Parallel K-NN in C++ and Fastflow

Final Project for Parallel and Distributed Systems (305AA) 2020/2021

Chenxiang Zhang
Department of Computer Science
University of Pisa
c.zhang4@studenti.unipi.it

Abstract

In the last few decades, the need of parallelism computation has continued to increase. In this work, we solve an embarrassingly parallel problem, K-NN, by implementing different architectures using C++ STL and FastFlow library. Then, we measure their execution time and analyze the results by comparing the performances using different metrics. We found out that overall the general performance follow the same trendline for all the architectures, with few quirks in the FastFlow version.

1 Introduction

The advancement in the technology together with the introduction of big data in the last few decade demands an efficient computation. This is achieved by implementing an highly and efficient parallelism code together with a multi-core hardware for any application that is computationally hungry.

In this work, we use the C++ language together with the FastFlow library [1] to solve a simple parallel problem K-NN for the final project of the course *Parallel and Distributed Systems*. We implement different architectures and compare their performances and results.

2 Experiments

2.1 Problem

"Given a set of points in a 2D space, we require to compute in parallel for each one of the points in the set of points the set of k closest points. Point i is the point whose coordinates are listed in line i in the file. The input of the program is a set of floating-point coordinates (one per line, comma separated) and the output is a set of lines each hosting a point id and a list of point ids representing its KNN set ordered with respect to distance."

The presented problem is a data parallelism problem. There is no dependencies during the computation of each point, which makes the problem embarrassingly parallel. The aim for this problem is to minimize the time needed to process the input data.

2.2 Architectures Design

To conduct our experiments we implemented four different architectures: Sequential, Parallel, FastFlow Parallel-For (FF-PF), and FastFlow Master-Worker (FF-MW). Given an input file that contains a number of 2D points, all the architectures will provide the same final result, that is each points paired with its closest k neighbors.

Sequential. The most simple architecture using only one single thread to solve the problem sequentially. The result of this architecture will be used to assess the correctness of the other versions. We will also use the time measurement of this architecture for the Speedup.

Parallel. This architecture implements a simple Map pattern using the C++ standard library. Given that the k-nn problem is an embarrassingly parallel problem, the set of input 2D points can be divided statically among the threads. Each thread calculates the range of the subset points it needs to compute. At the end of a worker's execution, its result is inserted in a shared variable using the lock system to prevent the race condition problem. The main thread waits for all the workers to finish their computation through a barrier system using the *join()* function, and in the end it sorts the shared variable preserving the order of the points.

FF-PF. This architecture uses the parallel-for pattern provided by FastFlow framework. The skeleton of this pattern is a Master-Worker, where the master is the scheduler and all the workers have a feedback channel towards the master. We set the static partition by assigning the parameter *chunksize* the value 0. Each worker computes a local partition that will be combined at the end, hence this implementation doesn't need a final sorting operation since the local partition is identified with a thread id, which preserves the final ordering during the merge.

FF-MW. This architecture uses the ff_Farm pattern provided by FastFlow framework. The skeleton of the farm is modified by replacing the standard emitter with a custom one, removing the collector, and adding the feedback channels from each worker to the emitter. The new emitter partitions the workload into #worker + 1 parts. All the partitions, except the last one, are distributed to each worker through a one-to-one channel. After the emitter has distributed all the work, it behaves as a worker and computes the last reserved partition and adds its result into the final result. Finally, the emitter acts now as a collector waiting the results of each worker through the feedback channels and adds them to the final results. At the end of all the computations, the emitter sorts the final results preserving the order of the points. This architecture design is similar to the FF-PF, the advantage is that it allow us to save one additional thread through the custom emitter. When running this architecture with parallelism equals to 1, the emitter will simply distribute all the work to a worker.

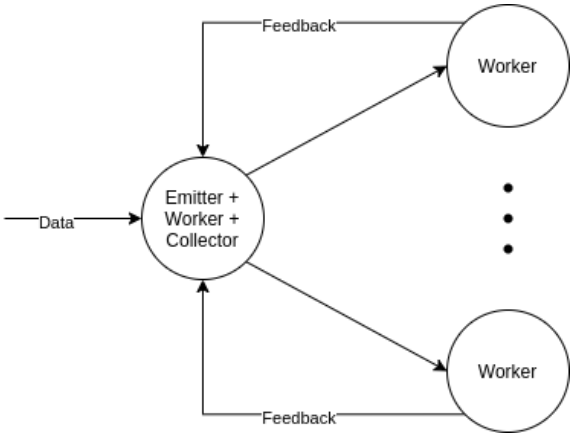

Figure 1: Architecture of FastFlow Master-Worker (FF-MW) version.

2.3 K-Nearest Neighbors Algorithm

Given all the *points* in the input file and a *range* defining the interval of points to compute the k-nn (in the sequential case *range* = (0, len(*points*)), we find the k-nn by using two nested loops where the first one loops from the *range* start to the *range* end and the second one loops over all the *points*. The

distance is calculated between the point in the first loop and the point in the second loop and it is either replaced or inserted in a data structure called *neighbors*. If the distance of the new point is less than the maximum point in the structure, then the maximum point is replaced with the new one. If the neighbors has less than K elements, then the new point is simply inserted.

The storage of *neighbors* is achieved using an efficient data structure *priority_queue* provided by C++. This structure is also known as a Maximum Heap of size K, maintaining the maximum element always on the root of the tree allowing a fast comparison of distance.

2.4 Experimental Setup

For the experiments, we compared the efficiency of all the architectures using increasing parallelism degree from 1 to 256. In order to reduce the presence of possible noise during the executions, each configuration is an average of 5 independent executions (time measurement μ). The measurement doesn't take into consideration the IO operations. All the experiments were run on the remote machine using the processor *Intel XEON PHI* 64 cores with 4 way hyperthreading and 1.3GHz computing power per core.

The input file was created using the code *generate.cpp*, it consisted in 100k 2D points generated pseudo-randomly in the range [-100000, 100000]. The number of neighbors for each point is K = 10. The function used to measure the distance is the Euclidean Distance.

We also try to empirically demonstrate the *Gustafson's Law* by running an experiment with a smaller dataset consisting in 10k elements and K = 1. This will demonstrate that as we decrease the dataset size, the efficiency will also decrease by exposing more the parallel overhead operations.

3 Results and Discussion

3.1 Expected Performance

We firstly define the completion time which is the time spent from application start to application completion. It is composed by IO operations read/write the data from/to a file and the computation of knn. The writing of the output to a file is optional.

$$T_{complete} = T_{read} + T_{knn} + T_{write}$$

We measured the performances of the Sequential architecture following the setup configuration described in the Section 2.4. The average results are $T_{read} = 734920\mu$, $T_{knn} = 311118000\mu$, and the $T_{write} = 846320\mu$. Since the IO operations are serial and given the Amdahl Law where if we prior fix the quantity of data then the Speedup is limited by the time needed for the serial computation. In this work we are only interested in optimizing the performance of T_{knn}, which is composed by an emitter time T_e, collector time T_c, and the actual computation time of the neighbors T_w.

$$T_{knn} = T_{seq} = T_e + T_w + T_c$$

The functions used as metric to measure the performances of the parallel architectures are the Speedup, Scalability and Efficiency.

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

$$Scalability(n) = \frac{T_{par}(1)}{T_{par}(n)}$$

$$Efficiency(n) = \frac{Speedup(n)}{n}$$

3.2 Comparison C++ and FastFlow

The performances on all the three different architectures follow a similar trendline among all the different metrics. As we can notice from the Figure 2, both the Speedup and Scalability follow an almost ideal growth until the parallelism reaches around 60. Afterwards, the growth starts to slow down until the parallelism reaches around 130 where it also almost reaches its maximum of parallel performances in both the metrics. From now on, the Speedup and Scalability will not show any significant improvement with the increasing of the parallelism. The slow down is caused by the increase of thread overhead operations with respect to the time spent in computing the data. Furthermore, we can observe a common heavy spike drop pattern among both the FF versions at parallelism around 64, 128, and 194. This is a problem likely related to the FF framework, since our C++ Parallel version presents more mild and irregular spikes.

The maximum Speedup reached for the architectures Parallel, FF-PF, and FF-MW are respectively 72, 76, and 65. While for the Scalability the maximum are respectively 79, 74, and 67.

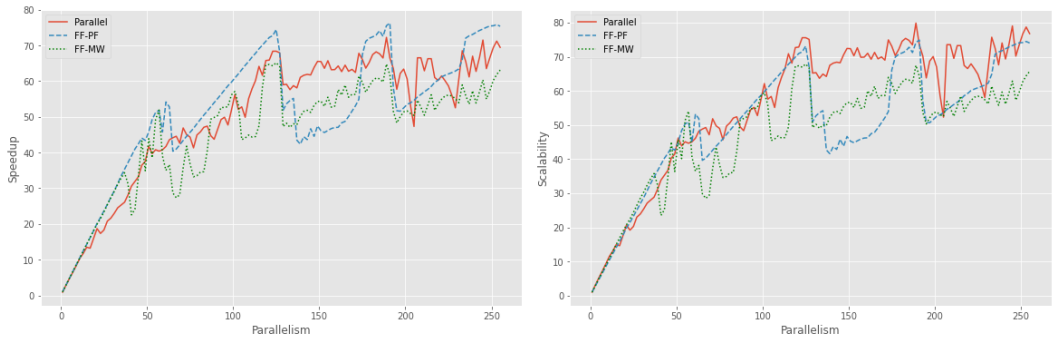

Figure 2: (left) Speedup and (right) Scalability comparison among all the parallel architectures.

The Efficiency reported in the left Figure 3 also highlights a common general trendline, but presents a noticeable differences in the start. All the FastFlow architectures retains a good efficiency until the parallelism reaches around 60. On the other side, the C++ Parallel version presents immediately a drop while both the FastFlow versions are able to maintain an high efficiency around 0.95%.

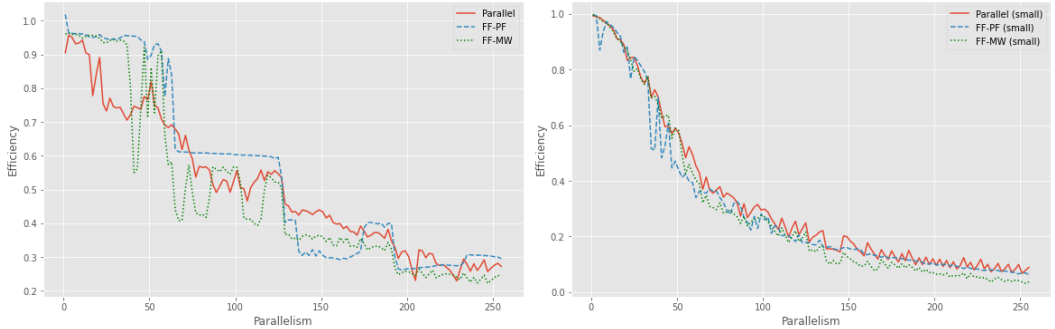

Figure 3: (left) Efficiency comparison among all the architectures. (right) Efficiency comparison when the dataset is small, composed of 10k points with K=1.

The right Figure 3 is where we confirmed empirically the *Gustafson's Law*. As we decreased the amount of data computation, the efficiency dropped with respect to the dataset with more data.

4 Conclusion

In this work, we have implemented three different architectures to solve an embarrassingly parallel problem, K-NN. The different versions consist in using purely the C++ STL for parallelism or to use the library FastFlow using the off-the-shelf skeletons. We found out that overall the general performance follow the same trendline for all the architectures: as the parallelism degree increases together with a fixed amount of computation, the performance growth decreased due to the Amdahl Law. The results from the FastFlow-ParallelFor skeleton was the most efficient in terms of Speedup, Scalability and Efficiency. But it also presented a sudden drop of performance at certain parallelism degree (64, 128, 194). While the C++ STL version follows an almost stead growth. Nevertheless, the easiness and the great performance that comes with the usage of FastFlow makes it a solid solution when dealing with the modern parallel problems.

4.1 Run the Project

Open the Makefile and check that the variable FF_ROOT is assigned to the correct FastFlow folder. Run the command *make all* to build all the executable. Run the executable to see which are the parameters needed for input.

We provide two simple bash scripts used during the experiments for testing the architectures:

- **run_single.sh**. This script runs an executable using a for loop to perform multiple iterations and outputs the average result.
- **run_complete.sh**. This script runs an executable with two for loops. The first one iterates over the parallelism degree range, and the second one iterates over the number of iterations for each parallelism. The output is formatted as a csv file, where every line has two values: *parallelism_degree, average_time*.

References

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *Fastflow: High-Level and Efficient Streaming on Multicore*. Programming multi-core and many-core computing systems (eds S. Pllana and F. Xhafa), 2017.