Distributed Systems HW-4 Q2 Report

Team Number: 55

Team Members: Mitansh Kayathwal, Pradeep Mishra

Roll Numbers: 2021101026, 2023801013

General Instructions

The instructions to build and run the code can be found in the `README.md` file present in the `Q2/` directory. A `Makefile` has also been provided. I have assumed that there are exactly 5 servers running parallelly (Named `server1` through `server5` in Makefile). A single client is present which takes input from the user and provides the corresponding output after performing the correct Remote Procedure Calls.

Implementation Quirks (Including Efficient Considerations(Bonus))

1. I use concurrent server querying at the client side. The client uses goroutines to query multiple servers concurrently. This is an efficient approach as it allows parallel processing of data from different servers, reducing the overall query time.
2. At the client side, I also use gRPC streaming to receive neighbors from each server. This is efficient because:
   a) It allows processing of results as they arrive, rather than waiting for all results.
   b) It reduces memory usage as the client doesn't need to store all results at once.
3. I have maintained a max-heap of size k to efficiently keep track of the k nearest neighbors across all servers. This is **crucial for efficiency** because:
   a) It limits memory usage to O(k) regardless of the total number of points processed.
   b) It allows quick (O(log k)) insertion and removal of elements.
   c) It automatically keeps the k nearest neighbors without sorting all points.
4. Each server loads its portion of the dataset from a file at startup. This is efficient as it avoids repeated file I/O operations during query processing.
5. The server streams the k nearest neighbors back to the client. This is efficient because:
   a) It starts sending data as soon as it's available, reducing latency.

b) It allows the client to process results incrementally.
c) It reduces memory usage on the server as it doesn't need to store all k neighbors before sending.

Thus, concluding the discussion on efficiency, I have mainly leveraged distributed processing using multiple servers thus allowing us to process queries faster through parallelization. The use of a max-heap of size k ensures that only the k best results are kept in memory, regardless of how many total points are processed across all servers, this is crucial at maintaining a good time complexity. The use of goroutines allows for parallel processing of results from multiple servers, improving overall query performance.

$TimeComplexity : O(s * k * log(k))$

s ⇒ Number of servers

k ⇒ No of nearest neighbours required to be computed

How gRPC helps us to facilitate distributed computing in this example?

a) Client-Server Communication:

gRPC enables us to connect a single client to multiple servers at once. In my case, I have 5 servers 50051 through 50055 which are connected to by the client simultaneously. Every server has it's own pre-partitioned dataset which it uses to answer client queries. Every server streams it's part of the local k-NN output to the client which eventually computes the global k-NN. This effective distribution of the load helps in doing parallel processing on the data-set.

b) Service Definition (using Protocol Buffers):

The KNNService is defined using Protocol Buffers, which provides a language-agnostic way to specify the service interface. This makes it easy to implement the service in different languages and ensures consistency between client and server. Thus, it allows us to have different languages for client, server and any other services which are part of our system.

c) Streaming:

gRPC supports streaming both at client and server side. I have utilized **streaming from the server** to the client to solve this question, which allows us to improve efficiency of data transfer. At the client side, I maintained a Max-Heap with at maximum k elements, this helps us to improve the system performance, which we will analyze soon. The FindKNearestNeighbors method uses server-side streaming to send multiple Neighbor responses back to the client. This is efficient for sending large amounts of data or results that are computed incrementally.

d) Concurrent Processing:

The client uses go-routines to concurrently send requests to all server instances. This parallelism allows the client to efficiently gather results from multiple sources simultaneously, a key aspect of distributed computing.

e) Aggregation of Results:

The client aggregates results from all servers, sorts them, and maintains them in a Max-Heap of size at maximum k. This demonstrates how we can combine distributed results to produce a final output.

Comparison between gRPC and MPI in terms of communication models, usability, and scalability

gRPC and MPI are both used for distributed computing but have different characteristics:

a) Communication Models:

- gRPC: Uses a client-server model with remote procedure calls. It's based on HTTP/2, supporting unary calls, server streaming, client streaming, and bidirectional streaming.
- MPI: Uses a peer-to-peer model where all processes can communicate directly with each other. It supports point-to-point and collective communication operations.

b) Usability:

- gRPC:
  - Easier to learn and use, especially for developers familiar with web services.
  - Provides automatic code generation for client and server stubs.
  - Supports multiple programming languages, making it versatile for heterogeneous systems.
  - Well-suited for microservices architectures and cloud-native applications.
- MPI:
  - Has a steeper learning curve and requires more low-level programming.
  - Primarily used in C, C++, and Fortran, with limited support for other languages.
  - Provides a rich set of specialized functions for parallel computing tasks.
  - Better suited for high-performance computing (HPC) and tightly-coupled parallel applications.

c) Scalability:

- gRPC:
  - Scales well for loosely-coupled distributed systems.
  - Supports load balancing and service discovery, making it easier to scale horizontally.
  - Better for systems with dynamic topology or cloud environments.
  - May have higher overhead for very large numbers of fine-grained communications.
- MPI:
  - Excellent scalability for tightly-coupled parallel applications.
  - Can efficiently handle a very large number of processes (thousands or more).
  - Optimized for high-performance environments like supercomputers.
  - Better suited for applications requiring low-latency, high-bandwidth inter-process communication.

d) Network Efficiency:

- gRPC: Uses HTTP/2, which provides features like header compression and multiplexing. Efficient for internet-scale distributed systems.
- MPI: Optimized for high-speed, low-latency networks common in HPC environments.

e) Fault Tolerance:

- gRPC: Built-in support for timeouts, retries, and load balancing. Better suited for handling node failures in loosely-coupled systems.
- MPI: Traditional implementations have limited built-in fault tolerance. Failures often require application restart, though some modern implementations are improving in this area.

f) Interoperability:

- gRPC: High interoperability between different languages and platforms.
- MPI: Limited interoperability, primarily used within the same language and on similar systems.

Performance analysis for different sizes of datasets and k

Data Collection & Plotting:

The script for metrics collection can be found at: `/Q2/client/data_collection.go` .

Table - 1:

Dataset Size = 1000000

No of Servers	Time taken (ms)	Value of k
1	337.91	1
1	358.95	10
1	356.02	100
1	372.78	1000
1	368.35	10000
1	418.08	100000
3	163.78	1
3	234.18	10
3	187.89	100
3	202.16	1000
3	201.57	10000
3	290.67	100000
5	141.03	1
5	126.16	10
5	138.94	100
5	145.02	1000
5	150.46	10000
5	322.61	100000
7	112.61	1
7	143.79	10
7	147.31	100
7	104.92	1000
7	129.95	10000
7	431.19	100000

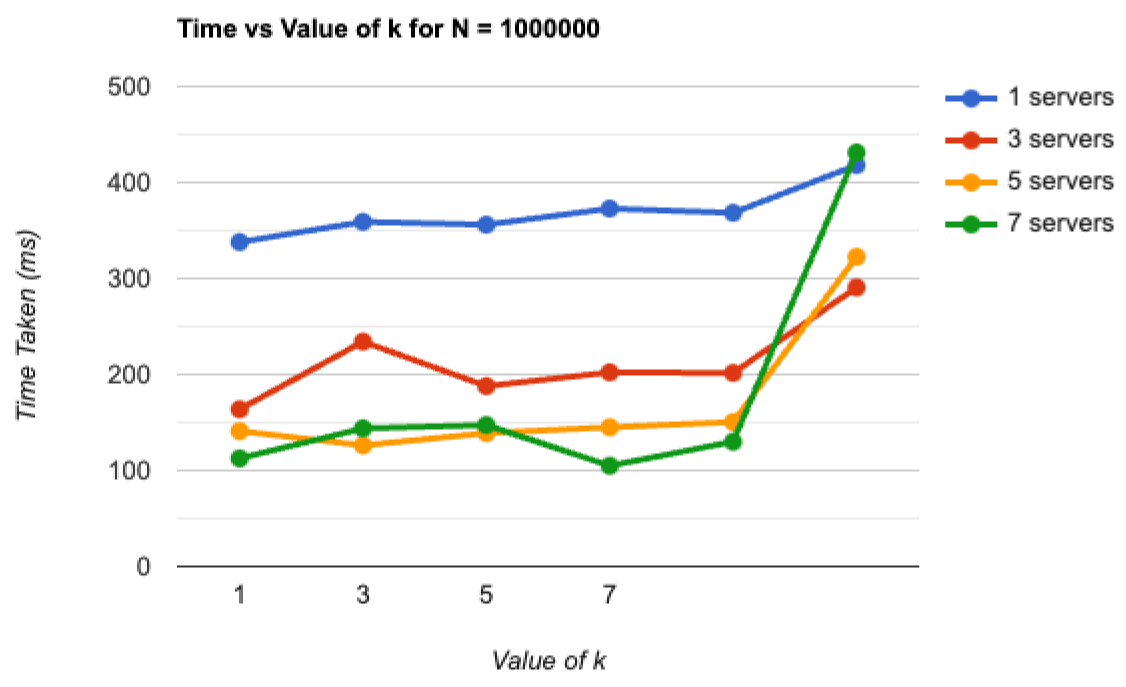
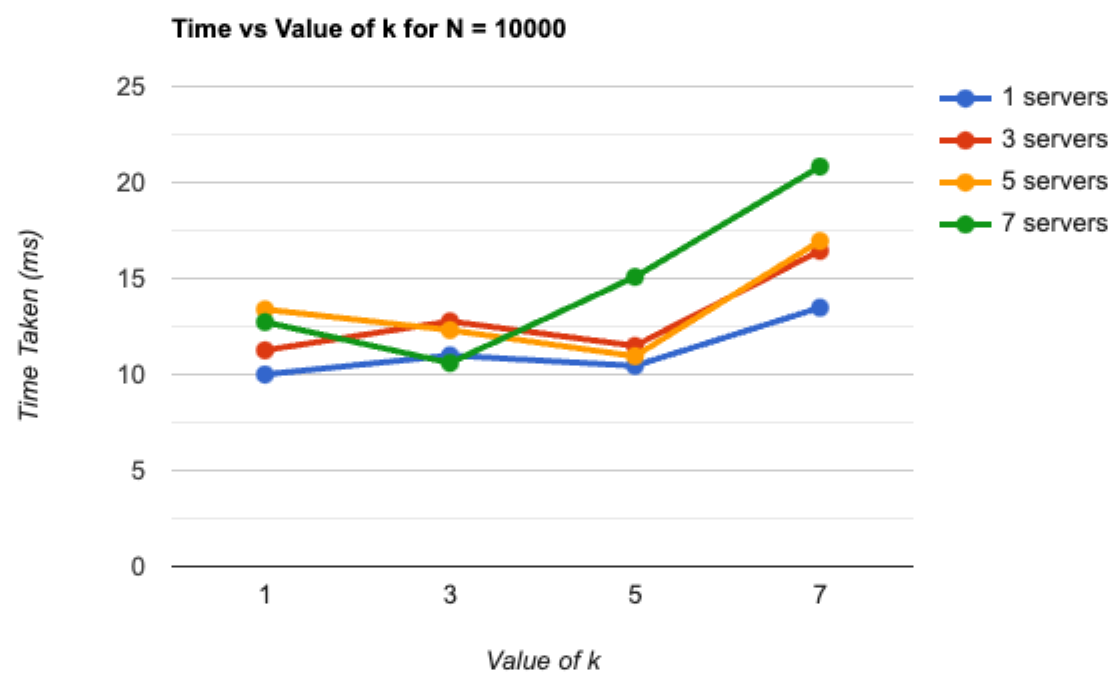


Table - 2:

Dataset Size = 10000

No of Servers	Time taken (ms)	Value of k
1	10.00	1
1	10.99	10
1	10.45	100
1	13.48	1000
3	11.26	1
3	12.77	10
3	11.49	100
3	16.44	1000
5	13.39	1
5	12.31	10
5	10.95	100
5	16.95	1000
7	12.73	1
7	10.60	10
7	15.08	100
7	20.83	1000



Inference

Table 1:

Key Observations:

1. **Scaling with Number of Servers:**
   - For small k values (1-100), increasing the number of servers significantly reduces computation time.
   - With 1 server, times range from ~338ms to ~373ms.
   - With 7 servers, times decrease to ~105ms to ~147ms for the same k range.
2. **Impact of k Value:**
   - For each server configuration, there's a slight increase in computation time as k increases.
   - The increase is more pronounced for very large k values (10,000 and 100,000).
3. **Diminishing Returns:**
   - The performance improvement from 1 to 3 servers is substantial.
   - The improvement from 3 to 5 servers is noticeable but smaller.
   - The difference between 5 and 7 servers is minimal for most k values.

Analysis:

1. **Improvement on Large Datasets:**
   - Distributing the workload across multiple servers allowed for parallel processing of the large dataset.

2. **Performance with Large k Values:**
    - For very large k values (10000 and 100000), we see a performance degradation across all server configurations.
    - This is due to:
      a) Increased communication overhead between servers to merge and sort larger result sets.
      b) More memory-intensive operations as larger heaps are maintained.
3. **Optimal Configuration:**
    - For this dataset, 5 servers seem to provide a good balance between performance and resource utilization.
    - The marginal benefit of adding more servers diminishes after this point.

Table 2:

Key Observations:

1. **Single Server Performance:**
    - For small k values (1-100), a single server often outperforms multi-server configurations.
    - The difference is small, but consistent across these k values.
2. **Large k Value (1000) Performance:**
    - For k=1000, multi-server configurations perform worse than a single server.
    - Performance degradation is more pronounced as the number of servers increases.

Analysis:

1. **Why Single Server Performs Better:**
    - **Overhead of Distribution:** For a small dataset, the time taken to distribute the data and aggregate results across multiple servers outweighs the benefits of parallel processing.
    - **Data Locality:** A single server can keep all 10,000 points in memory, allowing for efficient processing without network communication.
    - **Reduced Coordination:** No need for inter-server communication and result merging, which introduces latency.
2. **Performance Degradation with Multiple Servers:**
    - **Network Latency:** The time taken to send data between servers becomes a significant portion of the total computation time for small datasets.
    - **Merging Overhead:** For larger k values (e.g., 1000), the time taken to merge results from multiple servers becomes substantial relative to the actual computation time.

Conclusion

The performance characteristics of the KNN algorithm vary significantly based on the dataset size, the number of neighbors (k) being searched for, and the number of servers used. For large datasets, distributing the workload across multiple servers provides substantial benefits, especially for moderate k values. However, for small datasets, the overhead of distribution will outweigh the benefits of parallel processing.