EPFL

Privacy Preserving KNN Graph Contruction

Mohamed Yassine Boukhari

June 9, 2023

Advisor: Prof. Dr. Anne-Marie Kermarrec
Supervisors: Dr. Rafael Pires and Rishi Sharma

Scalable Computing Systems Lab
School of Computer and Communication Sciences

Abstract

A k-Nearest-Neighbors graph is a widely-used data-structure in Machine Learning for recommendation systems. In a decentralized collaborative filtering system, every node computes its distance to other nodes based on some similarity metric between their profiles. Since these profiles correspond to the tastes of the users, this is obviously an enormous privacy threat. We propose an implementation of a privacy preserving k-NN graph construction for decentralized recommender systems by applying private set intersection and union cardinality for computing the Jaccard similarity metric between user profiles and study its impact in terms of privacy, communication overhead and computational costs. We also propose ways to improve our system in terms of performance and security.

Contents

Chapter 1

Introduction

Recommender systems have significantly transformed the digital experience, using complex algorithms to curate personalized content based on individual user behaviors and preferences. They have found application in a multitude of platforms, from Netflix to Amazon, enhancing user engagement and satisfaction by aiding users in navigating the vast array of choices available.

As these systems continue to expand and accommodate a growing user base, the need for scalability becomes imperative. Traditional, centralized recommender systems can struggle to handle vast amounts of data and users, leading to efficiency bottlenecks. As a result, the concept of decentralized collaborative filtering has gained traction.

This is where the role of the k-Nearest-Neighbors (k-NN) graph becomes particularly significant. Serving as a popular data structure in machine learning and especially within recommendation systems, the k-NN graph is integral to the operation of decentralized collaborative filtering. In such a system, each individual node computes its proximity to others by employing a similarity metric based on their profiles, i.e., past interactions with various items. The nodes then identify their k nearest neighbors who have exhibited similar past preferences.

However, the disclosure of these profiles, which inherently reflect users' tastes, introduces an enormous privacy risk. Given the increasing internet usage and the consequent surge in data collection, exposing such information to other users amplifies the potential for privacy breaches, as it enables the inference of personal attributes, preferences, and potentially sensitive data. Therefore, the need to safeguard the personal data of users becomes a critical concern amidst the move towards more scalable, decentralized systems.

The main objective of this project is to develop a system capable of constructing k-NN graphs without compromising user privacy and integrate it into the *decentralizepy* framework. We leverage the private set intersection cardinality (PSI-CA) protocol to privately compute the jaccard similarity between users. Additionally, we examine the costs associated with such a system, which includes both computational overheads and communication expenses. The overarching aim is to strike a balance between ensuring robust user privacy protection and maintaining efficient system performance. The main artifacts are a high performance C++/Python library for PSI-CA and an extension of *decentralizepy* with both a normal and private k-NN nodes.

Firstly, In chapter 2 we give a brief presentation of the PSI-CA protocol, followed by a decentralized k-NN graph construction algorithm and an overview of the *decentralizepy* framework. Then, in chapter 3 we present our serverless PSI-CA protocol and the corresponding threat model. It also introduces our dedicated library for PSI-CA and outlines its core implementation details, followed by a detailed explanation of our extensions to *decentralizepy*. Next, Chapter 4 shows an evaluation of the performance of our library and presents a detailed analysis of the costs associated with the private construction of a k-NN graph.

Chapter 2

Background

2.1 PSI-CA Protocol

Private set intersection cardinality (PSI-CA) protocols enable two parties having Sets S_1 and S_2 to compute the size of the intersection of their sets $|S_1 \cap S_2|$ without revealing any specific details about the elements each party possesses. in the following paragraph we introduce a PSI-CA protocol by De Cristofaro et al [1]

The protocols requires a group $\mathbb{G}_0$ of prime order p, a subgroup ($\mathbb{G} \leqslant \mathbb{G}_0$) of prime order q such that $q|p-1$ and 2 hash functions $H : \{0,1\}^* \rightarrow \mathbb{G}$ and $H' : \{0,1\}^* \rightarrow \{0,1\}^{2\epsilon}$ given some security parameter ϵ that can be determined to match the security requirements of the system.

Before the start of the message exchange, both parties map their inputs c_i and s_j to elements of $\mathbb{G}$ by applying the first hash function $H(.)$ resulting in the sets hc_i and hs_j. At the start of the protocol, the client exponentiates its set items hc_i with a random exponent R_c and sends resulting values a_i to the server, which exponentiates them with its own random value R_s then shuffles the resulting values a'_s and sends them to client. Then, the server sends the output of a one-way function, $H'(.)$, computed over the exponentiations of server's items hs_j raised to R_s. Finally, client tries to match $H'(.)$ outputs received from the server with $H'(.)$ outputs computed over the shuffled a'_i values raised to the exponent $(1/R_c)$ to strip them from the initial mask. Client learns the set intersection cardinality by counting the number of such matches. The computational complexity and communication overhead are linear in the sizes of the two input sets. A more detailed overview of the protocol is given in the following figure 2.1.

Client		**Server**
$C = \{c_1, \ldots, c_v\}$		$S = \{s_1, \ldots, s_w\}$
$R_c \leftarrow\$ \mathbb{Z}_q$		$(\tilde{s_1}, \ldots, \tilde{s_w}) \leftarrow \prod(S)$
$\forall i\ 1 \leq i \leq v :$		$\forall j\ 1 \leq j \leq w : hs_j = H(\tilde{s_j})$
$hc_i = H(c_i)$		
$a_i = (hc_i)^{R_c}$		
	$\xrightarrow{\{a_1, \ldots, a_v\}}$	
		$R_s \leftarrow\$ \mathbb{Z}_q$
		$\forall i\ 1 \leq i \leq v : a'_i = (a_i)^{R_s}$
		$(a'_{l_1}, \ldots, a'_{l_v}) = \prod'(a'_1, \ldots, a'_v)$
		$\forall j\ 1 \leq j \leq w : bs_j = (hs_j)^{R_s}$
		$\forall j\ 1 \leq j \leq w : ts_j = H'(bs_j)$
	$\xleftarrow{\{a'_{l_1}, \ldots, a'_{l_v}\}, \{ts_1, \ldots, ts_w\}}$	
$\forall i\ 1 \leq i \leq v :$		
$bc_i = (a'_{l_i})^{1/R_c} \mod q$		
$\forall i\ 1 \leq i \leq v :$		
$tc_i = H'(bc_i)$		

Output $= |\{ts_1, \ldots, ts_w\} \cap \{tc_1, \ldots, tc_v\}|$

Figure 2.1: PSI-CA protocol by De Cristofaro et al

2.2 Decentralized k-NN Graph construction

Many methods have been proposed to construct k-NN graphs in decentralized systems. A typical approach in such a system [2] is for every node to start from a few initial neighbors then use a P2P epidemic protocol to converge towards a neighborhood containing the k most-similar other nodes in the system according to some similarity metric. An example of such an algorithm is presented in the following algorithm [1].

Starting from a random neighborhood, nodes repeatedly select a random neighbor and exchange their current neighborhood with them of then use the gained information to select more similar neighbors. Similarly, when receiving a new neighborhood pushed to them, nodes update their neighborhood with the new nodes they have just heard of. The intuition behind this greedy procedure is that if A is similar to B, and B to C, C is likely to be similar to A as well. To avoid local minima, this greedy procedure is often complemented with a few random peers.

Algorithm 1 Greedy decentralized k-NN algorithm executing at node p

```
Require: graph $G$, number of rounds $n$, number of nearest neighbors $k$
 1: $\Gamma(p) \leftarrow rand(k, G)$
 2: while $round \leq n$ do
 3:     $q \leftarrow$ one random neighbor from $\Gamma(p)$
 4:     send $\langle$push, $\Gamma(p) \cup \{p\}\rangle$ to $q$ ; request $\Gamma(q)$ from $q$
 5:     $cand \leftarrow \Gamma(p) \cup \Gamma(q) \cup \{r$ random nodes$\} \setminus \{p\}$
 6:     $\Gamma(p) \leftarrow \mathrm{argtop}^k_{g \in cand}(\mathrm{sim}(p, g))$
 7: end while
 8: procedure On Receiving($\langle$push, $\Gamma'\rangle$)
 9:     $cand \leftarrow \Gamma(p) \cup \Gamma' \setminus \{p\}$
10:     $\Gamma(p) \leftarrow \mathrm{argtop}^k_{g \in cand}(\mathrm{sim}(p, g))$
11: end procedure
```

2.3 Decentralizepy Framework

Decentralizepy [3] is a distributed framework designed for decentralized machine learning. It facilitates the development and emulation of large-scale networks across various topologies. The framework comprises numerous modules such as communication, models, datasets, sharing, among others. The focus of our work, however, is on the node module. We augment this module with two implementations: one for regular k-NN node and the other for PSI-CA k-NN node enriching its functionality to enable simulating k-NN graph construction.

Chapter 3

Implementation

3.1 Serverless PSI-CA

3.1.1 Overview

In the previous chapter, we presented a PSI-CA protocol between a client and a server where only the client learns the set intersection cardinality. However, our system operates in a decentralized manner. Consequently, we had to adapt this protocol into a peer-to-peer (P2P) protocol where every node can play both roles of client and server based on who started the message exchange and reacts according to the nature of the received message.

If *node 1* wants to start a PSI-CA protocol with *node 2*, it will mask its elements with a random exponent exactly like in the first phase of the original protocol then sends the masked elements to *node 2* along with the **Request** flag. Upon receiving a message with the **Request** flag, *node 2* will proceed to the second stage of the protocol and responds with a message containing its hashed randomized items, the values received previously from the other peer raised to the random exponent and the **Response** flag. When a **Response** message is received, the first node completes the final stage of the protocol, computes the intersection cardinality and sends the result to *node 2*. After the protocol, both parties learn the intersection cardinality of their sets. The protocol is presented in 3.1.

Public Parameters

The public parameter remain the same as the original protocol: a group $\mathbb{G}_0$ of prime order p, a subgroup ($\mathbb{G} \leqslant \mathbb{G}_0$) of prime order q such that $q|p-1$ and 2 hash functions $H : \{0,1\}^* \to \mathbb{G}$ and $H' : \{0,1\}^* \to \{0,1\}^{2\epsilon}$ given some security parameter ϵ that can be determined to match the security requirements of the system.

Node 1		**Node 2**
$C = \{c_1, \ldots, c_v\}$ $R_c \leftarrow\$ \mathbb{Z}_q$ $\forall i\ 1 \leq i \leq v :$ $hc_i = H(c_i)$ $a_i = (hc_i)^{R_c}$		$D = \{d_1, \ldots, d_w\}$ $(\tilde{d_1}, \ldots, \tilde{d_w}) \leftarrow \prod(D)$ $\forall j\ 1 \leq j \leq w : hd_j = H(\tilde{d_j})$
	Request : $\{a_1, \ldots, a_v\}$ $\xrightarrow{\hspace{4em}}$	
		$R_d \leftarrow\$ \mathbb{Z}_q$ $\forall i\ 1 \leq i \leq v : a'_i = (a_i)^{R_d}$ $(a'_{l_1}, \ldots, a'_{l_v}) = \prod'(a'_1, \ldots, a'_v)$ $\forall j\ 1 \leq j \leq w : bd_j = (hd_j)^{R_d}$ $\forall j\ 1 \leq j \leq w : td_j = H'(bd_j)$
	Response : $\{a'_{l_1}, \ldots, a'_{l_v}\}, \{td_1, \ldots, td_w\}$ $\xleftarrow{\hspace{4em}}$	
$\forall i\ 1 \leq i \leq v :$ $bc_i = (a'_{l_i})^{1/R_c} \mod q$ $\forall i\ 1 \leq i \leq v :$ $tc_i = H'(bc_i)$ $\mathbf{R} = \lvert\{td_1, \ldots, td_w\} \cap \{tc_1, \ldots, tc_v\}\rvert$		
	Result : R $\xrightarrow{\hspace{4em}}$	

Figure 3.1: Serverless PSI-CA

3.1.2 Complexity

Computation

The computational complexity is linear in the sizes of the two input sets. Let $n = |C|$ and $m = |D|$.The first node performs n hashing operations and n exponentiations modulo p with an exponent of $|q|$ bits in the first phase of the protocol and the repeats the same number of operations in the final stage. The second party performs m hashing operations and $m + n$ exponentiations modulo p with an exponent of $|q|$ bits. This results in $\mathcal{O}(m + n) = \mathcal{O}(|C| + |D|)$ operations in group with order p, $\mathcal{O}(m + n)$ $H(.)$ hashes and $\mathcal{O}(m + n)$ $H'(.)$ hashes.

Communication

Let $n = |C|$ and $m = |D|$. The protocol requires 3 messages in total:

- **Request :** with a size of $n \cdot |p|$ bits.
- **Response :** with a size of $n \cdot |p| + m \cdot \epsilon$ bits.
- **Result :** constant size.

The total communication overhead is $\mathcal{O}(n \cdot |p| + m \cdot \epsilon)$.

3.1.3 Threat Model

Honest but curious adversaries

We claim that With semi-honest adversaries, our protocol ensures both correctness and privacy of the involved parties. The intuition behind this claim is that for a party to infer information about the other parties inputs or the elements in the intersection it would require breaking the Discrete Logarithm *(DL)* and Decisional Diffie-Hellman *(DDH)* assumptions, as well as inverting the Hash function H'. Furthermore, the permutations $\prod$ and $\prod'$ guarantee that there is no disclosure of information about the elements of the intersecting set based on their location in the **Request/Response** messages. You can refer to [1] for a formal proof of these assumptions.

Malicious adversaries

Malicious adversaries can supply malicious inputs to the protocol leading the other parties to compute an incorrect intersection. However, based on the same assumptions in the previous paragraph, the malicious nodes cannot gain any information about the other sets.

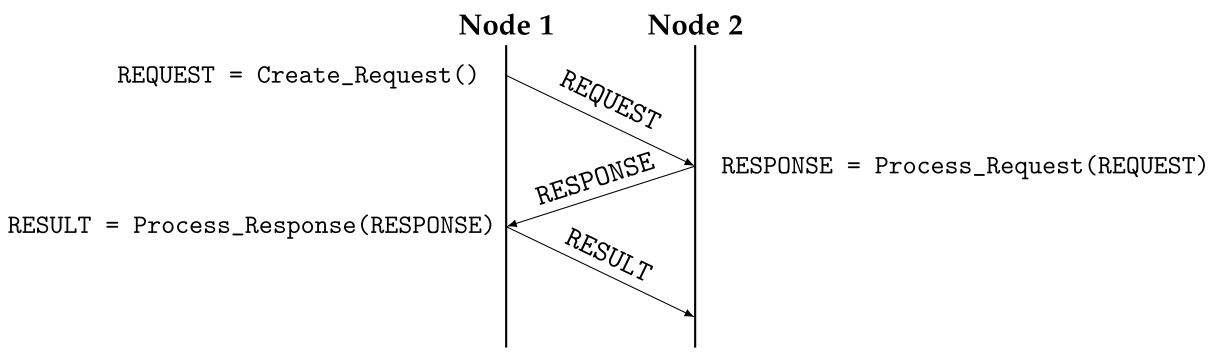
3.2 PSI-CA Implementation

3.2.1 Fpsica Library Overview

To properly be able to study the effect of PSI-CA on k-NN graph construction, it was necessary for us to develop an efficient implementation of the protocol from the ground up. We decided to separate the protocol implementation from the *decentralizepy* framework to allow more flexibility in the development and enable the implementation to be used outside the framework for other projects. Our goal was to strike a balance between performance and ease of use. As a result, we chose to implement the core operations of the protocol in C++ for its speed, and then provided bindings to a Python interface using pybind. Based on this architecture, we have constructed a high-performance Python module called **fpsica** (fast private set intersection cardinality) that offers PSI-CA functionality. It has been designed for easy installation on your system via the Python package manager, *pip*. This approach ensures that our implementation not only delivers superior performance, but is also easy for developers to use in a variety of contexts, meeting the needs of both speed and accessibility.

Required Functionalities and Workflow Description

Based on section 3.1, we can divide the Serverless PSI-CA protocol into three main phases presented in the scheme below:


Node 1
Node 2
REQUEST = Create_Request()
REQUEST
RESPONSE = Process_Request(REQUEST)
RESPONSE
RESULT = Process_Response(RESPONSE)
RESULT

Every node should be able to perform all three operations: `Create_Request()`, `Process_Request(req)` and `Process_Response(resp)` which will constitute the main interface exposed by our library. In addition, we require the following message types: `REQUEST`, `RESPONSE` and `RESULT`.

Security Specifications

We the decision to utilize Elliptic Curve Cryptography (ECC) over alternative finite field cryptography methods. ECC has been chosen due to its superior performance characteristics and greater efficiency. Unlike other methods such as RSA, ECC can achieve the same level of security with significantly smaller key sizes. This feature allows for faster computations and less resource-intensive operations.
We adopted the widely used NIST P-256 curve, also known as the prime256v1. This specific curve was chosen as it provides 128 bits of security, balancing robust security with computational efficiency.
For the hashing algorithm, we adopted SHA-256 (Secure Hash Algorithm 256-bit). SHA-256 was selected due to its widespread adoption, robustness, and strong security properties. It offers 128-bit collision resistance and 256-bit preimage resistance.

Code Organization

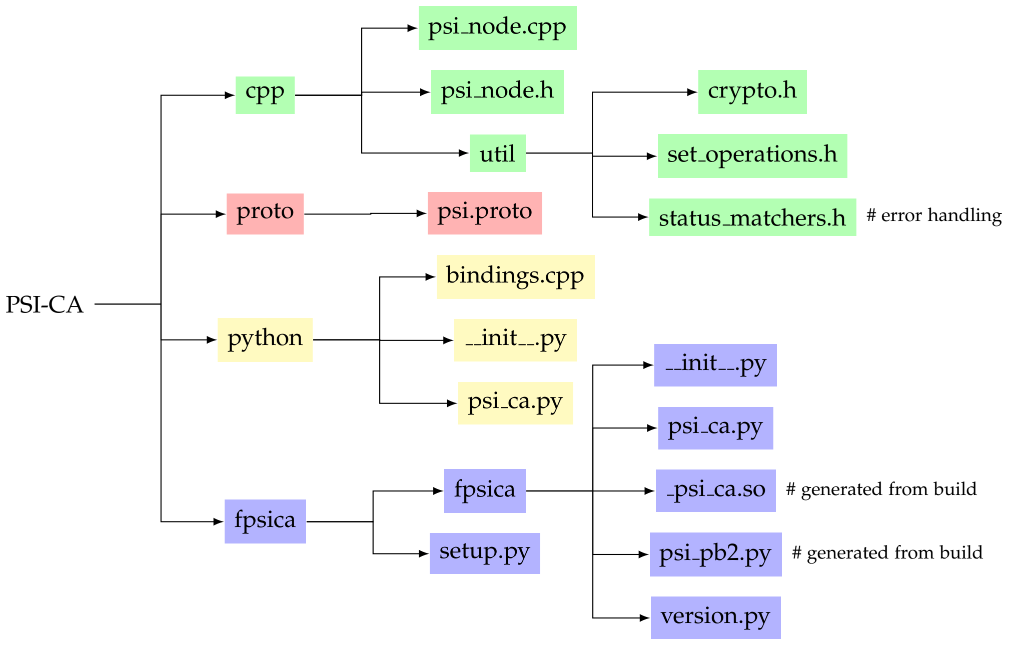


Figure 3.2: package architecture

The **PSI-CA** package is organized into four main directories, each with a distinct purpose and role in the library.

The **cpp** directory, highlighted in green, houses the core functionality of the package written in C++. The files *psi_node.cpp* contain Serverless PSI-CA node implementation and and *psi_node.h* defines the public interface of the class. The **util** sub-directory consists of utility files like *crypto.h* where the sha256 hashing procedure is implemented, *set_operations.h*, and *status_matchers.h* that provides helpers for error handling.

The **proto** directory, in red, contains the file *psi.proto*. This is the protocol buffer file defining the structure of data to be serialized and de-serialized during the interaction between different parts of the system.

The **python** directory, colored in yellow, hosts the Python bindings and utility functions. These allow users to interact with the C++ core from Python.

Lastly, the **fpsica** directory, depicted in blue, contains both the main Python module of the library and the setup script. Within the **fpsica** sub-directory, there are several files, including *__init__.py* which defines the modules fpsica exposes, *psi_ca.py* which defines the psi_ca module, *_psi_ca.so* (the compiled C++ module), *psi_pb2.py* (generated from the proto file for Python usage), and *version.py* (versioning information).

Setup and Build System

Our choice of build system fell on Bazel version 6, owing to its performance advantages. Its user-friendly interface encourages a more streamlined development process, making it a reliable choice for our implementation.

In terms of functionality, we provide a setup script for Bazel, ensuring a hassle-free setup process for developers intending to build upon or modify our work. Moreover, we automated the build procedure in order to generate the *fpsica* package in a straight forward manner.

3.2.2 C++ Implementation

Dependencies

In our C++ development, we primarily depended on the Google Abseil (ABSL) and Google Private Join and Compute libraries.

The Google Abseil (ABSL) library is a collection of C++ code that includes a variety of types, synchronization primitives, and utilities drawn. These augment the existing C++ standard library and fill in gaps in functionality and performance.

On the other hand, the Google Private Join and Compute library is an implementation of the Private Join and Compute functionality. This library exposes a performant and easy to use interface for ECC which we used to implement our protocol.

API

Our C++ module primarily exposes the `PsiNode` class, which represents a node in the Private Set Intersection Cardinality (PSI-CA) protocol. The key methods included in this class are:

- `CreateWithNewKey()` and `CreateFromKey()`: Generate a `PsiNode`, either with a new key or from an existing one.
- `CreateRequest()`: Begins the PSI-CA protocol by creating a request from a given input set.
- `ProcessRequest()`: Handles a received request and generates a response based on the input set of the node.
- `ProcessResponse()`: Interprets a received response and calculates the cardinality of the set intersection.

```
class PsiNode {
    public:
        // Create a new node with a new key
        static StatusOr<unique_ptr<PsiNode>>
           CreateWithNewKey();

        // Create a new node from a given key (used for
           test puposes)
        static StatusOr<unique_ptr<PsiNode>> CreateFromKey
           (const string& key_bytes);

        //Stage 1 of PSI-CA
        StatusOr<Request> CreateRequest(Span<const string>
            inputs) const;

        //Stage 2 of PSI-CA
        StatusOr<Response> ProcessRequest(const Request&
           request, Span<const string> inputs) const;

        //Stage 3 of PSI-CA
        StatusOr<int64_t> ProcessResponse(const Response&
           response) const;
};
```

Code Listing 3.1: Node API

3.2.3 Serialization

In our system, we employed Protocol Buffers (protobuf) version 3 for the serialization of messages. We defined three message types: `Request`, `Response`, and `Result`. This allowed us to define the structure and fields of each message in a language-agnostic manner benefiting from efficient serialization and de-serialization of messages. A more precise description is presented below.

```
//Request : {a_1,... , a_v }
message Request {
  repeated bytes elements = 1;
}

//Response : {a'_1,... , a'_v }, {td_1, ..., td_w}
message Response {
  message Masked {
    repeated bytes elements = 1;
  }
  message Hashed {
    repeated bytes elements = 1;
  }
  Masked masked = 1;
  Hashed hashed = 2;
}

//Result : R
message Result{
  int32 intersection_size = 1;
}
```

Code Listing 3.2: message definitions

3.2.4 Python Implementation

fpsica Python Modules and API

As previously noted, the Python API of our library mirrors the C++ API in terms of functionality. It provides the same set of operations necessary for executing the PSI-CA protocol.
Along with the core PSI-CA operations, our API also includes functions to control the serialization and de-serialization of messages exchanged between the nodes.

```
class Node:
    @classmethod
    def CreateWithNewKey(cls) -> Node
    @classmethod
    def CreateFromKey(cls, key_bytes: bytes) -> Node
    def CreateRequest(self, inputs: List[str]) -> Request
    def ProcessRequest(self, request: Request , inputs:
        List[str]) -> Response
    def ProcessResponse(self, response: Response) -> int

class Request:
    def ParseFromString(self, request: str) -> None
    def SerializeToString(self) -> str

class Response:
    def ParseFromString(self, response: str) -> None
    def SerializeToString(self) -> str

class Result:
    def ParseFromString(self, result: str) -> None
    def SerializeToString(self) -> str
```

Code Listing 3.3: fpsica API

bindings

We used pybind11, an open-source library specifically designed for creating Python bindings from C++ code. In the "bindings.cpp" file, we defined the necessary bindings that connect our Python classes and methods to their corresponding components in C++ and Protocol Buffers. A simple representation of the bindings is given in the following figure 3.3.

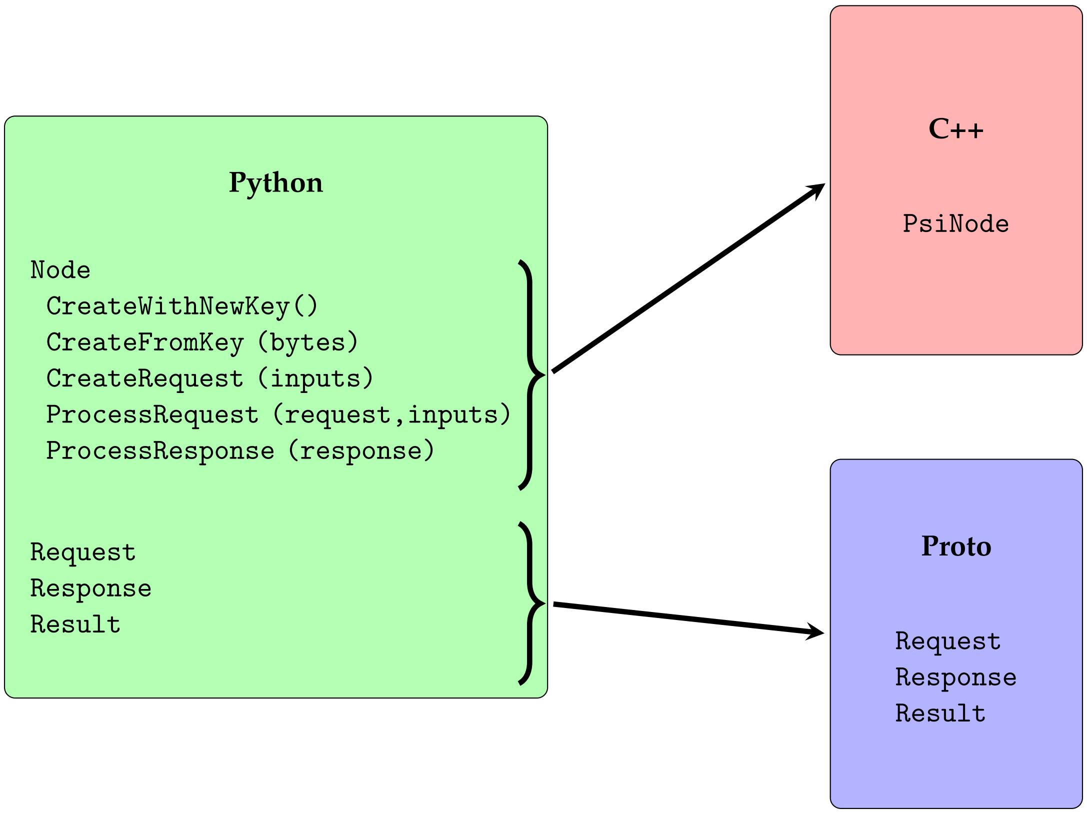


Figure 3.3: Overview of Python to C++/Proto bindings

3.3 PSI-CA KNN

The main goal of our project was to investigate the cost of PSI-CA protocol on the construction of k-NN graphs. For this purpose, we used our library to extend the *decentralizepy* framework with a `PSICAKNN` node class. In the following section, we are going to explain more in depth our implementation.

3.3.1 Initial Configuration

Parameters

To instantiate the new class, it necessitates the specification of two additional parameters: the number of iterations in the graph construction procedure `knn_rounds`, and the number of neighbors `k`. These parameters are supplied by the user before the start of the program.

Supported topologies

The architecture of our system requires that every node is able to communicate with every other node in the network. This necessity underscores the requirement for a fully connected graph as the underlying topology. It is important to mention that this requirement is imposed by the communication module of *decentralizepy*.

Datasets

The current implementation supports only the MovieLens [4] ml-latest-small dataset.

3.3.2 User Profile Construction

In our system, we utilize the Jaccard similarity as our metric to measure the similarity between users. It is defined as the size of the intersection divided by the size of the union of the sample sets.
This metric requires each user to have a profile that reflects his preferences. These profiles are constructed from the individual ratings given by each user to different movies by including the movies ids for which the user has given a rating above a certain threshold. This ensures that the profile reflects the movies that the user has a strong preference for, allowing the Jaccard similarity metric to provide meaningful comparisons between users.

3.3.3 Graph Construction Procedure

Intuition

We build upon the p2p greedy algorithm presented in the background chapter to design our graph construction procedure:
At the beginning, each node in the network starts the procedure with a random set of neighbors. At the end of every round, the node selects a new set of candidates by asking one of its neighbors for its neighbors and complements this set with a number of random peers. At the start of the next round, the node runs the PSI-CA protocol with its set of candidates and updates its neighborhood based on the obtained similarities. The algorithm is

given in 2.
In the proposed procedure, nodes exchange only the identifiers of their neighbors. This strategy allows nodes to determine which other peers might be similar, making them suitable candidates for running the PSI-CA protocol and maintaining the privacy of their profiles in the same time.

Algorithm 2 PSI Cardinality KNN algorithm executing at node p

```
Require: graph $G$, number of rounds $n$, number of nearest neighbors $k$
 1: $\Gamma(p) \leftarrow rand(k, G)$
 2: $cand \leftarrow \Gamma(p)$

 3: while $round \leq n$ do
 4:     for all $c$ in $cand$ do
 5:         RUN PSI CA WITH NODE $c$
 6:     end for
 7:     $\Gamma(p) \leftarrow \text{argtop}^{k}_{g \in cand \cap \Gamma(p)}(\text{sim}(p, g))$
 8:     $q \leftarrow$ one random neighbor from $\Gamma(p)$
 9:     send $\langle \text{push}, \Gamma(p) \rangle$ to $q$ ; request $\Gamma(q)$ from $q$
10:     $cand \leftarrow \Gamma(q) \cup \{r \text{ random nodes}\} \setminus \{p\}$
11: end while

12: procedure ON RECEIVING($\langle \text{push}, \Gamma' \rangle$)
13:     $cand \leftarrow cand \cup \Gamma' \setminus \{p\}$
14: end procedure
```

Concurrency and Optimisations

Performance optimization is a central focus of our project. Executing the PSI-CA protocol sequentially, where we wait for every candidate node to respond before moving on to the next would incur a huge overhead. Recognizing this potential bottleneck, we chose to adopt a highly concurrent implementation strategy. This allows us to run multiple instances of the protocol in parallel, substantially reducing the time required for each k-NN round to end.
We made the choice to have 3 parallel threads where each thread is responsible for some aspect of the program: a main thread, a sender and receiver. The main thread is responsible for creating the initial requests to send to the neighbors, updating the set of candidates and the organizing the rest of the procedure. In addition, it controls the other threads through synchronization primitives. The receiver thread waits for message and reacts according to the nature of received message by either updating the node's information or creating some response to send. The sender thread polls a queue

(which is a thread safe structure) and sends the popped messages to the other nodes. In the following pages, we provide an overview of the design as well as the set of algorithms (3, 4, 5) used for each thread.

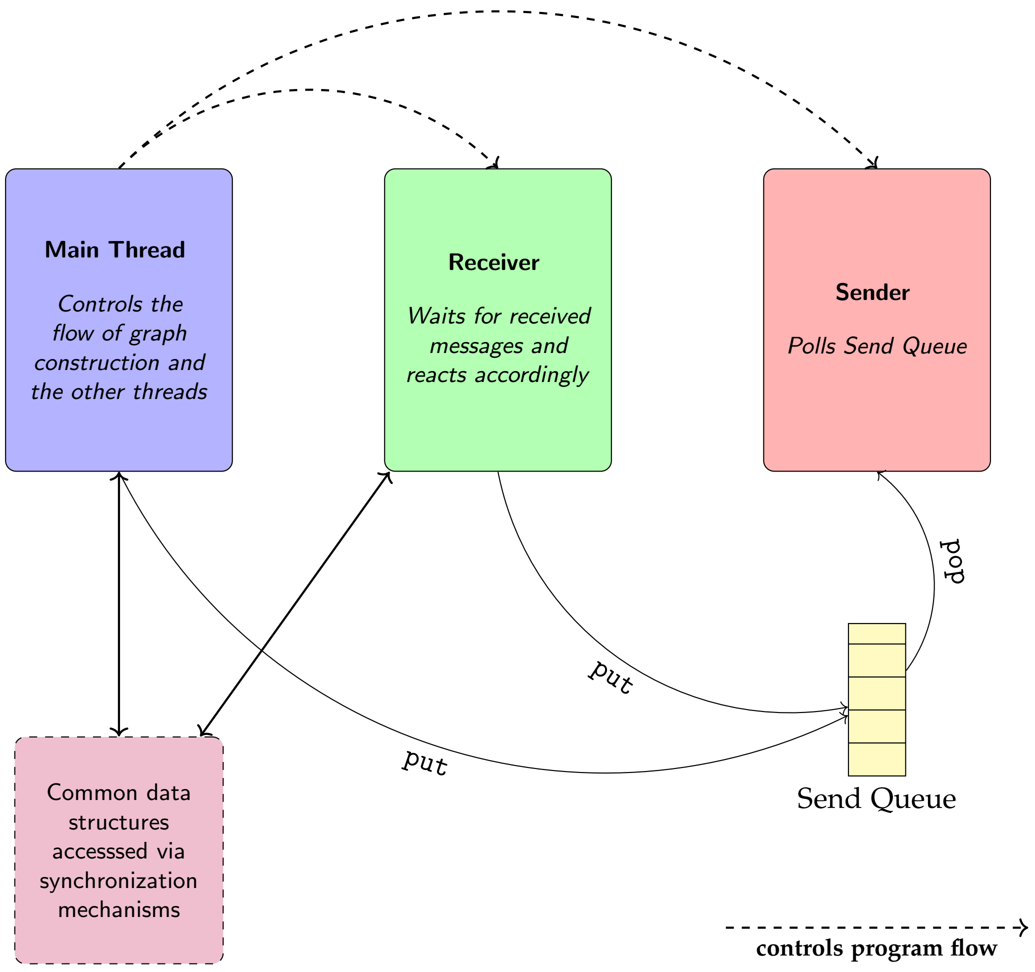

Main Thread
Controls the flow of graph construction and the other threads
Receiver
Waits for received messages and reacts accordingly
Sender
Polls Send Queue
pop
put
put
Send Queue
Common data structures accesssed via synchronization mechanisms
controls program flow

Figure 3.4: PSI Cardinality node design

We try to further optimise our implementation by reducing the communication overhead in the following manner: every node keeps a small state of previously engaged nodes and adds only new peers to the set of fresh candidates. This allows us to run the protocol once for every pair of nodes.

Algorithm 3 Main Thread

```
 1: $\Gamma(p) \leftarrow k$ random neighbors from $G$
 2: $current\_cand \leftarrow \Gamma(p)$
 3: $next\_cand \leftarrow Set()$
 4: $round \leftarrow 0$

 5: START($receiver\_thread$)
 6: START($sender\_thread$)

 7: while $round \leq knn\_rounds$ do
 8:     for all $c$ in $current\_cand$ do
 9:         $psi\_request \leftarrow$ create_request( )
10:         send_queue.put$\langle$ $c$ , $psi\_request$ $\rangle$
11:     end for

12:     $q \leftarrow$ one random neighbor from $\Gamma(p)$
13:     send_queue.put$\langle$ $q$ , random_discovery_request( ) $\rangle$

14:     WAIT FOR KNN ROUND TO END

15:     $next\_cand \leftarrow \Gamma(q) \cup \{r \text{ random nodes}\} \setminus \{p\}$
16:     $\Gamma(p) \leftarrow \text{argtop}^{k}_{g \in (current\_cand \,\cup\, \Gamma(p))}(\text{sim}(p, g))$
17:     $current\_cand \leftarrow next\_cand$
18:     $next\_cand \leftarrow Set()$
19: end while

20: for all $n$ in $G$ do
21:     send_queue.put$\langle$ $n$, knn_bye $\rangle$
22: end for

23: JOIN($sender\_thread$)
24: JOIN($receiver\_thread$)

25: return
```

Algorithm 4 Receiver Thread

```
1: $knn\_byes \leftarrow Set()$
2: On Receiving $psi\_request$ from $q$:
3:     $response \leftarrow$ process_request($request$)
4:     send_queue.put$\langle\ q\ ,\ response\ \rangle$

5: On Receiving $psi\_response$ from $q$:
6:     $result \leftarrow$ process_response($response$)
7:     $sim \leftarrow result/(|response.masked| + |response.hashed| - result)$
8:     send_queue.put$\langle\ q\ ,\ sim\ \rangle$
9:     $\Gamma(p) \leftarrow \text{argtop}^k_{g \in (q \cup \Gamma(p))}(\text{sim}(p, g))$

10: On Receiving $psi\_result$ from $q$:
11:     $\Gamma(p) \leftarrow \text{argtop}^k_{g \in (q \cup \Gamma(p))}(\text{sim}(p, g))$
12:
13: On Receiving $random\_discovery\_request$ from $q$:
14:     $next\_cand \leftarrow next\_cand \cup \Gamma(q)$
15:     $rd\_response \leftarrow \Gamma(p)$
16:     send_queue.put$\langle\ q\ ,\ rd\_response\ \rangle$

17: On Receiving $random\_discovery\_response$ from $q$:
18:     $next\_cand \leftarrow next\_cand \cup \Gamma(q)$

19: On Receiving $knn\_bye$ from $q$:
20:     $knn\_byes \leftarrow knn\_byes \cup \{q\}$

21: if $|knn\_byes| = |G| - 1$ then
22:         exit_sender $\leftarrow$ true
23:         return
24: end if
```

3.3.4 Regular KNN

In order to examine the cost of privacy preserving graph construction, we implement a `KNN` class that runs a similar procedure to the `PSICAKNN` but without running the 3 steps PSI-CA. We replace it by a 2 steps message exchange where a node sends its profile to the peer that responds with the similarity measure of their 2 profiles.

Algorithm 5 Sender Thread

```
1: $knn\_byes \leftarrow Set()$
2: while True do
3:     $q,m \leftarrow$ send_queue.pop$\langle\ \rangle$
4:     if send_queue not $empty$ then
5:         SEND($q,m$)
6:         if $m$ is $knn\_bye$ then
7:             $knn\_byes \leftarrow knn\_byes \cup \{q\}$
8:         end if
9:     else
10:        if $|knn\_byes| = |G| - 1$ and EXIT_SENDER then
11:            return
12:        end if
13:    end if
14: end while
```

Chapter 4

Evaluation

In this section, we provide a comprehensive evaluation of our system based on several metrics. We specifically focus on the quality of the k-NN graph, the communication overhead, and the execution time. To evaluate the cost of the PSI-CA protocol, we compare our privacy preserving implementation to the regular k-NN node. Apart from what we discussed above, we also benchmark the PSI-CA C++ and Python APIs to provide a well-rounded evaluation of implementation.
We use the machines sacs002-sacs005@iccluster. All the machines have the same specifications with processor Intel(R) Xeon(R) E-2288G CPU @ 3.70GHz and 64GB of memory running Ubuntu 22.04.2 LTS with linux kernel 5.11.0. We run every experiments a total of 5 times and report the mean and the standard deviation of every studied metric.

4.1 Performance Evaluation of PSI-CA API

In this section, we examine the performance of our PSI-CA API independently from the *decentralizepy* framework. The key metric evaluated is the execution time required for each operation, which we measure as we progressively increase the size of the input set.

C++ API

From the graph presented below, a clear linear relationship is evident between the size of the input set and the execution time for all three operations. This pattern is consistent with our prior theoretical analysis.

Even when the input set size reaches 10,000 elements, a size far exceeding the usual requirements of our system, the execution time remains less than one second. In practical use cases, our profiles are constituted of fewer than 100 elements. Under these conditions, our system performs remarkably well, with a combined execution time for all three operations of less than 30 milliseconds.

These findings corroborate the efficiency of our PSI-CA API, even under highly demanding conditions.

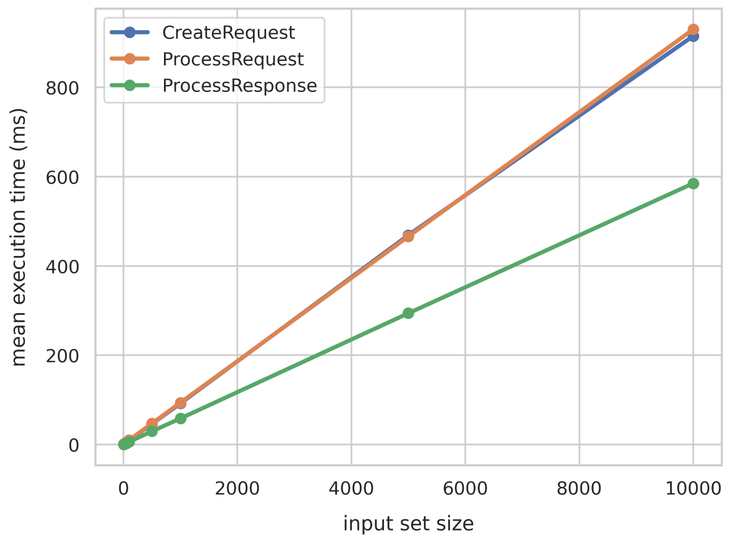


Figure 4.1: microbenchmarking the C++ API

Python API

From the figure 4.2, we can see that the python API behaves in a similar way to the C++ API. The execution time is linear in the size of the input set and the 3 operations take an execution time smaller than 20 ms for a set size of 100 and between 60 ms (`ProcessResponse`) and 160 ms (`ProcessRequest`) for an input size of 1000 elements which is more than enough performance for our application.

4.2 PSI-CA k-NN Evaluation

4.2.1 k-NN Quality

In this section, we provide a convergence analysis of our graph construction algorithm by studying the quality of the graph in terms of the number of performed rounds. The quality of a k-nearest neighbors graph is evaluated by determining its average similarity and comparing it to the average similarity of the ideal graph $quality(G) = \frac{avg_{sim}(G)}{avg_{sim}(G_{ideal})}$.

For every pair of (graph size , # of neighbors k), we log the average similarity for every iteration in the graph construction procedure and divide it by the

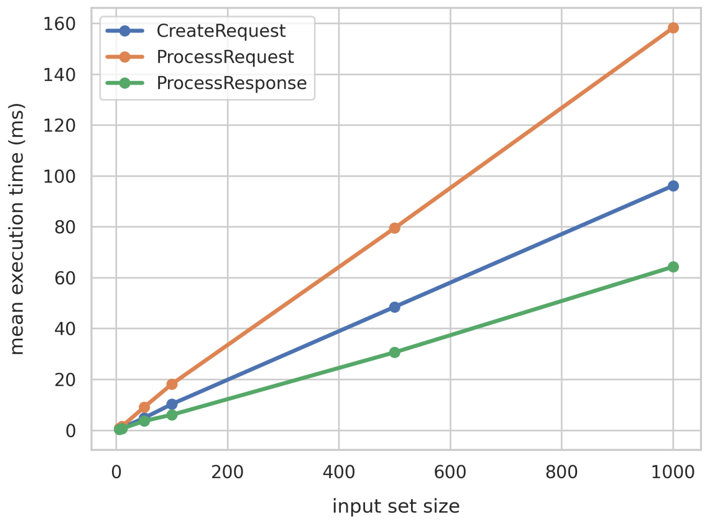


Figure 4.2: microbenchmarking the Python API

average similarity of the ideal k-NN which we have previously constructed. We present the results of our experiments in 4.3.

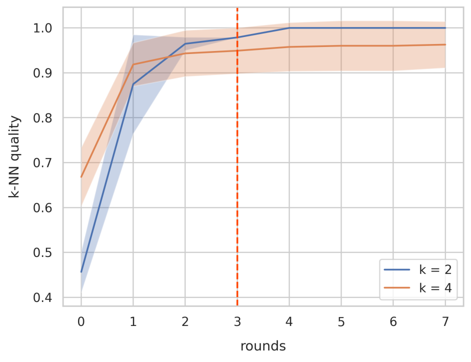
The plots in figure 4.3, show that in all the tested cases our system reaches more than 80 % quality after $log_2(graph\ size)$ iterations which confirms the claims in [2] that the convergence time of the greedy construction algorithm is $\mathcal{O}(log_2(graph\ size))$. In addition, we can clearly see that we reach a near ideal graph with a number of iterations smaller than $graph\ size/2$ in all our runs.

4.2.2 Communication Overhead

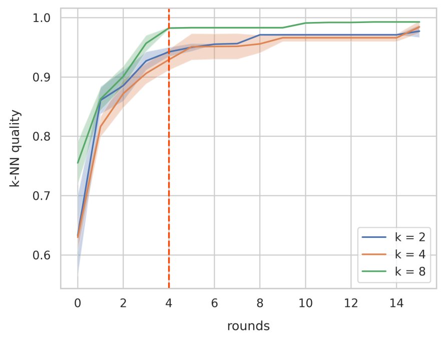
The communication overhead is measured by comparing the total number of exchanged bytes with the standard KNN class across varying graph sizes and k values. Our findings 4.4 reveal that the PSI-CA version necessitates 3.2 to 4.0 times more communicated bytes. This significant overhead mainly results from two factors.

First, the PSI-CA protocol demands an additional message exchange `Response`. From Figure 3.1, this `Response` appears to be approximately twice as large as the initial message. This is due to the inclusion of both sets, which undergo a process of hashing and masking, thereby increasing the message size.

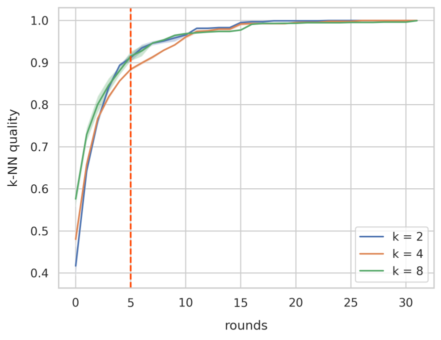
Second, the protocol involves the serialization of elliptic curve elements. These elements are more complex to represent compared to integers and consequently, require more bits. This complexity translates into a higher cost when these elements are transmitted over the network, further contributing to the communication overhead.



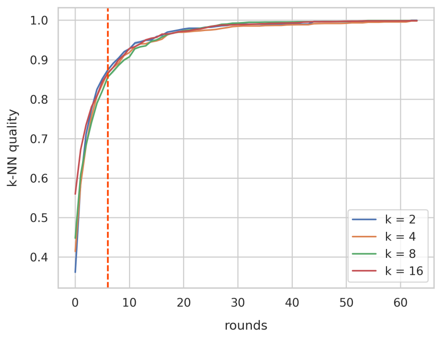
(a) 8 nodes



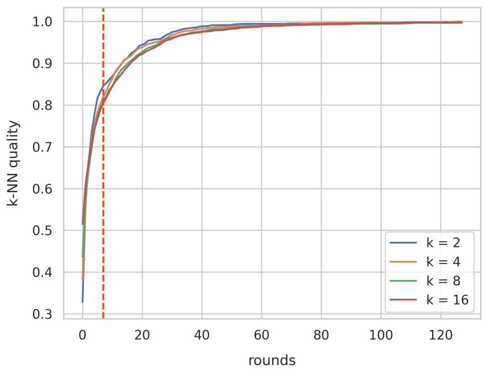
(b) 16 nodes



(c) 32 nodes

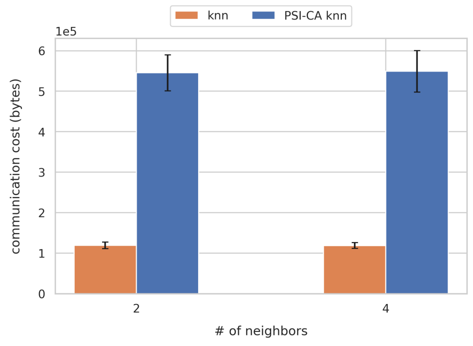


(d) 64 nodes

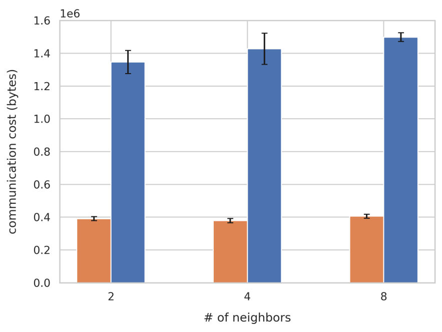


(e) 128 nodes

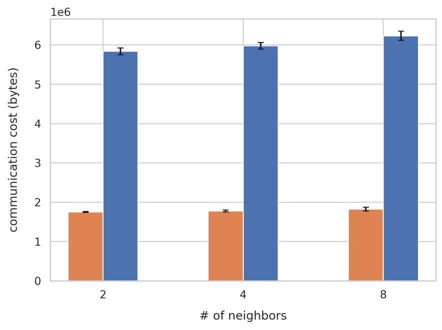
Figure 4.3: PSI-CA k-NN graph quality per round



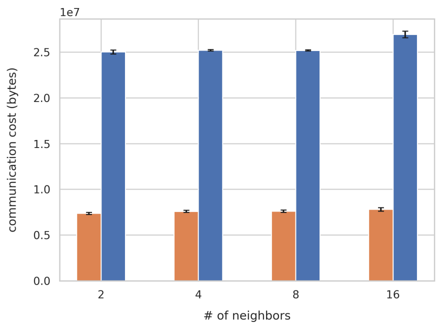
(a) 8 nodes



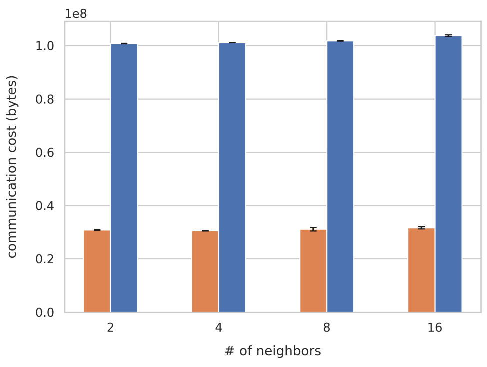
(b) 16 nodes



(c) 32 nodes



(d) 64 nodes



(e) 128 nodes

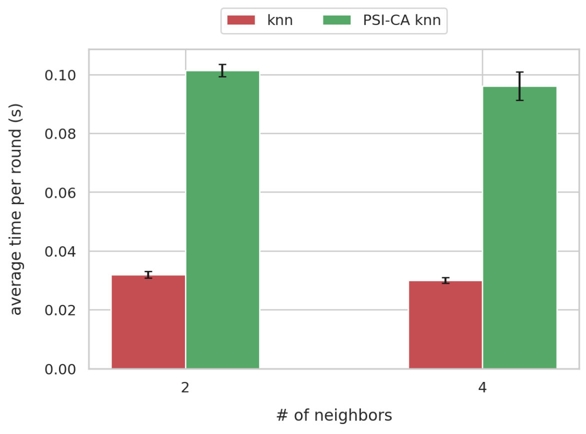
Figure 4.4: Total number of exchanged bytes for regular and PSI-CA k-NN

Despite the increased communication overhead, the PSI-CA protocol still demonstrates reasonable performance. In fact, the overhead is closely aligned with the theoretical limit as dictated by the design of the protocol. This indicates the efficiency of the protocol, ensuring that while there may be an increase in communication overhead compared to the standard KNN class, it is not beyond what would be expected given its design parameters.

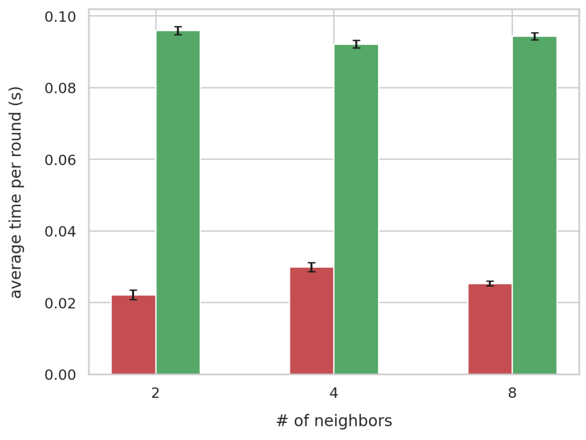
4.2.3 Computation Overhead

We assess the computational overhead of PSI-CA by measuring the average time taken per round during the graph construction procedure. Our results indicate that a round in a PSI-CA node is between three to four times slower than a regular node round, while maintaining an average of less than 0.1 seconds. This overhead is attributable to the PSI-CA computation itself, as the node needs to conduct the computation for the ProcessResponse phase for the candidates it selects, in addition to the ProcessRequest step for the requests it receives from other nodes. Additionally, PSI-CA requires more message exchanges than the regular procedure, resulting in more congestion in the send queue and increased wait time for messages from other peers, leading to additional latency.

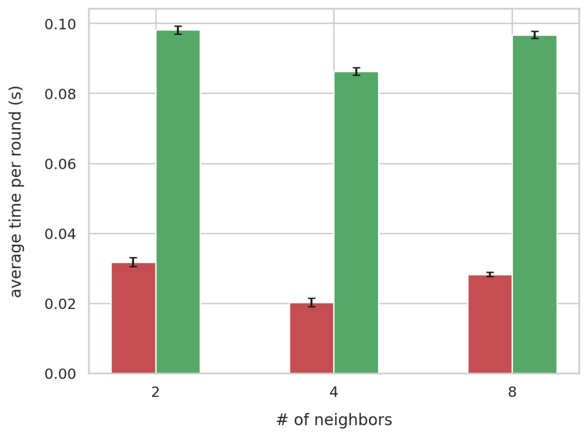
It is important to note that these benchmarks were performed in a cluster setting where communication latency is negligible and the computation latency is the dominant factor which doesn't represent the realistic settings for such a system.



(a) 8 nodes



(b) 16 nodes



(c) 32 nodes

Figure 4.5: Average time per k-NN round for regular and PSI-CA k-NN

Chapter 5

Conclusion & Future Works

In conclusion, the system we have implemented successfully achieves privacy-preserving k-nearest neighbor graph construction, providing privacy and correctness against semi-honest adversaries while not leaking information in the malicious adversaries model. However, it is important to acknowledge that our system introduces a non-negligible overhead in terms of communication and computation.

Despite the inevitable overhead, the performance of our system remains reasonable even when scaling the number of nodes. The time and communication costs are still comparable to the base case and do not increase exponentially. Thus, it is justifiable to accept the small compromise in performance to ensure user privacy.

Furthermore, it is worth noting that there is still room for improvement in both performance and security aspects of our system. In the upcoming paragraphs, we will explore potential enhancements and advancements that can be made to address these areas.

Authorized PSI-CA

Our current implementation of PSI-CA doesn't include any authorization mechanisms which can lead to impersonation attacks. One interesting solution to this issue is the use of Authorized PSI-CA [1].

Size Hiding PSI-CA

The PSI-CA protocol we built upon to design our system by default reveals the sizes of the sets of the 2 parties. This can be a security concern as an adversary may use this information to gain knowledge about the users [5]. There are some works that address this specific problem by introducing Size Hiding private set intersection.

Homomorphic Encryption

In this project we explored only one privacy enhancing technology which is private set intersection cardinality. A possible alternative approach is the use of homomorphic encryption. There has been some works about PSI from homomorphic encryption [6] that claim reducing the communication overhead compared to other PSI protocols.

Improving `fpsica` Performance

There is still room for improvement in our library implementation as our current C++ code doesn't involve any parallelization. A possible optimization would be to have multiple threads perform the computationally expensive cryptographic operations on disjoint chunks of the input/received sets which would significantly reduce the execution time especially for large sets.

Bibliography

[1] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. Cryptology ePrint Archive, Paper 2011/141, 2011. https://eprint.iacr.org/2011/141.

[2] Simon Bouget, Bromberg Yerom, François Taiani, and Anthony Ventresque. Scalable anti-knn: Decentralized computation of k-furthest-neighbor graphs with hyfn. pages 101–114, 05 2017.

[3] Akash Dhasade, Anne-Marie Kermarrec, Rafael Pires, Rishi Sharma, and Milos Vujasinovic. Decentralized learning made easy with decentralizepy. In *Proceedings of the 3rd Workshop on Machine Learning and Systems*, EuroMLSys '23, page 34–41, New York, NY, USA, 2023. Association for Computing Machinery.

[4] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), dec 2015.

[5] Giuseppe Ateniese, Emiliano De Cristofaro, and Gene Tsudik. (if) size matters: Size-hiding private set intersection. Cryptology ePrint Archive, Paper 2010/220, 2010. https://eprint.iacr.org/2010/220.

[6] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. Cryptology ePrint Archive, Paper 2017/299, 2017. https://eprint.iacr.org/2017/299.