scalability*, because in this case $T_{par}(1) \approx T_{seq}$.
Also, we can formalize the *efficiency* ϵ, using $s(n)$, as:

$$\epsilon(n) = \frac{nw}{T_{overhead}(nw)} \times \frac{1}{nw} = \frac{1}{T_{overhead}(nw)} \tag{4}$$

which is a very powerful metric that proves that the only element that can decrease the efficiency of the implementation is the overhead, which in our case can be seen as the sum of the time required to allocate the threads and the time required by the synchronization.

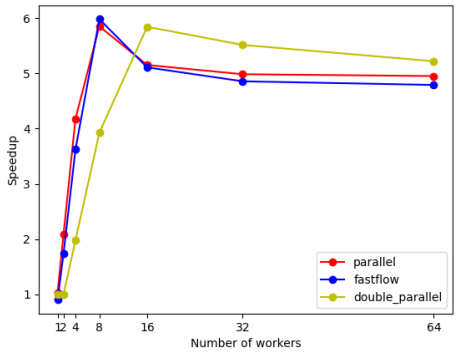
3.2 Real Results

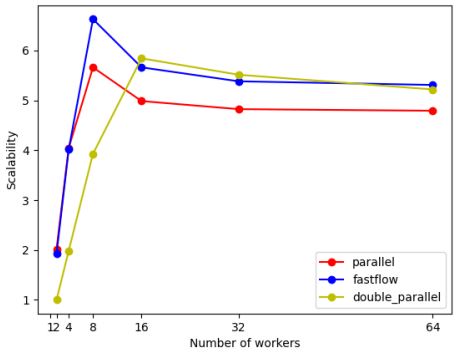
When measuring the performances of the implementations one should be very aware of the stochasticity of the measurement, since the results may vary based on low level factors, like the scheduling done by the operative system, the usage of the resources at a given time, background processes etc. To overcome this problems, all the measurement reported in this document have been performed using the scripts stated in section **Benchmarking**. All the results reported here have been acquired from the execution of such benchmarking scripts on a machine with a clock speed of $@1.4MHz$ with 256 processing elements distributed over 64 real cores.
The following table of results has been produced considering $K = 20$, 10 iterations per result and $50'000$ points in the input space:

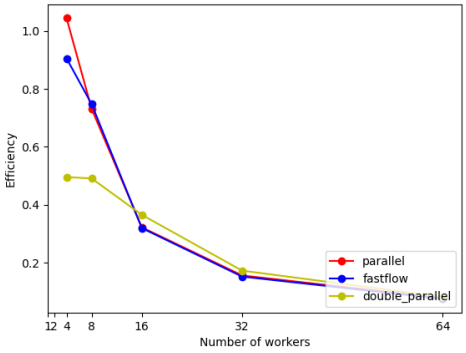
#threads	parallel	fastflow	double parallel
sequential	135352219		
1	131005715	150016332	-
2	64857780	77806088	-
4	32390563	37358656	68271533
8	23137265	22632509	34457520
16	26275162	26495230	23166295
32	27156176	27872661	24547015
64	27338698	28254441	25933739
128	27643799	28438641	25803273
256	27397251	28665697	26297492

Table 1: Results obtained with $K = 20, N = 50000$

From this data we can plot the speedup and efficiency:
Please note that to make the plots more visible I cut off the points for $nw = 128$ *and* $nw = 256$ *because, as stated in the table, their values are approximately the same*







3.3 Discussing the Results

From the data it's clearly visible how the speedup, the scalability and the efficiency of the program are very constrained. In fact the speedup curve increases up to a factor of approximately 6, decreases a bit and then remains constant. The first intuition that may occur is that this behaviour is caused by the increasing of the overhead produced by the number of threads in use. Even if this is obviously true, we can identify a stronger cause. If the overhead was the real problem in this case we should expect that the curve would have decreased when the number of threads was increased, but that's not really the case since the curve becomes pretty much a constant after the point of maximum speedup reached.

The main issue in this particular case is the cost of the sequential portion of the execution, which is very hard to overcome. As discussed in the Parallel Design section, parallelizing only the outer cycle doesn't require any synchronization on the data structure, however that means that the inner cycle -that in the results presented has length of 50′000 iterations- will be computed entirely sequentially which, on the machine taken into consideration, it's an enormous effort. At the same time, parallelizing the inner cycle requires synchronization on the data structure and the overhead caused by that

increases with the number of threads in use. Even in the case of the implementation that parallelizes both cycles, both problems appear.
There seems to be a very delicate trade-off between the amount of sequential work and the amount of overhead due to synchronization, that results in poor scalability and efficiency performances for all the solutions tested.
In other words, taking into consideration the Amdahl's law, we could say that the sequential portion of computation f has a large enough impact on the computation to quickly overwhelm the benefits of parallelizing the portion $1 - f$.

3.4 Another possible approach

Another solution that came to my mind to reduce as much as possible the sequential portion of the algorithm while lowering as much as possible the synchronization overhead was to use `std::unordered_map` to store local minima found by the threads. This implementation would behave just like `double_par.cpp`, with the exception that instead of using a `std::mutex` to synchronize the workers that compute the minima of a given point, the same workers would use a local minima data structure to obtain the K local results and then add that data-structure to an `std::unordered_map`. At the end of all the iterations that hashmap would contain $nw \times N$ data structures containing the local minima computed without the need of synchronization. Considering that a this point all the workers are free, it would be possible to assign to each worker a subset of points and in parallel merge the results stored in the hashmap. Then, as always, synchronize the output to the file.
This solution could get better performances in terms of scalability because:

- the pure sequential portion of code is limited to merging nw `std::vectors` of size K -which is typically very small, in the order of 10^1-, which could imply huge benefits;
- the cost of the sequential portion of code is $\mathcal{O}(K \times N)$ where N is the number of points assigned to each thread in the phase of merging the results, which is not very large since K is typically very small and N inversely proportional to the number of workers;
- in this case there's no synchronization on the data structure since the workers use local structures. One could argue that the access to the hashmap should be synchronized because they're not thread safe: even if that's true, inserting elements in parallel using different keys should never result in error, provided that the hash function is correct.

Unfortunately I didn't have the time to realize a working implementation of the approach, hence I couldn't produce its results.

4 Folder Overview & Commands

4.1 Make and Running Benchmarks

To make all the executable simply run `make` from `/SPM-Project`. To make only a specific executable please run:

- `knn_sequential` to build the executable of the sequential implementation;
- `knn_parallel` to build the executable of the standard C!! implementation;
- `knn_fastflow` to build the executable of the Fastflow implementation;
- `knn_double_parallel` to build the executable of the standard C!! implementation that parallelizes both cycles.

To compute the average time of execution please run navigate to `/SPM-Project/scripts` and run:

- `sequential_benchmark` to benchmark the sequential implementation;
- `parallel_benchmark` to benchmark the standard C++ implementation;
- `fastflow_benchmark` to benchmark the Fastflow implementation;
- `double_benchmark` to benchmark the standard C++ implementation that parallelizes both cycles.

Please remember that `sequential_benchmark` requires as parameters `number of iterations K [optional]input_file` while all the other scripts require as parameters `number_of_iterations K number_of_workers [optional]input_file`. Please keep in mind that -if specified- the `input_file` must be present in `/SPM-Project/bin`.
To generate data please run `python3 /SPM-Project/scripts/generate_data.py [optional] -n <#points> -l <lower bound> -u <upper bound>`

4.2 Folders content

- `bin` contains the executables and some useful pre-made input files;
- `data` contains output data and generated data;
- `fastflow` contains a local version of fastflow. In the folder received it's setup for my machine;
- `lib` contains useful libraries used in the implementations;
- `scripts` contains scripts to benchmark and generate new data;
- `src` contains the source code of the implementations.