K Nearest Neighbours - gRPC

Distributed Systems : Assignment 4 - Q2

October 21, 2024

INTERNATIONAL INSTITUTE OF
INFORMATION TECHNOLOGY

HYDERABAD

1 Introduction

The major assumption is that there are no *server crashes*.

Assume there are n data-points in total, and s servers. We have to find the k nearest-neighbours of some arbitrary data-point.

1.1 Sequential Execution

Without distributed computation, we would have to check the distance from the data-point sequentially, for each prospective neighbour, while pushing to a max-heap that compares distance from our given data-point. The size of this max-heap never exceeds k. This leads to a total time-complexity of:

$$O(n \cdot log(k))$$

1.2 Parallel Execution

If we could instead distribute approximately $\frac{n}{s}$ neighbours across s servers, then each server could calculate its local k-nearest neighbours in time $O(\frac{n}{s} \cdot log(k)$. Since these execute parallely, the time complexity of this section is:

$$O(\frac{n}{s} \cdot log(k))$$

Note: we assume minimal / constant time taken for communication, here.

Then, we gather these results into a list of s collections of k nearest neighbours. These can be merged efficiently using divide-and-conquer, first getting the k nearest-neighbours of the left half, then the right, then merging them together. The time taken by this can be formulated as:

$$T(s) = T(\frac{s}{2}) + O(k \cdot log(k))$$

$$\rightarrow O(k \cdot log(k) \cdot log(s))$$

Thus, the total time-complexity is:

$$O(\frac{n}{s} \cdot log(k) + k \cdot log(k) \cdot log(s))$$

Thus, **gRPC** facilitates a significant speed-up by allowing us to distribute expensive computation across servers.

2 gRPC vs MPI

2.1 Communication Models

- *MPI* uses a tightly-coupled message passing model - processes directly send and receive messages to/from each other. This allows for low-level control and optimisation of communication.
- *gRPC* uses a more loosely-coupled model, where client processes invoke remove functions on server processes. The purpose of gRPC is to abstract away the nitty-gritties to make it easier to develop distributed applications, but it may have higher overhead compared to direct message passing.

Here, "coupling" simply refers to the amount of inter-dependence between the processes running. To be succint, *gRPC* is far less "aware" of how and what the server is doing than *MPI* allows.

2.2 Usability

- *MPI* in general is more complex. It requires emssage-passing and explicit coordination between processes, and we have to coordinate effectively to ensure no adverse results.
- *gRPC* has a simpler interface that allows us to focus on higher level tasks, while abstracting away communication details. We can even implement the server and client in different languages if we wish.

2.3 Scalability

2.3.1 Efficiency

- *MPI* is designed for efficiency. We may not see significant overhead as we scale if we use MPI because of the tightly-coupled parallel computing.
- *gRPC* is more suitable to service-oriented, loosely-coupled architectures. We may see more significant communication overhead if we scale to a large number of processes, as may be the case in problems like the KNN problem.

2.3.2 Integrability

- *MPI* would be more difficult to integrate with larger services due to its tightly coupled architecture.
- *gRPC* would be easier to integrate with a number of services due to its service-oriented, loosely coupled architecture.

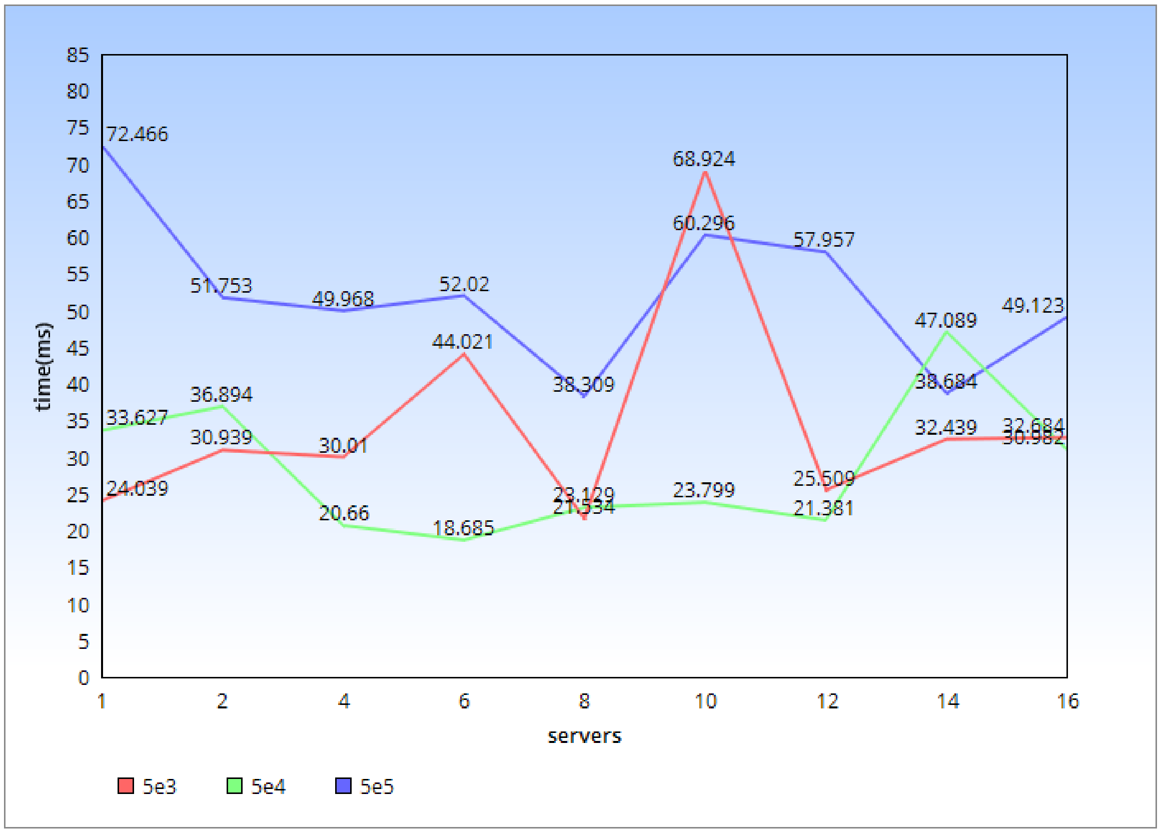
3 Performance Analysis

Theoretically, the time should decrease regularly with an increase in the value of the number of servers. But practically, an increase in servers brings about a significant increase in message overhead, often leading to the time taken increasing.

Note that the value of *nproc* on my system is 16.

3.1 Varying Values of n

Here, $k = 100$.



Clearly, we can see that adopting a distributed gRPC model is more practical for larger datasets than smaller ones. This is likely because for smaller ones the performance improvement brought about by the distributed computation is not worth the overhead caused by communication. If we have less data, we might as well calculate in a single process.

For larger datasets, we see a marked decrease in execution time with an increase in the number of servers. Parallel execution is better than serial for all values with a dataset size of *5e5*. For a value of *5e4*, the values are almost always better, but the time increases as the number of servers approaches *nproc*. This could also be because of other programs running on the system interfering with parallel execution.

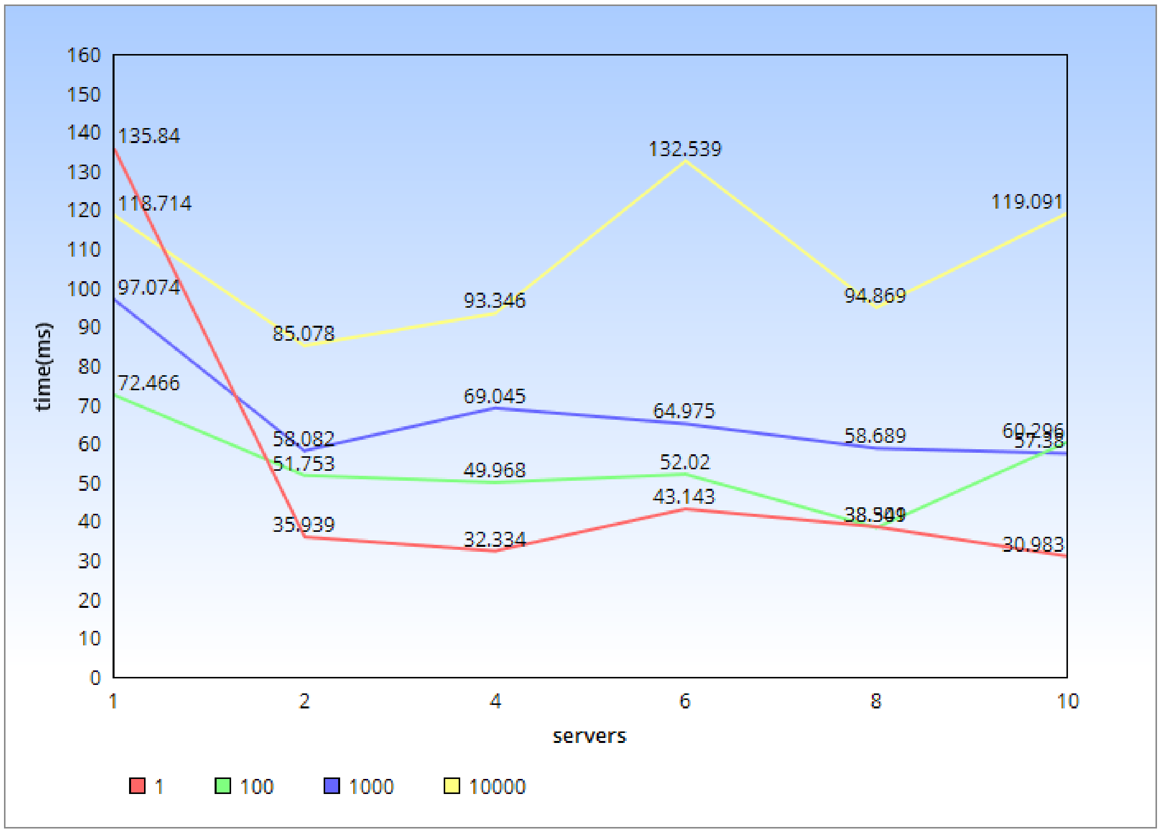
For a value of *5e3*, we see no significant improvements.

Also, trivially, the execution time seems to increase with n. Thus:

- increase in n increases time taken
- increase in s decreases time taken regardless of n, but is more noticable for higher n, as the communication tradeoff becomes significant for lower n.

3.2 Varying values of k

Here, $n = 5e5$.



Here, we return to our total time complexity approximation:

$$O(\frac{n}{s} \cdot log(k) + k \cdot log(k) \cdot log(s))$$

We notice that the second term does not decrease with increase of s and increases with increase in k. Thus, as we increase the value of k, the value of the distributed computation decreases as the work done by s in decreasing the complexity of the first term, is un-done by k in the second term.

This aligns with our results. While the performance improvement brought about about by more servers is drastic for $k = 1$, and clearly noticable for $k = 100$ and $k = 1000$, the improvement is minimal for $k = 10000$. This is likely because the second term (the final gathering of results) dominates the time improvement brought about by the initial local computation of results.

Thus:

- time taken increases with increase in k
- as k increases, the time improvement brought about about by s decreases.