Parallel and Distributed Systems final report: K-NN

Master degree in Computer Science

Teachers: Prof. Marco Danelutto, Prof. Massimo Torquati

UNIVERSITÀ DI PISA

Fabio Murgese

April 1, 2022

Contents

1 Overview

This is the final report of `Parallel and Distributes Systems: Paradigms and Models` course. The task is to code a parallelized version of the *K-Nearest Neighbors* algorithm: one version using standard threads (`STL`) and another version using the FastFlow (`FF`) library. The input is a .csv file where there are stored a set of points in 2D space with each i-th point saved in a different line; I have to compute in parallel the set of *k-closest* points for each one of the input points, ordered by distance. The output file contains line per line the different sets of neighbors for each of the points, identified by ids.

2 Implementation design

2.1 Design choices

The most important thing to notice at first, it's that our K-NN is a data parallel problem or, to be more precise, an *embarrassingly-parallel problem* because there is little to no effort to separate the problem into a number of parallel tasks: in our case, the computation of an item is totally independent from the computation of another one. As specified in the overview we will see two different implementation of this algorithm: one implemented using the *StandardLibrary* and one implemented using *FastFlow*. For now, let's discuss the `Sequential` version, made to create the main structure to be parallelized then, and also to see timings in each part of the program. In this way we could highlight weaknesses of the implementation like bottlenecks and target this pieces of code with some improvements.

2.1.1 Sequential version

The actual implementation of the sequential version takes into account a partition of the problem into three different subproblems: "**Read**", "**Compute**" and "**Write**". The reading section is accounted for reading the input .csv file and populate a data structure - a vector of pairs; the .csv file is structured in order to have the i-th point at the i-th line of the file defined as a couple of coordinates `<x, y>`.
After the reading is completed, we have the computing step: for each data point in our vector of points we compute the distance between points, with the only foresight to not compare

the same point with itself. The distance computed is the well know *Euclidean distance* for two dimensional points (in the formula below p and q).

$$d(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2} \tag{1}$$

Right after this step, we need to sort our vector of neighbors in order to retrieve the first k elements that can be considered *nearest* neighbors of the point taken into account. This process has been carried out by using the *make_heap* function starting by the list of all neighbors and then the *sort_heap* function to effectively make the sorting. We exploited these two commodities of the `<algorithm>` library in order to avoid defining a specific comparator function to manage the comparison operator between the two elements of the sorting. By creating a Min Heap only in the end of the process - when we already have all the neighbors stored in a vector - may result in a waste of memory because of the double storage of the neighbors; but, in this way, we compute and store all the distances for each of the data point and we could vary the k parameter without having to compute again the distances to retrieve different *k-nearest* neighbors: we just need to take the *first-k* items from the just built Heap (but it's not possible for the actual implementation of the algorithm).
In the end, we write to file the list of nearest neighbors for each of the points in the data structure.

2.1.2 STL version

As we have seen in the sequential version of the program, the first part is the data reading. We need to wait until the *ReadFile* task is finished: we need all the points to start computing the various distances. So, we have a first T_{reader} that we cannot skip.
Then, we can start populating our thread pool with nw workers: we could exploit a `Farm` pattern thanks to the fact that, after having read all the data points, we could emit tasks to be given out. But in this specific case, all points are indexed in a data structure and distances have to be calculated, so there is no need to *emit* tasks: they can be seen as integers to be computed, ranging from 1 to N - being N the size of our data structure. We can notice that in this scenario, we don't need to wait for an emitter thread to distribute tasks, nor a queue to pop tasks from. In this way, each of the tasks will need to compute the exact same thing: they will take more or less the same amount of time. So, each worker will take its $worker_{ID}$ as a starting point and iterate until it reaches the end of the data structure, incrementing each time its index by nw - the number of workers. Once the KNN of a single node is computed, the worker will send the result to the `Writer` - our *Collector*

thread - having solved the mutual exclusion problem of multiple workers accessing a single queue by giving to each of the worker a different deque to write out. After this, the worker move on to the next data point and repeat the process.
In the end, the Writer has just *nw* queues to look for: it tries to *pop* from a specific queue cycling through the queues. Being the job of the Writer shorter than the computation task of the other workers, it's been no necessary to introduce a *sleep time* for the Writer.

2.1.3 FF version

For what concerns the *FastFlow* implementation we have a `ParallelForReduce` structure. In a nutshell, we first build a sequence of neighbors for each data point (the proper `ParallelFor` step) by exploiting a Min Heap as for the `STL` version; then, we reduce these partial results, in the form of strings, by creating a single string from each of them by their concatenation. We have the same Reading part again. After this, the ParallelFor paradigm starts and with it the lambda function that defines its behavior: it computes all the tasks and append the results to a local string. This string will then be passed to the Reduce part, responsible for the concatenation of these partial strings to the final one, in the end written to the output file.

2.1.4 Improvements

- Writer (`STL`, `FF`): append to a local string instead of writing to the output file, in order to reduce the number of system calls. Low impact on performances: the Writer isn't a bottleneck in our system (not even with the 256 threads from the Xeon PHI).
- Pinning threads to specific cores with *CPU affinity* (`STL`). Medium impact on performance (up to $\simeq 34\%$). The reason to pin threads to specific CPUs is that if a thread happens to move due to (de)scheduling policies to another CPU, data needs to be transferred as well. Maybe in some cases may end up in close proximity, accessing lower levels of caches, but in other cases it could impact the performace a lot.

```
[f.murgese@c6320p-2 knn]$ ./average_time.sh knn_stl input_huge.csv 10 64 3
3503.51 ms
3476.01 ms
3666.47 ms
On average: 3548.66 milliseconds
```

Figure 1: Without thread pinning.

```
[f.murgese@c6320p-2 knn]$ ./average_time.sh knn_stl input_huge.csv 10 64 3
2659.32 ms
2653.34 ms
2615.26 ms
On average: 2642.64 milliseconds
```

Figure 2: With thread pinning.

Another possible improvement that could be carried out in the future is the use of a MinHeap of just k elements (it would affects both `STL` and `FF` versions). As stated previously, I left a bigger Heap that includes all the N elements to save one time all the neighbors and their relative distances in order to have the possibility to change dinamically the k parameter and retrieve different *K-NN*.

3 Expected performances

As previously discussed, Reading is unavoidable, hard to parallelize and very fast to be done, so it's not worth it. For this reason T_{reader} is always paid on any execution: Sequential, STL and FF. As for the Reading, the Writing is not parallelizable so there's always a T_{writer} to pay in each execution. For the parallel implementation I merged this cost inside the *Knn* timing because the Writer is a thread that pops concurrently from the Workers jobs, and it would be very hard to time it.
To have an idea of the timings, I executed the *Sequential* version of the program taking the timings for each of the sections, in order to understand how the parallel version would behave.

```
[f.murgese@c6320p-2 knn]$ ./knn_seq input30k.csv 10
Reading input file computed in 92695 usec
Knn computed in 356309393 usec
Writing to output file computed in 7350 usec
```

Figure 3: Results of the Sequential version for 10-NN with 30000 points (input30k.csv)

We have a total of 356316743 microseconds ($\simeq$ 5.93 minutes). So, in the parallelized version we should expect an ideal time of:

$$T_{\text{Ideal}} = T_{\text{reader}} + \left(\frac{T_{\text{sequential}}}{nw}\right) + T_{\text{writer}} \tag{2}$$

We have to consider that the machine where the experiments have been run is the Xeon PHI with 64 cores with quad hyperthreading, so we have available up to 256 workers. So we expect $\frac{356316743}{256} = 1391862.27$ microseconds to compute all the tasks if we exploit the machine at its maximum. So we have a total time of 1491878 microseconds (1.491 seconds).

4 STL and FF performances

All timings were taken with the `utimer.cpp` class used during the course. Here, I don't take into account the constant T_{reader}.

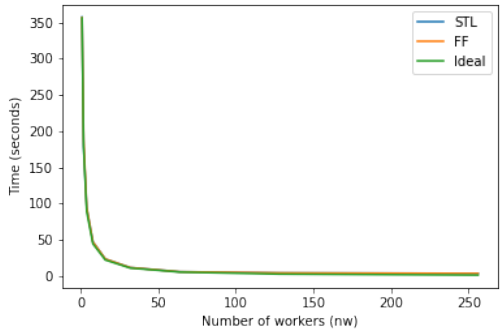

Figure 4: Plot of the performance.

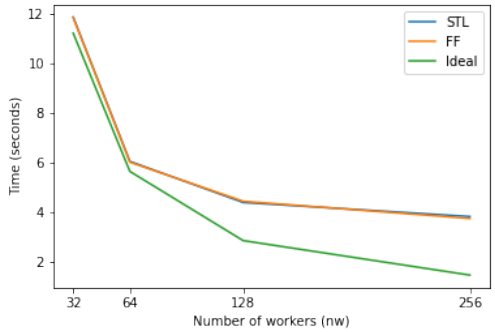

Figure 5: Results from nw = 32.

It's very hard to distinguish the lines of the two implementations, even in the second image where I wanted to show more in detail what happens with a higher number of workers (starting from 32 workers), but the results are, again, very comparable. As we can see we get very close to T_{Ideal} up to 64 threads, that are the actual number of cores of the machine. Once we exceed that, the performance starts getting worse. For this reason, 64 is the *plateau* for both our parallel versions of the program: no need to pay extra cores in order to achieve much better performance.

As stated before, both STL and FF implementations have very similar results. For this reason I will show calculations once: they will hold for both versions. I had 3849289 microseconds for computing Knn and writing to output file with *nw* set to 256, the maximum amount of workers the machine made available. Actually, we could consider that 256 adds a little of descheduling time because of the additional Writer thread that sums up to the 256 workers already taken for the proper K-NN computation. Anyway, I have a total time of $94090 + 3849289 = 3943379$ microseconds (3.943 seconds), almost three times the ideal timing. Let's now calculate the main measures:

Speedup (**n**) $= \frac{T_{\text{Seq}}}{T_{\text{Par}}(N)} = \frac{356.316}{3.849} \simeq 92.57$

Scalability (**n**) $= \frac{T_{\text{Parallel}}(1)}{T_{\text{Parallel}}(N)} = \frac{357.669}{3.849} \simeq 92.95$

Efficiency (**n**) $= \frac{T_{\text{Ideal}}\ (N)}{T_{\text{Parallel}}\ (N)} = \frac{1.491}{3.849} \simeq 0.387$

These results referes to the executions using the *input30k.csv* file made up by 30000 points and with a k parameter equal to 10.
To analyze the **Gustafson Law** let's see how it scales from a file of 10000 points to a bigger file of 30000.

Input size	**Speedup**	**Scalability**	**Efficiency**
10000	67.88	70.35	0.34
20000	93.11	93.57	0.40
30000	92.57	92.92	0.39

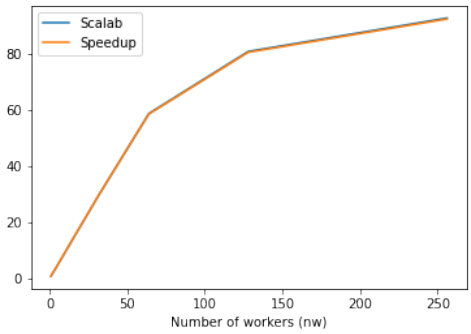

Figure 6: Scalability and Speedup.

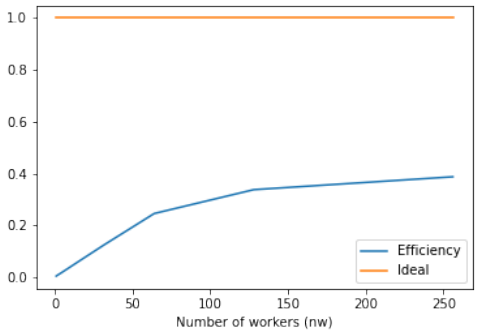

Figure 7: Efficiency.

All the executions timings are saved in the tables of the Appendix.

5 Differences between results and expectations

As we can see from the results, I couldn't achieve the theoretical results: those are estimations, not results actually achievable in a real scenario, as we have stated. A first limitation can be found into the actual number of cores of the machine; then, the fact that maybe the machine has been used from other students during my tries occupying some threads, or just by routine procedures the machine has to run. Also, the writer procedure didn't help because I expected a better handling of the different queues to write the results to the output file; maybe this it's better handled by the `Fastflow` library as we can see it reaches sligtly better results with respect to the `STL` version of the program with the highest number of workers and data.

In the end, another factor that might be taken into account is *load balancing*: I chose a simple divide pattern in order to not have to choose where to assign the tasks that are left out from the initial division of the tasks; this is famous for not exploiting data locality as much as it could. By the way, the dataset is very large so faults could still happen but, in this way, I tried to minimize that risk.

6 Building and running the model

To build and run the project I created a `Makefile`. First, you have to download the Fastflow library inside your working directory and then build the project by using the commands

```
git clone https://github.com/fastflow/fastflow.git
make
```

To test a specific version *executable*

```
./executable inputfile k nworkers
```

To compute the timings needed for the execution of a specific version there is the script *average_time.sh*, that iterates the execution a specified number of times, printing the timings needed for the *Knn* computation (Workers + Writer) and their average. Use the command

```
./average_time.sh executable inputfile k nworkers niterations
```

7 Appendix

KNN_STL	number of points	k	nw	read time	knn time (micros)	knn time (s)
input_short.csv	1000	10	1	4217	329800	0.329
input_short.csv	1000	10	2	4185	171339	0.171
input_short.csv	1000	10	4	4504	95746	0.095
input_short.csv	1000	10	8	4146	64366	0.064
input_short.csv	1000	10	16	4129	40600	0.04
input_short.csv	1000	10	32	4553	35036	0.035
input_short.csv	1000	10	64	4468	41017	0.041
input_short.csv	1000	10	128	4464	65641	0.065
input_short.csv	1000	10	256	4529	125680	0.125
input_long.csv	10000	10	1	36583	36938073	36.938
input_long.csv	10000	10	2	36699	18439707	18.439
input_long.csv	10000	10	4	28345	9482151	9.482
input_long.csv	10000	10	8	36818	4740476	4.74
input_long.csv	10000	10	16	36316	2410066	2.41
input_long.csv	10000	10	32	35979	1250690	1.25
input_long.csv	10000	10	64	36412	710177	0.71
input_long.csv	10000	10	128	36208	532702	0.532
input_long.csv	10000	10	256	36907	525203	0.525
input_huge.csv	20000	10	1	64919	154295476	154.295
input_huge.csv	20000	10	2	66263	77598863	77.598
input_huge.csv	20000	10	4	67080	42568936	42.568
input_huge.csv	20000	10	8	58509	20280353	20.28
input_huge.csv	20000	10	16	67316	10171101	10.171
input_huge.csv	20000	10	32	66458	5117722	5.117
input_huge.csv	20000	10	64	55714	2675023	2.675
input_huge.csv	20000	10	128	58536	2428315	2.428
input_huge.csv	20000	10	256	67152	1649605	1.649
input30k.csv	30000	10	1	92558	357669417	357.669
input30k.csv	30000	10	2	93177	179388169	179.388
input30k.csv	30000	10	4	93962	94594603	94.594
input30k.csv	30000	10	8	95266	47408067	47.408
input30k.csv	30000	10	16	82730	23671664	23.671
input30k.csv	30000	10	32	94269	11884935	11.884
input30k.csv	30000	10	64	94558	6071949	6.071
input30k.csv	30000	10	128	92316	4415697	4.415
input30k.csv	30000	10	256	94090	3849289	3.849

KNN_FASTFLOW	number of points	k	nw	read time	knn time (micros)	knn time (s)
input_short.csv	1000	10	1	4600	306707	0.306
input_short.csv	1000	10	2	4290	164446	0.164
input_short.csv	1000	10	4	4500	88565	0.088
input_short.csv	1000	10	8	4577	41975	0.041
input_short.csv	1000	10	16	4583	24875	0.024
input_short.csv	1000	10	32	4557	19549	0.019
input_short.csv	1000	10	64	4597	20743	0.02
input_short.csv	1000	10	128	4609	26483	0.026
input_short.csv	1000	10	256	4609	56014	0.056
input_long.csv	10000	10	1	37175	36008936	36.008
input_long.csv	10000	10	2	36274	18881446	18.881
input_long.csv	10000	10	4	37226	9450620	9.45
input_long.csv	10000	10	8	37290	4734197	4.734
input_long.csv	10000	10	16	37288	2376831	2.376
input_long.csv	10000	10	32	37275	1209620	1.209
input_long.csv	10000	10	64	37161	641089	0.641
input_long.csv	10000	10	128	37342	498291	0.498
input_long.csv	10000	10	256	37142	438812	0.438
input_huge.csv	20000	10	1	67346	154000943	154
input_huge.csv	20000	10	2	67516	80690480	80.69
input_huge.csv	20000	10	4	67391	40478912	40.478
input_huge.csv	20000	10	8	67358	20263217	20.263
input_huge.csv	20000	10	16	67547	10156096	10.156
input_huge.csv	20000	10	32	68063	5105248	5.105
input_huge.csv	20000	10	64	67307	2621611	2.621
input_huge.csv	20000	10	128	67479	1963752	1.963
input_huge.csv	20000	10	256	66264	1732447	1.732
input30k.csv	30000	10	1	95150	356890487	356.89
input30k.csv	30000	10	2	94032	188304620	188.304
input30k.csv	30000	10	4	94761	94537741	94.537
input30k.csv	30000	10	8	95072	47208598	47.208
input30k.csv	30000	10	16	95162	23667940	23.667
input30k.csv	30000	10	32	95433	11855674	11.855
input30k.csv	30000	10	64	95197	6037520	6.037
input30k.csv	30000	10	128	95129	4466334	4.466
input30k.csv	30000	10	256	94221	3769793	3.769