UNIVERSITÀ DI PISA

SPM Project: K-NN

Jacopo Massa - 543870

Academic Year 2020/21

Contents

Introduction

The goal of this project is to design and implement a parallel application that computes the K-NN algorithm. So, given a set of 2D points, it computes (for each of them) the set of k *closest points*, retrieving them onto an output file.

Two applications have been developed:

- the former uses `Standard Template Library (STL)`'s threads and data structures.
- the latter exploits the `FastFlow (FF)` C++ framework.

Results obtained by both of them are discussed and compared later in the report.
In addition to these versions, also a *sequential* one has been implemented because sequential timing is necessary to compute some statistics.

1.1 K-NN Structure

To better understand the implementations, let's see how **K-NN algorithm** works, defining the list of operations that all versions have in common:

READ input file: a *.csv* file containing 2D points coordinate (as $\{x,\ y\}$) is read line by line, assigning an ID to each point.

COMPUTE distance: for each pair of **different** points we need to calculate the distance between them to define the set of *k-closest points*, sorted by distance (from closest to farthest).
In this project **Squared Euclidean Distance** has been used as *distance metric.*

WRITE output file: all found sets are written on an output file (one per line) preceded by the ID of the corresponding point.

Implementation

2.1 Data structures

In both `STL` and `FF` I've used some auxiliary classes that helped in different ways:

- **Point class** defines a point with its coordianates and the method to compute the distance. I've also overloaded the insertion (<<) and extraction (>>) operators to have a faster way to read/write points.
- **utimer class** computes timings of portion of codes, exploiting the RAII pattern. (*The same we used during the course).*
- **queue class** implements a FIFO thread-safe queue, used to enqueue already computed points that can be printed on file. (*The same we used during the course).*
- **Heap class** implements an heap, exploiting the `priority queue` from the Standard Library, with a custom comparator that allows to compare two `Points` instead of primitive types.
  Actually, the heap is a fixed-size *MaxHeap* (size is equal to the hyperparameter K). Further details are provided in the Section 2.4.

2.2 `STL` version

First of all, we need to **read** data. This part is blocking with respect to the rest of the program, because we must populate a data structure with **all** the points to compute the distance among **all possible** pairs.

Then there's the population of a thread-pool (implemented as `vector<thread>`) with a number of workers given as input to the program. If we focus for a moment on the type of problem we are facing, we immediately notice that single K-NN tasks have no correlation among themselves, and as we have to compute all of them (for each point), we don't need an *emitter* that partitions tasks or a *queue* from where workers can pop them.

So, I've decided to implement a **static scheduling** policy, where each worker has the same chunk of tasks to execute (*chunk_size = total_size / # workers*). After a thread has **computed KNN** for a point, it will push the **set of IDs** of the *k-closest points* into its associated queue (**each worker has its own queue**).

Another thread (which is not part of the original threadpool) *collects* outputs provided by workers. It starts working along with other threads, by checking all queues in a round-robin fashion. A particular value (called `EOS` or *End-Of-Stream*) is pushed in the queue when a thread has finished its job. By doing so, the collector can focus on other queues, "jumping" the ended ones.

Having a queue for each thread, avoids the *mutual exclusion problem*, in fact each queue has only one writer (the *i-th* thread) and one reader (the collector).

Last operation is done by the collector, which has to **write** on the output file all the informations it pops from queues. To avoid a large number of system calls, I've opted to append all the results to a string which is written only once, after all workers have ended their jobs.

Actually, there will be many system calls, because the system will buffer the entire string and write it partially every time, but it isn't a real bottleneck.

2.3 `FF` version

This version works exactly as the previous one, but the implementation is done by using constructs provided by `FastFlow`. Obviously this version is shorter because `FastFlow` abstracts some parts of the implented pattern, so we have only to point out the right lambda functions (similar to the task assigned in the previous version).

In particular, for this problem, I've opted for `ParallelForReduce` pattern exploiting again the independent iterations. This pattern works with two lambdas:

- ***mapF***, that is (to make a parallel with the previous version) the task computed by each thread, but slightly varied. Here a worker doesn't generate a list of IDs, but directly **the string to append** to the final result.
- ***reduceF***, which is the (previous) collector task, which makes the real concatenation to the string that will be written in output.
  We can notice how this is not a classical *reduce* operation, but doing so we avoided to have a "manually-managed" string shared among all workers, since all the logic is managed by the framework.

Also here I've opted for a static scheduling, by setting the `chunk_size` parameter of *parallel_reduce* function equal to 0 (which means contiguous chunk of iterations).

2.4 Additional info

Until now I provided a detailed description of the two versions. But in this section I want to explore more in depth some aspects of used data structures and decisions I've made during the implementation phase.

2.4.1 Squared Euclidean Distance

In the first version of the project, the "classical" *Euclidean Distance* was used as distance metric. So for each pair of point the program computed a square root (`sqrt`), which is a (computationally) complex operation. But, as we are talking about *distances*, *Squared Euclidean Distance* is more suitable, and omits the redundant square root calculation, exploiting its *monotonicity* to not change the distances' ordering.

I've noticed a marked improvement in performance after this update.

2.4.2 Left-out Task Management

As described in Section 2.2, for `STL` I opted for a static scheduling, that divides *more or less* equally tasks among workers. But obvisously, if the number of threads in the pool is not a multiple of the number of tasks, some tasks will remain outside the partition. The number of left out tasks will be, in the worst case, equal to *#workers -1*.

To overcome this issue I used an ***atomic*** integer counter (`notAssignedIdx`), initially set to the index of the first left out point. This counter needs to be *atomic*, so that each thread can access and increment it in one clock cycle, so I didn't have to manage shared access to it. Overhead introduced by this atomic counter is **negligible**, in fact *utimer* class timed 0/1 usec, that compared to a single K-NN task computation (4048 usec) will have no impact on final timings.

Each worker, after having processed the assigned task chunk, will try to process one of the left out point, by checking if `notAssignedIdx < #points`, otherwise it will end its stream, sending the `EOS`.

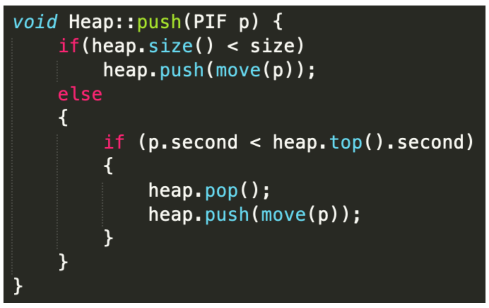
Hence, tasks that are not initially partitioned, are picked by workers themselves, whenever they are free.

2.4.3 MaxHeap

Intuitively, as the project request is to retrieve points sorted by distance in ascending order, the first solution I thought of was a *MinHeap*: everytime the program computes a distance it pushes in the heap a pair made by the `ID` of the neighbour and its distance from the "task point". A first drawback was that I had to store all the points in the heap (a huge space consumption for large datasets), and then pop the first k elements from top. Another issue of this implementation was that at each *push*, if a point as an high distance, it's necessary to traverse the entire heap to place it, increasing the timing needed for this operation.

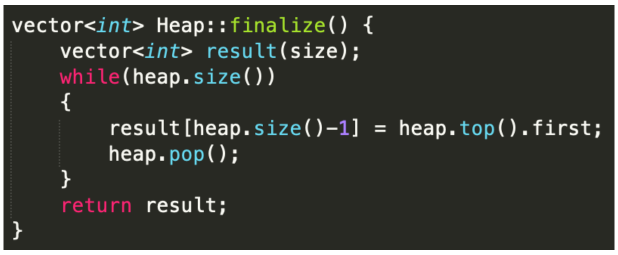
By simply switching the *MinHeap* into a `MaxHeap`, **I improved timings** by a lot, because when pushing I could discard every point which had an higher distance with respect to the top, and replace the top with the new point, otherwise. In addition, I could manage the heap size, limiting it to k, reducing also the space consumption.

All the heap dynamics were internally managed by the `std::priority_queue`, I only overrided the `push` method (2.1), and to obtain points in the right order, I reversed the heap when finalizing it (2.2).



```
void Heap::push(PIF p) {
    if(heap.size() < size)
        heap.push(move(p));
    else
    {
        if (p.second < heap.top().second)
        {
            heap.pop();
            heap.push(move(p));
        }
    }
}
```

Figure 2.1: Heap `push` method.



```
vector<int> Heap::finalize() {
    vector<int> result(size);
    while(heap.size())
    {
        result[heap.size()-1] = heap.top().first;
        heap.pop();
    }
    return result;
}
```

Figure 2.2: Heap `finalize` method.

Performances

3.1 STL and FF performances

To evaluate the two implementations I've calculated the following measures, for several thread pool sizes (`n`):

- $speedup(n) = \frac{T_{seq}}{T_{par}(n)}$
- $scalab(n) = \frac{T_{par}(1)}{T_{par}(n)}$
- $\epsilon(n) = \frac{T_{id}(n)}{T_{par}(n)}$ (efficiency)

where T_{seq} has been computed using the sequential version, and:

$$T_{id}(n) = T_r + \frac{T_{knn}(n)}{n} + T_w$$

Theoretically the ideal execution time has to be expressed as the sequential execution time divided by parallelism degree, but in this case, *reading* and *writing* are considered as constant, because they are not parallelized.

To better understand the formula, I recall what I explained in Sections 2.2 and 2.3: T_r is the initial time needed to populate the data structure, **reading** all the points to work on, so it can't be avoided, therefore it must considered for all versions. On the other hand, T_w is the time needed by the *collector* thread to pop results from workers' queues and **print** them on the output file. Also this operation is not parallelizable, so we must consider this execution time too.

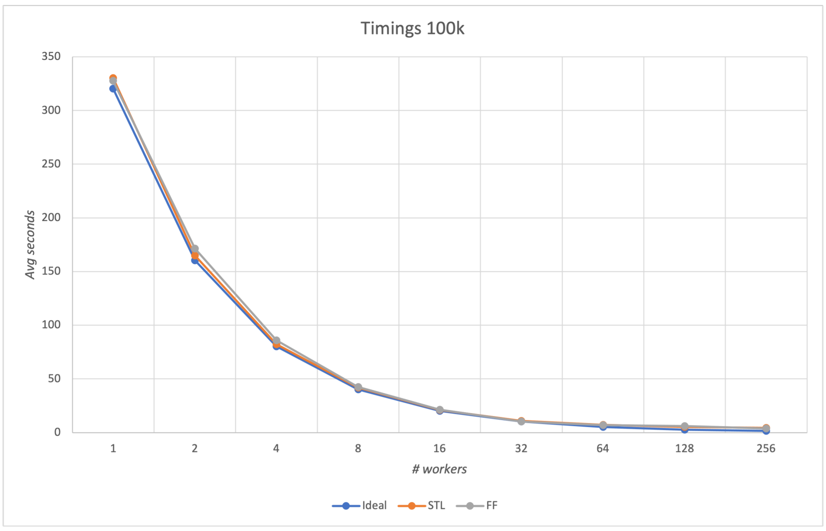


Figure 3.1: Timings 100k.

As stated at the beginning of this Section, I've done several measurements, ranging variables as follow (all with ***k=20***):

- **#Threads** $2^i \ \forall i \in (0, \dots, 8)$
- **#Points** $(10k, 60k, 100k)$

All the results and associated graphs are collected in the `res.xlsx` file linked to the project. I'll show and discuss here only the ones related to the middle case (*#Points = 100k*).

As we can see from Figure 3.1, both `STL` and `FF` seems to be overlapped to the *ideal* one, but if we zoom the last part of the graph (Figure 3.2) we can notice how up to **32** workers, timings are more or less equal, but increasing the number of threads we move away from the ideal time, with a consequent performance decrease.

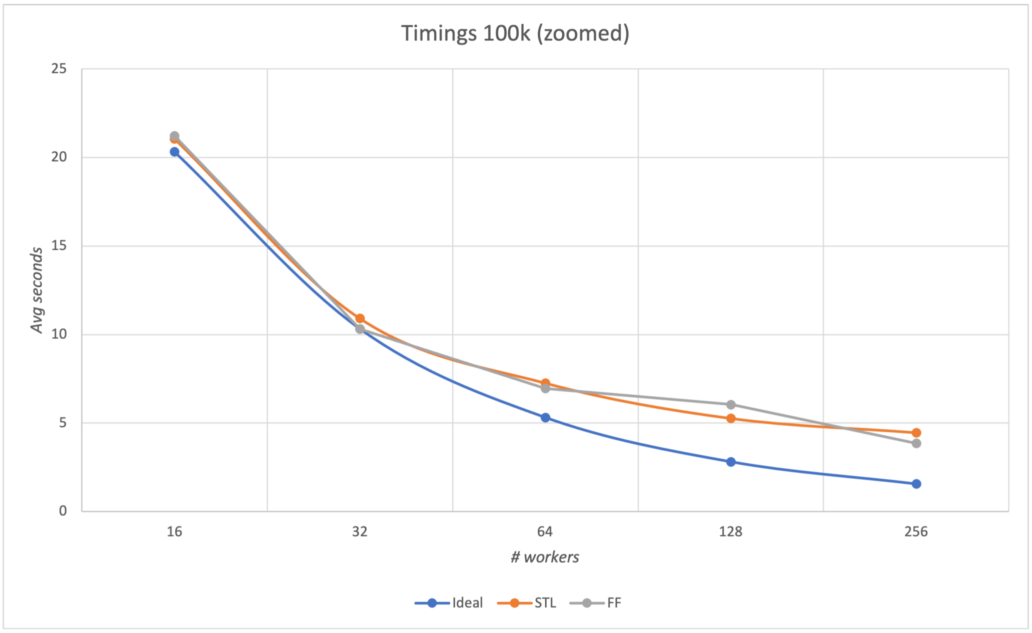


Figure 3.2: Timings 100k (zoomed).

What as been said so far can be confirmed by looking at the graphs of measured statistics (Figures 3.3 and 3.4). In fact, *speedup* and *scalability* no longer grow exponentially, but linearly after 32 workers (even earlier). The same happens to the efficiency which starts decreasing after our critical point.

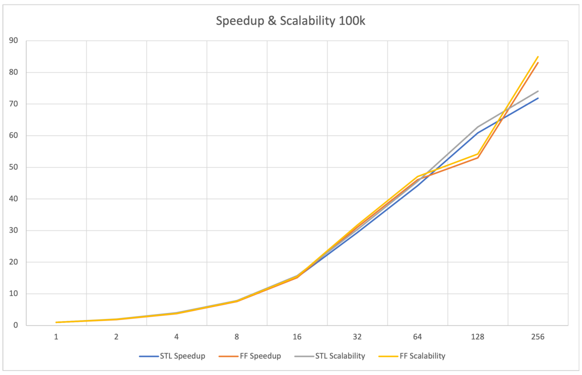


Figure 3.3: Speedup and scalability 100k.

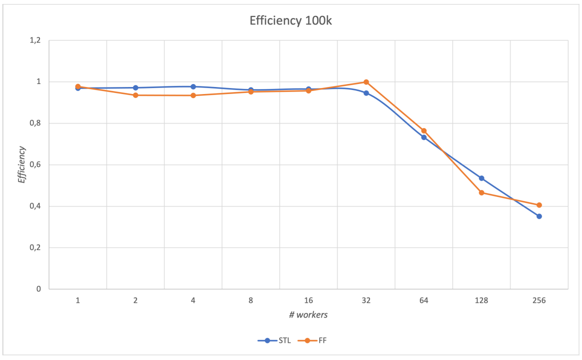


Figure 3.4: Efficiency 100k.

I decided to show the *100k* case, because it is the highest one. But, if we look more in depth the whole results, we notice that performances increase by increasing the data size too (as ***Gustafson's Law*** states). The following table summarize results (taken at the critical point *threads=32*) for the different data sizes:

Table 3.1: Results' summary (at $\#workers = 32$).

Data size	T_{par} (sec)		Speedup		Scalability		Efficiency	
	STL	***FF***	***STL***	***FF***	***STL***	***FF***	***STL***	***FF***
10k	0,18	0,15	18,27	22,13	18,91	24,1	0,75	0,91
60k	4,29	4,34	28,2	27,7	28,52	28,7	0,92	0,91
100k	10,9	10,32	29,4	31,04	30,1	31,75	0,95	0,99

The only difference I've observed between `STL` and `FF`, is that the latter better performs especially with an higher number of workers, maybe because *FastFlow* manages more intelligently the *load balancing* (especially the left out tasks).

However `FF` has retrieved in general better results, so better times more or less in each case I tried.

3.2 Expected VS Observed Results

One last consideration is about **theoretical results**. Metrics I've used has the following known asymptotes:

- **Speedup & Scalability** $\Rightarrow$ \# *workers*
- **Efficiency** $\Rightarrow$ 1

Comparing these values with the ones in table 3.1, we are really close to theory, especially for efficiency that reaches 0,99.

I didn't expect reaching higher values for several reasons. First of all, I wasn't the only user of the machine on which I ran the tests, so machine's cores could be busy doing other jobs, altering (a few) of my timings.
In addition, I opted for static scheduling to make the most of data locality, but I couldn't exploit a lot because, even if I partioned points among workers, each of them needed the whole dataset to perform K-NN.

Build and run the project

To build and run the project you can use the given `Makefile`, whose main targets are *stl, ff* and *seq*, to build all the three program versions.

So, first of all, you need to run the following command:

```
$ make
```

Afther that, you need to generate some points to work on. Use the given `Python3 "generate.py"` script, by running the following command:

```
$ python3 generate.py <seed> <len> <max>
```

This command will generate a file named `input_{len}.csv` in the subfolder `input`. Once you have generated some points you can make a single run of the program with the following command:

```
$ ./{version} <k> <workers> <inputfile>
```

where `version` can be *stl* or *ff*.

For the sequential version, command doesn't take the number of workers as input:

```
$ ./seq <k> <inputfile>
```

Another script is given to make an average of timings, by executing several times the same version (*stl* or *ff*):

```
$ python3 avg.py <version> <iterations> <k> <workers> <inputfile>
```