University of Pisa
Graduate Programme in Computer Science

K-Nearest Neighbors(KNN)

Yohannis Kifle Telila

Student ID : **621821**

Parallel and distributed systems: paradigms and models
Academic year 2021/2022

Contents

Chapter 1

Introduction

1.1 Problem description

In this project I will be analysing and comparing performance of different implementations of K-Nearest neighbours algorithm. The problem statement is provided as "given a file containing 2D points separated by comma, one point per line the problem is to find K Nearest points for every points in the file". The distance is measured using euclidean distance measurement. Point i is the point whose coordinates are listed in line i on the file. 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 problem of finding a K-nearest neighbour of a given points can be parallelized to speed up the computation. We can categorize this problem as **data parallel computation** or **embarrassingly parallel computation** because its possible to divide our computation to a number of workers and combine the result at the end and communication between the threads(workers) is not needed.

In this report, I will be discussing the results I get from three different implementation of the problem and improvements on the original implementation to get better performance of the problem. The first method I used to implement is pure STL C++ in sequential method. This implementation is provided in the file **stl_seq_knn.cpp**. Secondly, I used the same implementation and parallelized it to speedup the computation using multiple threads. The implementation of this version can be found in **stl_par_knn.cpp**. Thirdly, I used openMP `parallel for` to implement the same problem in a parallel form. The code for this implementation can also be found in the file `openmp_par_knn.cpp`. Finally, I used FastFlow parallel for reduce version where the implementation can be found `ff_pf_knn.cpp` in the file provided along with this report file.

1.2 Algorithm

As stated in the problem statement, we are interested in finding the k-nearest neighbours for each given points in 2D space. To find k-nearest neighbour of a point, the first solution would be to sort all the points based on the distance, from nearest to the furthest and selecting the top k nearest points which will be our K-nearest neighbours for that point. But this would require to store all the points which might not be efficient in case we have huge amount of points. The best solution would be to store only the result in priority queue of size k and continuously update our k points and discard other points. The pseudo code for sequential implementation is provided below.

Algorithm 1 K-nearest neighbour

```
1: points2D ←read2Dpoints(filename)
2: vector < knn_res >  knn_imp_result
3: for i = 0, 1, 2, ..., points2D.size() − 1 do
4:     knn_i ←get_knn(points2D, points2D.size(), i,k)
5:     _Insert(knn_imp_result,knn_i)
6: end for
7: print(knn_imp_result)
```

`knn_res` is a user-defined data structure containing `int index` to store index of the point and `vector<int> knn_index` to store index of all the nearest points. The method `get_knn(points2D,points2D.size(), i, k)` will receive the 2D points read from the file, the size of the 2D points, point index $\boldsymbol{i}$ and number of nearest neighbours $\boldsymbol{k}$ and returns a vector containing $\boldsymbol{k}$ nearest points for the point $\boldsymbol{i}$. The pseudo code for this method provided at Algorithm 2.

The completion time or T_c of this algorithm can be estimated as the sum of time it take to read the points from a file T_{read}, the time it takes to compute the nearest neighbours of the points $T_{compute}$ and the time it takes to print the result to a file T_{write}.

$$T_c = T_{read} + T_{compute} + T_{write} \tag{1.1}$$

The time $T_{compute}$ is the largest of all three time measures which could be reduced significantly by farming the computation to multiple workers. The time T_{read} and T_{write} are done sequentially and can't be parallelized. These two time measures are accountable for sequential fraction of our code. But for parallel version of our code the sequential fraction of our code would also include the time it take to divide the task to the threads(workers) T_{divide} and $T_{collect}$. If we managed to make small change in our implementation it could be possible to remove the time it takes for partial results collection $T_{collect}$. So the T_c for parallel version is given as.

Algorithm 2 get_knn method pseudo-code

```
for j = 0, 1, 2, ..., points2D.size() − 1 do
    if J not equal i then
        d ←measure_euclidean_distance(points2D[i], points2D[j])
        if kneighbours_size < k then
            _Insert(kneighbours,d,j)
        end if
        if kneighbours_top_distance > d then
            _kneighbours.remove_from_top()
            _Insert(kneighbours,d,j)
        end if
    end if
end for
for i = 0, 1, 2, ..., k − 1 do
    _Insert(knn_res_i,kneighbours.top().index)
    _remove_from_top_kneighbours.
end for
return knn_res_i
```

$$T_c = T_{read} + T_{divide} + T_{compute} + T_{write} \tag{1.2}$$

The sequential fraction for parallel version would be the sum of T_{read}, T_{divide} and T_{write}. The parallel fraction would be $T_{compute}$.

Test Data	T_{read} (sec)	T_{write} (sec)
input_20k_s123.txt	0.157	53.364
input_50k_s456.txt	0.373	262.247
input_100k_s789.txt	0.737	1125.713

Table 1.1: Read and write time for $k = 50$

For the sake of the experiment, I excluded the T_{read} and T_{write} for each experiments which can be achieved by placing $-d$ flag during execution. Detail will be provided in the last chapter.

Chapter 2

Parallel Implementation choices

In this section, I will discuss different parallel implementation choices I experimented with.

2.1 C++ STL Parallel

My first implementation was to parallelize the for loop computation on line 3 of Algorithm 1 using threads. There is the second for loop inside `get_knn()` method on algorithm 2 which could also be parallelized. If we were to parallelize this for loop we need to synchronize the workers which creates an additional overhead. Depending on the available resources, we could potentially get a better performance parallelizing both the outer and inner for loops. For this experiment I only parallelized the outer for loop and the inner for loop is performed sequentially by each workers.

The amount of task between the workers is distributed in a static partition schema. I divided the data-set into equal size of chunks based on the number of workers and the first worker gets the first chunk, the second worker will get the second chunk and so on. This way all the workers will get fairly equal amount of work. The work load of the workers is also fairly balanced since each workers get same amount of data. Ideally, the time it takes to compute the k-nearest neighbours of any point should be equal.

Each workers writes their partial results `par_res_chunk` of the knn computation to the global result `knn_par_result`. Since many workers could try to access the global result at the same time, I used `std::mutex` to synchronize access to this global variable `knn_par_result`. The C++ STL implementation can be visualized as figure 2.1

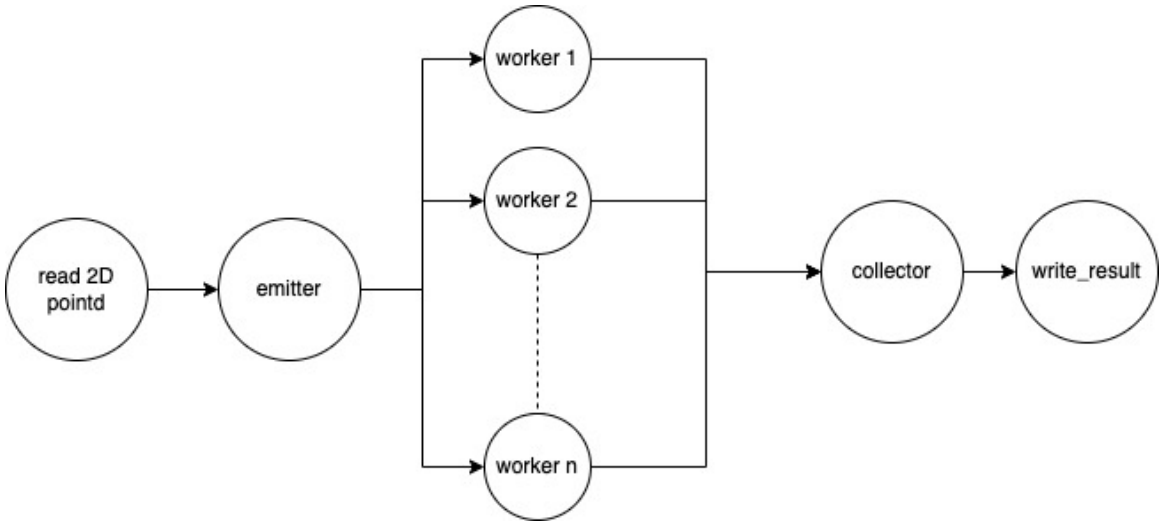


Figure 2.1: C++ STL implementation

The code implementation is found under the name `stl_par_knn.cpp` and the parameter to this implementation are the number of workers, the value of `k`, input file data containing the 2D points and optional flag `-d`.

2.2 FastFlow

I used a `ff::parallel_reduce` method which provides easy interface to map operation over the number of workers and do the reduction easily for iteration in-dependant operations. I used static partitioning, which will finally produce ordered output.

2.3 OpenMp

For the openMp version I used `#pragma omp parallel for`, which spawns a number of threads where each threads executes a chunk of of the entire for loop. I used the static scheduling for the task distribution among the workers. Diagram for this implementation can also be visualized as in 2.1. Synchronization of the thread is performed using `#pragma omp critical` where the partial result computed by the threads is merged with the global result.

Chapter 3

Experiment setup and performance measurements

To test the performance of my implementations, I used a remote machine with processor ***Intel Xeon PHI 64-cores*** **with 4 way hyper-threading**. Each core has the speed of 1.30GHz. In every experiment I excluded the time for for reading the file and writing the results. But the time for dividing task and collecting the partial results from the workers in parallel implementations is included in the time measurement. The read and write time is provided in table 3.1

The test was performed on $nw \in \{1,2,4,8,16,32,64,128,256\}$ and all execution time was measured using `utimer.cpp` class which uses RAII (Resource acquisition is initialization) for measuring time. the experiments were performed with k=**50** and the time was averaged over three runs.

3.1 Data-set

As the input for the algorithm and to test the performance of the algorithm, I generate three different size of 2D points files. Namely, `input_20k_s123.txt`, `input_50k_s456.txt` and `input_100k_s789.txt` which all can be found inside `/data` folder.

Test Data	Size	Range	Seed
input_20k_s123.txt	20,000	(-100,100)	123
input_50k_s456.txt	50,000	(-100,100)	456
input_100k_s789.txt	100,000	(-100,100)	789

Table 3.1: Dataset used for experiment

These three data-set would help us understand how our algorithm would behave in smaller, medium and bigger data sets. The code used for the data-set generation

can be found at `generate2d.cpp`. Guide on how to run this code is provided in the last chapter.

3.2 Speed up(S_p)

The speed up of a parallel algorithm is given as the ratio of best execution time in sequential algorithm over to the time for parallel algorithm. Mathematically speed-up can be expressed as.

$$S_p(nw) = \frac{T_{seq}}{T_{par}(nw)} \tag{3.1}$$

The speedup indicates the improvement in performance from adding more processing units to the system. Ideally, we would expect the speed-up to be equal to the number of workers(threads) but this is usually will never be quite achieved because of the overhead that will be introduced as we add more processing units to the system.

3.3 Scalability(S_c)

Scalability of a parallel program with nw workers can be expressed as the ratio between time spent using only one processing unit over time spent using nw workers. This can be expressed mathematically as.

$$S_c(nw) = \frac{T_{par}(1)}{T_{par}(nw)} \tag{3.2}$$

The value we would get from calculating scalability will tell us how much we can improve when we add more computing resources. Again, ideally we would expect a linear increase in scalability as we add more workers to our system but its impossible to achieve this result because of the overhead that will be introduced.

3.4 Efficiency(E_f)

The Efficiency of a parallel program is the ratio of speed-up we obtained over number of workers or processors used which can be expressed as.

$$E_f(nw) = \frac{Sp_{nw}}{nw} \tag{3.3}$$

The max value for $E_f(nw)$ is 1.

Chapter 4

Results

In this chapter I will be discussing the results found by running the implementation choices described under chapter 2. Only the plots for the `input_100k_s789.txt` data-set will be provided in this section. The plot for the rest of the data set will be provided at the end of this report page.

4.1 Sequential implementation

The sequential implementation is the one provided in algorithm 1. The code for this implementation can be found at `stl_seq_knn.cpp`. The table 4.1 summarizes result I achieved over the three data sets.

Threads	Test Data	Time(sec)
1	input_20k_s123.txt	14.7877
1	input_50k_s456.txt	90.069
1	input_100k_s789.txt	349.335

Table 4.1: STL C++ Sequential result

This result will be used to measure the speed up of the parallel implementations.

4.2 Speed-up comparison

From the speed-up graph of the three implementation choices depicted in Figure 4.1 we can understand that the FastFlow version of the parallel program achieved a better performance compared to the other two as the number workers grows.

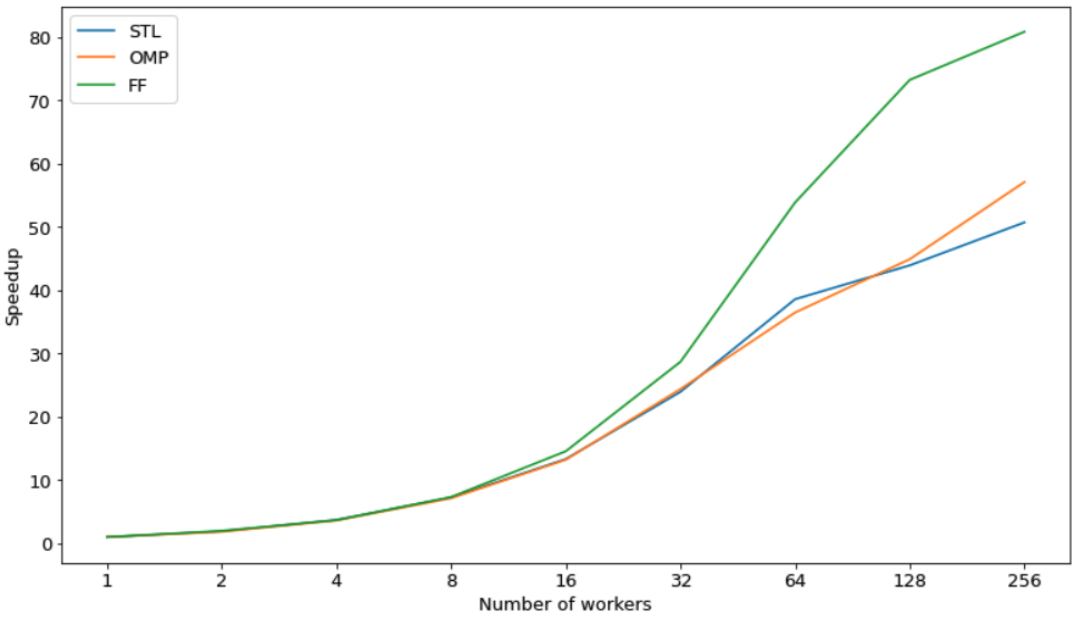


Figure 4.1: input_100k_s789, Speed-up plot, K=50

From the plot, we can observe that all three methods followed the ideal speed-up until the number of workers reaches **32**. While the FastFlow version showed even a more closer speed-up value to the ideal speed-up until the number of workers reach around 50, the other methods diverged and plateaued from ideal speed-up. This is mainly caused by the overhead and bottlenecks that comes with the increase in the number of workers. In the next chapter, I will discuss more in detail possible improvements on openMp and STL Cpp version to increase speed up.

4.3 Scalability comparison

The same observation as in the speed-up can be seen in the scalability plot. We would expect an increase in scalability as the number of workers gets bigger.

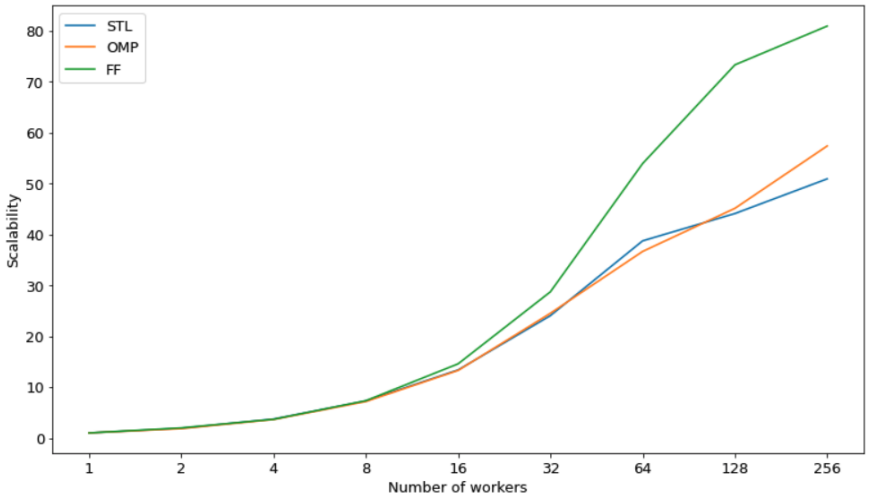


Figure 4.2: input_100k_s789, Scalability, K=50

As we have discussed in chapter 3, it could difficult to get to the ideal scalability which we can observe from the plot. As the number of workers increase, the scalability becomes more and more less than the ideal value.

Interestingly, as we would expect, as the amount of data-set gets bigger, the overheads will be become more and more negligible. We can observe this from the result of the three data-set. With data-set containing 20,00 points, the overhead starts to make an effect starting from around nw = 8. Whereas, with the data-set 50,00 points, the effect of the overhead stops being negligible starting from nw=32. For the data-set with 100,000 points, the critical points is around nw=50. This is what is stated in the Gustafson's law.

4.4 Efficiency comparison

Here again the efficiency quickly drop for the two implementation openMp and STL C++ after 8 number of workers where as the efficiency was between 1 and 0.8 until the number of workers reaches 70 for the FastFlow version.

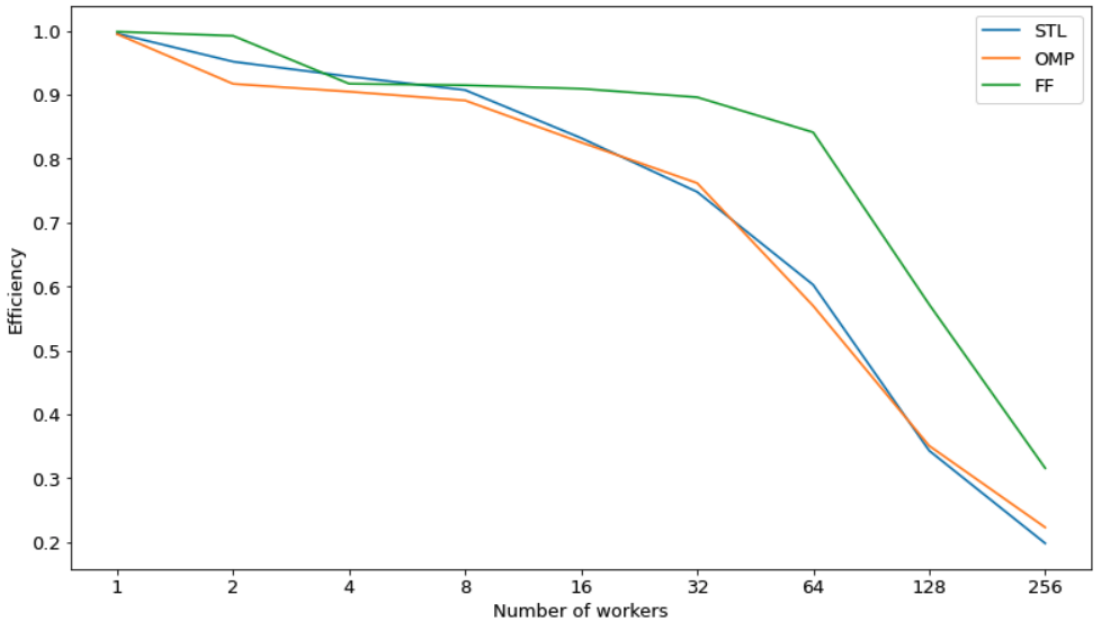


Figure 4.3: input_100k_s789, Efficiency, K=50

Chapter 5

Performance improvement

From the previous result chapter, it is clear that the two implementation choices, STL CPP and openMp didn't quite show a good performance. The obvious one reason is the bottleneck, where each thread writes their partial result to the global result. Let's see this improvement in detail and the result I was able to get.

5.1 Removing bottleneck

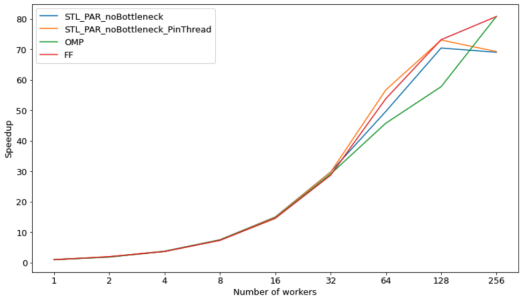
One way to avoid this bottleneck is to create a vector of size equal to the number of workers and passing the address of the index to the corresponding thread. This way, each thread will be able to write their results to the global result without the need of exclusive access to the global result. The updated implementation of parallel STL C++ can be found at `stl_par_knn_nobottleneck.cpp` and for openMp the code can be found at `openmp_par_knn_nobottleneck.cpp`.

5.2 Pinning thread

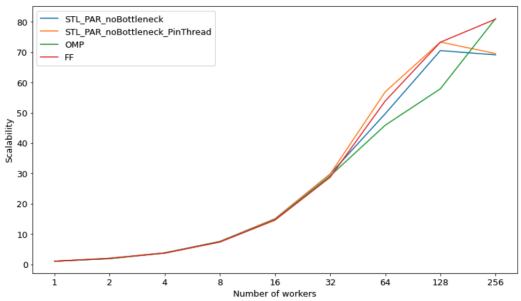
The other potential event that could cause additional overhead is the movement of threads from one core to another. During the execution the threads may be subjected movements to another core and this is totally handled by the operating system. When a thread is moved to another core, all the data that is stored in a cache will also be moved to the new core which will cause an overhead. The solution to this event is to pin the threads to a core during their entire execution period. Thus, saving the time it take to move one threads from one core to another.

5.3 Updated Result

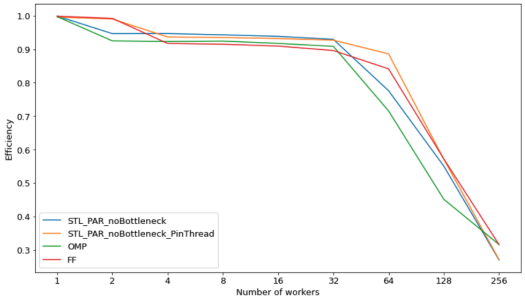
I re-run the test with the updated version of the implementations and I was able to almost match the performance I was able to get using FastFlow. Again, I will only be



(a) Speed up



(b) Scalability



(c) Efficiency

Figure 5.1: input 100k s789, K=50

showing the results I managed to get using the data-set `input_100k_s789.txt` here.

From the plot we can understand that FastFlow and optimized STL C++ parallel implementation showed a better performance in all three measurement. Interestingly, from the plot we can observe that the the movement of the threads from one core to another indeed increases as the number of workers increases; creating an overhead.

Chapter 6

Conclusion

In a smaller data-set, openMp seems to be performing better, where as in bigger data-set FastFlow and optimized STL parallel starts to shine. Though getting the ideal result is impossible, I was able to get relatively good performance in all implementations.

This project helped me to understand how to analyze and optimize sequential code for much better possible execution time and how to wisely utilize available resources. Even though using pure Standard Template Library gives you much more flexibility in your implementations and design choices, It is amazing to see how the libraries I used provided me similar or better performance with a simple and small lines of codes. Especially, I am surprised by the performance gain and simplicity to use FastFlow.

6.1 Future works

To store k-nearest neighbours, I used a priority queue over a vector with custom compare function which gives us a time complexity of $O(n)$ in insertion and deletion operations. We could improve this complexity by using a heap which will give us a time complexity of $O(logn)$ in insertion and deletion operation and $O(1)$ in finding max and min which could be a better improvement if we were to use bigger $k-value$. I left this change for future improvements.

It is also possible to remove the square root found in the formula to calculate euclidean distance. We could save additional computation time without changing the final result.

6.2 Folder structure and how to build project

In the file attached along with this report, all implementation files and necessary scripts to successfully run the program will be found. The file contains 6 folders.

- `\bin` folder contains all the executable files.
- `\data` folder contains the three data-set used for testing the different implementations.
- `\fastflow` is where the FastFlow library is cloned.
- `\outputs` folder contains the output of each algorithms or implementations. Each line contain a point id and list of space separated k points closest to the point.
- `\src` folder contains utimer class for measure running time of the algorithms and utility functions and data structures used in the program.
- `\Test-result` folder contains test result data and visualization of the plots.

To build the project issue the `make all` or just `make` command from inside the folder `\Multi-Thread-KNN`. This command will generate the executables inside `bin` folder. Running each executable with no parameter will provide the correct parameters and their orders. Each executable have `-d` optional flag which basically ignores writing the result to the file.

To run a single script, the following issue can be issued from inside the project folder. Important thing is that, `/bin/stl_seq_knn` only has two important parameter, `k-value` and `test-data`.

```
#!/bin/bash
$ ./bin/stl_seq_knn 50 data/input_50k_s456.txt          #optional -d
```

For the other implementations, there is `nw`, number of workers important parameter. In the following command `nw` is provided as 256.

```
#!/bin/bash
$ ./bin/stl_par_knn 256 50 data/input_50k_s456.txt    #optional -d
```

Additionally, one bash script called `executeTest.sh` is provided to aid the testing of the implementations. The parameter for this script are `k-value` and `runs`. Running time of each algorithm will be averaged over the number of `runs`.

```
#!/bin/bash
$ ./executeTest.sh 50 3     # k-value = 50 and runs=3
```

To generate test data `/bins/generate2d` executable can be used. The parameters are `start`, `end`, `seed`, `amount` and `file name` respectively. For `file name` you don't need to specify file extension. The following command will generated 1000 2d points, each coordinate value between -100 and 100 with 123 seed value. File will be automatically stored in `data/input_1k_points.txt`

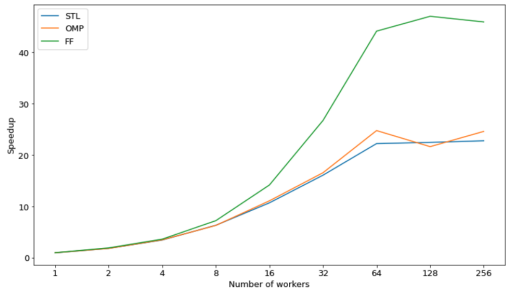
```
#!/bin/bash
$ ./bin/generate2d -100 100 123 1000 input_1k_points
```

Bibliography

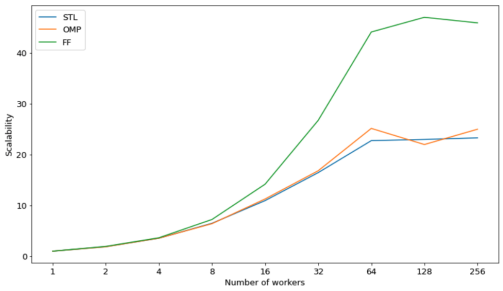
[1] M. Danelutto, DISTRIBUTED SYSTEMS: PARADIGMS AND MODELS, 2014.

[2] Structured Parallel Programming: Patterns for Efficient Computation - Michael McCool, James Reinders and Arch D. Robison.

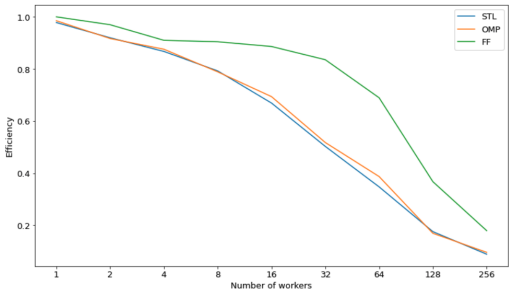
[3] https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf



(a) Speed up

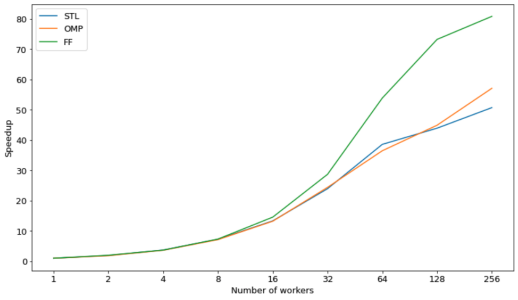


(b) Scalability

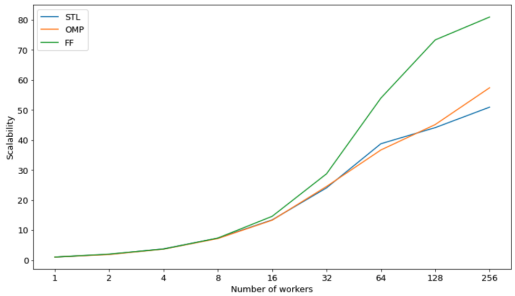


(c) Efficiency

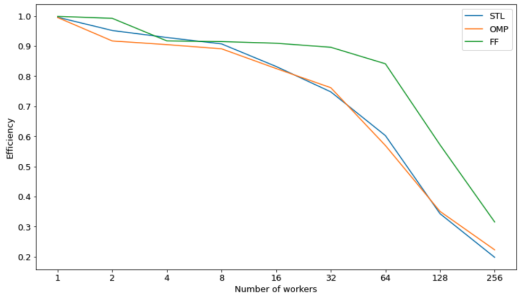
Figure 1: input_20k_s123.txt, K=50, with no performance improvement



(a) Speed up

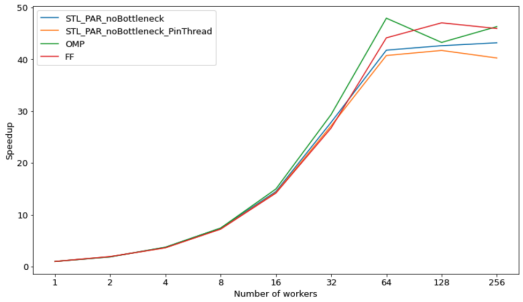


(b) Scalability

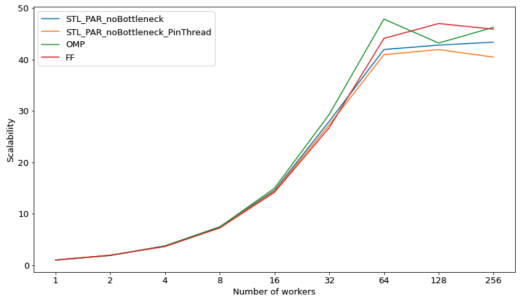


(c) Efficiency

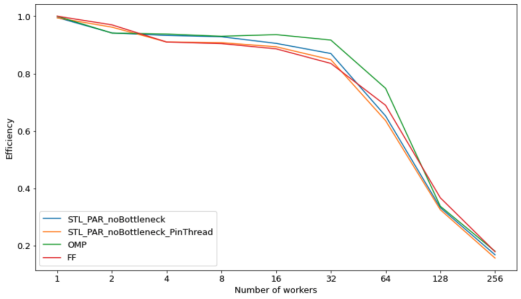
Figure 2: iinput_50k_s456.txt, K=50, with no performance improvement



(a) Speed up

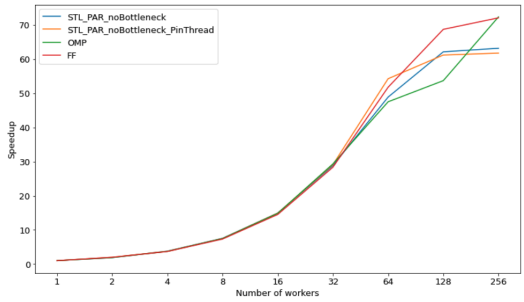


(b) Scalability

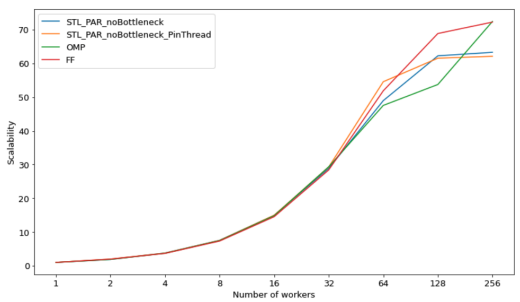


(c) Efficiency

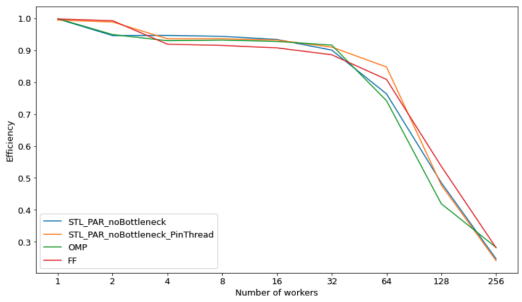
Figure 3: input_20k_s123.txt, K=50, with performance improvement



(a) Speed up



(b) Scalability



(c) Efficiency

Figure 4: input_50k_s456.txt, K=50, with performance improvement