Parallel implementation of k-Nearest Neighbors

Marco Natali (m.natali10@studenti.unipi.it)

9 June 2021

1 INTRODUCTION

This project will analyze and construct a parallel implementation of finding the k-Nearest Neighbors of all 2D points where k Nearest Neighbors are defined as follows:

Def (k Nearest Neighbors)**.** Given a point $(x, y) \in \mathbb{R}^2$ and given a distance measure $d : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ we define k Nearest Neighbors the k nearest point to the given point computed by distance measure d.

Three different Implementation are provided:

- Sequential Implementation, used as a baseline for comparing performances.
- Parallel Implementation using C++ thread.
- Parallel Implementation using FastFlow library.

2 IMPLEMENTATION

As already mentioned in 1, we provide 3 different implementation, which differ only on how all k Nearest neighbors computation is implemented, so shared code has been declared in a header file called "utility.h" and defined in "utility.cpp" accessible by all implementations.
Common interface provided are the following:

DISTANCE FUNCTION: I have implemented the Euclidean Distance as a measure for distance between two points defined as

$$d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

k NEAREST NEIGHBORS COMPUTATION: to compute k Nearest neighbors function I have implemented the compute_knn function, where there is also a lambda function used to define the comparison between points needed to sort points by their Euclidean distance.

In addiction are defined read function to obtain 2D points from an input file and a write function used to write k Nearest Neighbors for each point on an output file.
To represent a 2D point I choose to use a 3-tuple with an identification number, a double x value, and a double y value.
To generate file with n 2D points I have created a C++ script called "generate_points.cpp" that generate n 2D points with a certain range of x and y values.

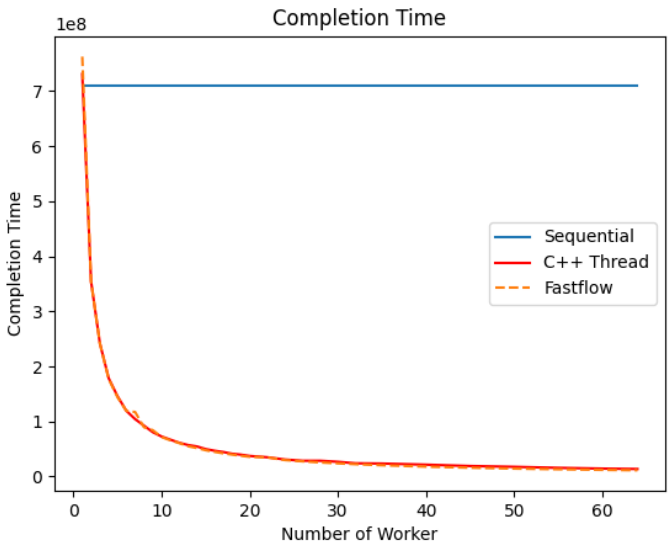

Figure 1: Completion Time achieved with Sequential, C++ Thread and Fastflow version with 50000 points and $k = 10$

2.1 Parallel Implementation using C++ Threads

In the C++ thread implementation I use a set of threads to compute in parallel the computation of finding k Nearest Neighbors for each point.
As Parallel pattern I have chosen to use a map pattern, since our computation is data-driven and consist to apply n times the same function, which is finding the k nearest neighbors of a point. I have chosen to avoid parallelizing the computation of k nearest Neighbors for a point to avoid introducing a more complex model, which would not improve the performance of our model.
Each thread receives a range of points which compute k Nearest neighbors and it is not needed resource management since all points are accesses only in reading, and also we create in advance the result vector, so for each point, we only write on their position in the result vector.
For load-balancing, no particular precautions have been taken, since each thread receives a more or less equal number of points which we need to compute k Nearest Neighbors (depends whether the number of points is divisible or not by the number of threads).

2.2 Parallel Implementation using Fastflow library

For Fastflow parallel implementation, I use the parallel_for instruction which allows parallelization of loops with independent iterations. This is implemented on top of the farm building block and it has a master-worker structure where the master is the Scheduler of loop iterations. As before only computation of all k nearest neighbors has been parallelized since it was not convenient to parallelize computation of k nearest neighbors of a point.
Also I avoid using any resource management since I create in advance the result vectors, where I save all k nearest points of each point. Also, no resource access restrictions are done to access points since to compute nearest neighbors are required only reading operations.

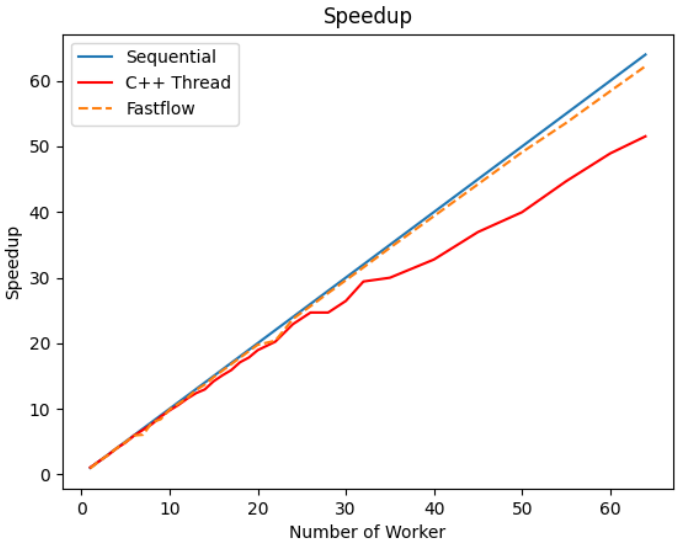

Figure 2: Speedup achieved by C++ Thread and Fastflow version compared with ideal ones, using 50000 points and $k = 10$

3 PERFORMANCE ANALYSIS AND EVALUATION

To compare performances of parallel implementations, two measures are adopted: completion time and speedup.

Def (Completion Time)**.** Completion Time is defined with the following formula

$$\text{Completion Time} = T_{end} - T_{start}$$

where T_{start} is the time when the computation start and T_{end} is the time where ends computation.

Def. Speedup measure is defined with the following formula

$$\text{Speedup}(n) = \frac{T_{seq}}{T_{par}(n)}$$

where T_{seq} is the completion time in our sequential implementation and $T_{par}(n)$ is the completion time spend in our parallel implementation with a parallel degree equal to n.

Given n points the time requires to compute k Nearest neighbors of a point is given by $n-1$ computation of Euclidean distance which requires $O(1)$, so a total cost of $n-1$. Given the time of the computation of a point, we have that the sequential time computation is $n * n - 1$ and $T_{par}(n) = n - 1 + T_{split} + T_{threadcreation}$, so the speedup obtained from our parallel implementation is

$$\begin{aligned} \text{speedup}(n) &= \frac{n * (n-1)}{n - 1 + T_{split} + T_{creation}} \\ &\approx \frac{n * (n-1)}{n-1} \approx n \end{aligned}$$

This upper bound however is over-optimistic because it considers overhead required to create a thread and split range data negligible with $n \gg num\_thread$.

4 RESULTS AND COMPARISON

I choose to compare our different comparisons using Speedup and Completion time, as introduced in 3 and in Figure 1 there is a graphical representation of Completion Time of Sequential, C++ Thread and Fastflow version using 50000 points and computing $k = 10$ Nearest Neighbors.

With 1 thread we have that C++ Thread version outperform Fastflow version since it seems to have less overhead and also sequential version perform better since it has no overhead to create a thread and compute ranges. Also, it can be noted that completion time continues to decrease with an increase in the number of processing units, which means that the tasks assigned to each worker require more time to be computed than the time to set up and manage concurrent activities.

In figure 2 we have speedup obtained by our two parallel implementations compared with ideal ones, where we can note that up to 30 processing units both C++ thread and Fastflow version have a speedup near ideal ones. With more processing units fastflow version still has speedup compared to ideal speedup, instead C++ Thread version has decreasing performances, due to thread creation overhead and time to manage concurrent activities.
This better performance of Fastflow is derived from the fact that Fastflow is an optimized framework for implementing parallelism, instead the version using C++ threads requires a higher time to manage parallel activities. From results obtained in experiments is possible to derive that both Fastflow and C++ thread version achieve more or less the ideal theoretical speedup and already with 50000 point we note that the choice to add a processing unit will improve performance since the overhead for thread creation is less than the time to compute k-Nearest neighbors.
As the number of workers increases, the Fastflow version performs better, since it is an optimized framework for parallel computation, and in C++ thread version to achieve the same performance will require more concurrent activities management.