UNIVERSITÀ DI PISA

Master Degree in Computer Science

ICT Solutions Architect curriculum

Final report for SPM's project: K-NN

2020-2021

Teacher:
Prof. Marco Danelutto

Student:
Giulio Purgatorio
ID: 516292

Contents

1 Overview

This is the report for the final project of `Parallel and Distributed Systems: Paradigms and Models` course. I had to implement two parallelized C++ versions of the **K-Nearest Neighbors** algorithm: one using standard threads and one using the `FastFlow` library. The input is a .csv file where, for each line, there are 2D coordinates for the i-th point, and the output is another text file with the previous nodes and their K nearest neighbors (ordered by distance), again one per line.

2 Implementation design

2.1 Main design choices

The first thing to notice about a K-NN problem is that we're talking about a **Data Parallel Problem**, and a particular one: we're in the field of the so-called *embarrassingly-parallel problems* because the computation of an item is totally independent from the computation of another one (and therefore it is a perfect example for parallel programs).
As specified in the short summary, there are two versions of this project and, from now on, I will refer to them as the *StandardLibrary version* (`STL`) and the *FastFlow version* (`FF`). But first, let's discuss the Sequential version, made to create the main structure to parallelize and also to see timings in each part of the program: this way it was possible to see where bottlenecks were and target them with some improvements (discussed in 2.4).

Algorithm 1: Sequential version

Input : A $.csv$ file, where point i is on the i-th line as such: $x_{\text{coord}}, y_{\text{coord}}$

Output: A $.txt$ file, where on the i-th line there's a point and a list of K nearest points, with format: $ID : \{ID_1, ID_2, ..., ID_{\text{K}}\}$ such that, assuming $d(A, B)$ as distance metric between point A and B:

$d(ID, ID_1) \geq d(ID, ID_2) \geq ... \geq d(ID, ID_{\text{K}})$

if *$input\_file.csv$ doesn't exist or is empty* **then**
    return
end

Read points in the $input\_file.csv$ and populate the $data$ structure

foreach *Data point* $i \subset data$ **do**
    foreach *Data point* $j \subset data$ **do**
        if $i \neq j$ **then**
            Calculate $d(i, j)$ and store it in a vector;
        end
    end
    Partial sort the vector (up to K);
end

Print results to the output file $output\_sequential.txt$;

We can see that the main structure already tells us that are three macro-areas to address: **"Read"**, **"Calculate"** and finally **"Print"**. From now on, I'll denote $data.size()$ as N.

2.2 Actual implementation: STL

The first part is, of course, **data reading**. We cannot start working on any task until the *read()* is finished, because the computation has to explore the distance to all the other points. This means that there is a time T_{reader} that cannot be skipped in any way.

After this, we start populating our thread pool with nw workers. It would be possible to follow a `Farm` pattern, where, after reading all the input data points, we could emit tasks to be given out. But looking at our case's scenario, there is no real reason to "emit" tasks, because all

points are indexed in the $data$ structure and have to be calculated. For this reason, tasks can simply be seen as integers to be computed, ranging from 0 to $N-1$. This has also a minor effect on performance, considering that one thread isn't busy distributing tasks.
So, there's no need for neither an emitter nor a queue to pop tasks from. Considering that all tasks have to compute the exact same thing (which implies a strict correlation in work done), we can safely assume that they will roughly take the same amount of time (doing the exact same number of tasks). So each worker can take its own $Worker_{\text{ID}}$ as starting point and iterate until it reaches N, incrementing its index each time by nw (number of workers). Once the KNN of a node is computed, the worker will send its results to the *Collector thread* (which we will call the `Writer` from now on): the mutual exclusion problem (because with a single queue/deque workers would contend mutual access) is solved by giving to each worker a different deque towards the Writer. Once this is done, the worker can simply move on to compute the next data point and repeat the process.
Finally, the Writer has exactly nw queues to look for: the simplest way to cycle through this is to use a Round Robin approach. At the current version, the writer just tries to *pop()* from the specific queue: this can change with a *timed wait*, but after seeing that the Writer actually takes a much shorter amount of time writing to the file with respect to the workers, I opted to leave it as it is without further improvements.

2.3 Actual implementation: `FF`

In this version instead, I opted to write a more concise code by exploiting what the library had to offer. This made the code a lot shorter, with respect to the `STL` one. The structure is similar to the previous one because I've used a `ParallelForReduce`. The reason is again the evident ParallelFor pattern that produces outputs that have to be somehow concatenated.
A quick overview is that we have again the same Reading part. Immediately after this, the ParallelFor paradigm starts with its own lambda function, computing all the tasks and appending results to a local string: this string will then be passed to the Reduce part, which will append all the strings received and print them in the output file. To do the fewest possible number of system calls, both versions will append everything they receive to a local string and write just once. Of course this case of "Reduce" isn't the usual one, because we're not actually decreasing

the number of characters to append, but it's still a clean way to implement it.

2.4 Improvements for efficiency

In this subsection, I'll describe some efficiencies improvements I've made while developing the final version of the program.

To have a first simple example to test, I had a farm-like approach with a queue being filled by tasks emitted by the main method right after the workers' creation. All workers would create a simple vector with the distances with all the other points, then they would (partial) order it and finally write results to the same shared queue. The Writer would then simply pop one at a time and immediately print it to the final output file.

These improvements are ordered in ascending order of impact.

- (STL) **Removing the first queue** to pop from by indexing correctly workers with their own IDs (`Owner Computes`).

  Impact on performance: minimal ($\simeq$ same results)

  Discussion: the queue is again a deque, so writing to (or popping from) it is done separately. Therefore, emitting indexes to compute happened very fast and didn't slow down at all. I think the minimal difference is just the little waiting time between workers waiting to pop and the fact that one worker started late because of the needing for the *Emitter*, but the computation was long enough and therefore no real impact was given.

- (STL) Using **different (de)queues** towards the Writer instead of a single one.

  Impact on performance: Small nw: low ($\leq 1\%$). High nw: ($\simeq$ same results)

  Discussion: at first I thought it would increase performance by a lot, but it didn't. So I went back to analyze a bit better the 3 main timings and, by temporarily removing the Writer thread from the "Workers vector" that was being joined, I've noticed that it doesn't require that much time at all (with respect to the Workers' timings). The performance increase is noticeable when using a smaller number of threads, but when we start having higher ones probably the blocking Round Robin on the Writer side behaves similarly to the original problem.

- (STL, FF) Writer: **append to a local string** instead of writing to the output file, in order to reduce the number of system calls.

  Impact on performance: low ($\leq 1\%$)

  Discussion: reducing the number of system calls is obviously a good thing to aim at, but for the same reason specified in the previous point, the Writer isn't a bottleneck in our case (not even with the 256 threads from the Xeon PHI).

- (STL) **Pinning threads** to specific cores with CPU affinity.

  Impact on performance: medium (up to $\simeq 34\%$)

  Discussion: as we've seen during lessons, it's possible to pin threads to specific CPUs. The reason why this is useful is that if a thread happens to move due to (de)scheduling policies to another CPU, the data needs to be transferred as well. In lucky cases, threads may end up in "close" proximity, maybe accessing lower levels of caches, but in other cases, it could impact the performance a lot.

- (STL, FF) Using the monotonicity of the square root function to **simplify calculations**.

  Impact on performance: high ($\simeq 40\%$)

  Discussion: At the beginning, I was using the euclidean distance to calculate the distances between all points. At the current version, both implementations of the project will call the euclidean distance without actually computing the square root because it's a redundant operation. For example, having point B and C at distance 3 and 4 respectively, the final output doesn't change if they're classified as 9 and 16, but this way we avoid a complex operation like `sqrt(...)`.

- (STL, FF) Using a **MaxHeap** of K elements instead of a simple vector that was being (partial) sorted at the end.

  Impact on performance: high ($\simeq 60\%$)

  Discussion: How: First of all, I wanted a MaxHeap to have the "most distant" node as the root of the structure. This way, if a node has a higher (or equal) distance value than the root it can be discarded without actually traversing the heap. Instead, if the distance is lower, I can safely replace the root with the new element and then *heapify()* the structure.

Why: this has two different improvements. The first is, of course, time. Having K elements to sort is way faster than N. But another important factor is space: instead of saving $O(N * nw)$ data by storing in a vector all distances, we just need to save $O(K * nw)$. We could reach the same memory consumption with the vector approach, but it would require sorting it every single iteration and dropping the element, which will be then paid on the execution time's side.

3 Expected performances

As it was discussed in Section 2.1, reading is unavoidable, kind of hard to parallelize and definitely not worth it in terms of impact because it is very fast (with respect to the other parts of the program): therefore, T_{reader} is always paid on any execution (both versions).
The same goes for writing: this is definitely not parallelizable and so there's always a T_{writer} cost to pay in each execution. In both versions, I opted to merge this cost inside the Worker timings because the Writer is a thread that pops concurrently to the Workers jobs, and it'd have been hard to time it (or, by putting it in a sequential manner, it wouldn't go as fast as it could).
Finally, to have an idea on the "Worker timings", I executed and timed the *Sequential version* of the program and, by subtracting the T_{reader} and T_{writer} I could have an idea of how the `STL` and `FF` versions would behave.

Results for the Sequential version with 100000 points ($input\_long.csv$) and 20-NN:

```
[g.purgatorio@C6320p-2 spm]$ ./knn_sequential input_long.csv 20
Reading input file computed in 303915 usec
KNN computed in 186236042 usec
Writer computed in 43162 usec
```

So a total of 186583119 microseconds ($\simeq$ 3 minutes and 6 seconds).
In the parallelized versions I can then expect an ideal time T_{Ideal} of:

$$T_{Ideal} = T_{\text{reader}} + (\frac{T_{\text{sequential}}(Workers)}{NumberOfWorkers}) + T_{\text{writer}}$$

Considering that the machine provided (Xeon PHI) had up to 64 cores with quad hyperthreading (so up to 256 workers), we then expect to have $\frac{186236042}{256} = 727484.53$ microseconds

to compute all the tasks when using the machine at its maximum, so a total time of 1074561.54 ($\simeq 1$ second).

4 Overview of STL and FF performances

All timings were computed with the `utimer.cpp` class we used during the course, exploiting the `RAII` pattern. The following pictures do not take into account the constant T_{reader}.

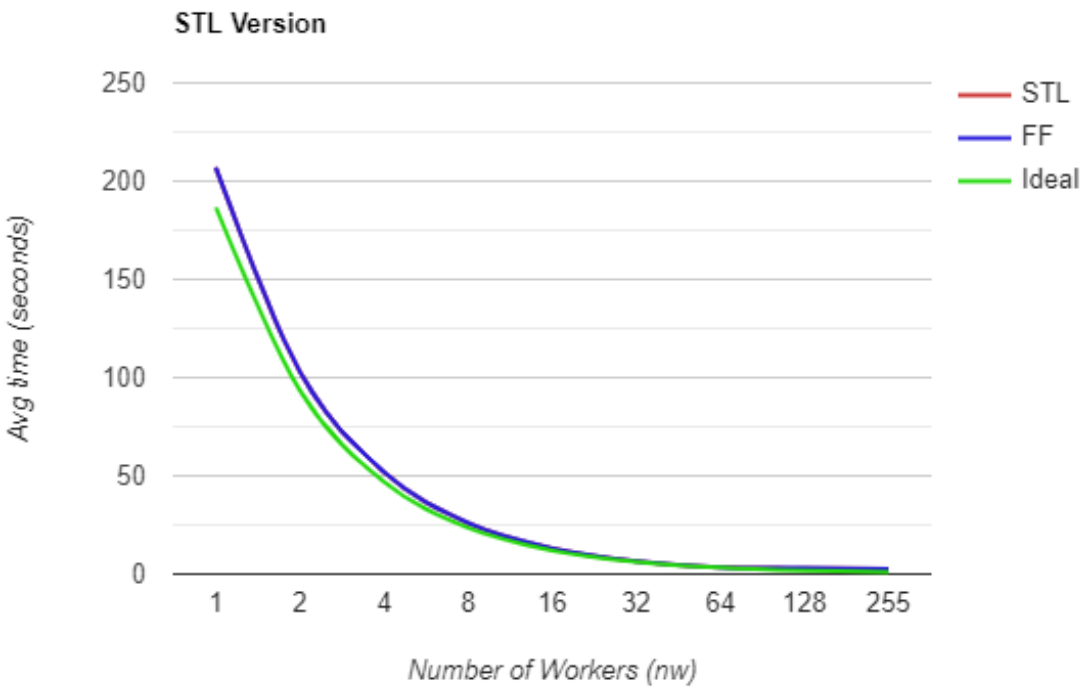

Figure 1: Workers + Writer timings for STL and FF versions

It's hard to distinguish the two lines from the two implemented versions: they are practically one on top of the other, sharing similar results. For this reason, a "zoomed version" (starting from 32 workers) of this picture is provided, also to discuss theoretical bounds (Figure 2).

As we can see, up to 64 threads (which is exactly the number of cores of the machine) we get very close to the ideal time T_{Ideal} and, once we exceed that, it starts to perform worse.

Therefore 64 is our plateau for both versions of the program: no reason to really pay the cost

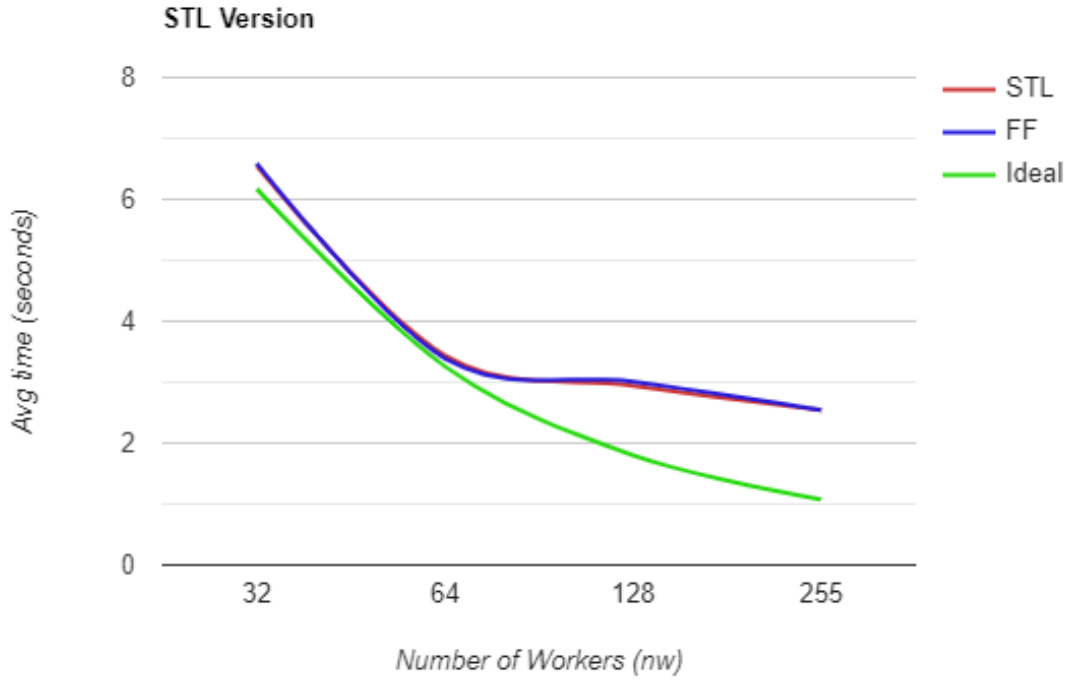

Figure 2: Zoomed version

of using higher resources because we do not gain enough from it.

4.1 Mathematical overview

Considering that both versions got very close to the same result, I will only show calculations once: they will hold for both STL and FF.

Results were in the range of 2590.09 ms for computing the KNN calculations **and** the Writing part with $nw = 256$, but this is just a bit improved by using $nw = 255$ so that one Thread will be reserved for the Writer (less descheduling), reaching 2539.49 milliseconds on average. Counting this as the best result, we now need to add the constant T_{reader} as we said before, getting a total of: $303.915 + 2539.49 = 2843.405$ milliseconds. This is almost three times our ideal timing (see Section 3). Anyway, lets now see the main properties to calculate:

Speedup(n) $= \frac{T_{\text{Seq}}}{T_{\text{Par}}(N)}$ consequently is: $\frac{186.583}{2.843} \simeq 65.62$, or, in our plateau, $\frac{186.583}{3.741} \simeq 49.86$

Scalability(n) $= \frac{T_{\text{Parallel}}(1)}{T_{\text{Parallel}}(N)}$ is: $\frac{208.092}{2.843} \simeq 73.18$ or, in our plateau, $\frac{208.092}{3.741} \simeq 55.61$.

Efficiency(n) $= \frac{T_{\text{Ideal}}(N)}{T_{\text{Parallel}}(N)}$ is: $\frac{1.074}{2.843} \simeq 0.377$ or, in our plateau, $\frac{3.561}{3.741} \simeq 0.95$.

Again, all results were computed on a $N = 100000$ input file and with $K = 20$: we could

Figure 3: Speedup (STL and FF)

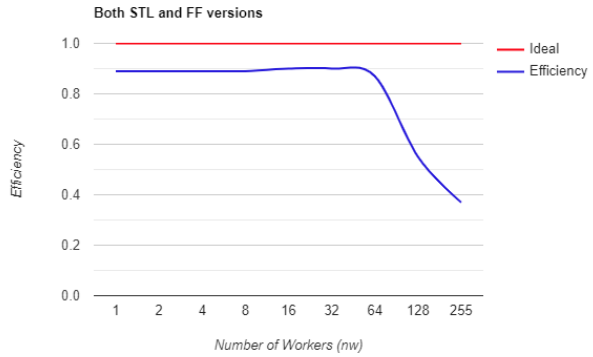

Figure 4: Efficiency (STL and FF)

achieve better speedups as stated by the **Gustafson Law** by using larger data sizes, but results are very good and, in the case of *efficiency*, close to its theoretical maximum ($\simeq 1$).

Input size	Speedup	Scalability	Efficiency
(input.csv) **10000**	17.85	20.52	0.41
(input_medium.csv) **50000**	29.33	33.12	0.58
(input_long.csv) **100000**	49.86	55.61	0.95

Table 1: Gustafson Law's scaling with different input sizes

5 Reasoning about differences observed among expected and measured performance

As usual, the theoretical result could not be achieved: these estimations simplify the real-world scenario and give us a maximum that we cannot achieve, but we should aim at.

The first limitation we find is the actual number of cores of the machine.

Then, the fact that the machine might have other tasks to do: maybe other students occupying cores, or maybe just routine tasks from the machine itself.
Finally, the implementation of different queues to write to didn't help as expected, causing the Round Robin approach to not perform. This is probably better handled by the `FastFlow` library as we can see that it reaches slightly better results when the number of cores increases.
A final point may also be the way Load Balancing was handled. In our case, I opted for a static divide pattern, the cyclic one. This had the advantage of not having to define who would need to do tasks that were out of the $\frac{N}{nw}$ part *(e.g. 10 data, 3 workers, 1 task is left out)*, but it's famous for its cost of not exploiting data locality as much as it could. Anyway, I've chosen this also because the "data" had to be the whole dataset, which is very large, so faults are still going to happen.

6 Building and running the project

To build and run the project, I've created a `Makefile`. Once the zip is extracted to a folder, build everything by executing the command

```
make
```

Then, to test a specific version x, just (all parameters are mandatory):

```
./x input_file K-hyper num_of_workers
```

In the folder, there's also a script that iterates the same command a specified number of times, prints the timing needed (only) by all the Workers plus the Writer for each iteration and at the end prints the average (in milliseconds). It wraps the previous command and so can be used as:

```
./average_time x input_file K-hyper num_of_workers num_of_iters
```