Parallel and Distributed Systems project report K-Nearest Neighbours

Ristori Alessandro
Master Degree in Computer science.
a.ristori5@studenti.unipi.it, 619618
Parallel and Distributed Systems, Academic Year: 2020/2021
Date: 3/02/2022

Contents

1 Introduction

The K-Nearest Neighbours is a well known classification method used to classify datapoints given their spatial representation without the complexity of models such Neural Networks. A point is classified in a determined class by a majority vote of its k neighbours, where k is an hyperparameter and the only one that KNN needs.

The aim of the project was not a classification task or to learn the best value of k, but to find the nearest k neighbours for all the points in a 2d space and writing the results into another file. The input dataset is composed by pairs of coordinates which represents the points, the output file contains an ordered list for each point with their k nearest neighbours ordered by distance. Moreover, the main purpose of the project was to find a way to parallelize this modified version of KNN with both C++ standard library threads and FastFlow library.

2 Implementation

In this section I'll discuss of the main implementation choices taken during the development of the project by also giving an overview on the method itself.

2.1 KNN overview

Before we dive into the heart of the section we need to contestualize KNN inside the project. KNN is a perfect example of **Data Parallel problem** and, more precisely, an **embarassingly parallel** one, because the **computation of a subtask is independent to the others**, therefore those subtasks don't need any kind of communication or synchronization. The task can be summarized as the composition of three major steps:

- **Read all the points** on a 2d space from a file.
- **Compute KNN**.
- **Write results** on file.

Without a doubt the most computationally expensive and interesting part is the second one, which I will talk more about on subsection 2.2.

2.2 Sequential implementation

As I said at the end of section 1, the project required two different versions of the same task, one using C++ threads from stdlib and one using FastFlow, but both share the same main structure and, therefore, I thought that it would be useful to show it in algorithm 1 (which is the sequential version of KNN method) in order to better understand how it works.

Algorithm 1: KNN sequential

```
   Input: txt file containing a pair of coordinates for each row representing the points,
          the index of a point is the row on which such pair is located and the value of k
   Output: string which will be written on a file where the i-th row is in the following
           format: $point_i :< neigh\_id_1, distance_{i1} >, ... < neigh\_id_n, distance_{in} >$
 1 Read points from file and store them into points
 2 Priority queue knn
 3 String knn_results
 4 for point i in points do
 5     for points j in points do
 6         if $i == j$ then
 7             continue
 8         calculate d(i, j)
 9         if $knn.size() < k$ then
10             knn.emplace(point[j])
11         else if $d(i, j) < knn.top()$ then
12             knn.pop()
13             knn.emplace(point[j])
14     while !knn.empty() do
15         p = knn.pop()
16         knn_results.append(to_string(p))
17 Write knn_results to file
```

The main choice was to **keep a single priority queue globally** with k as its dimension, which is emptied at the end of the first inner loop in order to write the results in the final output string, this is surely better than keeping as much priority queues as the number of points, which would have been a serious waste of space the more the latter went up.

Moreover, the priority queue was preferred since it can retrieve istantly (i.e.: with constant cost) the most distant point in the structure so the only thing that has a notable cost is the reestablishment of the priority condition which is $O(log(k))$, way better than $O(klog(k))$ that would have taken by ordering a normal array of k points.

2.2.1 Sequential cost

The total cost of the first inner loop is given by the cost of computing distances, $O(n)$, and the cost to mantain the priority condition for a complexity of $O(nlog(k))$, this part of the method will be known as *knn* and its completion time will be T_{knn} from now on; at the same time, the building of the ouptut string has cost $O(klog(k))$ and its completion time will be known as $T_{BuildString}$, so the final cost of the KNN computation is:

$$O(n(nlog(k) + klog(k))) = O(n^2log(k) + nklog(k)) = O(n^2log(k))$$

The previous statement holds always true if $k << n$, which is the case of this project; we can now define the completion time of the sequential version as:

$$T_{seq} = T_{Read} + n(T_{knn} + T_{BuildString}) + T_{Write}$$

where n is the number of points, T_{knn} is the time spent on finding the k closest points and $T_{BuildString}$ is the time spent to build the output string. T_{Read} and T_{Write} are autoexplicative and they represent the **serial part** of the source code.

2.3 Parallel implementations

As required from the project, I developed two more implementations of KNN, one using C++ threads and the other using the FastFlow library.

2.3.1 C++ threads

The parallel implementation with threads from the standard library of C++ is straightforward since it is nearly the same as the sequential one with some tweaks. Each thread has assigned the same number of points in an extremely easy way, given nw as the number of workers. the **workload** is defined as $\frac{n}{nw}$ and the **excess amount of work** as $n\%nw$ which needs to be redistributed between the threads, so at maximum we need to redistribute $nw - 1$ of excess workload and, since $nw << n$, this is highly negligible with respect to the KNN computation.

Each thread knows where its partition starts and ends given its workload and id, therefore the threads work on their partition at the same time just like a *Map* template. Moreover, **each thread returns a string** (just like for the sequential implementation) which is saved in an array in position i that corresponds to the thread id and at the end the results are written on the output file by combining one string at time, like in a *Reduce* fashion.

The only cons from this implementation come from the **increased number of priority queues**, which is now the same as the number of threads, and the **overhead cost** since we need to split and merge the work.

2.3.2 FastFlow

The fastflow implementation is nearly identical to the C++ threads one and even simpler since I did not need to load balance the workload manually. I opted to use a **ParallelForReduce** since it works in the same way as I did for the stdlib threads version (i.e.: a Map followed by a Reduce, keep in mind that in this way the Map's collector is substituted by the Reduce).

2.3.3 Parallel cost

Both C++ threads and FastFlow implementations rely on algorithm 1, so their cost is similar but, since the outer loop can be done in parallel now, the weight of the main argument that contributes the most (i.e.: $n(T_{knn} + T_{BuildString})$ is now divided by the number of workers.

Even though the weight of the most expensive part is alleviated, **the cost coming from the task subdivision and final merge should be taken into account as pure overhead** $T_{ov} = T_{Split} + T_{Reduce}$. Hence, the total cost of the parallel implementations can be defined as:

$$T_{par}(nw) = T_{Read} + nwT_{ov} + \frac{n(T_{knn}+T_{BuildString})}{nw} + T_{Write}$$

Where nw is the number of workers or, more technically speaking, the **parallelism degree** and everything else was the same as discussed during the sequential version.

3 Results

This section will show what kind of setup was used to test the performances of our implementations and what results were achieved.

3.1 Expected results

First of all **I expected the completion times of both implementations to be somewhat similar** given their same structure, this implies that they will also have some **similarity on their speedup and others metrics**.

Speaking about the speedup, we know that it is equal to $\frac{T_{seq}}{T_{par(nw)}}$ and that the ideal one is exactly the parallelism degree, nw; this means that, in a real world environment, the achievable speedup is $s(nw) < nw$, let's check it this fact holds:

$$s(nw) = \frac{T_{seq}}{T_{par}} = \frac{T_{Read}+n(T_{knn}+T_{BuildString})+T_{Write}}{T_{Read}+nwT_{ov}+\frac{n(T_{knn}+T_{BuildString})}{nw}+T_{Write}}$$

It is safe to assume that T_{Read} and T_{Write} are negligible with respect to the total completion time so everything reduces to:

$$s(nw) = \frac{n(T_{knn}+T_{BuildString})}{nwT_{ov}+\frac{n(T_{knn}+T_{BuildString})}{nw}}$$

And this is surely less than nw as we know from the lectures.

3.2 Setup

Before we talk about the results I decided that it was necessary to explain some choices beforehand:

- The results were all obtained on the **64-core with four-way hyperthreading Intel XEON-PHI**.
- For each configuration **I did not take into consideration T_{Read}, T_{Write} and T_{ov}** since they were nearly the same for each configuration with the same amount of points, moreover the most interesting part is the *non-serial* one, which is $n(T_{knn} + T_{BuildString})$.

- **Each configuration was run ten times**.
- The **chosen value for k was 10 for all the configuration**, hence all the results in the next subsections are averages.
- The **parallelism degree was decided to be** 2^i where $i \in \{0, .., 8\}$.
- **Times were obtained using the utimer class** that the professor showed during the lectures.

3.3 Results on the XEON-PHI machine

I decided to run tests on 2d spaces with **10k, 20k, 50k and 100k points** where every point was drawn by an uniform distribution over [0, 10]; the number of points was decided as such in order to see **how each implementation perform going from a sparse spaces to densely populated ones**. First of all it was necessary to calculate the amount of time spent by the execution of the sequential knn version in order to get values needed to obtain the other metrics (e.g.: speedup and efficiency), Table 1 shows the results achieved by such implementation.

Sequential 10-KNN execution times (μsec)			
10k points	**20k points**	**50k points**	**100k points**
4187051	15973321	95360616	375058248

Table 1: Results with the sequential implementation.

As one would expect, the time does not scale linearly with the number of points since the cost of algorithm 1 is $O(n^2 log(k))$, something more interesting arose during the execution of C++ threads and FastFlow versions whose results are shown in Table 2.

Parallel and FastFlow 10-KNN execution times (μsec)								
nw	**stdlib threads**				**FastFlow**			
	10k	**20k**	**50k**	**100k**	**10k**	**20k**	**50k**	**100k**
1	4195076	16573213	98816008	377017508	4198343	15989082	96277045	406293436
2	2150114	8006220	50251728	201949754	2124308	8018496	48329593	201459406
4	1324868	4072286	27676599	99481003	1072248	4034389	2422181	99968804
8	726914	2030168	13578018	51113708	539050	2021035	12098202	50136748
16	322046	1087086	6741971	24312895	316082	1062674	6183317	26882276
32	182296	677542	3472759	14382776	210621	588005	3461644	14591443
64	133958	434134	2151317	8876960	141363	400733	2278688	9016368
128	99786	308973	1590471	6123643	106730	318886	1665072	6326216
256	99371	256716	1046949	3969818	95743	274393	1122244	4312146

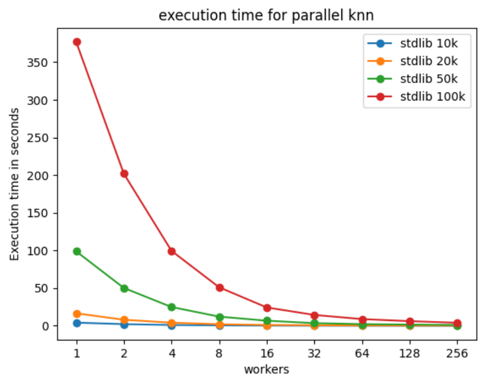
Table 2: Results with the stdlib threads and FastFlow implementations.

3.4 Results analysis

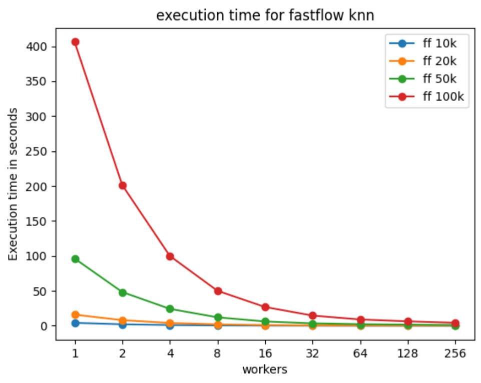
As first thing I want to underline is the fact that **the results are extremely similar between the two implementations** since, as I said in subsection 2.3, the two versions are basically the same with some minor changes.

3.4.1 Execution times

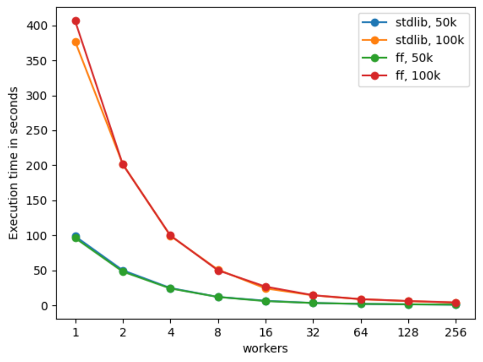
The execution times comparison (only for 50k and 100k points) is shown in Figure 1



(a) C++ threads



(b) FastFlow

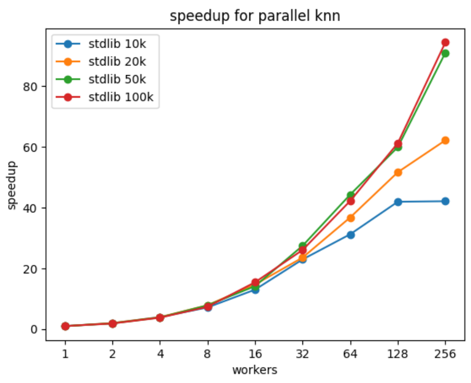


(c) Comparison

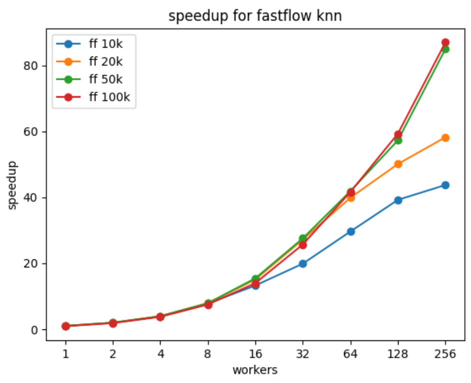
Figure 1: Execution times for C++ threads and FastFlow versions and their comparison

3.4.2 Speedup

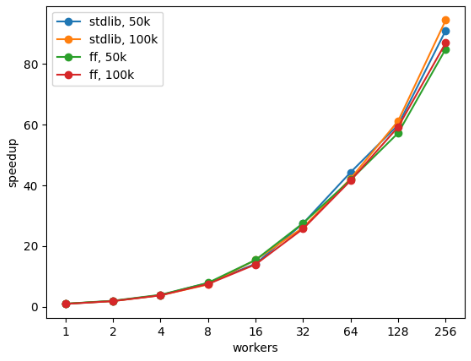
Let's now see and compare the speedup obtained by both implementations as shown in Figure 2



(a) C++ threads



(b) FastFlow



(c) Comparison

Figure 2: Speedup for C++ threads and FastFlow versions and their comparison

The **speedup seems to scale well with the number of workers as long the number of points goes up**, in fact I could clearly see that for sparse 2d spaces (e.g.: 10k and 20k points) **the speedup seemed to reach a sort of plateau** the more I increased nw.

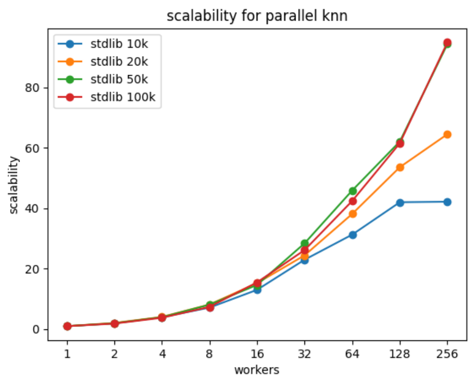
Since the execution times are the same **it was natural that the two versions would have similar speedup**, this will be the same for the next metrics that we are going to see.

3.4.3 Scalability

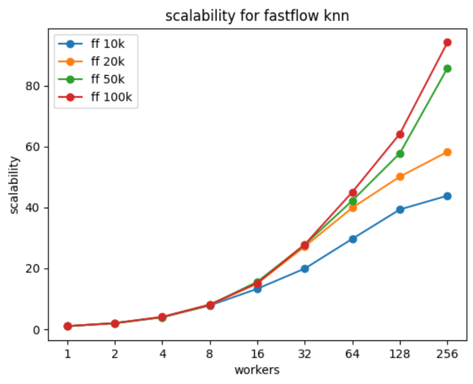
Scalability is the ratio between the parallel execution time with parallelism degree equal to 1 and the parallel execution with parallelism degree equal to nw, the ideal value is the same as for the speedup (nw).

$$scalab(nw) = \frac{T_{par}(1)}{T_{par(nw)}}$$

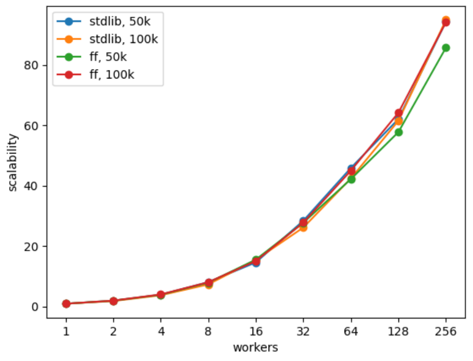
It measures how the parallel implementation achieves efficiently better performance the more the parallelism degree increases.



(a) C++ threads



(b) FastFlow



(c) Comparison

Figure 3: Scalability for C++ threads and FastFlow versions and their comparison

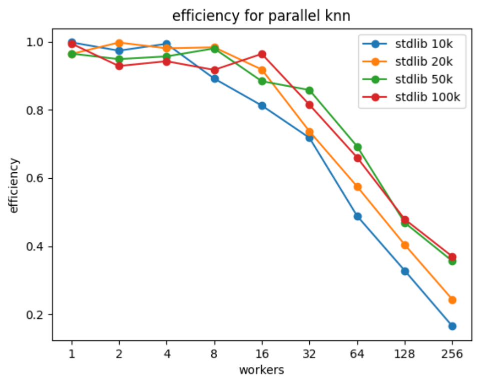
What we have discussed briefly during the speedup analysis holds for the scalability too.

3.4.4 Efficiency

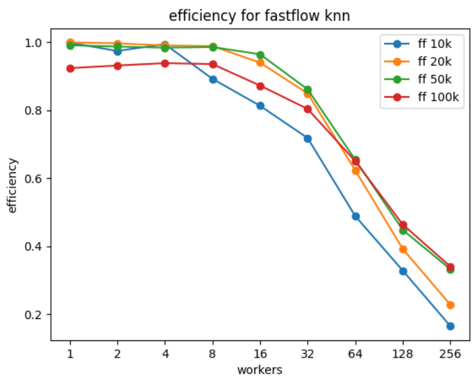
As last metric of the analysis it was natural to choose efficiency which is the ratio between the ideal execution time and the actual execution time.

$$\epsilon(nw) = \frac{T_{id}(nw)}{T_{par}(nw)} = \frac{T_{seq}}{nwT_{par}(nw)} = \frac{s(nw)}{nw}$$

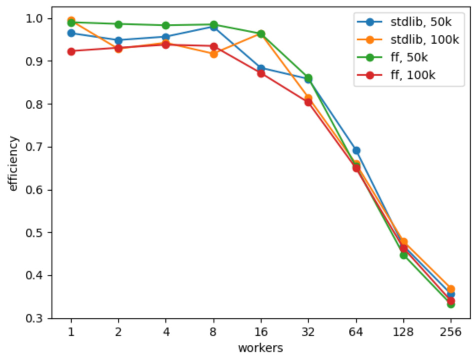
The formula holds since $T_{id} = \frac{T_{seq}}{nw}$, moreover **efficiency measures how a parallel implementation is making use of the available resources**. The ideal value for the efficiency is 1 given the fact that the speedup is bound by nw.



(a) C++ threads



(b) FastFlow



(c) Comparison

Figure 4: Efficiency for C++ threads and FastFlow versions and their comparison

The most interesting thing is the fact that the efficiency has a strange plot from 2 to 16 workers as shown in Figure 4, this could be caused by the congestion on the XEON-PHI machine from other students while I was working on it (but I do not want to blame them as everyone needs to do project) or by routine tasks of the machine itself.

Another thing is the fact that **configurations with more points had a better efficiency than the ones working on sparse spaces**, but this comes to no surprise since the former had an higher speedup than the latter.

4 Improvements

Meanwhile I was working on the project many other solutions or improvements came to my mind that at the end were not implemented for various reasons, in this subsection i will explain some of them.

4.1 Parallelize everything

First, is possibile to parallelize both the outer loop and first inner loop from algorithm 1, however this comes with the price that the threads working on the inner loop have to insert or remove points from the priority queue by ”communicating” with the others, this implies an higher overhead. I could also parallelize the reading and writing part but since their cost was small with respect to the KNN computation I thought that it would not have any major improvement.

4.2 Compute more efficiently the distances

Then it is possible to compute the distances by calculating the upper or lower triangular distance matrix, but even this solution comes with an higher overhead since a thread would need to access two priority queues at the same time (for two different points i and j) and one of them could be accessed at the same instant by another thread (this is the same ”communication” problem as before). Moreover the computation of the distances in this way has the same cost of my solution, $O(n^2)$, so it was discarded right away.

5 Running the project

To build everything needed to test the project you can make use of the Makefile just by typing *make* once you have extracted the zip file.

Afterwards, you can generate datasets of points with *generate_points.py* using it in the following way (it will create 10000 points by using a uniform distribution between 0 and 10 if no arguments are passed):

```
python generate_points.py --n 100000 --file datasets/points.txt
```

Then you can run every implementation easily, the following are just examples:

```
./knn_sequential datasets/points.txt 10
./knn_parallel datasets/points.txt 10 256
./knn_ff datasets/points.txt 10 256
```

For each implementation the second argument is the value of k, meanwhile the third argument, for both stdlib threads and FastFlow implementations, is the number of workers. If you want to execute more times the same implementation or to execute more than one at time, use the *compute_knn.py* method:

```
python --file datasets/points_10k.txt --k 10 --nw 256 --runs 10 --execute spf
```

The last argument specifies what versions to run (s for sequential, p for parallel and f for FastFlow).