Parallel implementation of k-Nearest-Neighbors

William SIMONI (w.simoni@studenti.unipi.it)

3 February 2022

Problem definition

Given a set of 2d points, the problem consists of finding, for each element, the k nearest points in terms of Euclidean distance.

1 Parallel Design

We can parallelize the process by a map of the KNN function over the set of points, followed by a reduction that merges the results. Assuming, with a certain confidence, that computing KNN for a point requires more or less the same time needed to compute KNN for another one, we can divide the dataset equally among the threads at the beginning of the computation. Moreover, we do not need synchronization during execution since threads will read from the shared data structure (to get the point coordinates) and add the result to a local solution. Eventually, the local solutions are merged and written in a file. Note that using a local solution should avoid performance issues such as false sharing.

2 Implementation

2.1 KNN function

Given a set of points P, a point p such that $p \in P$, and a parameter K, the KNN function returns the ID of the K nearest points to p in P.

The set of points P is represented by a PointVector object, which is a vector whose elements are struct called Point. Each Point contains an x and y field, which are float variables.

The function calculates the euclidean distance (without the square) between p and all the other points in P and uses a max heap of size K to maintain the ID of the K closest neighbors. Every max heap entry is a <key, value> pair where the key is the distance between p and another point q, while the value is the ID of q.

In the end, it reads the max heap and returns a string containing the ID of the K nearest neighbors ordered from the most nearby to the furthest.

The algorithm complexity is $O(n \log(K))$ in time where n is the size of P. Indeed, in the worst case, the input file will contain points in decreasing order of distance from p. So, the function will have to pop/push elements from/in the max heap at every iteration. Since both the pop and the push costs $O(\log(K))$ and the iterations are n, the final complexity is $O(n \log(K))$.

2.2 Sequential program

We can divide the sequential program execution in three main parts:

- **read**: it reads the file indicated as program argument and creates a data structure containing all the points to which apply the KNN.
- **knn**: the program applies KNN to all points. It maintains the KNN results in one single string, say RESULT. At each iteration, the string returned by KNN is appended to RESULT.
- **write**: it writes RESULT in a txt file.

2.3 Parallel implementation

The fastflow and the C++ Thread implementations are quite similar. The function executed by the threads is indeed exactly the same in both the implementations:

```
1       auto f = [&](const long start, const long stop, const long thid)
2       {
3               string result = "";
4
```

```
5               for (int i = start; i < stop; i++)
6               {
7                   result += knn(pv, i, K) + "\n";
8               }
9
10              results[thid] = result;
11      };
```

The thread initializes its local result in row `3`. Then it applies the function to its partition of points. In detail, it appends one after the others the strings returned by the KNN function (rows `5` to `8`). Eventually, it puts its local solution into the global array `results` (row `10`).

In order to have the thread id, the fastflow implementation uses `parallel_for_idx`. A second provided solution uses `parallel_for_reduce`. However, this second solution performs worse than the first.

2.3.1 Thread affinity

Compiling with `-D DOPINNING`, the executable will linearly pin the threads. As shown in the performance section, this leads to lower latency in most cases.

3 Experiments

In this section, we will use the following notation:

- We will indicate with T_{seq}, the latency of the sequential program.
- We will refer to the Fastflow implementation that uses `parallel_for_idx` with the name FF_idx. While, we will call FF_red the second implementation, which uses `parallel_for_reduce`.
- We will use T_{READ}, T_{KNN} and T_{WRITE} to refer to the execution time needed by respectively the read, the KNN, and the write phases.

If not indicated, the measuring unit used is the microsecond.

3.1 Experiments setup

- Experiments used two datasets: the first one of 20.000 points and the second of size 100.000. These datasets were generated uniformly using the `generateinput.py` script.
- For all the tests, we used $K = 15$.
- Time is measured by computing the mean execution time over five trials.

3.2 Maximum speedup

The speedup, with parameter n, is defined as the ratio between the latency of the sequential program and the latency of the parallel program run with n workers.

As the Amdahl law states, we can divide the code into two parts: serial e non-serial. The serial part should also consider the overhead coming from the creation and the join of the threads, but for this analysis, let us say that $T_{serial} = T_{READ} + T_{WRITE}$ and $T_{non\_serial} = T_{KNN}$. Therefore, the serial fraction is

$$f = \frac{T_{READ} + T_{WRITE}}{T_{READ} + T_{KNN} + T_{WRITE}}$$

Knowing f, we can then compute the maximum achievable speedup as $1/f$. Table 1 shows T_{serial}, T_{non_serial}, f, and the corresponding maximum speedup for the two datasets. The times are obtained from the execution of the **sequential** program.

Table 1: Sequential program measurements

dataset	T_{serial}	T_{non_serial}	f	**max speedup**
20.000	100.833	8.251.777	1.207 %	82.85
100.000	450.435	204.189.058	0.220 %	454.54

As expected by the Gustafson law, increasing the dataset size reduces f, and by consequence, increments the maximum speedup. However, for this problem, increasing the dimension of the data just to increment speedup is not a viable way; if we are interested in a dataset of 20000 points, it does not make sense to add other points merely to increase speedup.

3.3 Latency

Latency is defined as the time spent between the moment a given activity receives input data and the moment the same activity delivers the output. For our problem, we can define the latency as $T_{READ} + T_{KNN} + T_{WRITE}$. Table 2 and Figure 1 show the latencies of the different implementations with different amounts of workers. The ideal time is computed as follows:

$$\text{ideal time} = \frac{T_{non\_serial}}{\text{number of workers}} + T_{serial}$$

T_{serial} and T_{non_serial} are the serial and non-serial times of the sequential program.

We can observe the following:

- The fastest implementations overall are the STL+pinning and the Fastflow version with `parallel_for_idx`.
- The Fastflow implementation with `parallel_for_reduce` is slower up to half a second with respect to the implementation with `parallel_for_idx`. Probably, this is due to some synchronization mechanism that is not present in the former implementation.
- The latency dramatically increases in all implementations except the basic STL one moving from 63 workers to 65. The same happens from 127 to 129 and from 191 to 193. After the increment, latency lowers again until the next multiple of 64, which just happens to be the core count of the machine. The latency increase is so evident that the STL version has the best latency in 65, 129, and 193.
  The only difference between STL and STL+pinning is merely the linear pinning of the threads. Thus, pinning threads have something to do with this behavior.

3.4 Serial fraction

As already mentioned, $T_{serial} = T_{READ} + T_{WRITE}$. Looking at Figure 2, T_{serial} is almost constant whatever is the number of workers.

As Figure 3 shows, the larger is the dataset, the smaller is the serial fraction. In the 100.000 points dataset, after more or less 60 iterations, the serial fractions maintain values around 10% reaching a maximum of 14%

Table 2: Latency for 100.000 points. In each row, we indicate in bold the fastest implementation.

n_w	STL	STL+pin.	FF_idx	FF_red	ideal
1	205.646.446	206.074.535	**205.468.008**	205.941.296	204.639.494
63	5.664.079	4.233.280	**4.199.095**	4.799.401	3.691.531
65	**5.575.468**	6.135.887	6.145.603	6.550.336	3.591.805
127	3.958.646	**3.487.182**	3.498.116	4.552.743	2.058.223
129	**4.182.045**	4.665.289	4.700.607	4.899.302	2.033.296
191	3.538.414	**3.416.468**	3.432.608	4.163.691	1.519.488
193	**3.734.918**	4.013.847	4.043.628	4.221.820	1.508.409
255	3.296.322	**3.254.353**	3.272.017	3.467.118	1.251.176

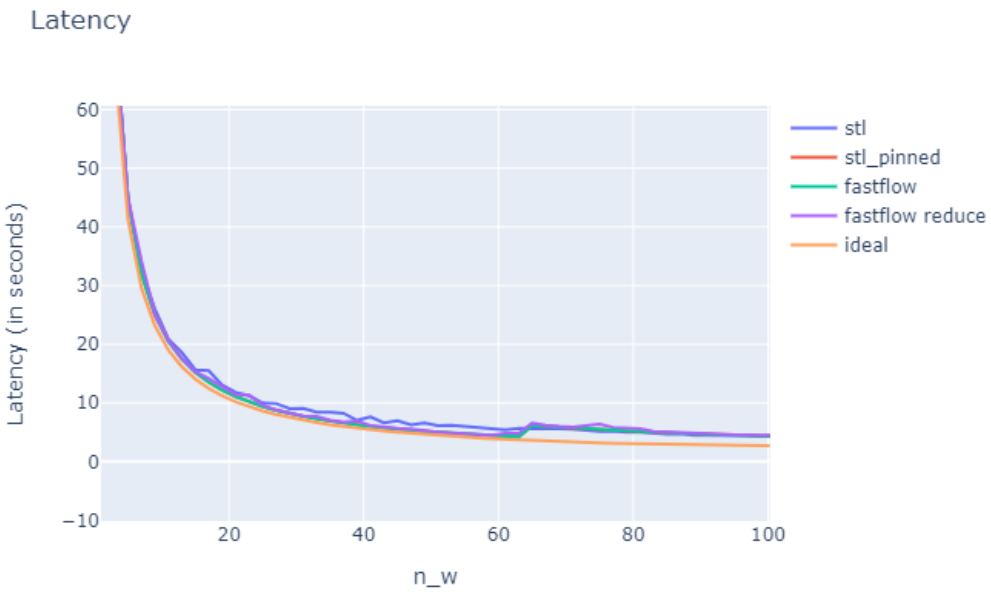


Figure 1: Latencies (in second) of the different implementations.

when the number of workers is 255. With 255 workers, we achieve a speedup of 62 (see Table 3), which is approximately $455^1 \times 0.14 = 63,56$. Therefore,

[1]Speedup upper bound given by the Amdhal law.

we can see the serial fraction plot as complementary to the speedup. The higher is the fraction, the closer the speedup is w.r.t. the maximum speedup. Therefore, with 20.000 points, we are closer to the maximum speedup than with 100.000 points.

3.5 Speedup

In the case of the map pattern, the ideal speedup is the number of workers employed in the computation. However, considering the bound given by the Amdahl law, say α, the true ideal speedup is $\min\{\alpha, n\_w\}$.

Figure 4 and Figure 5 picture how the speedup evolves, increasing the number of workers. The speedup appears to be close to the ideal curve until 63 workers. Table 3 indicates the speedup achieved with 255 workers, which ideally is when we should have the maximum speedup.

For the 100.000 points dataset, a good choice of workers can be 127. After 127, the speedup slightly grows but remains around 60. With 20.000 points, the speedup decreases after 63 workers. The FF_red implementation has the worst drop, passing from a speedup of 27 with 63 workers to the 18.3 reported in the table. Therefore, the best choice is for sure 63 for all the implementations.

The best version overall is the STP + pinning implementation. FF_red is, on the other hand, the implementation that behaves worst.

Table 3: Speedup using 255 workers

dataset	**STL**	**STL+pin.**	**FF_idx**	**FF_red**	**ideal**
20.000	29,14	29,33	26,5	18,3	83
100.000	62,08	62,88	62,5	59,02	255

3.6 Efficiency

The efficiency, with parameter n, is defined as $speedup(n)/n$. With 100.000 points, all the implementations, except the STL, have an efficiency close to or above 80% when n is less than 64. The efficiency starts decreasing as soon as we use more workers than cores.

When we have 20.000 points, the efficiency decreases even before 64 workers, probably due to a larger serial fraction.

Figure 6 shows the efficiency with 100.000 points according to the number of workers.

3.7 Accelerators

Due to the map nature of the problem, I tried to run a GPU implementation written by Vincent Garcia (and others) on a Colab Machine [2]. Unfortunately, with 100.000 points, the program returns memory errors. With 20.000 points, the GPU implementation had $T_{KNN} = 0.044$ seconds against the 0.18 seconds needed by the STL+pinning implementation with 255 workers. So, as expected, GPUs, even using a Colab GPU (the NVIDIA Tesla k80), beat the CPU counterpart for map patterns. A second very likely cause is that their implementation is better than the one proposed here.

4 Conclusions

The XEON PHI can handle 256 threads with 64 cores. However, the performance gain declines as soon as we start using multiple threads per core. As figure 5 shows, with 63 workers, we have a speedup of 49. Adding 192 workers, we only have an increment of 14. A second problem that may have led to a lower performance boost is that other users were probably using the machine during the tests. Of course, when dealing with other patterns, hyperthreading may be more proficuous (probably with a different pinning).

4.1 How to build and run

To build the project, execs `make` in the project folder. Each parallel executable have three parameters, in order, the input file name, K and the number of workers. For example: `./FastflowPar ./data/100000.txt 10 8`.

All the datasets are in the data directory. Tests directory contains python Jupiter notebooks with several plots generated with plotly.

You can run a test using test_" implementation name".sh. In the file, you can set several parameters, such as the K, or the dataset for the test.

[2] https://github.com/vincentfpgarcia/kNN-CUDA

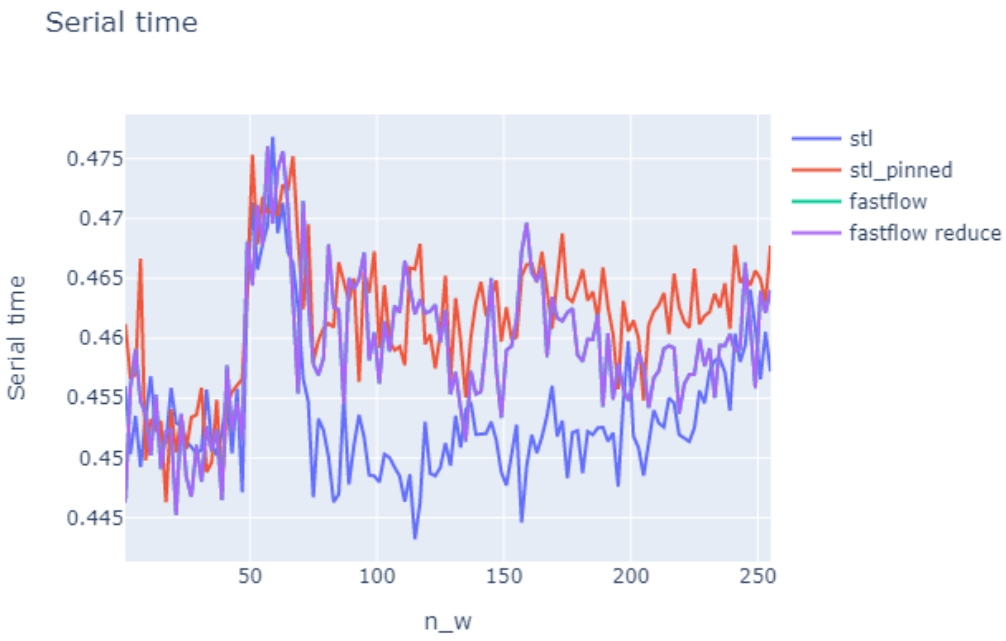
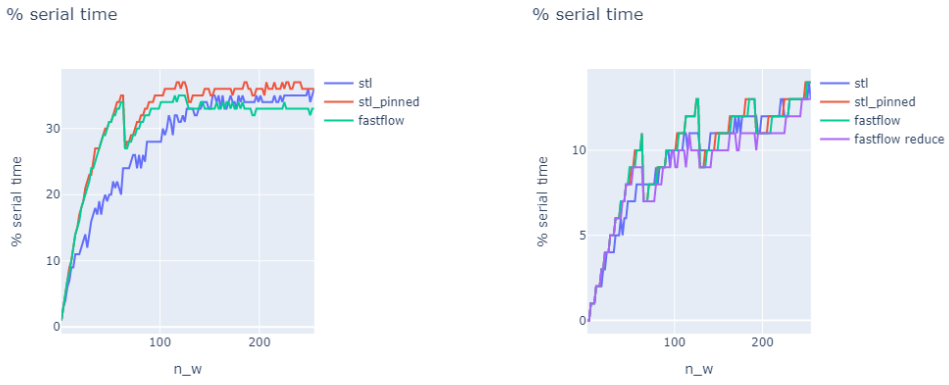


Figure 2: Serial fraction time (in second) of the different implementations for the 100.000 points dataset.



(a) 20.000 points

(b) 100.000 points

Figure 3: % serial fraction of the different implementations

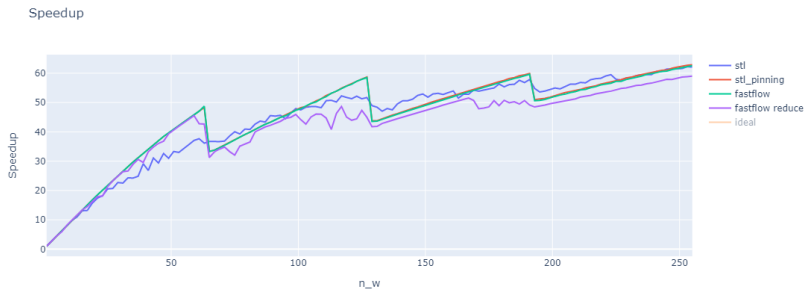


Figure 4: Speedup over the 100.000 points dataset

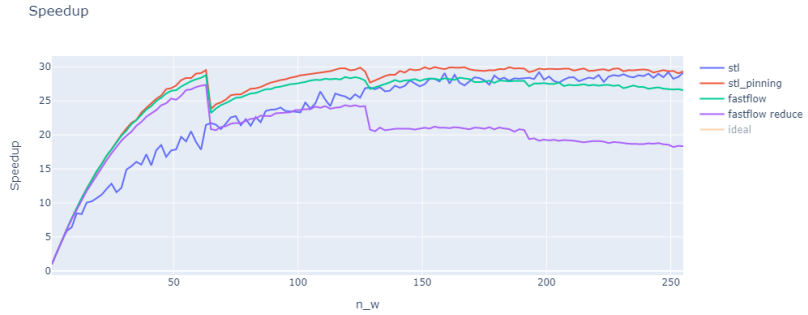


Figure 5: Speedup over the 20.000 points dataset

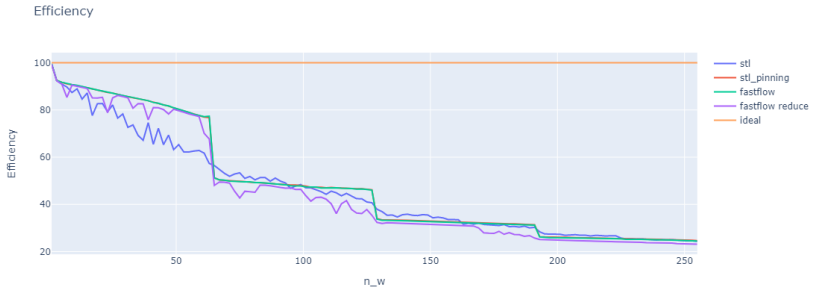


Figure 6: Efficiency with 100.000 points dataset