K-Nearest Neighbors (KNN)
Application project

Nicola Gugole (619625)

June 9, 2021

Final project presented for the
Parallel and distributed systems: paradigms and models
course

University of Pisa
Artificial Intelligence
A.Y. 2020/2021

Contents

1 Introduction **1**

2 Application Analysis and Expectations **2**

3 Implementation Choices **3**

4 Experimental Scenario and Setup **5**

5 Experimental Results **6**

5.1 XEON PHI - 5 knn 6

5.2 XEON PHI - 200 knn 8

5.3 I7-8750H - 5 knn 9

5.4 Efficiency Analysis 9

6 Conclusions **10**

7 Referenced Figures **11**

1 Introduction

In this report the aim is to study how to parallelize the computation of KNN, starting from a theoretical model and then digging into technical details of the implementation.
Finding the K Nearest Neighbors of a 2 dimensional point is a problem which algorithmic solution is composed of few simple steps:

1. take set of points in a given space
2. order the points by increasing euclidean distance with respect to the reference point
3. take as result only first K ordered positions

The ordering mentioned in **step 2** is actually a huge bottleneck which is useless most of the times. In fact, when k is reasonably small with respect to the number of points (as is needed in most realistic cases), the sorting cost can be avoided by instead going through all points only once.
This can be achieved by maintaining a support structure where top k points

are kept based on the distance with a reference point. An alternative method, which effectively lowers the theoretical complexity per point from $O(n \log n)$ to $O(n \cdot k)$.

Such a problem can be heavily parallelized and what this report shows are the results with two different implementation, first using only the *C++ stdlib threads* and then using the *FastFlow library*.

2 Application Analysis and Expectations

The nature of this application strongly implies the use of a *data parallel pattern* since all points are given a priori (input is a file) and most of all no dependencies are present between points.
Iterations are therefore completely independent, which leads to favoring the use of Map pattern with a Reduce in the end, needed to compose the resulting string which is finally output to a file (reduce is implemented differently in the two project cases: *FastFlow* and *C++ Threads*).
Consequently, talking about **expected speedup results**, one should expect timings to follow Map pattern logic.
Going more in detail, the *completion time* can be defined as:

$$CT = \ t_{read} + t_{computeKNN} + t_{write}$$

Where reading and writing times are part of the *serial fraction* of the application.
Therefore, in such an application all speedups take place in the computation of KNN. In particular, given m points in a 2D space:

$$t_{computeKNN} = \ m \cdot t_{singlePointKNN} + t_{resultReduce}$$

Leading to an expected speedup, given nw workers:

$$speedup(nw) = \frac{t_{computeKNN}}{nw \cdot (t_{split} + t_{merge}) + \frac{t_{computeKNN}}{nw}}$$

Hence $speedup(nw) < nw$, as is obviously known.

3 Implementation Choices

Implementations are similar and they both have an interface which answers to two different user needs:

1. *generate, read, compute, write* (if one wants to create new tests, saved by default to *points.txt*):
   executable_name n [seed] [pardegree] [k]
   - **n**: number of points to be generated
   - **seed**: for replicability (*default: 123*)
   - **pardegree**: number of workers (*default: 1*)
   - **k**: number of nearest neighbour per point (*default: 5*)
2. *read, compute, write*:
   executable_name filename [pardegree] [k]
   - **filename**: path to file containing points in space
   - **pardegree**: number of workers (*default: 1*)
   - **k**: number of nearest neighbour per point (*default: 5*)

Taking a closer look at the practical implementation, both the parallel structures are fairly standard:

- **C++ threads:** implemented as a Map pattern, each thread is given an equal amount of points for which knn has to be computed. This follows a *static partitioning* fashion since all points require the same amount of work. Moreover, to efficiently produce the resulting string to be output to file, each thread constructs a local resulting string, behaving as a *local reduce*. The final global reduce is consequently composed of *nw* iterations and is done sequentially. This kind of implementation, mixed with the extremely low amount of updates on shared variables (done only once at the very end of each thread's execution) lead to avoiding problems such as *load balancing* and *false sharing*.
- **FastFlow:** implemented with a *ParallelForReduce* object, which in *FastFlow* is constructed as a *Farm* where the collector has been removed.

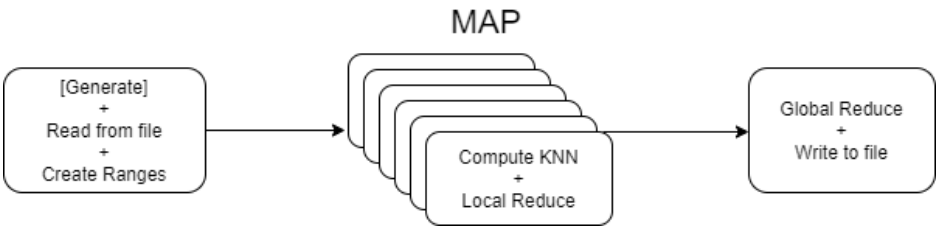

Figure 1: C++ stdlib implementation

Each thread is given again the same amount of work by selecting *static partioning* mode. The *map* part stands in finding the knn and updating a local string for later final *reduce*, which just serves to create the overall string.
Since this application is a one-shot, meaning that all the work can be done in one *parallel_reduce* call, in the implementation the object is non-instantiated, preventing useless overhead. The method *ff::parallel_reduce* is instead called directly, creating threads and destroying them at the end of the call. Note that since the reduce variable is a string, the identity value is in this case the empty string.

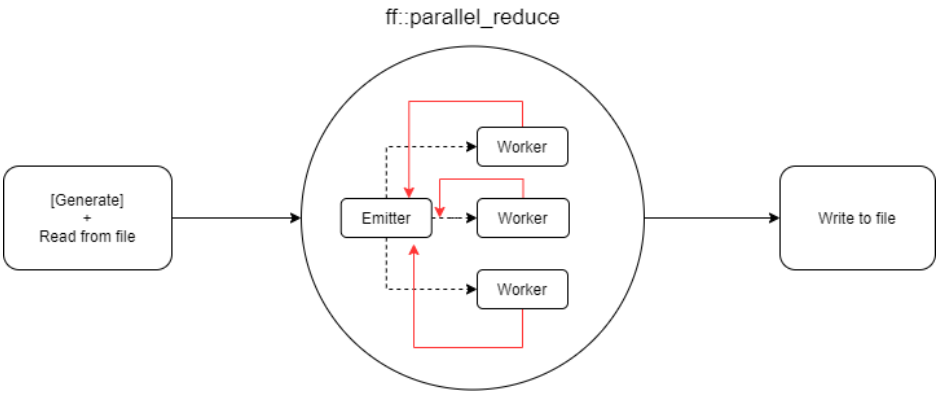

Figure 2: FastFlow implementation

4 Experimental Scenario and Setup

Results shown in the next section are captured by execution on an institutionally provided machine and on my personal computer:

- **64-core with 4 way hyperthreading Intel XEON PHI** (frequency of 1.30GHz)
- **6-core with 2 way hyperthreading Intel i7-8750H** (frequency of 2.20GHz)

Other preliminary informations:

- Timings reported in next section are expressed in microseconds (using *std::chrono*) and are taken **averaging** 5 different executions per configuration
- *Parallel time* and *sequential time* are the execution times without taking *generation, reading* and *writing* times into consideration (*read/write* times are shown in Table 4)
- Each implemented algorithm (in *XEON PHI* case) was tested with $p \in \{2^0, 2^1, ..., 2^8\}$
- Each implemented algorithm correctness was tested with the sequential execution
- Points were created using a pseudo-random generator

For execution on your machine the setup is straightforward: unzip and run *make all* on bash, single executable interfaces were previously explained.

5 Experimental Results

Experiments shown in this section focus initially on running both requested algorithms with an increasing amount of points in the 2D space.
After that, to highlight the effect of increasing the knn number, similar but reduced experiments are presented.
Lastly, mainly for fun, algorithms were run on my local machine (*Intel i7-8750H*) to see the effect of a better processor in action.

5.1 XEON PHI - 5 knn

Experiments were conducted taking into consideration 2D spaces composed of 10k, 50k and 100k points. The aim was to get a deeper analysis and highlight the execution differences from quite sparse to fairly largely populated spaces. Looking at Table 1 it can be noticed how the two approaches do not give extremely different results but at the same time some careful assertions can be done. Looking at the 10k case, *FastFlow* under-performs with respect to the C++ implementation, reaching a *knee point* at parallel degree **128**. This can be motivated by the slightly more complex structure underlying the *ParallelForReduce* object, which overhead is enough to weight on the execution performance when the workload is low. In the 50k case the differences become even subtler, with the C++ implementation still having the lead, but removing the *knee point* previously presented by *FastFlow*.
The 100k case allows for an even greater *overhead hiding*, keeping once again the two approaches near in timings. *FastFlow* actually out-performs the C++ implementation when the parallel degree reaches its maximum, an event which also happens with an higher number of knn (shown in next section). This behaviour might be partly explained by the presence of a sequential *reduce* at the end of the C++ implementation, adding overhead as the parallel degree increases.

Speedup and *scalability* results are shown in Figure 3, expressing once again what discussed above.
The *speedup* was calculated as $sp(nw) = T_{seq}/T_{par}(nw)$.
The *scalability* was calculated as $scalab(nw) = T_{par}(1)/T_{par}(nw)$.
As one might expect, the increase in number of points in a 2D space translates to an increase in the workload and therefore an higher speedup. The *speedup* curve is anyway far from ideal, in fact the algorithm behaves almost as expected up to parallel degree **32** (in the sparser spaces it diverges even before) before diverging from the ideal line.
The divergence can be explained by the fact that the workload per thread decreases as the number of threads increases, reaching a point where the *overhead of forking, joining and communicating with threads* becomes an observable aspect in the overall execution time, not anymore hidden by the parallel execution.

Running algorithm for 5 knn (μsec)						
ParDegree	FastFlow 100k pts	C++ Thread 100k pts	FastFlow 50k pts	C++ Threads 50k pts	FastFlow 10k pts	C++ Threads 10k pts
Seq	302249989	302249989	74598775	74598775	2983577	2983577
1	312475379	312402353	76247524	76339278	3080964	3117707
2	166124554	167633212	40609916	40598418	1646762	1640429
4	84786583	84413922	20703797	21020605	835612	824171
8	42508942	42446516	10710799	10940204	459655	417741
16	22970225	21652235	5736439	5761780	255735	217530
32	12334895	13778168	3650744	3708465	174810	172635
64	8091502	7811788	2515890	2528439	128839	117621
128	5769036	5697786	1843226	1780527	104615	99030
256	4629888	4803251	1356079	1339770	110730	97269

Table 1: execution times taken on XEON PHI machine

5.2 XEON PHI - 200 knn

This section serves to prove how increasing the workload translates into a better hiding of the parallel overhead costs and therefore to better *speedup* and *scalability*.
Once again, peeking at Table 2, the two approaches produce similar performances, with *FastFlow* slightly out-performing C++ implementation in some cases and viceversa.
Increase in performance is outstanding looking at the *speedup* and *scalability* results shown in Figure 4. For example, the *speedup* with respect to the 5 knn case improved from an approximate **60x** to **80x** when the 2D space is populated by 100k points, and improved from **30x** to **60x** when the 2D space is composed of 10k points.

Running algorithm for 200 knn (μsec)				
ParDegree	FastFlow 100k pts	C++ Thread 100k pts	FastFlow 10k pts	C++ Threads 10k pts
Seq	581492740	581492740	21404969	21404969
1	583198820	592576606	21496165	21555433
2	306626274	301769180	11436365	11430849
4	155455865	152000501	5800799	5775530
8	78801933	81472581	2954318	2980150
16	39893459	41084033	1740906	1635407
32	24446799	25847542	884632	1076252
64	13662300	14342256	619351	616957
128	9781801	9083830	406581	406278
256	6830085	7241098	355512	366355

Table 2: execution times taken on XEON PHI machine

5.3 I7-8750H - 5 knn

This last section provides the execution times produced on a machine with much less cores but a better performing CPU. Results shown in Table 3 give an idea of how better is the performance on a more recent processor, providing a sequential time which is almost **11x** the *XEON PHI* sequential time on the same task.
Although this is true, working on a laptop, having better sequential performances and a more restricted degree of parallelism translate into a worse parallel performance, as shown in Figure 5.

Running algorithm for 5 knn (μsec)		
ParDegree	FastFlow 100k pts	C++ Thread 100k pts
Seq	28118863	28118863
1	28159372	28161610
2	16046766	16401383
4	9547729	9568565
8	5788809	5801995
12	6943231	7009267

Table 3: running times taken on i7-8750H machine

5.4 Efficiency Analysis

The three experiments performances can be wrapped up with an efficiency analysis based on Figure 5.
Efficiency is calculated as: $\varepsilon(nw) = T_{seq}/(nw \cdot T_{par}(nw))$.
As already noticed comparing the *XEON PHI* experiments, an higher workload leads to a better speedup, therefore it is not a surprise to see how configurations with more points resulted to be overall more efficient, the same can also be said about having an higher workload by increasing the number of knn.
Looking instead at results on my local machine (*I7-8750H*), where the execution times are orders of magnitude lower, the drop in efficiency is considerably faster compared to results on the *XEON PHI*.

6 Conclusions

Bringing the experience to a close, this project shows how an application to compute KNN for a given 2D space can be implemented efficiently with a fairly standard parallel pattern, either using C++ threads or using an off-the-shelf library as *FastFlow*.
Results shown are definitely similar using both implementation, showing the heavier weight of *FastFlow* implementation when the workload is low, a weight which fades away as the workload is increased.
From the implementation point of view, working with C++ Threads surely gives you more control as the implementation becomes extremely low level, but kudos to *FastFlow* for giving us an extremely straightforward, abstract and compact interface to parallel coding, delivering a faster implementation with respect to coding with C++ threads.

7 Referenced Figures

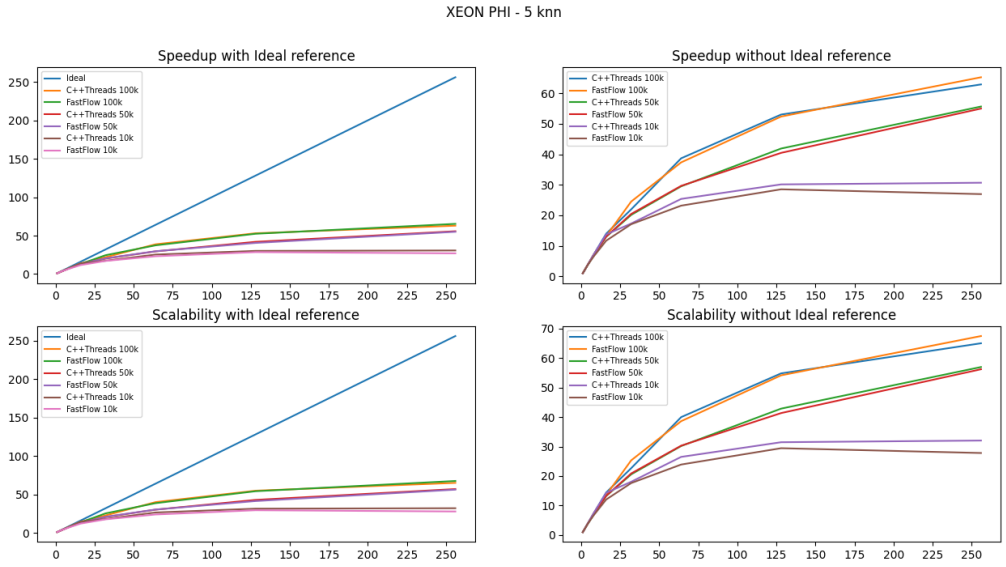

Figure 3: Speedup (first row) and Scalability (second row) results - Comparing with (first column) and without (second column) the ideal line

Read-Write time scaling (μsec)		
Execution Configuration	Reading Time	Writing Time
(10k pts - 5 knn)	11791	5006
(50k pts - 5 knn)	54954	18182
(100k pts - 5 knn)	108784	37705
(10k pts - 200 knn)	11320	84634
(100k pts - 200 knn)	109705	1409487

Table 4: read-write times taken on XEON PHI machine

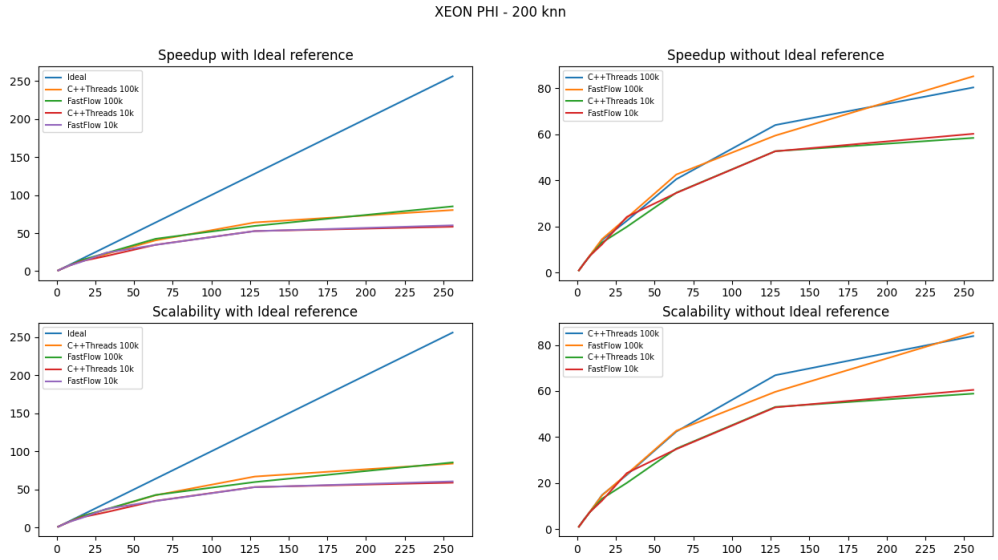

Figure 4: Speedup (first row) and Scalability (second row) results - Comparing with (first column) and without (second column) the ideal line

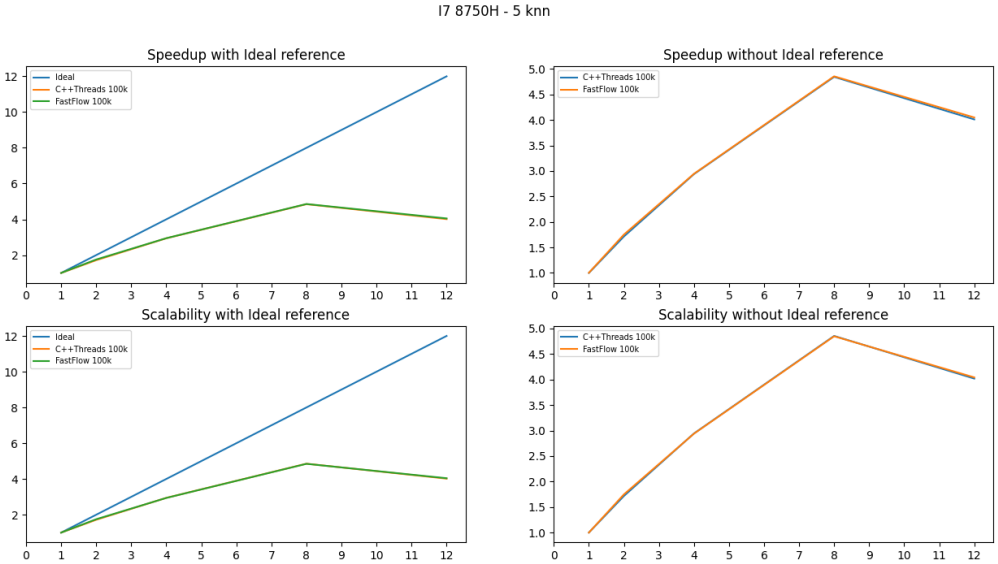

Figure 5: Speedup (first row) and Scalability (second row) results - Comparing with (first column) and without (second column) the ideal line

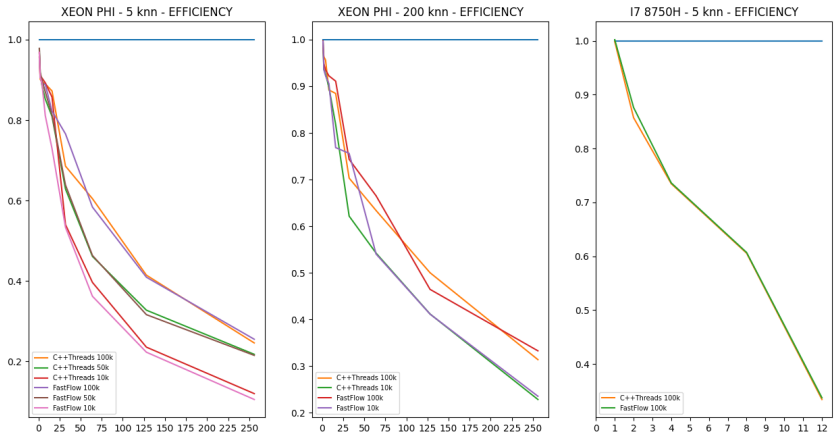

Figure 6: Comparing the three experiments in terms of efficiency