2015

Exploring Techniques for Providing Privacy in Location-Based Services Nearest Neighbor Query

John-Charles Asanya
University of Central Florida

EXPLORING TECHNIQUES FOR PROVIDING PRIVACY IN LOCATION-BASED SERVICES NEAREST NEIGHBOR QUERY

by

CHARLES N. ASANYA
B.S. University of Central Florida, 2007
M.S. University of Central Florida, 2009

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2015

Major Professor: Ratan K. Guha

ABSTRACT

Increasing numbers of people are subscribing to location-based services, but as the popularity grows so are the privacy concerns. Varieties of research exist to address these privacy concerns. Each technique tries to address different models with which location-based services respond to subscribers. In this work, we present ideas to address privacy concerns for the two main models namely: the snapshot nearest neighbor query model and the continuous nearest neighbor query model.

First, we address snapshot nearest neighbor query model where location-based services response represents a snapshot of point in time. In this model, we introduce a novel idea based on the concept of an open set in a topological space where points belongs to a subset called neighborhood of a point. We extend this concept to provide anonymity to real objects where each object belongs to a disjointed neighborhood such that each neighborhood contains a single object. To help identify the objects, we implement a database which dynamically scales in direct proportion with the size of the neighborhood. To retrieve information secretly and allow the database to expose only requested information, private information retrieval protocols are executed twice on the data. Our study of the implementation shows that the concept of a single object neighborhood is able to efficiently scale the database with the objects in the area.

The size of the database grows with the size of the grid and the objects covered by the location-based services. Typically, creating neighborhoods, computing distances between objects in the area, and running private information retrieval protocols causes the CPU to respond slowly with this increase in database size. In order to handle a large number of objects, we explore the concept of kernel and parallel computing in GPU. We develop GPU parallel implementation of the snapshot query to handle large number of objects. In our experiment, we exploit parameter tuning. The results show that with parameter tuning and parallel computing power of GPU we are able to significantly reduce the response time as the number of objects increases. To determine response time of an application without knowledge of the intricacies of GPU architecture, we extend our analysis to predict GPU execution time. We develop the run time equation for an operation and extrapolate the run time for a problem set based on the equation, and then we provide a model to predict GPU response time.

As an alternative, the snapshot nearest neighbor query privacy problem can be addressed using secure hardware computing which can eliminate the need for protecting the rest of the sub-system, minimize resource usage and network transmission time. In this approach, a secure coprocessor is used to provide privacy. We process all information inside the coprocessor to deny adversaries access to any private information. To obfuscate access pattern to external memory location, we use oblivious random access memory methodology to access the server. Experimental evaluation shows that using a secure coprocessor reduces resource usage and query response time as the size of the coverage area and objects increases.

Second, we address privacy concerns in the continuous nearest neighbor query model where location-based services automatically respond to a change in object's location. In this model, we present solutions for two different types known as moving query static object and moving query moving object. For the solutions, we propose plane partition using a Voronoi diagram, and a continuous fractal space filling curve using a Hilbert curve order to create a continuous nearest neighbor relationship between the points of interest in a path. Specifically, space filling curve results in multi-dimensional to 1-dimensional object mapping where values are assigned to the objects based on proximity. To prevent subscribers from issuing a query each time there is a change in location and to reduce the response time, we introduce the concept of transition and update time to indicate where and when the nearest neighbor changes. We also introduce a database that dynamically scales with the size of the objects in a path to help obscure and relate objects. By executing the private information retrieval protocol twice on the data, the user secretly retrieves requested information from the database. The results of our experiment show that using plane partitioning and a fractal space filling curve to create nearest neighbor relationships with transition time between objects reduces the total response time.

To my family

ACKNOWLEDGMENTS

First, I will like to give thanks to the Lord God who is my refuge and my fortress. I praise and glorify you with all my heart for giving me the perseverance and presence of mind throughout this journey. You have lifted me up and shown me the path when I found myself on a crossroad, and for that I owe an everlasting gratitude.

I will also like to use this opportunity to thank my major professor Dr. Ratan K. Guha for being my able adviser and for his assistance throughout my PhD endeavor. Dr. Guha has taught me valuable lessons in learning and research. His eat pie by the slice approach to research has helped me develop into a researcher that I am today and hope to continue throughout my career. His experience, encouragement, and invaluable input have helped advanced my work as a researcher, and for that I am grateful. I will also like to extend my gratitude to the members of the dissertation committee starting with Dr. Damla Turgut for her encouragement and willingness to accommodate my request as a committee member even with several timeline shifts and unforeseen circumstances. I will also like to thank Dr. Mostafa Bassiouni for his input especially during my proposal presentation which encouraged me to advance my research. I also offer great thanks to Dr. Ram Mohapatra and Dr. Cliff Zou who without hesitation accepted my request as a committee member even on a short notice. To all these committee members who despite their busy schedule were able to take time off to be part of my education career I owe my gratitude and I feel honored.

To the most important people in my life, my family, my gratitude and love are everlasting. To my father Isaac Asanya, your discipline and believe in me has been the foundation that has carried me through life and inspired me to reach for the pinnacle of education. I thank you for your encouragement and unwavering support for education. I still remember vividly as a little boy reading newspaper to you while you lay on the sofa nodding your head as I read it out loud. Though at the time I could not totally comprehend what I was reading, but right then, the torch for the thirst of knowledge was lit. Today, I carry this torch with joy and pride knowing how proud you will be of me.

My gratitude also goes to Lockheed Martin Corporation who provided me the opportunity and assistance to further my education.

Finally, I will like to extend my gratitude to the Department of Engineering and Computer Science of the University of Central Florida. Thank you for providing me with the resources for my research, for that I am very grateful.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AP	Access Point
*C*NNQ	Continuous Nearest Neighbor Query
*c*PIR	Computational Private Information Retrieval
CS	Common/Central Server
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
CUDA	Compute Unified Device Architecture
FIFO	First In First Out
GPS	Global positioning system
GPU	Graphical Processing Unit
I/O	Input/output
LBS	Location Based Services
Mbps	Megabits Per Second
MBR	Minimum Bounding Rectangle
MQMO	Moving Query Moving Object
MQSO	Moving Query Static Object
MPI	Message Passing Interface
NIDS	Network Intrusion Detection System
NNQ	Nearest Neighbor Query
NVD	Network Voronoi Diagram
ORAM	Oblivious Random Access Memory
OS	Operating System

OT	Oblivious Transfer
PIR	Private Information Retrieval
POI	Point of Interest
QOS	Quality of Service
QRA	Quadratic Residuosity Assumption
RAM	Random Access Memory
SC	Secure Coprocessor
SM	Streaming Multiprocessor
SNNQ	Snapshot Nearest Neighbor Query
SP	Scalar Processor
SQMO	Static Query Moving Object
SR	Service Request

CHAPTER ONE: INTRODUCTION

Advancement in mobile technology and wireless communication has provided opportunities for developers to create complex applications. This new computing paradigm is driven by the high-powered GPS enable mobile devices with location information access capability. Examples of this type of application includes: (1) the store locator application described in [1] that allows users to quickly find store location with the help of location intelligence; (2) the proximity-based marketing which allows companies to send advertisements to individuals within a specific geographic location; (3) the travel information application that updates subscribers with traffic or weather information. Other examples include the nearest neighbor locator which allows users to locate nearest point of interest (POI), and the reverse 911 used to alert subscribers in times of emergencies. These types of applications are referred to as location-based services (LBS) and are growing in popularity and becoming ubiquitous.

LBS-Nearest Neighbor Query

Location-based services are information and entertainment services that are accessible through portable device and mobile network. LBS providers deliver the service through user mobile device using the base transceiver station (BTS) that the mobile device is communicating with [2], or by the more accurate global positioning system (GPS) satellite. In [2], the services were classified into push and pull. In push, users within target areas receive solicitation from advertisers or solicitors without requesting. Some examples are: (1) shoppers in a department store alerted of items on sale when within the targeted section of the store; (2) travelers on an international trip informed of a discounted international phone plan when within the international terminal of the airport.

In a pull system, user actively requests information before receiving it. For example, a traveler driving through a city may decide to spend the night in the city and requests for the nearest lodging place. Other examples include visitors to an area who wish to dine on an ethnic cuisine and requests for the specific nearest ethnic restaurant, or a commuter walking around searching for the nearest taxi. Specifically, user issues a query referred to as nearest neighbor query (NNQ) to the LBS provider requesting for the nearest POI to their location. LBS responds in one of two types of NNQ called snapshot or continuous nearest neighbor query. Snapshot nearest neighbor query (SNNQ) is the response to a user query that represents a snapshot of a point in time. Basically, user issues a query to the LBS requesting for the nearest POI, when the nearest POI changes, another query is required from the user. Continuous nearest neighbor query (CNNQ) on the other hand is the response that continuously updates the user with the nearest POI as the user location or the nearest POI changes without another query from the user. To customize the service, user has to disclose some personal information. Though LBS have provided convenience and information access that is unimaginable years ago; nevertheless, privacy advocates have raised concerns on the negative fallout from a possible collusion between LBS provider and an adversary. Some users will be reluctant to use the service for fear that doing so will compromise their personal information and expose them to danger, embarrassment or reprisal. Therefore, protecting user personal information is necessary for these types of services. Note that POI and user are interchangeable for object.

Protecting Private Information in LBS-NNQ

Nearest neighbor query is a query issued in order to locate the nearest POI assuming the objects are statically or dynamically restricted within the edges of a spatial network S_N. A user that wish to receive from the LBS the nearest POI to its location within the S_N has to disclose to the LBS together with the service request (SR) message the raw position information which will allow the LBS to customize the request. If for instance a user SR message is represented as SR(*23.53'28"N, 71. 42'36"W,q*(*i*)) which contains user location (*23.53'28"N, 71.42'36"W*) and desired information *q*(*i*), SR exposes the raw position data of the user. An untrustworthy provider or third party intermediary can collaborate with adversary to misuse this information. NNQ is considered private if a user such as Alice issues a query to request for the nearest casino to her location and Bob is unable to determine Alice's location or information she requested. This requires that Alice location and desired information remains anonymous to all parties. Providing anonymity to user private information is a challenge, but it has been receiving lots of attention from researchers. Interestingly, SNNQ and CNNQ pose different challenges when it comes to protecting user private information. It is important to note that privacy requirement varies for different users and type of information desired. For instance, a user looking for the nearest gas station may have different privacy requirement from a user looking for the nearest gambling place. In the rest of this chapter, we briefly describe the different areas covered in our work with our contributions.

Providing Privacy in SNNQ

In a SNNQ, a user device issues a query to the LBS server requesting the nearest POI such as a gas station, restaurant or hospital. Server responds to the user request and terminates the session. As the nearest POI changes due to a change in the user or the POI location, the user does not receive update. Any update due to a change in location will require that the user issue another query. To protect private information of the user, several proposals like [3] assume the existence of other k -1 users with similar profiles as the target user which allows the system to aggregate the queries in order to obscure the profile of the target user. Other methods rely on the honesty of the server or third party [4], while some of the methods release too many database information to the user and therefore require the user to filter through the data to find its nearest POI as in [5]. While each of these solutions has proven to be effective, we found various limitations to the techniques. For instance, a query may never be answered for lack of existence of other k -1 users in the target area. Also, releasing large amount of database information engages resources longer than may be desired, and LBS may not want to release too many of their valuable information if there is no financial benefit. To address this, we introduce space mapping model based on abstracting the concept of an open set in a topological space where points in the space belongs to a subset called neighborhood of a point. We combine it with a database design that dynamically scales in direct proportion with size of the object to help a user relate a position with the location of information in the database. We execute private information retrieval (PIR) protocol twice on the data in the database to secretly retrieve the information and minimize the amount of information released. The design allows us to scale the database server based on the size of the area and number of objects for performance.

Contributions

The main contributions are summarized as follows:

1. We propose a new model to protect user private information based on abstraction of sets in a topological space as a solution to cloaking and reliability on a server or third party.
2. We propose to scale the database in direct proportion with the size of the object to obscure and relate object position with information in the database.
3. We provide experimental evaluation of our proposal with respect to the execution time and complexity to show feasibility and scalability of the methodology and compare the complexity with related work.

Parallel Implementation of a Private SNNQ

The increasing popularity of LBS-NNQ results in rapid increase in the number of subscribers and information processed. This increase results in the growth of the size of the database. Usually, the security protocol used to encrypt the data, creating neighborhoods, computing distances between objects in the area causes the CPU to respond slowly with the increase in database. A faster computation mechanism therefore is required to support this growth. Interestingly, several researches address this issue by implementing multi-threading in a CPU [6]; however, the limited number of cores in CPU limits the amount of improvement that can be derived from it. We propose to use the array of streaming multiprocessors (SM) in graphical processing unit (GPU) and the inherent parallel computation capability of the GPU to reduce the processing time. To do that, we exploit computation sharing by running operation in parallel where necessary to reduce computational workload and speed up the computation. The challenge in GPU computing is to find enough parallelism in the algorithm that will account for the transfer overhead between the CPU and the GPU device.

Contributions

The main contributions are as follows:

1. We propose to use the streaming multiprocessors of GPU and the parallel computing platform and programming model of compute unified device architecture (CUDA) to address the increase in response time as the size of the database increases.
2. We provide experimental evaluation of our parallel implementation and compare with CPU serial implementation to show the speedup of GPU over CPU.
3. We conduct performance evaluation of the GPU, and then provide analysis and model to predict GPU execution time.

Providing Privacy in SNNQ with Tamper-Resistant Hardware

As an alternative to space mapping and to minimize resource usage and network transmission time, we explore hardware approach by using secure coprocessor (SC) computing technology to obscure user private information. Commercially available secure coprocessor was shown in [7] to assist privacy. This approach can support traditional query and can have lower overhead. Secure coprocessor is a tamper-resistant hardware device designed such that upon a break-in the intruder cannot learn or change the internal state except through normal I/O channels or by forcibly resetting the entire secure coprocessor. Lately, there has been significant research on using this method to protect data; however, most are implemented to protect data in a cloud computing environment. The work in [8] and [9] set up a secure channel between user and trusted processor to transmit data. However, the solution fails to prevent leakage of information that could assist adversary to determine the type of memory access, memory location accessed, or what data is read or written by the client. Proposal in [10] seem effective but only hides user’s location.

We propose to use secure coprocessor in LBS-SNNQ to protect both user location and information accessed. We propose to implement oblivious random access memory (ORAM) methodology to obfuscate database access pattern. ORAM is a probabilistic random access memory (RAM) with a memory access pattern that allows a client to obscure its access pattern to a remote storage server. The secure coprocessor accesses LBS server through the ORAM methodology and secretly computes user desired information and then exchange encrypted information with the user.

Contributions

Our contributions are as follows:

1. We propose to process user query using secure coprocessor to hide both user location and information accessed and minimize size of data transferred.
2. We propose to access LBS server using ORAM methodology to obfuscate database access pattern.
3. We provide analysis of the emulation of the secure coprocessor to show improved performance over software approach as the size of the object increases.

Providing Privacy in CNNQ

In a spatial network the position of a mobile object will change as the object moves from one point to another. Depending on the trajectory or the object motion, the change may lead to a change in the nearest POI to a user position. For a user to issue a query each time a change occurs will be inefficient. CNNQ supports this mobility, that is, user device issues a query for the nearest POI, server responds to the initial request, but also continues to update the user with the most current nearest POI as the user or the POI location changes. Processing private nearest neighbor query for a moving object is not an easy task. Proposals in [11], [12], [13] and [14] have been suggested to find nearest neighbor but with no consideration on user privacy. While these suggestions were effective and extends the research in the area of CNNQ, users in LBS-CNNQ still runs the risk of having their private data exposed which can be misused by adversaries. Solutions based on snapshot query as some researchers suggested will be cost prohibitive as the frequency of the changes increases. Proposal in [15] introduces privacy technique for LBS-CNNQ, though effective, it works like SNNQ. As stated in [11] and [16], it is inefficient and infeasible for a moving object to issue a query at every point of the line segment as in SNNQ. The problem gets more complex when we consider that the mobility of the objects involved can affect relationships in a CNNQ. This relationship leads to what is referred as mobile query static object (MQSO) (e.g. a user searching for the nearest gas station while driving through a city), moving query moving object (MQMO) (e.g. a user driving through a town searching for nearest friend who may be driving through the town), and static query mobile object (SQMO) (e.g. a user at a bus station searching for the nearest bus to the station). In our work, we only consider MQSO and MQMO.

In MQSO-CNNQ, the user (query) is dynamic while the object (POI) is static. If for instance, the nearest neighbor to a user at a location *l* is p_1, and $p_1$is the POI, as the user moves to another location *l'*, its nearest neighbor at this location may change depending solely on the motion and trajectory of the user.

In MQMO-CNNQ, both the user and the POI are dynamic within a spatial network. If the nearest POI to a user at a location *l* is p_1 the nearest POI at a future location *l'* depends on the trajectory and motion of both the user and the POI. Location of the user and the POI can change simultaneously. POI location is only known to the server at time *t*. To guarantee that server returns nearest POI, the location of the POI at a future time *t'* should also be known by the server. Server cannot be updated offline as in MQSO due to the possibility of a change in the POI location after a query has been initiated. In order for server to continuously respond with the precise POI, the location of the query and the POI at time *t* and *t'* must be known. Given the frequency of updates that may be involved in this type of query, privacy technique used in SNNQ will be too costly to implement. In order to implement the same privacy technique as in

SNNQ, large amount of information has to be exchanged between the user and the server throughout the query segment.

As a solution, we propose plane partitioning using Voronoi diagram to bind objects in a region, and use continuous fractal space filling curve to determine proximity and create relationships between the objects in a path. We also implement a database that dynamically adjusts to match the size of the objects in the path. We also execute PIR protocol twice on the data to secretly retrieve the information and minimize the amount of information released.

Contributions

Our contributions to private CNNQ are as follows:

1. We propose plane partitioning and continuous fractal space filling curve using Voronoi diagram and Hilbert curve order to create nearest neighbor relationship through a path.
2. We treat object location as a function of time and introduce the concept of transition and update time that indicates where and when a nearest neighbor will change to deal with inefficiencies of multiple queries caused by the constant changes in object location.
3. We provide experimental evaluation with respect to complexity and network communication time to show an efficient scalable design.
4. We implement previous work and compare the response time with our design to show that our design reduces the total response time.

Organization of the Dissertation

The rest of the work is organized as follows: In chapter 2, we review related works starting with existing approaches to privacy in LBS-SNNQ. We then discuss the direction taken by researchers to reduce the response time in a NNQ. We discuss works on alternative method of privacy using hardware, and finally in chapter 2, we present previous works on privacy in LBS-CNNQ. In chapter 3, we explore privacy technique in LBS-SNNQ starting with the problem definition. We then present the model for our solution and implementation. We follow with experimental evaluation. In chapter 4, we discuss parallel computing using GPU and CUDA architecture to address the increase in response time in the SNNQ due to the increase in the size of the database, and then follow with our implementation of the SNNQ. We then conduct experimental analysis to compare serial and parallel implementation of our proposal. Then, we present GPU Run-time modelling for the snapshot query. In chapter 5, we explore the idea of a tamper-resistant hardware using secure coprocessor computation to protect private information in LBS. We discuss the experimental evaluation and compare with software approach. Chapter 6 discusses LBS-CNNQ and presents our privacy solutions for MQSO and MQMO types of CNNQ. We follow with experimental analysis of the implementation. We then implement related work and compare with our design. Then, we present analysis on the impact of large prime in software approach. Finally in chapter 7, we followed with conclusion and future work.

CHAPTER TWO: RELATED WORKS

In this chapter, we review relevant works on private LBS-NNQ and other works on data privacy and parallel computing. We start by presenting previous privacy techniques on LBS-SNNQ. We follow with work on parallel implementation of a NNQ. We then discuss alternative approach to privacy using tamper-resistant hardware and then finish with work done on LBS-CNNQ.

Snapshot Nearest Neighbor Query

As stated in [17], three main models are used to protect user private information in LBS-NNQ. Among them are: two-tier spatial transformation, three-tier spatial transformation and cryptographic technique.

Two-tier spatial transformation approach provides direct communication between user and the LBS. It is mostly implemented through technique like *k*-anonymity. In this model, the LBS are prevented from distinguishing user query from other *k*-1 queries by generalizing the query. Shin et al [18] designed two-tier system based on anonymizing user profile. User specifies the extent that location and profile information should be generalized such that the generalized area includes at least other *k*-1 users with identical profile as the user. This will prevent the LBS from determining the exact query origin. Another two-tier approach was proposed in [3], [4] and [19]. In these techniques user specifies the minimum level of anonymity desired, the desired maximum spatial and temporal resolution, and the acceptable waiting interval. If other *k*-1 users exist in the region within this interval, all the queries will be combined into a cloaking region and then send to LBS server. If on the other hand no other query exists in the area within this interval, the user query will be dropped. Cloaking region was proposed in [20] to increase the location privacy of the user at additional computation. The problem with these techniques is the dependency on other queries and large coverage area. There is no guarantee that another query will exist in the region within the time interval, and even when they exist the profile may be substantially different than that of the target query. Therefore, there is a possibility the query may go unanswered.

Three-tier spatial transformation is another way to protect user private information in LBS. As implemented in [21] and also in [4] using a trusted third party, user communicates directly with the trusted server which then redacts user information before transmitting to LBS. Kalnis et al. [22] proposed three-tier idea by using trusted anonymizer. The trusted anonymizer collects user current location and anonymizes the query before sending to LBS. User locations are sorted and grouped using Hilbert Space-filling curve order. Each user can decide level of anonymity required ranging from 1 (no privacy required) to user cardinality (maximum privacy). Idea that allows the entire mobile node to communicate with external server through a central anonymity server was proposed in [23]. The central server is part of the trusted computing base. During initialization, mobile nodes will set up authenticated encryption connection with the anonymity server. The anonymity server will then decrypt the message, removes any identifiers, and distorts the position data according to the prescribed cloaking algorithm to reduce the re-identification risk. However, these techniques rely on total honesty of the third party server and also provide a single point of failure.

Another technique used in LBS-NNQ to protect user private information is cryptography. This model is implemented through encryption of user information. Cryptography in most cases provides direct communication between user and LBS sever thereby eliminating the need for a third party. It encrypts user query prior to sending to the server. A hybrid technique combining oblivious transfer (OT) and homomorphic encryption to protect user location information and user profile was proposed in [24]. It introduced a proxy party that interacts with multiple services and collects payment from subscribed users. The proxy then distributes the money to the services on behalf of the users without the proxy or the services learning the exact relationship between user and LBS. Since this protocol uses a proxy, privacy depends on the honesty of the proxy. The focus of [25] is on improving the work of [24]. The authors use OT twice to eliminate the need for a proxy. It requires multiple LBS providers to coordinate with a central server. It uses symmetric encryption which makes it hard for key management as number of users increase.

The most relevant to our approach is the method proposed in [5] and [26]. The technique is based on the design of a database represented as a square matrix of size *n* and PIR protocol that relies on quadratic residuousity assumption (QRA). User encrypts its request to the server. Server runs PIR protocol on user request and responds to the user. The user decrypts the response using prior knowledge of some large prime. However, as used in [5], it reveals too many information to the user. It also increases transmission time and user computation.

Our proposal is based on the single database design and computational private information retrieval (*c*PIR) protocol of [27]. We use space mapping technique that creates single object neighborhood that scales to the database server and then implement PIR protocol twice to minimize the amount of information released by returning only the requested information to the user. It also reduces the server transmission and user computation cost.

Parallel Computing in a Nearest Neighbor Query

Increase in the number of LBS subscribers result in increase in the information processed by the LBS and subsequently increase in the size of the database. Finding distances between points, sorting computed distances, selecting reference points, searching and updating locations, query processing on Voronoi-based index, and a host of other technique used in nearest neighbor search can be computational intensive as a result of this increase. Several parallel implementations have been suggested to reduce this computational burden involved in processing nearest neighbor query. CPU multicore architecture was proposed in [28] where the architecture has a primary site and a number of secondary sites. The relevant sites are activated simultaneously. It traverses R-trees in parallel to collect the most promising nearest neighbor. The upper tree level is maintained in the primary site, while the leaves and corresponding data objects are stored in the secondary sites. The efficiency measure was the response time of a given query. However, this method may not be ideal for LBS private query since computation is distributed among multiple sites. In [6], database search of age-wise groups similar to *k*-NN search was performed on a desktop using multicore processor to take full advantage of parallel programming. While computation cost can be significantly reduced through multithreading in a multicore CPU; nevertheless, the limited number of cores in average CPU limits the improvement that can be achieved through multithreading. GPU provides the needed improvement. GPU parallel implementation was used in [29]. It uses hashing to construct *k*-NN algorithm exploiting the multiple cores and data parallelism to achieve linear space and time complexity. In [30], NVIDIA CUDA API was used in *k*-NN.

Our proposal is similar to [30] leveraging GPU inherent parallel computation capability and CUDA API to reduce the computational burden in SNNQ as a result of the increase in database size. We took advantage of the independent of the data points to maximize the parallelization. We experimented with parameter tuning to determine a more efficient parameter size for GPU computation. The algorithm is split between CPU and GPU. The part where the output data points cannot be represented without relation to each other is implemented on the CPU, while the part where the output data is independent is implemented on the GPU. The massive reduction in the GPU computation compensates for the data transmission between CPU and GPU thereby reducing the overall run time.

Tamper-Resistant Hardware Private Data Computation

Alternative technique to protect private information is to use tamper-resistant hardware. Several tamper-resistant hardware techniques have been proposed to protect private information. In [7], it was shown that privacy can be achieved using commercially available coprocessor through algorithm that both provides asymptotically optimal performance and also promises reasonable performance in real implementations. A prototype that gives reasonable performance on dataset sizes up to about 10, 000 was implemented in [31]. In [32], diverse tools that ranged from ORAM to switching networks and the snort network intrusion detection system (NIDS) were used to build a trusted computing secure hardware that enhanced client privacy. Tamper resistant hardware that uses a secure trusted coprocessor to process private information was proposed in [8] and [9]. It works by a user setting up a secure channel with a trusted coprocessor. This approach has been stated to perform better than cryptographic technique [33]. However, none of these techniques was implemented for LBS private query. Also, some of them do not prevent information leakage that could assist adversary to determine the type of memory access which could lead to revealing information accessed. In [10] and [34], idea of a trusted platform module (TPM) using secure hardware-aided PIR to provide privacy in LBS was proposed. User sends encrypted location information to query scheduler. The scheduler forwards the encrypted information to the TPM who decrypts the message and then uses PIR to retrieve information form LBS database. It encrypts the request and sends back to the user. This proposal does not prevent disclosure of the information accessed. It only hides the location. It also does not hide access pattern which can leak information that may compromise location privacy. A proposal that uses SC to encrypt and shuffle the entire database was implemented in [35]. SC blindly retrieves user request from the database and encrypts it before sending it back to the user. It then stores the information in a cache. Each subsequent request is first checked in the cache before accessing the database to prevent the server from deciphering information accessed based on consecutive query. However, cache can quickly fill up. It also focuses on location privacy and not requested information privacy.

We explore tamper resistant hardware as an alternative to protect private information in LBS-SNNQ. Specifically, we use secure coprocessor to perform operation on a user request. The SC and the user encrypt their information before transmitting. The SC accesses database information using path ORAM of [36]. This technique optimizes the system by focusing on a path rather than the entire database.

Continuous Nearest Neighbor Query

Several solutions including [15] have been suggested to prevent query from divulging user private information in CNNQ. The proposal partitions area into cells using Voronoi graph. The contents of each cell form cooperative group with centered common server (CS). User sends request to CS. The CS takes its own location and a neighbor CS location that the user is heading to as two anchor points and chooses other continuous anchors in the line segment between them. The CS organizes users to form cooperative *k*-anonymity group according to their moving trend, it removes user identity and actual location, and then sends request to LBS. The CS sends each snapshot query of continuous query request with an anchor which the user has not yet passed. The problem with this technique is that it issues query at each segment of the anchor point. It also depends on third party which may not be trustworthy and can provide single point of failure. Another downside is that it depends on other users having similar moving pattern. A design that issues query based on pre-determined uniformly distributed static point was proposed in [37]. It finds the vertex of a cloaked area and the nearest static point to the vertex, and then finds the nearest object to the static query point as an approximate answer for the vertex. It repeats the same process when the nearest neighbor changes. This technique works like snapshot query. The query response will be approximate and not exact nearest neighbor. Also, privacy depends on third party anonymizer. Chow and Mokbel in [38] propose *k*-sharing region. A cloaked spatial region contains at least *k* users, and the region is also shared by at least *k* of these users. Query is issued based on the cumulative cloaked region which will stop adversary from linking query to a specific user. However, it does not say how a user continuously receives nearest neighbor from

LBS. It will also be difficult to maintain exactly *k* users in the entire cloaked region; therefore, query may never be answered.

We propose idea that combines Voronoi diagram and Hilbert curve order to create nearest neighbor relationships. We then create transition time to determine the point where nearest neighbor will change. Server runs PIR protocol of [27] on user request and creates a data queue in increasing order. At each transition point, server pops data from the queue and updates the user with the nearest neighbor.

CHAPTER THREE: PROVIDING PRIVACY IN LBS-SNNQ

This chapter presents the details of our proposed technique to protect private information in LBS-SNNQ. The problem definition is such that given points of interests $P = \{p_1, p_2, \cdots, p_n\}$ where n is the number of p contained in an area A and p_i is a single POI. If a user u positioned in a location l which is inside A issues a query to receive the nearest p from LBS, a private SNNQ should allow LBS server to return to the user u located in l a p whose distance to u is $\leq$ to all other $(P - p_i) \in A$ without revealing the l or the p_i to the server or any other party. Anonymizing user private information based on cloaking a region as in [3] is an effective way to implement private query in LBS-SNNQ. Nevertheless, the growing popularity of LBS has led to increase in the size of area and objects covered. This growth will lead to a rise in the amount of information in the database which may increase database response time. We view this problem as one that can be represented as points in a Hausdorff space where each point belongs to a subset and the database scaled to the size of the subset in the space.

Specifically, we address the issue by adopting the concept of points in a set where each point belongs to a subset called neighborhood of the point that satisfies Hausdorff axioms of topological space. A space is said to be Hausdorff if for any two distinct points in a topological space there exists disjoint open sets containing the two points. Neighborhood of a point is a set containing the point where the point can be moved some amount without leaving the set. We abstract this concept to create object space where object space represents an area A which contains the POI P.

Solution for SNNQ

The idea is to impose a $\sqrt{n}\ x\ \sqrt{n}$ grid *G* over a target area and then map the objects in the grid such that each object is contained in its own neighborhood, and then scale the database to the size of the neighborhoods in the area. Since a user is located in one of the neighborhoods, a user position can be related with location of information in the database. This will allow the database server to return to the user the exact nearest object contained in the same neighborhood with the user without revealing location information or any other POI.

To help explain our design methodology, we first define Hausdorff axiom of topological space. Hausdorff space is a topological space if any two distinct points in the space can be separated into disjoint neighborhoods. A set *S* is considered a topological space if each point $x \in S$ assigned to a non-empty family of subsets of *S* called the neighborhood *U* of *x* satisfies the following axioms:

1. Each point *x* belongs to a neighborhood, i.e., $x \in U$.
2. Any set which contains a neighborhood of a point is a neighborhood of that point, i.e., every superset of a neighborhood of a point $x \in S$ is a neighborhood of *x*.
3. The intersection of any two neighborhoods of a point is a neighborhood of that point.
4. Any neighborhood U of *x* contains a neighborhood *W* of *x* such that U is a neighborhood of each point in *W*

We extend this concept to our object space. We first define the object space in terms of a set *O*, where O is the set for all the p contained in A. We also define $V = \{v_1, v_2, \cdots, v_m\}$ to be the subsets of *O*, where v_i denotes a set that contains all p located in a cell c. We define $NHD =$

$\{N_{HD_1}, N_{HD_2}, \cdots, N_{HD_k}\}$ subsets of V to be the neighborhood of a p. Each $p \in O$ belongs to a subset called neighborhood N_{HD} of p.

Neighborhood Creation

The design is based on dividing an area into neighborhoods that contains the objects in the area such that each neighborhood contains only one object. We first map an area *A* into a square grid *G* of cells *c* similar to Ghinita et al. in [5], and we assign identification number to each *c* (*A* is the area covered by the LBS, while *G* is the grid imposed over the area, and *c* is the grid cells). Figure 1 is an example of an area mapped into a 4x4 grid and a cell within the grid containing 4 POIs. The user *u* is contained in one of the grid cell. For this example, each *c* in the grid must contain 4 POIs and will be padded if necessary to maintain a cell of size 4. Each *c* is identified with a unique identifier. The cells *c* from $c_1, c_2, \cdots, c_n$(n represents the size of the grid) are assigned the value of a unique identifier C_{Id}, i.e., $c_1 = C_{Id_1}$, $c_2 = C_{Id_2}$, and $c_n = C_{Id_n}$.

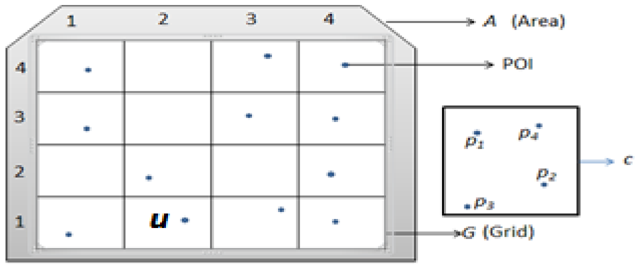


Figure 1 Area covered by a 4x4 grid with a cell containing four points of interests

Each v is divided into neighborhoods N_{HD} that satisfy Hausdorff space such that each p has its own neighborhood and $\forall N_{HD}\ \exists!\ p \in N_{HD}$.

We let the separation distance of p which is represented as $S_d(p)$ (distance from p to the end point of its N_{HD}) be $\frac{DI}{2}$, where DI is the diameter of the area of the N_{HD}. For each N_{HD} and p we assign location data $l_{N_{HD}}$ and l_p respectively based on the geographical coordinate of the area of the neighborhood and the POI. Each $p \in v_i$ is assigned the coordinate location data (l_{NHD}) of the N_{HD} that contains it to prevent a user from identifying the exact location of the objects in the area.

$$\therefore l_{N_{HD_j}}(p_i) \Rightarrow l_{p_i}$$

However, since $N_{HD_j} \not\cong p_i$,

$$\therefore l_{p_i} \nRightarrow l_{N_{HD_j}}(p_i)$$

Each N_{HD} is associated with a c. The N_{HD} is identified by the C_{Id} of the containing c and the $l_{N_{HD}}$ of the N_{HD}. Since N_{HD} is represented by C_{Id} and $l_{N_{HD}}$ so is the p contained in the N_{HD}, i.e.

$$N_{HD_i} = f(C_{Id_i}, l_{N_{HD_j}}) \Rightarrow p_i = f(C_{Id_i}, l_{N_{HD_j}})$$

but not vice versa. The p is now related to its N_{HD}. For example and as shown in Figure 2, if restaurant R_1 in a cell with a C_{Id} of 64 is contained in N_{HD} with a location data of 28.8 and 77.0 obtained from the conversion of the spherical coordinate of $l_{N_{HD}}$ to decimal degree, the R_1 will be assigned the value $l_{N_{HD}}$ and identified as R_1(64, 28.8, 77.0). It is also assigned an integer t (t from $1 \cdots k$) indicating the order of the location of information in the database, where k is the number of POI in a cell. The entire restaurant in the cell will be identified with the same C_{Id} and their respective *t* value together with $l_{N_{HD}}$ of their respective neighborhood. The restaurant R_1 for *t* = 1 will therefore be identified as $R_1(64, +1, 28.8, 77.0)$.

This information is stored in the database which aligns with *G* such that the database tuples are stored in a matching order with *G*.

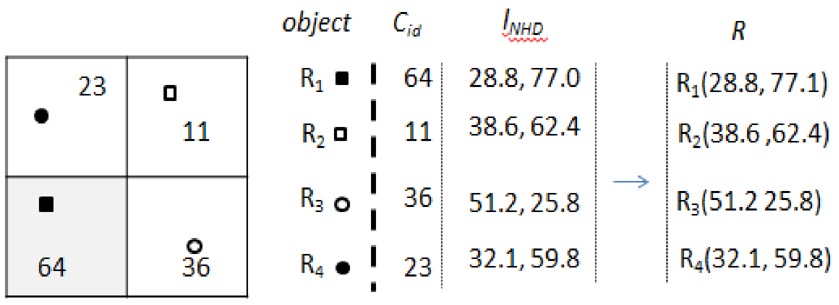


Figure 2 Identifying objects with their cell id and location data

Database Design

Our database design is similar to the work of [5] and [26] using single database design proposed in [27]. The design is such that the tuples are stored in a manner that will allow a user to relate the position with the location of information in the database. The database *D* is of size *n*-bit binary string and represented as a square matrix $M = \sqrt{n}\ x \sqrt{n}$ and indexed by D_i, where $D = \{p_1, p_2, p_3, \cdots p_n\}$. $M_{a,b}$ is the matrix element corresponding to D_i, where *a* and *b* is the index for the matrix row and column respectively. To prevent inference from the number of bits transferred, each matrix element is represented by equal number of bits *m*. This is equivalent to *m* matrices M[1] , M[2] , M[3]. . .M[m] one for each bit of the object in a cell. The computation cost for this will be *O(mn).* Objects in the grid are aligned with information in the database. Each cell can contain multiple objects; therefore, the same concept is applied for *k* objects in a cell. In this case $M_{a,b}$will correspond to $D_{i_{j \to k}}$, where *j* is from 1 to *k*, and (*k* = 1) $\Rightarrow D_{i_j} = D_i$. Each neighborhood contains a single object which is equivalent to m^k matrices (M[1] , (M[2]) , (M[3]) . . . , (M[m]))^k one for each bit of the object in a neighborhood. This will bring the computation cost to O(mn)^k. The design is such that the database can be scaled with the size of the objects for performance. User will request for the *i*-th bit value of *D.*

Private Information Retrieval

When a user wants to find the *i*-th bit value of D_i, to maintain privacy, the user will send encrypted query *q(i)* to the LBS server. Server will run PIR protocol on user request and responds with $r(D_i)$ where *r* is used for encryption. User will compute D_i from $r(D_i)$ to determine *i*. PIR is a protocol that allows a user to retrieve and interpret the value of the information retrieved from a database without revealing the exact information retrieved. Different flavors have been proposed over the years. The earlier version introduced in [39] requires replication of database. In our work, we explore computational private information retrieval (*c*PIR) protocol introduced by Kushilevitz and Ostrovsky in [27] which requires single database tuple distribution. It is based on the premise that there is no known function in polynomial bounded time that will allow a database of size *n* to differentiate between queries for *i*-th bit and *j*-th bit $\forall 1 \leq i, j \leq n$. This is based on the assumption that the computational resources in relation to the running time of the encryption and decryption including adversary algorithms is bounded by some polynomial function in *k.*

The PIR scheme is such that for a database *D* of *n* bit string from which a user wishes to obtain bit D_i while keeping secret index *i* from the database it requires that the user query divulge no information about *i.* As described in [27], *c*PIR is based on Quadratic Residuosity Assumption (QRA). Goldwasser and Bellare used QRA in [40], and it states that it has become computationally infeasible to determine the quadratic residues in modulo arithmetic of a large composite number *N* = *q*q'*, where *q, q'* are large primes. For instance, if *N* is a natural number define

$$\mathbb{Z}^{*}{}_{\mathrm{N}} = \{\boldsymbol{x} \mid \mathbf{1} \leq \boldsymbol{x} \leq \boldsymbol{N}, \boldsymbol{gcd}(\boldsymbol{N}, \boldsymbol{x}) = \mathbf{1}\} \qquad (1)$$

Define Quadratic Residuosity Predicate as $QN(y) = 0$ if $\exists x \in \mathbb{Z}_N^* \; x^2 = y \bmod N$ is true, and $QN(y) = 1$ if $\exists x \in \mathbb{Z}_N^* \; x^2 = y \bmod N$ is not true. If $QN(y) = 0$ (i.e., *y* is a square *y*) *y* is said to be quadratic residue *(QR),* and if $QN(y) = 1$ (i.e., *y* is a non-square *y*) *y* is said to be quadratic non-residue *(QNR).*

If the equation

$$\mathbb{Z}^{+1}{}_{\mathrm{N}} = \{\mathbf{y} \in \mathbb{Z}_{\mathrm{N}}^{*} \mid \left(\frac{\boldsymbol{y}}{\boldsymbol{N}}\right) = \mathbf{1}\} \qquad (2)$$

is true, then half of the numbers in $\mathbb{Z}^{+1}{}_{\mathrm{N}}$ are $\in QR$, and half are $\in QNR$. If *q* and *q'* are large enough $\frac{k}{2}$ bit prime, then for every constant *c* and a family of computational bounded polynomial function $f_{k_0}(y)$ there exists an integer k_0 such that $\forall K > k_0$

$$\Pr_{\mathrm{y} \in \mathbb{Z}_{\mathrm{N}}^{+1}}\left[\boldsymbol{f}_{\boldsymbol{k_0}}(\boldsymbol{N}, \boldsymbol{y}) = \boldsymbol{Q}_{\boldsymbol{N}}(\boldsymbol{y})\right] < \frac{\mathbf{1}}{\mathbf{2}} + \frac{\mathbf{1}}{\boldsymbol{k^c}} \qquad (3)$$

If equation 3 is true and k_0 is large enough, then the probability of differentiating between QR and QNR is not better than guessing, i.e., the server will not be able to decipher the information requested by the user by trying to find out if $y \in QR$ or $y \in QNR$. Readers interested in number theory or a good reference on quadratic residuosity predicate can read [41] and [42] respectively. We ran PIR protocol twice on user request to prevent the server from finding out user requested information and to minimize the amount of information released to the user.

Processing SNNQ

The following steps as shown on the high level architecture of Figure 3 will be followed in the processing of a private snapshot nearest neighbor query.

Step 1: Server in the offline phase maps the grid as described on the neighborhood creation section of this chapter and stores the information in the database.

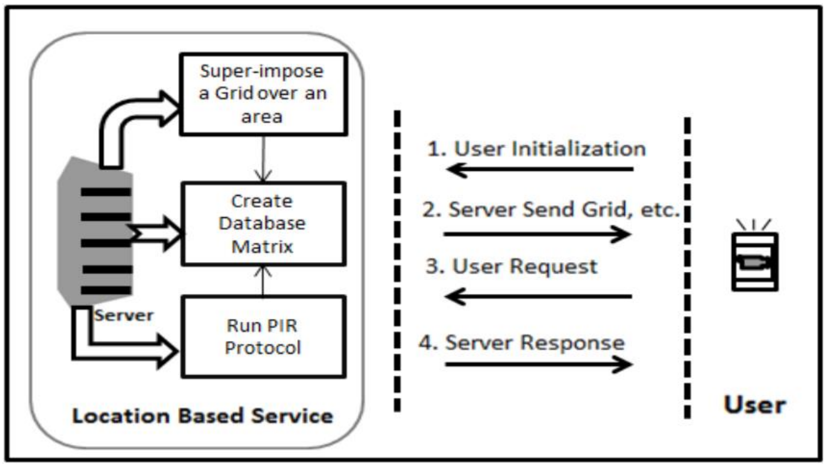


Figure 3 High level system architecture showing interactions between user and LBS

Step 2: When a user initiates a query for the nearest POI, server sends the grid and the POIs information to the user which is actually neighborhood information. User finds the cell it belongs to and focus only on the neighborhood contained in that cell. User then generates her own location data l_u based on its location. For each neighborhood in the user cell user computes with *O(1)* to *O(k)* the absolute difference in the distance $|l_{N_{HD}}(p) - l_u|$ to determine the closest neighborhood to the user location. Remember that the user will not know the exact location of p until server responds with the exact location. Initially, user only have the information of the neighborhood that contains the p.

Step 3: When a user determines the object closest to its location, it sends a cryptic request to the server. For example, for s = $\lceil\sqrt{n}\rceil$ and a database D organized as an *s* x s matrix M let $G_{2,3}$ be the grid that contains the user and let $M_{b,c}$ be the matrix element of interest to the user, user randomly generates modulus N = *q*q'* (equivalent to public key used in asymmetric cryptography) with a query message $y = [y_1, y_2 \cdots, y_s]$ and $x = [x_1, x_2 \cdots, x_s]$ such that $y_c \in QNR$ and $\forall j \neq c\ y_j \in QR$. Also $x_b \in QNR$ and $\forall r \neq b\ x_r \in QR$. User sends a query message PIR(c, b) to the server. Let the matrix column multiplication be represented as

$$z_r = \prod_{j=1}^{s} w_{r,j} \qquad (4)$$

and the row as

$$Z_\alpha = \prod_{r=1}^{s} z_{r(r,j)} \qquad (5).$$

where Z_α is a product of z_r and x_r within the range,

$w_{r,j}$= y_j^2 if $M_{r,j}$ = 0, otherwise $w_{r,j}$ = y_j if $M_{r,j}$ = 1 for all j = 1 ⟶ s. Also, $z_{r(r,j)} = x_r^2$ if $M_{r,j}$= 0, else = $z_{r(r,j)} = x_r$ if $M_{r,j}$= 1 for all r = 1 ⟶ s. Server runs PIR protocol twice on the user request. For every column and row server computes equations 4 and 5 respectively. It returns Z_α to the user with *O(1)* ≪ *O(s)* in [5]. User will then as was previously in [43] determine Z_α value by computing equation 6.

$$(Z_\propto^{\frac{q-1}{2}} = 1 \bmod q_1) \wedge (Z_\propto^{\frac{q'-1}{2}} = \bmod q_1) \qquad (6)$$

It will be easier to compute equation 6 since user knows q and q'. If equation 6 is true, $Z_\alpha \in QR$ else $Z_\alpha \in QNR$. Therefore, $M_{b,c}$ computes to 0 if $Z_\alpha \in QR$, else $M_{b,c}$ computes to 1 if $Z_\alpha \in QNR$. Remember that LBS server is unable to tell if the query encryption $(y, x) \in QNR$ or QR due to the computational intractability of a large prime.

Analysis and Results

In this section, we present the analysis and results for the experimental evaluation of our design. We start by describing the experimental setup. In the experimental study we present the results of the implementation with respect to the execution time. We conduct experiments to study the scalability of the design for database performance. We also analyze the complexity of the design and compare with [5]. We first study *CreateObjectNeighborhood* which we use for mapping objects to a neighborhood.

Note: Evaluation of the PIR cryptosystem is not part of this work since it has been proven to be secure in [27].

Experimental Setup

The algorithm was developed in C and C + + languages. All experiments (client and server side) were conducted on a windows 7 PC running Intel quad-core *i5*-2410M processor at 2.3GHz with 4GB RAM. We used as an input to our mapping algorithm the raw data provided by United States Board on Geographic Names (USBGN) [44]. All coordinates are given in the WGS 84 coordinate reference system. WGS 84 is the latest revision of the World Geodetic System which is used in mapping and navigation including GPS satellite navigation system. We use the map of Boston, Massachusetts an area of United States with geographic coordinate of 42° 21’ 29”N and 71° 03’ 49”W having approximately 48 square miles of land and 41 square miles of water.

For segmentation, we use the neighborhoods of Boston from list of the neighborhoods in Boston [45]. Two datasets were needed in the experiments: points of interests for which we use a set of spatial datasets provided by the City of Boston in [46], and user locations which were randomly generated since user location varies within the coverage area. Figure 4 shows part of the algorithm that updates object location as it is identified. Different sizes of grid *n* and objects *k* were used in our experiment to validate database scalability and performance. Different combinations of *k* and *n* were used to evaluate the design. We use Haversine Formula in [47] to calculate in decimal degree the distance between two points. Haversine Formula does not consider the non-spheroidal (ellipsoidal) shape of the Earth; therefore, we disregard the problem of overestimating trans-polar distances and underestimating trans-equatorial distances.

Initially, object location information is obtained and maintained in a binary search tree (BST), where object location information in node y (node on left sub tree of x) means that x.key>y.key and vice versa for node y (node on right sub tree of x). BST is used by the *UpdateDatabaseMatrix* routine. After initial creation, update of new location information is handled through insertion sort to the appropriate node to preserve the sorted order of the tree.

```
(* Updating object location *)
Input: Number of objects $k$, grid size $n$
Output: objLocation
Procedure $UpdateObjectLocation(k, n)$
1.    (* Initial Update *)
2. while $numbOfUpdates < (k * n)$
3. do $temp = l_p$;
4.   if $temp < min$
5.          do $*objectLoc = temp$;
6.      else  $*objectLoc = min$;
7.          do insertion_sort($objectLoc, l\_p, k$);
```

Figure 4 Algorithm updating objects locations

Execution Time

We conduct series of experiments to determine execution time to create neighborhoods for the objects so that each neighborhood contains no more than one object. Neighborhoods are created in the algorithm shown in Figure 5, and the execution time results are shown in Figure 6. In the first experiment we measure and analyze the execution time to create neighborhoods for the objects in order to show performance impact of various sizes of grid and object.

```
(* Imposing grid and creating object neighborhood *)
**Input:** Object number $k$, grid size $n$, Area $area$
**Output:** Neigborhood of object $N(p)$
**Procedure** $CreateObjectNeighborhood(k, n, A)$
**Procedure** $CreateCells(totalArea)$
1. do compute totalArea=area
2. do compute dimension = (totalArea)/gridSize;
3. do compute cellArea = dimension;
4. do CreateCells(totalArea);(* call recursive function *)
5.    while (totalArea >areaOfCell)
6.        do compute totalArea = totalArea - dimension;
7.        do compute cellArea = cellArea + dimension;
8.        do CreateCells(totalArea);
```

Figure 5 Algorithm creating neighborhoods

It is important to note that execution time will vary slightly based on operating systems allocation of threads to the processor; therefore, for measuring execution time we averaged the time taken over 100 runs. Figure 6 shows the execution time to create neighborhoods over an area mapped into a grid of size 16. As shown in the figure, object size has minimal impact on the computation time. As the number of objects in a cell increases by 4 (from 4 to 32), execution time fluctuates between 139ms and 237ms. The execution time seems to be independent of *k*. We averaged execution time over the range of different *k* values which resulted in 166ms for each *k* for *k* from 4 − 32 for a given *n*. We observe steady increase in execution time as we increase the size of the grid; however, as shown in Figure 6 and Figure 7 (n = 16 and n = 64 respectively) the proportionality of the increase is less than 1. It is interesting to note that since the increase is less than 1 for *n* in the range of 16 to 64, the system will scale well for different populations. The significant of this is that we can adjust *k* and *n* within this range and the execution time will not be far from the average time. Figure 8 shows the execution time in milliseconds for different sizes of grid *n* and relatively small *k* values. It shows that time is also unaffected by *k*. This is very important in a densely populated area. Since *k* has little effect on the execution time, we can appreciate the impact in an area with large concentration of *k* since we can reduce the size of *n* to the smallest possible value and increase the size of *k* in a cell and still achieve desirable result.

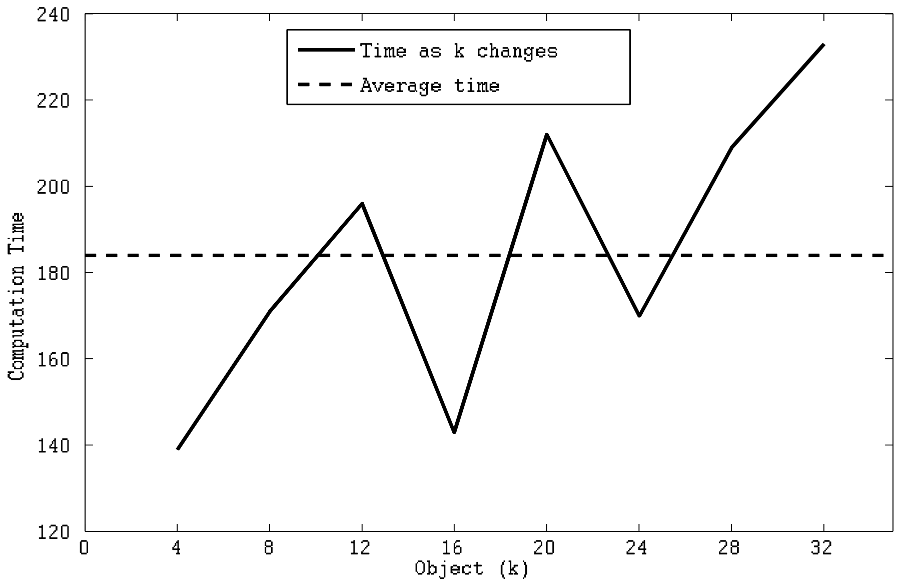


Figure 6 Computation time (ms) to create neighborhoods as k changes over an area with n=16

There are three observations to be deduced from Figure 9. First, when *n* is small, an increase between (1.5 and 2.5)*n* has little impact on execution time. Second, if *n* is large, same increase has more significant change in execution time. Third, for the same *n* the time difference is more noticeable for large *n* than small *n* as *k* increases. Similarly, Figure 10 shows the time for large *k*. These observations play a significant role in mapping an area in order to achieve better performance. For instance, in the first observation, for a small densely populated area with large concentration of *k*, increasing the size of *k* in a cell and reducing the size of *n* will produce less execution time. Increase in *n* within the threshold will result in negligible increase in execution time. In the second and third observations, increasing *n* for a large value of *n* will have significant increase in execution time as *k* increases; however, if the area is large enough to create multiple segments of small *n* where the aggregate will result in significant increase in computation time, then single segment with large *n* which will result in smaller *k* will be more desirable. In the alternative, we can create multiple databases for each segment while still keeping *k* small for a more improved performance.

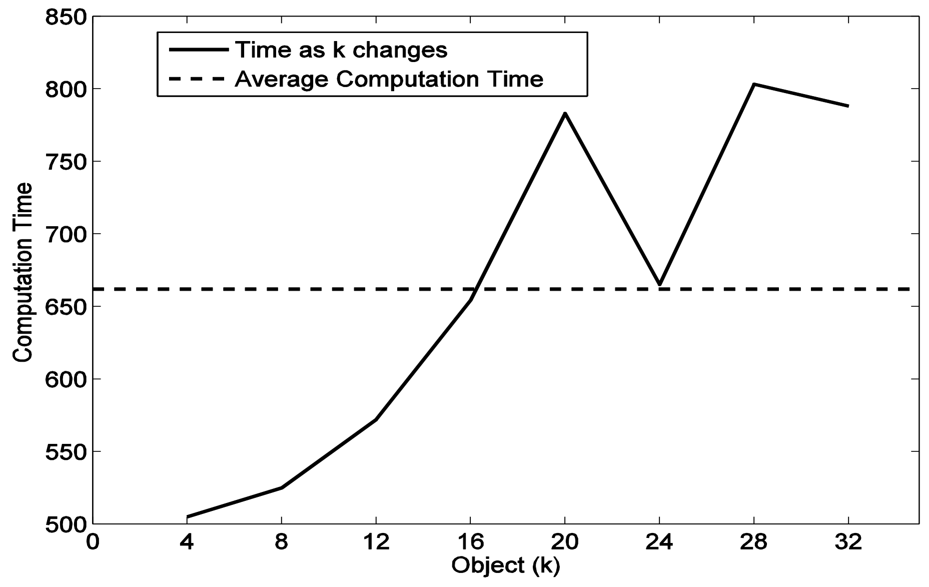


Figure 7 Computation time (ms) to create neighborhoods as k changes over an area with n=64

We also study the performance trend on the execution time for finding nearest POI in the *FindNearestNeighbor* module. The results are presented in Figure 11 and Figure 12 for the different values of *n* when *k* is set to 36 and with user location changing within the grid. Figure 11 shows computation time as user moves from cell to cell. We can see from the result that two things impact execution time: First, the grid size, and second, the cell number user belongs to, i.e., the time for a user to determine its location on the grid. In Figure 12, we vary *k* from 4 to 40. Again, the object size is less significant on the execution time. This also is very important because it provides opportunity for scaling the database in the case of a densely populated area.

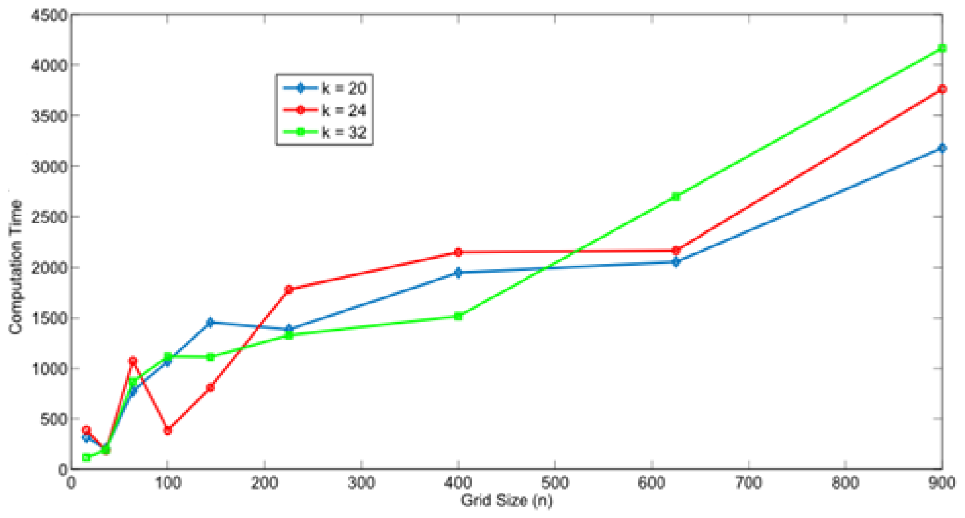


Figure 8 Computation time (ms) for various k =20, 24 and 32 as n gets large

Figure 13 shows the result for a case where user cell *c* remains the same as *n* and *k* varies. We observe a small change in computation time as the range of *n* changes from 16 upwards to 1225, and the ranges for *k* changes from 4 upwards to 40. Similarly, Figure 14 shows the result for a very large increase in *k* (up to 300 in a cell) while the user location changes within the grid and the size of the grid remains constant at16. In Figure 13 and Figure 14, we can see the impact of the size of *n* for users in the same cell. When *n* is small, execution time is reduced compare to large *n*. As the figures also show, significant time is spent by the user to determine the cell it belongs to. However, large *k* also shows significant impact on execution time. Therefore, size of *n* should be balanced with *k* to avoid small *n* that will result in large *k*, or for a very small *k* that will result in large *n*.

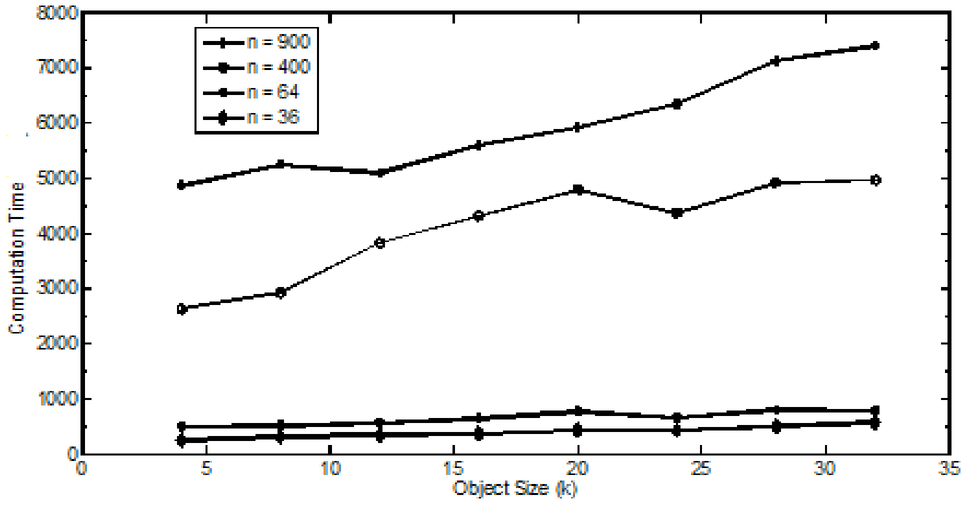


Figure 9 Large changes in *n* shows more significant change in time (ms) than small change in *n*

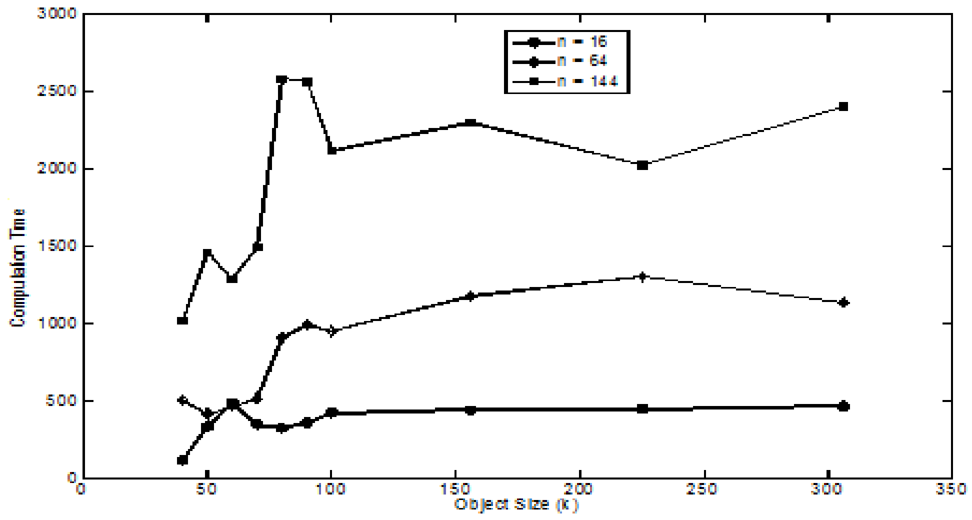


Figure 10 Large k showing very little change in time (ms)

The results in Figure 15 and Figure 16 is for *n* = 16 and *n* = 64 respectively with Figure 15 having a lower *c* number. As shown in in the figures, *k* has impact in execution time. Therefore, for a large area where *k* is sparsely located, creating a big size grid which will result in a few numbers of *n* and similarly small numbers of *k* will be more desirable. The lesson here is to make *n* reasonably small which will result in a relatively small number of cells thereby reducing the time for a user to find the cell it belongs to.

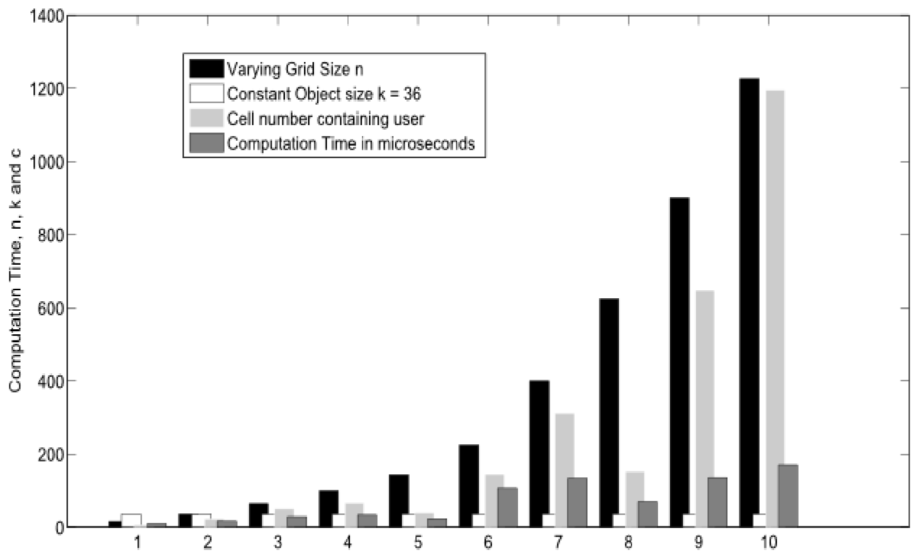


Figure 11 Computation time (ms) impacted only by n and user cell c for k=36

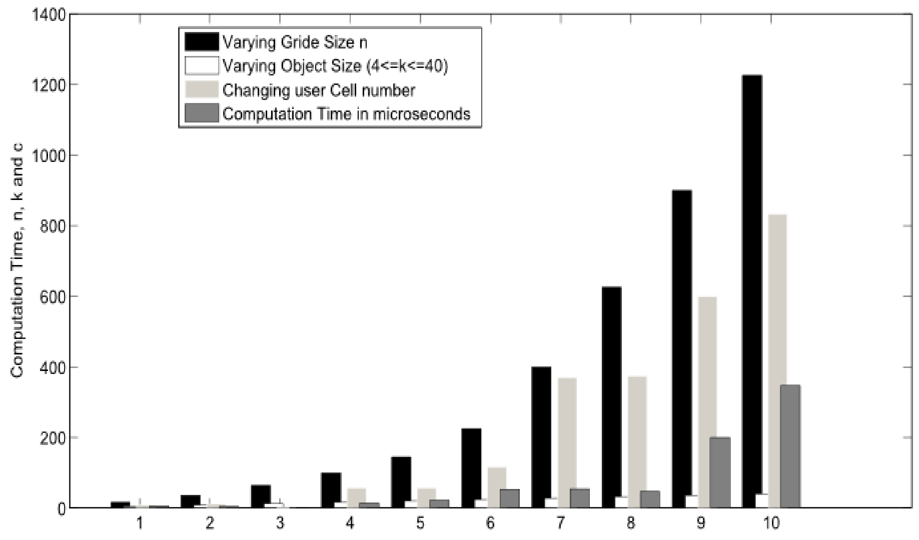


Figure 12 Computation time (ms) impacted only by n and user cell c for varying k (4,8…40)

Scalability

We describe scalability in terms of the ability to scale our design based on the size of the object and the area to achieve desirable performance. This is important in the case of urban area where objects concentrate in a fairly small quarters, or large area where objects are sparsely located. It allows us to scale the area to a few grids with a large concentration of objects in a grid, or create more grids with few objects contained in a grid. The database can then be scaled accordingly for performance. As shown in Figure 9 and Figure 10, during neighborhoods creation computation time remains fairly the same (with few outliers) as *k* increases. Similar observations were made as shown in Figure 15 and Figure 16 when finding nearest neighbor where we observed modest increase in computation time as *k* increases.

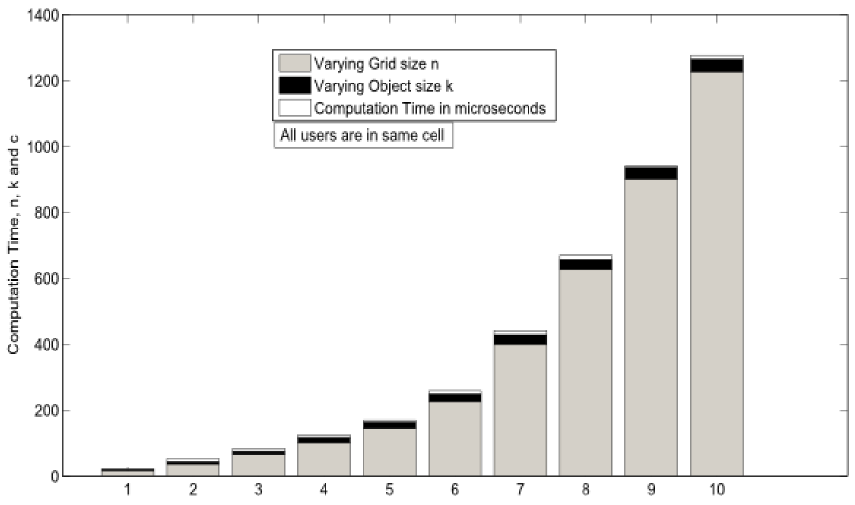


Figure 13 Computing time (ms) when n = (16,36,…1225), and k = (4,8,…40), and constant c.

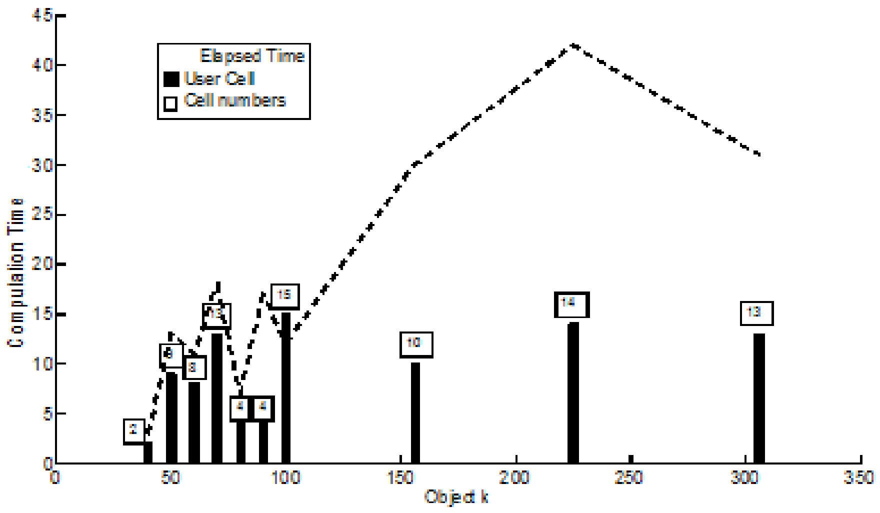


Figure 14 Time (ms) for large variation of k and constant n = 16, and varying c.

Complexity and Comparison

We analyze computation and communication complexity and compare with [5]. The results are shown in Table 1. In the comparison, we consider user side communication for sending PIR and the computation for decrypting server response. We also consider computation and the transmission cost on the server. As already established, the database is organized as a $\sqrt{n}\ x\ \sqrt{n}$ matrix where each matrix element is *m*-bit long.

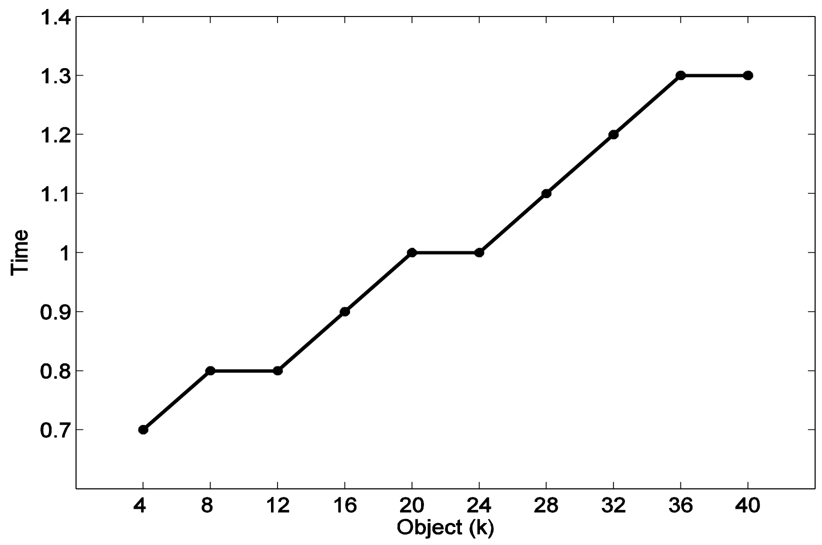


Figure 15 Computation time (seconds) for n=16, constant low c value, and varying k

The encryption modulus is p bits, and f*(p)* is the polynomial function used to denote the computation cost on the value of the p bits. We use double PIR which results in server computation cost of $O(\mathrm{f}(p)m\sqrt{n}) * 2$, which represents the cost for computing $Z = \prod_{r=1}^{s} z_{r(r,j)}$. This is more than the cost in [5]. However, the reduced cost in [5] comes at the expense of revealing and transmitting too many information to a user. Our computation cost for a user to decrypt the server response is reduced to $O(\mathrm{f}(p)m)$ since the double PIR allows the server to transmit only the desired POI. The same operation takes $O(\mathrm{f}(p)m\sqrt{n})$ in [5] because it sends the entire matrix column to the user which requires user to compute through the entire column in order to retrieve a single nearest neighbor. The downside in our design can be seen for very small $\sqrt{n}$ where [5] will perform better. However, that will require $\sqrt{n}$ to be 1 (single-dimensional matrix) which is less likely for the database representation.

Table 1 Computation and Communication Cost Comparison

Cost Type	Our Scheme	Ghinita et.al.
Computation (server)	$O(f(p)m\sqrt{n})2$	$O(f(p)m\sqrt{n})$
Computation (user)	$O(f(p)m)$	$O(f(p)m\sqrt{n})$
Communication (server)	$O(pm)$	$O(pm\sqrt{n})$
Communication (user)	$O(pm) * 2$	$O(pm)$

Server communication cost in our protocol is $O(pm)$, while [5] is $O(pm\sqrt{n})$. The improvement comes from implementing double PIR in the server which results in the server sending only the requested POI against the entire column of POI sent in [5]. For the communication complexity, in our design user has to send two PIR moduli p each for column and row encryption. The cost of communication for this transaction is $O(pm) * 2$ against $O(pm)$ in [5].

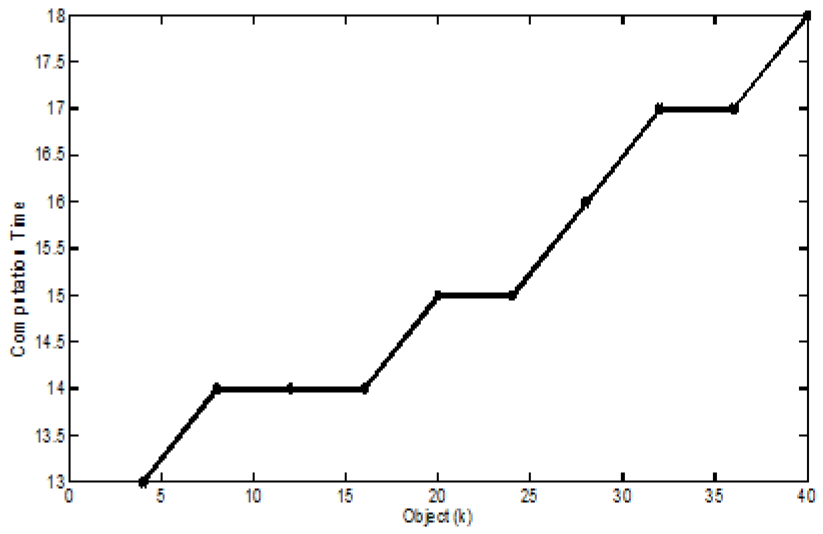


Figure 16 Computation time (seconds) for n=64, constant high c value, and varying k

In general, our design specifically creates neighborhood that satisfies Hausdorff axiom of topological space for each of the POI. Location data is assigned to the neighborhood which is then assigned to the POI contained in the neighborhood. This will prevent the user from deciphering the exact location of the POI from the information sent with the grid. The position of the location data of the neighborhood in the database is aligned such that when a user determines the closest neighborhood, it can match the position with the location of information in the database. The double PIR ensures that only the requested information is released to the user. By adopting the concept of Hausdorff space in our design we were able to divide an area into neighborhoods that contain a single POI. The design allows us to scale the database to the size of the neighborhood in the area so that a user can relate its location to information in the database thereby allowing the server to run PIR protocol on the information in the database and returns to the user only the desired information. Compare to design of [5], server sends the entire grid to the user with information of the POIs. User searches the entire data to find the POI with the shortest distance to its location. This technique exposes too many of the database information to the user, involves increase in the transmission cost, and increases the computation cost on the user.

Double PIR protocol used in our design is to minimize the amount of data revealed by the server as seen in [5] and to reduce the communication cost for transferring the data back to the user. User will only have information on the nearest object and no other object. Nonetheless, PIR has some obvious challenges, it has been criticized for been too costly to implement. For instance, large modulus is preferred for enhanced security, but as the size of the modulus increases so is the cost both in time and complexity. The increase in the object and size of the grid also leads to increase in the size of the database which increases the cost of the implementation. In later chapters, we will explore parallel computing and hardware approach to address these issues.

CHAPTER FOUR: PARALLEL COMPUTING IN SNNQ

We first design CPU serial implementation of the snapshot nearest neighbor query. Due to the limited resources and computational limitation of the CPU the response time increases as the size of the grid and the object increases. The increase in response time will affect quality of service in LBS. We address this issue by utilizing GPU and CUDA API to break the algorithm into smaller parts which are computed simultaneously in order to reduce the total computation time. GPU has been gaining lots of traction in the research community mainly due to its inherent parallel design. GPU is a device present in most modern PCs. They are designed for high speed graphics which are inherently parallel. It is made up of array of streaming multiprocessors which currently have cores up to 48 for Fermi, and 192 for Kepler architectures [48]. This allows scaling of the processor. Figure 17 is a sample GPU architecture depicting the host and device memory where CPU and GPU codes respectively execute.

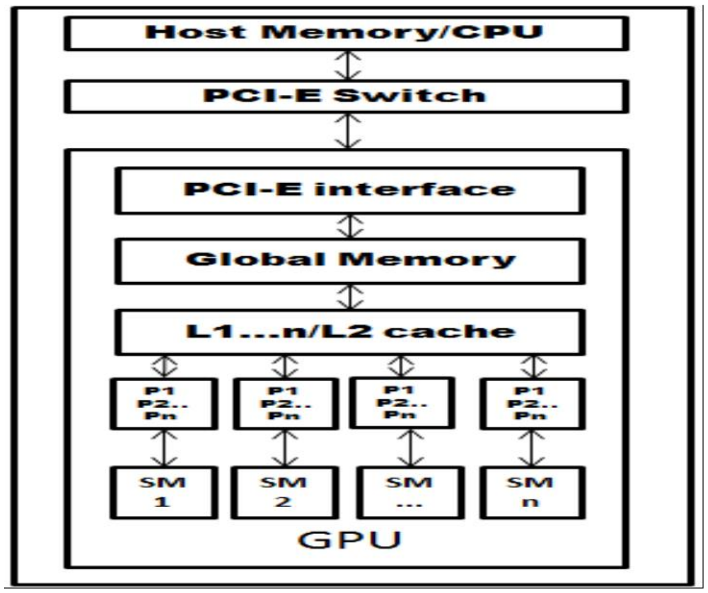


Figure 17 Sample GPU architecture showing streaming multiprocessors and the scalar processors

The $\mathrm{P}_1, \mathrm{P}_2 \cdots, \mathrm{P}_n$ in the figure represent single instruction scalar processor (SP) inside the SMs. Each SM is equipped with its own L1 cache, and the L1 caches are connected to an L2 cache.

The challenging part of GPU computing is to determine where and what to parallelize. Gieseke et al in [49] propose using GPU to implement nearest neighbor search on a refined classical *k*-d tree called buffer *k-d* tree. Proposed in [50] is a CUDA based parallel implementation of nearest neighbor search using GPU which achieved a speedup over serial implementation of the same algorithm. Silvestri et al in [51] exploited GPU in a continuous range queries and location update to achieve a 20x speedup over serial computation of the same dataset. They utilized CPU/GPU in all the computation phase to preprocess the data set such that objects in the same index grid cell are stored consecutively to optimize memory access. They also produce and transfer in parallel the list of object ids for each issued query. Though effective, the method that was used to apply parallelism to the cells may lead to load imbalance. Our design of single object neighborhoods helps to create data independent which is crucial for parallelization. The CPU computes the serial part of the algorithm. Using CUDA API, CPU code spurns thread the size of the total independent computation needed to perform the operations on the GPU. The threads run concurrently. At the completion of the operation, the data is transferred back to the CPU.

Not every application is suitable for parallel computing. Applications that qualify for parallel processing should have scalability, speedup, and efficiency. The amount of speedup that can be achieved in application is directly related to percentage of the program that can be parallelized. Amdahl's law in [52] states that a small portion of a program which cannot be parallelized will limit the overall speedup available from parallelization. If β represents the fraction of a running time a program spends on non-parallelizable parts and N is the number of processors; then,

$$\lim_{N\to\infty} \frac{1}{\frac{1-\beta}{N} + \beta} = \frac{1}{\beta}$$

represents the maximum speed-up that can be achieved by parallelizing the program. If P is the proportion of a program that can be parallelized, $(1 - P) = \beta$ represent the fraction of running time spent on non-parallelizable parts. Maximum speedup that can be achieved by using N processors can also be represented as

$$S(N) = \lim_{N\to\infty} \frac{1}{(1-P) + \frac{P}{N}} = \frac{1}{(1-P)}$$

This is how much faster the computation runs on parallel hardware.

Graphical Processing Unit

According to [53], CPU will idle if there are not enough processes to keep it busy. If there are too many small tasks with each blocking after a short period, the CPU will spend most of its time context switching and very little time doing useful work. With CPU, scheduling policies are often time multiplexing (dividing the time equally among the threads). The percentage of time spent context switching is directly proportional to the number of threads; therefore, large increase in number of threads decreases efficiency. In contrast, GPU requires thousands of threads to work efficiently. GPU is designed to handle stall conditions which are expected to happen with high frequency. With available pool of work GPU always has something useful to work on. When a GPU hits a memory fetch operation or has to wait on the result of a calculation, the streaming processors will switch to another instruction stream and return to the stalled instruction stream sometime later.

For the most part, CPU run single thread programs and are mostly dual or quad core devices. It calculates a single data point per core per iteration. GPU; however, runs in parallel by default. Instead of calculating just a single data point per SM, GPU calculates 32 data points per SM. This provides a 4x advantage in terms of number of cores over a typical quad core CPU, and also a 32x advantage in terms of data throughput. GPU provides high speed memory (shared memory) next to the SM which allows the programmer to safely leave data in this memory knowing that the data will not be destroyed by the hardware. It also serves as the primary communication mechanism among threads. Cook in [53] stated that there are two major differences in the task execution model of a GPU. The first is that groups of *N* SPs execute in a lock-step basis running the same program but on different data. The second is that because of the GPU huge register file switching threads have effectively zero overhead. GPU can support a very large number of threads and is designed in this way. Over the years, GPU has steadily outperformed CPU [54].

Compute Unified Device Architecture

CUDA is an extension of the C language that allows GPU code to be written in regular C. The code is written for the host processor (CPU) or the device processor (GPU). The host processor spawns kernels or multithreads tasks onto the GPU device. The GPU through its internal scheduler allocates the kernels to the GPU hardware. Cook in [53] explained that CUDA is ideal for problems that are extensively parallel, and that it can be used if a problem can be constructed and the output data points represented without relation to each other. This is ideal for our protocol where the output data are independent. CUDA splits problems into grids of blocks, each of the blocks contain multiple threads. The blocks can run in any order, however, only a limited number of blocks will execute at any point in time determined by the resource requirement of a thread block. Blocks execute end to end in any one of the SMs. It is normally done in a round-robin basis to allow equity distribution of blocks among the SMs. In its basic form, a grid made up of threads is split into blocks, and the blocks are split into warps. To perform action, the kernel provides some action and data, each thread then works on its individual part of the problem. Threads can swap data with one another under the coordination of either the warp or the block, but any thread coordination with the blocks can only be performed by the kernel.

CUDA program runs on the CPU and transfers needed data to the GPU, the GPU performs execution and returns the data to the CPU. The process continues until the application terminates. Based on this process, the total execution time Tex_{time} can be broken into parts as follows:

$$Tex_{time} = \beta + t_{HD} + \alpha + t_{DH} \tag{7}$$

The β in equation 7 represents the serial portion of program executed by the CPU. The parallel part executed by the GPU is represented by α, while t_{HD} is the time required to transfer the data from the host to the device, and t_{DH} represents the time needed to transfer the data from the device back to the host. Equation 7 seems contradictory to improving speedup when compared to

β, however, significant reduction in β accounts for the improvement. Note that equation 7 is correct if there are no overlaps between data transfer and kernel execution.

Task Model

Ideal algorithm for a GPU should be able to have enough parallelization that will significantly account for t_{HD} and t_{DH}. The algorithm in Figure 18 for the CPU-based code loops through the *gridSize* and the *objectSize* and creates and assigns objects to neighborhoods within a cell *c* contained in the grid *G*. The struct of each object has array called *objectNeighborhood[ ]* which maintains a list of all objects of a given neighborhood. The loop through the *cellNumber.size()* and *objectSize* identifies each neighborhood with their object and their cell number. For the GPU-based implementation, *computecellArea* and *CreateCells* are still present and execute on the CPU, the loops are performed by the GPU by first allocating memory and copying over the data using *cudaMalloc* and *cudaMemcpy* respectively.

Implementation

We create a sorted linear binary search tree (BST) that holds object locations. The *crtObjNHood* module creates the neighborhoods by computing the difference between the positions of two adjacent objects such that the spatial distance from object p_i to the edge of the neighborhood is $\frac{|l_p[i]-l_p[i+1]|}{2}$, i.e.,

$$\mathrm{l}_{\mathrm{N}_{HD}}(j) \equiv l_p(i) + \frac{|obj(l_p[i]-l_p[i+1])|}{2} \qquad (8)$$

For object *i* the location value $l_p(i)$ is $l_{N_{HD}}(j)$. The piece of code in Figure 18 runs on the CPU. Line number 11 launches CUDA kernel function that executes on GPU. The CUDA kernel in the algorithm of Figure 19 stores the value of the neighborhood which is assigned to the object for a given index in the variable $l_{N_{HD}}[j]$ for that same index. The "*for"* loop iterates *"tid"* times (index 1 to *tid*). This is translated to *"size"* threads in CUDA. Each thread executes the line

$$l_{N_{HD}}[j] = l_p[i] + \frac{|obj(l_p[i] - l_p[i+1])|}{2}$$

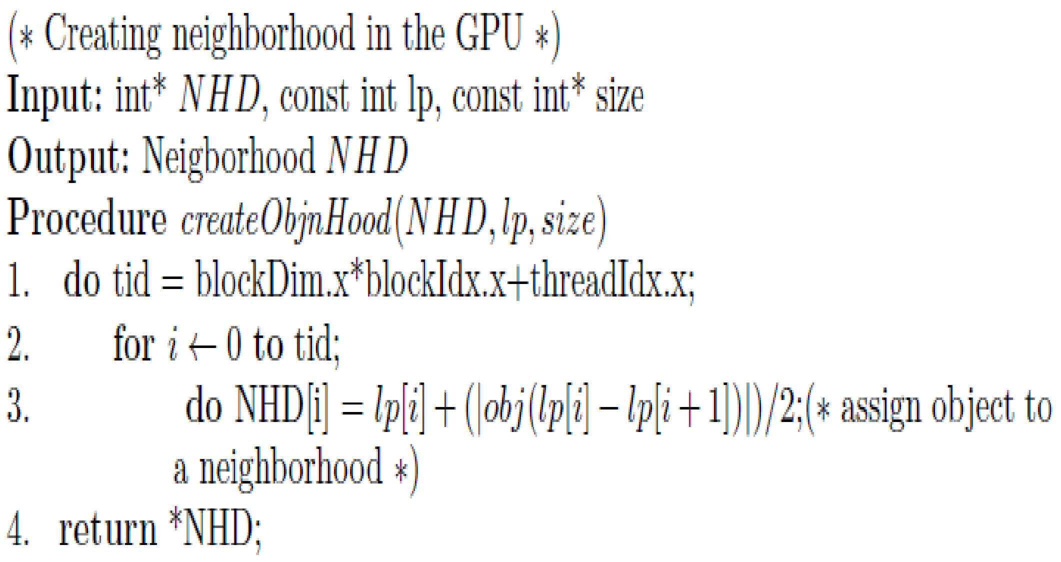
This is made possible by the design of the algorithm which creates no dependency between the iterations. The *_global* prefix tells the compiler to generate a GPU code rather than CPU during compilation. The GPU function is called with *"crtObjnHood<<<blocksPerGrid,threadsPerBlock>>>($l_{N_{HD}}$,l_p, SIZE)".* The "*blocksPerGrid"* parameter tells the GPU the number of blocks to be allocated for the thread execution, and "*threadsPerBlock"* tells the GPU how many threads in a block. Line 11 of algorithm of Figure 18 calls the GPU "*crtObjNHood"* module *"SIZE"* times and each call is made with a different thread. Each thread does four reads from memory, one subtraction, two additions, and one store operation, and then terminates.

```
(* Launching CUDA kernel function *)
Input: Object size $k$, grid size $n$, area $A$
Output: Neighborhood value $NHD$
Procedure crtObjnHood($k, n, A$)
1.  do define  SIZE=k*n;
2.  do define  SIZE_IN_BYTES(sizeof(unsigned  int)*(SIZE));
3.  do cpu_NHD=(*float)malloc(SIZE_IN_BYTES);
4.  do cpu_lp=(*float)malloc(SIZE_IN_BYTES);
5.  do cudaMemAlloc(&NHD,SIZE_IN_BYTES);
6.  do cudaMemAlloc(&lp,SIZE_IN_BYTES);
7.  do cudaMemcpy(NHD,cpu_NHD,SIZE_IN_BYTES,cudaMemcpyHostToDevice);
8.  do cudaMemcpy(lp,cpu_lp,SIZE_IN_BYTES,cudaMemcpyHostToDevice);
9.  do int  threadsPerBlock=512;
10. do int  blocksPerGrid=(SIZE+threadsPerBlock-1)/threadsPerBlock;
11. do createObjnHood<<<blocksPerGrid,threadsPerBlock>>>(NHD,lp,SIZE);
```

Figure 18 Code that runs on the CPU and launches kernel function

Analysis and Results

We provide analysis for the GPU parallel implementation of the SNNQ and compare the results with the CPU implementation. We did not start to define the core of our performance model in order to predict the GPU execution time. Our goal is to conduct statistical analysis based on performing variety of simulations and compare it with CPU execution time, and then define the performance model based on actual experiment. The testing computer is Aliewnware X51 R2 running windows 7 operating system and equipped with Intel core *i7-4770* processor with 12G RAM. Available in the system is NVIDIA GeForce GTX 760 Ti with 2G GDDR5. Table 2 shows the rest of the GPU specification. The threads are grouped into 32 thread groups called warp. Each warp is placed into the SM to be executed by the number of the SPs available in the SM. Understanding GPU architecture is important to avoid blocks that does not make full use of the hardware which could lead to drop in performance [53]. The values of the data sets used in our simulation were chosen to maximize performance. For the most part, the number of threads per block was based on achieving 100% utilization factor of a compute capability 3.0 GPU as prescribed in [53], and to maximize the number of threads to reduce the effect of memory latency.



```
(* Creating neighborhood in the GPU *)
Input: int* $NHD$, const int lp, const int* size
Output: Neigborhood $NHD$
Procedure $createObjnHood(NHD, lp, size)$
1.  do tid = blockDim.x*blockIdx.x+threadIdx.x;
2.      for $i \leftarrow 0$ to tid;
3.          do NHD[i] = $lp[i] + (|obj(lp[i] - lp[i+1])|)/2$;(* assign object to
            a neighborhood *)
4.  return *NHD;
```

Figure 19 Piece of code that runs on GPU to create neighborhoods

We create blocks with thread counts that are a round multiple of the warp size in order to have enough active warps in each streaming multiprocessor to sufficiently hide all of the different memory and instruction pipeline latency of the architecture in order to achieve maximum throughput. For instance, we create 512 threads (32 threads per warp) which results in 16 warps so that we can schedule up to 64 warps (32 x 64) for a total of 2,048 threads at a time in a single SM. For the most part, we chose our thread size to ensure that we have the maximum number of threads running in the SM. The size of the thread is also intended to minimize number of blocks with the hope of reducing the time for GPU to add and remove each block from the SM. As previously stated, the grid size in our protocol has to be a square. This presents some challenges when processing data which are not multiple of the kernel launch parameter. We use padding to fill those that are less than warp size. For instance, 8,100 threads were needed to be processed in one instance; however, this does not equate to a perfect multiple of a warp, we therefore padded it with 92 extra threads to make it perfect multiple of a warp. Our goal is also to make use of all the available SMs in every execution. For example, we have 5,376 threads to compute, we split it into 384 threads per block and then we have a total of 14 blocks to be allocated to the SMs. This will require each SM to have two complete blocks and no left over block. Since all the SMs are running the same kernel, it is likely their execution time will be the same. All the SMs will be available at the same time. Assumption here is that the allocation is done on a round-robin basis since NVIDIA does not disclose the methods of scheduling used. Also, in some cases we create large number of blocks to reduce the size of threads in a block when the blocks are not multiple of SMs in order to avoid the time delay for the leftover block to execute due to large kernels.

Table 2 GPU System Specification

Unit	Specification
CUDA Cores	1344
Stream Multiprocessors	7
Maximum Thread per	1024
Maximum Thread per SM	2048
Graphic Clocks (MHz)	915
Processor Clock (MHz)	980
Memory Clock	6.0Gbps
Memory Interface	GDDR5
Standard Memory Config	2G
Memory Interface	256-bit
Memory Bandwidth	192.2
Compute Capability	3.0

Comparing CPU and GPU Performance

We show here the results of the simulation we ran as the size of the database increases and compare the differences in performance between GPU and CPU with respect to run time. We studied the impact of parameter tuning for making CUDA calls to examine the effect on GPU execution time, and we provide analysis for deciding the right thread and block size. We then provide a model for predicting GPU execution time. Figure 20 and Figure 21 depicts computation time between CPU and GPU. It shows the result for the serial computation performed in two different computers with different specifications. The first computer was only used for CPU serial execution since it does not have CUDA-enabled GPU. We performed both CPU serial and GPU parallel computation using the second computer. The results show that

GPU significantly reduces the response time as the size of the database increases. GPU shows on the average an improvement factor of 23.9 over CPU using the same hardware.

Results of our experiment show that creating a small size threads per block for a maximum occupancy of the SM performed better when the size of the objects were small. The same observation was made for small size threads per block with 94% SM occupancy. However, the results also show that when the size of the objects are large, better performance will be achieved by creating large size threads per block that result in a 100% utilization factor. This is likely due to global memory latencies from GPU adding and removing each block from the SM. In other words, small thread size in a block performs better when the size of the objects are small, while large thread size in a block performs better when the size of the objects are large.

Some challenges were observed in our GPU implementation. Dealing with large data sets is one of them. As the size of the grid increases, so does the size of the database matrix and the modulus used for the encryption. These data which are stored in the system RAM are transferred to the GPU. The data easily exceeds the size of the system RAM causing data to be retrieved from the disk before sending to the GPU thereby causing transfer delays. Another challenge is the database structure which requires square matrix that may not always map to a perfect multiple of warp thereby requiring extra padding in the kernel launch parameter in order to create a thread that is multiple of a warp.

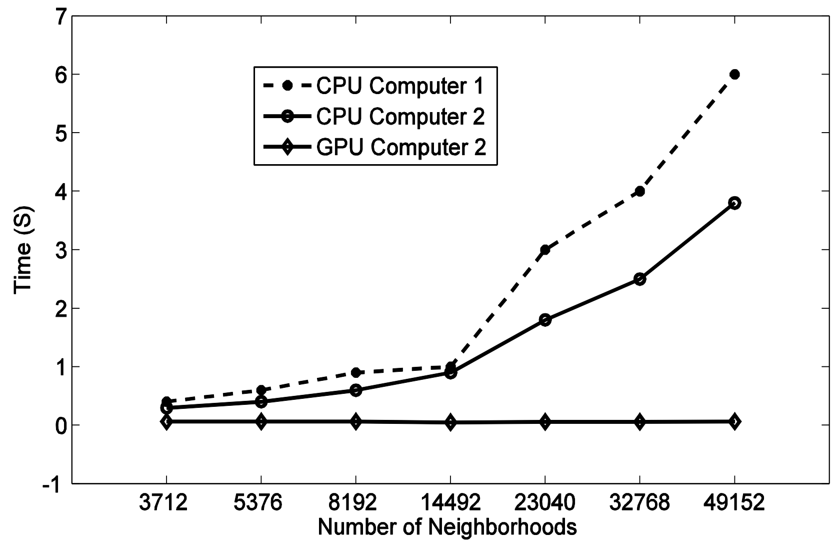


Figure 20 Time difference between GPU/CPU for thread size of 1024

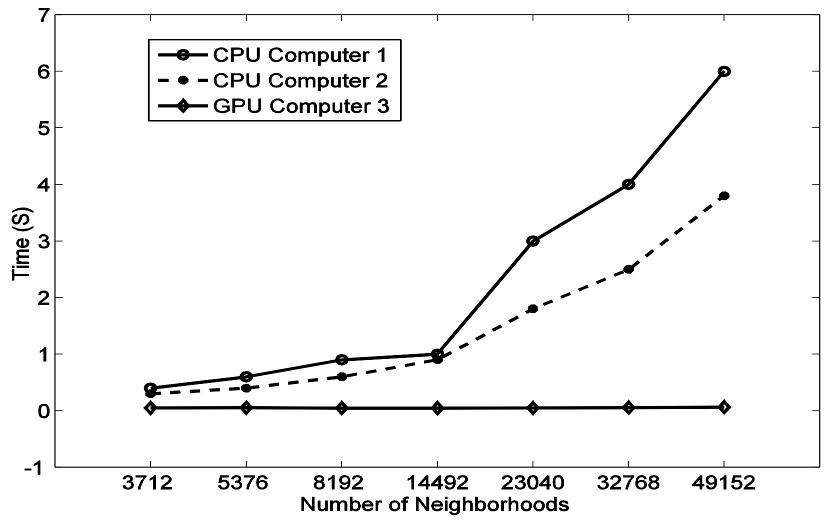


Figure 21 Time difference between GPU/CPU for thread size of 384

Modelling GPU Run-Time

Some work has been performed on GPU run time modelling. The work provided in [54] modelled execution time in multiple GPUs for algorithms whose GPU work scales linearly as the number of mobile objects increases. Other GPU modeling like the one in [55] modelled existing algorithm not similar to our work. Our goal is to create a model specifically for our algorithm and similar algorithms in order to provide a guide for determining the size of the parameters to achieve optimal result without knowledge of the intricacies of the GPU architecture.

We set to predict the execution time based on the size of the parameter use in the CUDA calls. We start by developing run time equation for an object using the run time for a block and for each additional block. We extrapolate the run time for a problem set based on the equation and then determine query execution time for a single object. This would be the time to create one neighborhood. To find query execution time for a single object, we first find out block execution time T_{BExe}. This is done by running several operations. We determine the block processing time K_t for adding and removing blocks to the SMs by running operations first with a single block and then by adding blocks to the SMs. The execution time T_{Exe} for a single object is then represented as follows:

$$T_{Exe} = \frac{T_{BExe}}{Thd_{Size}} \tag{9}$$

T_{BExe} is the execution time for a single block of the given problem, and Thd_{Size} is the size of thread in a block. From equation 9, we can extrapolate the GPU execution time T_{GPU} for a given problem. Let the number of blocks be B_{Num}, let P_s be the problem set, where $P_s = Thd_{Size} * B_{Num}$. If N_{SM} is the number of SMs contained in the GPU,

$$T_{GPU}(P_s) = T_{Exe} * Thd_{Size} * \left\lfloor \frac{B_{Num}}{N_{SM}} \right\rfloor + K_t * B_{Num} + T_{Exe} * Thd_{Size} \tag{10}$$

Equation 10 represents the total time a given problem will take to execute on the GPU with N_{SM} SMs. Note that T_{Exe} will be the same for each object since the same thread is being executed. Also, the last part of equation 10 ($T_{Exe} * Thd_{Size}$) is the time to execute the last block in the problem set. From equation 10 we can see the importance of creating blocks that are multiple of the SMs, because the whole operation waits for the last block to complete. We can also see that the optimal Thd_{Size} and B_{Num} depends on T_{Exe} (which depends on the complexity of the operation) and K_t. As shown in Table 3, this model has shown to be quite accurate for different object sizes. The large number of objects used in our model allows us to provide a precise average run time for the objects thereby giving a highly accurate approximation of the expected run time. It is important to note that this model will accurately predict GPU run time for applications which GPU execution scales similarly to ours.

Table 3 Run time (ms) for different thread sizes in a block

Total Object	1024/Blk	768/Blk	512/Blk	384/Blk	256/Blk
100,000	64.302	64.473	64.350	64.310	64.311
200,000	69.772	69.710	69.663	69.662	69.798
300,000	78.445	78.522	78.455	78.522	78.455
400,000	80.972	80.920	80.872	81.006	80.915
1,000,000	98.202	98.227	98.152	98.189	98.177
2,000,000	128.003	127.987	127.971	127.962	127.954
4,000,000	199.522	199.510	194.522	199.509	199.497

CHAPTER FIVE: HARDWARE ASSIST PRIVATE INFORMATION PROTECTION IN LBS-SNNQ

Alternatively, tamper-resistant hardware can be used to protect private information in LBS-SNNQ. Tamper-resistant hardware computing technology with secure coprocessor allows usage of onboard chip or microprocessor to carry out cryptographic operation thereby eliminating the need for protecting rest of the sub-system. In general, secure coprocessor refers to a physically secure subsystem of a larger host computer. The idea is for the coprocessor to provide a secure means of computation and data storage against an adversary even when they have access to the host system. Smartcards and Trusted Platform Module (TPM) are some of the ways that secure coprocessor can be deployed. In the design proposed in [56], [57], and [58] only the coprocessor is considered secure while the rest of the system including operating system (OS) is considered vulnerable to software attack. Other application of secure coprocessor technology is in cloud computing where network of remote servers hosted over the internet are used to store and process data. In this work, we propose to use secure coprocessor to protect user private information in LBS-SNNQ. This will help to minimize the amount of information transferred while still allowing a user to obtain the nearest POI to its location without adversary having the ability to decipher user location or the information obtained. This is achieved through encryption of transmitted information, obfuscated instruction execution using hardware architecture proven to be secure, and engaging in obscure access to a remote storage device. A user sends encrypted query to a secure coprocessor. The secure coprocessor processes user information and engages in a periodic and coordinated access to a remote server to obtain user requested information using ORAM methodology. It is important to note that using a secure coprocessor is different from using a trusted third party. In the remainder of this chapter, we explain the secure coprocessor and the tools used to implement our design. We provide experimental analysis and compare with the software approach.

Secure Coprocessor

A secure coprocessor contains CPU, bootstrap ROM, and secure non-volatile memory which cannot be physicality penetrated [59]. It is designed to meet rigorous security requirements that assure unobservable and unmolested running of the code residing in it even in the physical presence of an adversary [7]. The Input-Output (I/O) interface to the module is the only way to gain access to the internal state [59]. The device is equipped with hardware cryptographic accelerators that enable efficient and fast implementation of cryptographic algorithms such as DES and RSA [35]. Secure coprocessor [7] provides protection for code to execute and carry high-speed encryption process. Crucial for its operations are: resource management, communication, key management, and encryption services. A secure coprocessor is designed such that upon a break-in the intruder cannot learn or change the internal state except through normal I/O channels or by forcibly resetting the entire secure coprocessor [59].

Nonetheless, the security of a coprocessor can be breached through physical or software attack. In a physical attack the adversary have access to the hardware [33] and can dictate problem behavior by tampering with the hardware pins through the means in [60]. Software attack involves malicious use of software to remotely launch attack without physical access to the hardware such as the cache attack addressed in [61], [62], and [63]. In this work, we consider attack by an adversary from a remote location. The coprocessor considered meets the requirement for oblivious computation which includes: (1) same amount of work and time will be spent for each operation (i.e., all operations including saving information in the internal memory and discarding information cannot be differentiated). (2) Each user thread owns the coprocessor the entire time, therefore, no other thread will be able to access information been computed for another user. (3) Coprocessor actions and operations for external request must be obfuscated. The coprocessor accesses the server through ORAM methodology.

Oblivious Random Access Memory

ORAM is defined in [64] as a probabilistic random access memory (RAM) which has memory access distribution pattern independent of the input. It allows a client to obscure its access pattern to a remote storage server through continuous reshuffle and re-encrypt of the information stored in the remote server. It prevents any leakage of information that could assist adversary to determine the type of memory access (read or write), location accessed, or what data is read or written by the client. Different flavors of ORAM have been proposed, like the ones in [64], [65], [66], [67], [68], [69], [70], [36] and used in [71] to build a secure coprocessor architecture called Ascend which guarantees privacy for data stored in a cloud computing environment. ORAM model defined in [66] consists of a basic RAM decoupled into two interactive machines (CPU and memory module) which acted as Interactive Turing-Machine (ITM). The read-only communication tape of the CPU is associated with the write-only communication tape of the memory, and vice versa. The ORAM algorithm is such that an adversary can observe the physical storage locations accessed, but, it ensures that the adversary has negligible probability of learning anything about the true access pattern. Another flavor of

ORAM discussed in this chapter is known as Path ORAM. It is designed for optimization by focusing on a path rather than the entire database information.

Path-ORAM

To obfuscate the exact information accessed, most ORAM techniques are based on streaming the entire database. For our work, we implement Path ORAM of [36] to reduce the computational burden of streaming the entire database each time query is issued thereby reducing the access time. The design is based on security principles defined in [68]. It requires that no information be leaked about which data is being accessed, the last time data was accessed, if the same data has been accessed, access pattern (random, sequential, round-robin etc.), or whether the access is read or write. It is implemented with a binary tree data structure of height H and 2^H leaves. Data is stored in the tree node called bucket. Each bucket contains up to Z blocks, and the buckets are padded if necessary to always maintain block of size Z.

The sequence of data request as described in [68] is as follows:

$$\vec{x} := ((op_M, \alpha_M, data_M), \cdots, (op_1, \alpha_1, data_1)) \quad (11)$$

M is the length of data request sequence, and each op_i represents operations $read(\alpha_i)$ or a $write(\alpha_i, data)$. In other words, α_i is the identifier of the block been read or written, while $data_i$ is the data been written. The first index represents the most recent load or store data, and the last index M represents the oldest load or store operation. Given a sequence of data request $\vec{x}$ such that $A(\vec{x})$ is the random sequence of access to the remote storage place, secure ORAM is defined such that any two data request sequences $\vec{x}$ and $\vec{y}$ with access pattern $A(\vec{x})$ and $A(\vec{y})$ cannot be computationally distinguished by anyone except the client. The construction is correct if given an input $\vec{x}$ ORAM returns with a probability $\geq 1 - negl(|\vec{x}|)$ data which is consistent with $\vec{x}$. The data path is defined such that $i \in \{0,1,\cdots,2^H - 1\}$ represents i-th leaf node, where H is the height of the binary tree, and $i = location[\alpha]$ represents the block α currently associated with leaf node *i*. Leaf node *i* defines a unique path from leaf *i* to the root, and $\rho(i)$ denote sets of buckets along the path from *i* to the root. Also $\rho(i,h)$ represents the bucket in $\rho(i)$ at height *h* in the tree.

The path ORAM client maintains two data structures: stash *S*, and location map. The stash is used locally by the client to store a small number of overflow data blocks from the server. The client also stores location map where $i = location[\alpha]$ corresponds to block α currently mapped to i-th leaf node, i.e., block α resides in some bucket in the path $\rho(i)$ or in the client stash. During initialization (client stash is empty) server encrypts some dummy blocks and randomly assigns it to the buckets. The location map for the client contains random numbers between 0 and $2^H - 1$. The read and write processes are as follows:

$read \leftarrow Access(read, \alpha, none)$ from block α, and $Access(write, \alpha, data)$ to block α. The position of block α is remapped to a new random position. For a read operation if *i* is the old position for the block, then, read path $\rho(i)$ that contains block α. For a write operation update the data for block α, write back the path and any additional block from the stash. Block $\alpha'$is placed in a bucket at *h* if $\rho(i,h) \cap \rho(location[\alpha'],h)$.

Note that all the *Z* blocks are read from the bucket during a read operation and decrypted. For a write operation the encrypted blocks are written back into the bucket and padded if necessary to create size *Z* block. Path ORAM security as stated in [36] is such that if $\vec{x}$ is the sequence used for requesting size *M* data, the sequence of access $A(\vec{x})$ as seen by server is

$$P = location_M[\alpha_M], location_{M-1}[\alpha_{M-1}], \cdots, location_1[\alpha_1]) \quad (12)$$

and $location_j[\alpha_j]$ represents the address location α_j for the position map of the j-th load or store operation including paths $\rho\left(location_j\left(\alpha_j\right)\right) \forall j, 1 \leq j \leq M$ encrypted using random encryption. Each $location_i(\alpha_i)$ is randomly remapped after it is revealed to the server; therefore, $location_{i(\alpha_i)}$ is statistically independent of $location_j\left(\alpha_j\right)$ for $i < j \leq M \; and \; \left(\alpha_j\right) = (\alpha_i)$. Also, since address locations are independent of one another; hence, for $i < j \leq M \; and \; \left(\alpha_j\right) \neq (\alpha_i)$ $location_{i(\alpha_i)}$ is independent of $location_j\left(\alpha_j\right)$. Therefore,

$$\Pr(P) = \prod_{j=1}^{M} \Pr(location_j\left(\alpha_j\right)) = (\frac{1}{2^H})^M \quad (13)$$

If equation 13 is true, then, $A(\vec{x})$ cannot be computationally distinguished from a random access $A(\vec{y})$.

System Architecture

Our model is based on a secure coprocessor accessing external memory under the control of LBS server using path ORAM. Figure 22 depicts system high level interaction between user, secure coprocessor, and the LBS server. The problem solved in this model is the same as previously stated in software cryptography, i.e., protecting user private information from been disclosed to another party in LBS snapshot nearest neighbor query. Our technique is based on coprocessor technology that supports computation of arbitrary programs. The coprocessor serves as a client to the LBS server through path ORAM methodology access. The coprocessor can be located inside or external of a server. Our model consists of a single server with coprocessors.

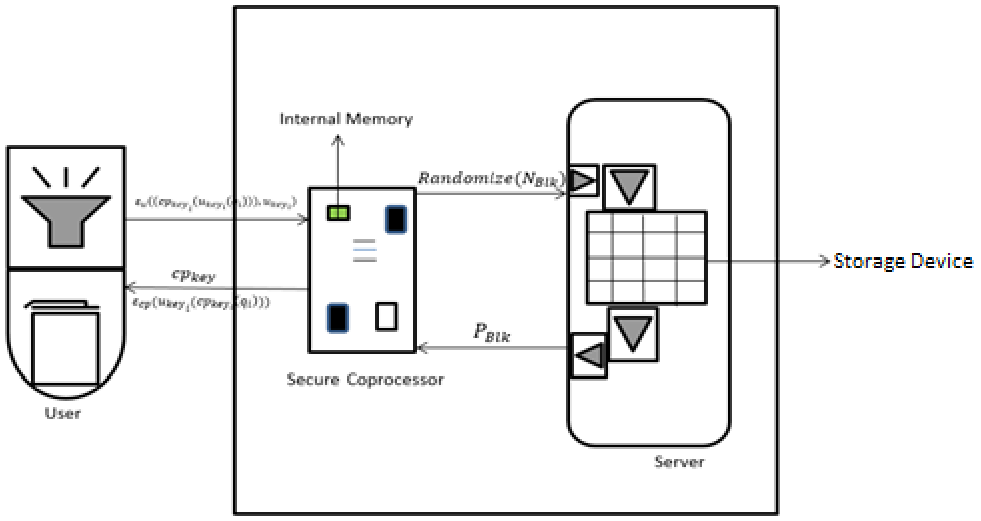


Figure 22 High level architecture of the interaction between user, secure coprocessor, and the server.

The coprocessor maintains a stash as in [36]. Each record is stored as a unit pair $(Info_i(loc_i, p_i))$ in unsecured storage device outside the coprocessor, where loc_i is the location information of the object p_i. The coprocessor interacts with the server to obtain information stored in the server using ORAM methodology. Object location information stored in the server is represented with a binary search tree (BST) data structure. All information must have the same number of bytes *B*. The BST is such that all buckets are of equal size blocks. Buckets are padded with dummy blocks if necessary. We use location data for the BST keys. Location data on the nodes on the left sub-tree are smaller than the key, while location data on the nodes to the right are larger than the key. Location information is stored by the LBS server. Random locations are picked in a balanced manner by the coprocessor. The location data of these random locations are used as key to the tree and stored in the coprocessor stash. For example, suppose three location data randomly picked and used for the keys are key_1 (28.2N, 72.5W), key_2 (28.5N, 72.8W), and key_3 (28.8N, 81.0W). The tree is such that all location data $key_1 < loc_i < key_2$ is left of key_2, while all $key_2 < loc_i < key_3$ is to the right of key_2. The coprocessor will use this structure to determine the access path for the user required information.

Assumptions

The assumptions include the following:

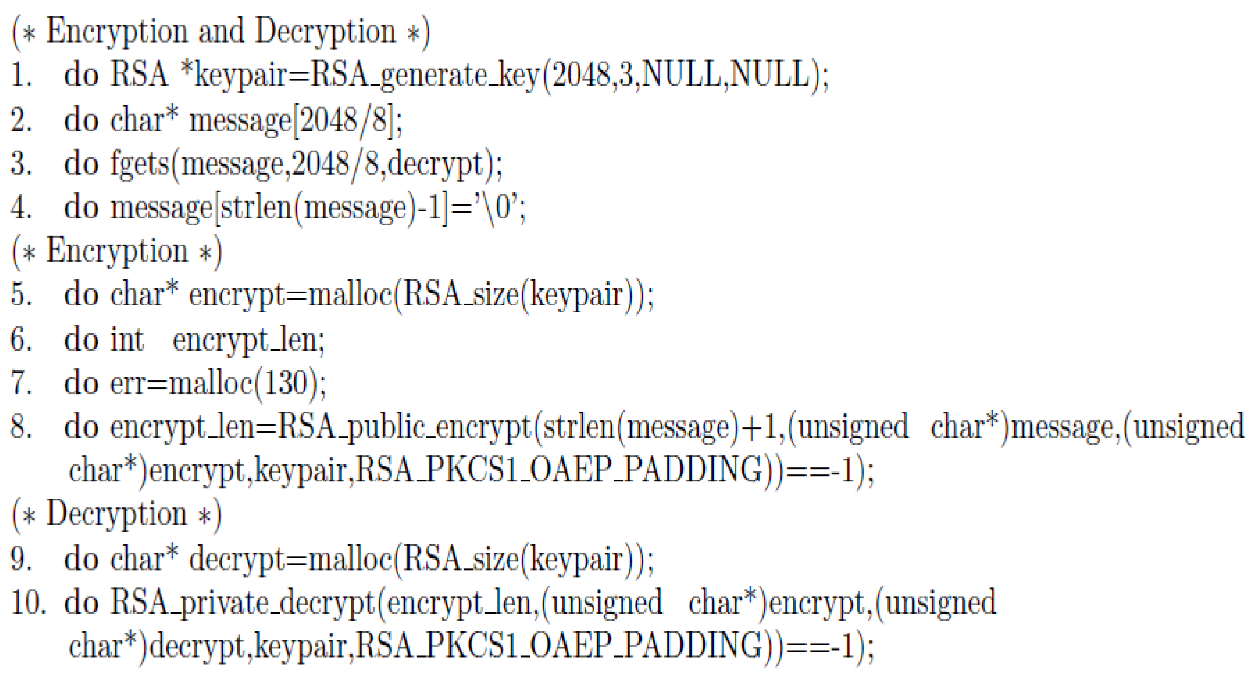
1. The secure coprocessor features a software architecture that allows developers to install and update their applications onto the device in a way that protects the privacy and security interests of the developer.
2. Attacker has no physical access to the hardware or that a direct physical access will not interfere with the computation.
3. The coprocessor is a tamper proof black box designed to satisfy requirements for oblivious computation as in [66] and stated in [71].

Algorithm Design

Secure coprocessor like IBM 4758 model runs its code on the application layer of the card. In our work, we follow similar design of [7] and build a framework that emulates a secure coprocessor card that runs the code to perform the task. The algorithm is designed to ensure oblivious conditions are met in the secure coprocessor operations. The first condition is satisfied by allocating to each thread equal amount of time for operations. Same amount of time will be spent writing to the memory information considered to be a hit and discarding information considered to be a miss. For instance, if X amount of information is processed by the secure coprocessor (where $X = x_1, x_2, \cdots x_n$), for $x_i \neq x_j$ if x_i is the desired information the same amount of time will be spent to write x_i to the internal memory and to discard x_j. The implementation is such that the time and effort spent for all operations will be indistinguishable. This will prevent adversary from distinguishing the type of operation performed. The coprocessor maintains two tables: $Table_1$ and $Table_2$. User initiates a query and sends a request to the coprocessor. The coprocessor implemented by the module *Sim_Secure_Coprocessor()* generates a unique id for the user and calls the RSA key generating function in line 1 of the algorithm in Figure 23 to generate a key. It stores the *Uid* of the user in $Table_1$ as $tableArray_1[\,Us_i][Uid]$ and sends (*Uid,key*) to the user. User then calls similar function to generate its own key, determines the desired information, and calls line 8 of the algorithm in Figure 23 to perform RSA encryption on the data. It then sends (*Enc(Uid,loc,info),key)* to the coprocessor, where *Enc(Uid,loc,info)* is the encrypted user query which contains user identification number, location, and information desired. Coprocessor receives the information and creates the process to perform the operation by calling *"pthread_create(tid, NULL, &process, NUL*L*)*". The coprocessor locks the process by executing *pthread_mutex_lock()* until after the thread execution. During the execution the coprocessor decrypts the information and checks if *Uid* exists in $Table_1$. If *Uid* exists it will store in $Table_2$ the user information as (*Uid,loc,info*). The coprocessor uses the key derived from *loc* to read from *Memory[i...] up to Memory[...k]* where k is the length of data request sequence. The simplest way to emulate memory is to treat it as a plain array of bytes as shown below:

$Data = Memory[key][info]; /*\ Read\ from\ Address1\ */$
$Memory[key][info] = Data; /*\ Write\ to\ Address2\ */$

The array stores the key and data (POI). The key determines the path of the information to be read and processed.



```
(* Encryption and Decryption *)
1.  do RSA *keypair=RSA_generate_key(2048,3,NULL,NULL);
2.  do char* message[2048/8];
3.  do fgets(message,2048/8,decrypt);
4.  do message[strlen(message)-1]='\0';
(* Encryption *)
5.  do char* encrypt=malloc(RSA_size(keypair));
6.  do int  encrypt_len;
7.  do err=malloc(130);
8.  do encrypt_len=RSA_public_encrypt(strlen(message)+1,(unsigned  char*)message,(unsigned
     char*)encrypt,keypair,RSA_PKCS1_OAEP_PADDING))==-1);
(* Decryption *)
9.  do char* decrypt=malloc(RSA_size(keypair));
10. do RSA_private_decrypt(encrypt_len,(unsigned  char*)encrypt,(unsigned
     char*)decrypt,keypair,RSA_PKCS1_OAEP_PADDING))==-1);
```

Figure 23 Encryption and decryption algorithm

The coprocessor runs through the loop to look for user desired information, *for(i=1;i<=sizeof(k);i++)* (*k* is the size of data read from memory) save $Info_i$ in $Table_2$ ($Info_i$ is the information read from memory). If $q_i = Info_i$ continue to save q_i in $Table_2$ till the end of the loop (q_i *is* the user desired information). For the first execution of the loop *Sim_Secure_Coprocessor()* calls *x=PAPI_get_real_cyc()* to determine the amount of clock cycle for the operation. For the rest of the operation it uses the *x* amount of clock cycles for the execution. This satisfies the first oblivious condition. After streaming the entire path and performing the operation to determine user request, the memory is randomly reshuffled and remapped in a write operation. This satisfies the third condition. Application for each user query allocates separate address space that contains entire streamed path. The *pthread_mutex_unlock(&c)* is called at the end to release the resource and allow other threads to execute. This satisfies the second condition. A pending request is created and queued for any request prior to the expiration of the clock cycle. This will ensure that no two threads are granted access to the coprocessor at the same time. Performance is degraded by the wait, but it helps to enforce the oblivious condition.

Using ns-3 node class we create three nodes for the secure coprocessor, the LBS server, and the user. We provide methods for managing the versions of the abstractions for all the nodes. We use specializations of class Application called *UdpEchoClientApplication* and *UdpEchoServerApplication* for data transmission between user and the secure coprocessor. The coprocessor tasks include: key management, encryption, data computation, data transfer, data read and write on the host memory. For encryption we implemented RSA since most existing commercial secure coprocessor already implements RSA. We use OpenSSL for the cryptographic functionality. The sample RSA algorithm shown in Figure 23 does the encryption and decryption process and returns the encrypted data.

Processing SNNQ

Suppose a user wants to receive the nearest POI to its location, the user and the coprocessor engages in the following protocol:

1. Initialization phase: User declares its intention for the nearest POI, the coprocessor creates address space for the user with a unique id and generates random key ($cp_{ub_{key}}$ and $cp_{ri_{key}}$) following the guides provided in [72], and send $cp_{ub_{key}}$ to the user.
2. User identifies its location and desired POI and creates query message q. User generates its own key ($u_{pu_{key}}$ and $u_{pr_{key}}$) and encrypts the query using $cp_{ub_{key}}$ and sends ($\varepsilon_u(cp_{ub_{key_i}}, q)u_{pu_{key}}$) which contains the encrypted message and its own public key to the coprocessor. Only the coprocessor will be able to decrypt the message since it has the $cp_{ri_{key}}$. If $cp_{ub_{key}}$ or $u_{pu_{key}}$is compromised the attacker will still not be able to obtain any useful information.
3. The coprocessor uses $cp_{ri_{key}}$ to decrypt the message, and saves it in the internal data structure.
4. The coprocessor uses path ORAM to obtain from the server information desired by the user. It first identifies the key that relates to the user desired information. It streams the entire path looking for the block with correct record with $O(\log(N))$ search, where N is the total block size on the tree. If the number of blocks in a path is P_{blk} then for $1 \leq j \leq P_{Blk}$ if $j = q_i$ (q_i is the information desired by the user and stored in the block) it will save the information in the internal memory; otherwise, it will discard the information with the same amount of work and time equal to the time to save it as in [7]. Each user thread owns the coprocessor the entire time; therefore, no other thread will be able to access information been computed for another user.

5. Once a path is accessed it is completely remapped to a new location according to process in [36]. The coprocessor sends to the user encrypted query response using $u_{pu_{key}}$ as $\varepsilon_{cp}\left(u_{pu_{key_i}}(q_i)\right)$. It deletes the information from the internal memory.
6. User upon receipt of the information decrypts it with its private key to find its nearest POI. The same process is repeated for subsequent queries

Summary of Results

We evaluate our design by emulating the secure coprocessor card. The host machine runs Ubuntu 9.04 with Intel i5-2540M CPU running at 2.60GHz and 4GB of RAM. The coprocessor capacity is based on working size of 4GB. Our script integrates with *udpEechoClient* and *udpEechoServer* module of the ns-3 network simulator. We evaluate the performance from the initialization phase with regards to response time and compare with software technique. For a fair comparison, we evaluate the software technique for the snapshot query on the same host machine running Ubuntu OS. We ignore all offline computations. The evaluation took into consideration all latencies caused by hardware emulation. Since the size of grid and object can have impact on performance, we examined two different objects and grid sizes. Table 4 shows the steps involved in each of the techniques compared excluding the offline phases. The comparison is based on the sequence of events (not necessarily similar) involved in the eventual retrieval of users nearest POI. For instance, in the hardware technique the coprocessor transmits the key to the user so user can encrypt the request. This is equivalent to server transmitting the entire grid to the user in the software approach to protect user location. Finally, we conduct complexity analysis.

Comparing Response Time

Table 5 shows the results for the two different approach when we compare the response time for *n* = 16 with *k* = 5, and *n* = 225 with *k* = 15 using 1024 bits key. The experiments show that the different *n* and *k* values have impact only on the software approach. It shows that the execution time using hardware approach is slightly over that of the software. However, while the time remains the same in all cases for the hardware approach, the same cannot be said of the software approach. We observe significant increase in cost for *n* = 225 with *k* = 15. The results show that software approach will be desirable over hardware when *n* is small, but as the size of *n* and *k* increases, hardware seems to be a better option than software technique. We deduced from the experiments that when it comes to initial transmission cost, the size of the area and object has impact on the software approach while there is no observable impact on hardware technique. The improvement observed in the hardware technique can partially be attributed to the fact that we are no longer hiding the object location or desired information to the server by sending the entire grid. Sending the entire grid proved to be costly as the size of the grid increases.

Complexity

We also conduct complexity analysis. Three design points affect complexity in the hardware approach. The first one is the external access to host memory (LBS server) through ORAM. In this face, coprocessor streams data from the database with $O(n)$, where *n* is the total size of record in the database. Since we implement path ORAM which streams only a subset of the database the complexity of our design is $O(\log(n))$. This shows an improvement of $(O(n) -$

$O(\log(n)))$ over techniques like [66]. The second design point is remapping the entire database which takes $O(n)$. The third one is finding and encrypting user desired information. If the length of the key is p bits while f(p) is the computation function for the p bits and if each POI is m bit long, it will take the coprocessor $(O(\log(n)) + O(n) + O(f(p)m))$ to search for user desired POI, remap the database, and run the encryption protocol. The equivalent process for the software approach takes $O\left(2f(p)km\sqrt{n}\right)$. In general, tamper resistant hardware performs better than software approach as the size of the area and object increases.

Table 4 Steps compared between hardware and software approach

Tamper Resistant Hardware	Software Cryptography
Coprocessor generates key and sends to the user.	Server sends grid and the POIs information to the user.
User identifies its location and desired POI and creates query message. User generates its own key and encrypts the query using key from coprocessor.	User finds the cell it belongs to and generates its own location data. For each neighborhood in user cell, user computes the absolute difference in the distance between neighborhood and its location. User randomly generates modulus N and query message and sent to the server.
Coprocessor decrypts the message, and saves it in the internal data structure.	Server runs PIR protocol on the user request, and for every column and row it computes two equations and returns Z_α to the user.
Coprocessor engages path ORAM server to obtain information desired by the user. It first identifies the key that matches the location of the user; it then streams in the entire path looking for the block with correct record.	User then determines Z_α value by computing equation 6.
Coprocessor completely remaps the path to a new location.	
User upon receipt of the information decrypts it with its private key to find its nearest POI.	

Table 5 Response time for software and hardware techniques

	Response Time *n* = 16, *k* = 5	Response Time *n* = 225, *k* = 15
Tamper Resistant Hardware	2751ms	2751ms
Software Cryptography	2431ms	5464ms

CHAPTER SIX: PRIVATE CONTINUOUS NEAREST NEIGHBOR QUERY

In this chapter, we extend our privacy protecting technique to address user private information protection in continuous nearest neighbor query. Protecting user private information in a continuous nearest neighbor query poses different type of challenge compare to a snapshot nearest neighbor query. Nearest point of interest to a user may change based on the movement of the user and the point of interest. In some instances, only one object changes its position, while in some both objects change their position simultaneously. Figure 24 shows two depictions of a continuous nearest neighbor query where in (a) only the user location changes, and in (b) the location of user and point of interest changes. The first instance is referred to as moving query static object (MQSO), while the second is known as moving query moving object (MQMO). Examples illustrated in Figure 24 shows a user *u* located in location l_1 with $p_1$as the nearest POI. For MQSO in the top of the figure when *u* location changes from l_1 to l_3 the nearest POI changes to p_6, while for MQMO in the bottom of the figure when *u* location changes from l_1 to l_3 the nearest POI is still $p_1$due to the simultaneous movement of the user and the POI.

The problem in a CNNQ is defined such that a user *u* and POI *p* located in location *l* contained in an area *A* at time *T* and moving at their own respective velocity *v* is at a new location *l'* at a future time *T'*. If *p* is the nearest POI to *u* in *l* located in *A* at time *T* then as the objects moves from *l* → *l'* at time *T* → *T'* private CNNQ shall continuously return to *u* the static or dynamic p located in A whose distance (*d*) from $u \leq \ \forall \ p$ located in A as u or p moves within the spatio-temporal network until query termination without disclosing *l*, *l'* or p to another party.

Some of the suggested ideas for anonymity in CNNQ include issuing a query at every point of a line segment. However, it is very inefficient to issue a query at each point of a line segment as in SNNQ discussed in [16] and [73]. Chow and Mokbel in [37] suggested interesting idea based on cumulative cloaked region that prevent linking query to a specific user. This suggestion demands the presence of other users in the area which may cause some delays in issuing a query. We view the problem as one that can be represented as a continuous vector in a space.

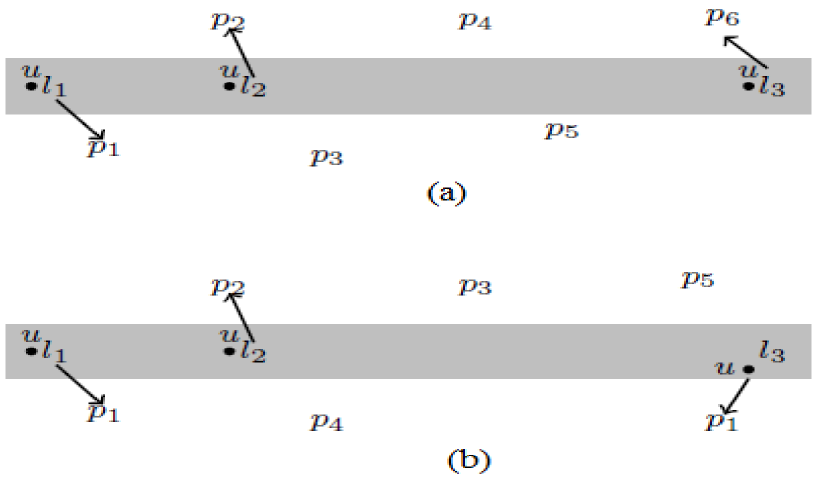


Figure 24 Moving query and static object (a) and Moving query and moving object (b)

We propose plane partitioning using Voronoi diagram and continuous fractal space filling curve using Hilbert curve order to create nearest neighbor relationship within a path of movement such that upon a single query server will continuously update a user with the nearest POI as the user or the POI location changes within the path without giving up user private information. We treat user location as a function of time to project transition time interval within the path where nearest POI will change. Additionally, we extend the concept in MQMO-CNNQ to deal with moving POI. Specifically, we introduce the concept of update time which is when the server updates current POI location. We also implement dynamically adjusting database that scales with the objects to help a user hide a location and relate it to information in the database. Finally, we execute private information retrieval protocol twice in the database to retrieve desired information.

Solutions for MQSO-CNNQ and MQMO-CNNQ

Our design strategy is to determine the next nearest neighbor to each of the POI in a path based on their proximity to one another by utilizing Voronoi diagram and Hilbert curve order. The distances between the points of interests is determined and used to determine where the nearest POI will change. Location of the objects within the spatial network is treated as a function of time, and the nearest POI will depend on the object velocity and elapsed time. A user issues a query to LBS server and specifies the time interval when the nearest POI will change. At each time interval the server will update the user with the nearest POI in the user's path without the need for the user to issue another query as object location changes. To help understand our design, we first explain the different models and tools that were used in the design.

Modelling Dynamic Object

To account for the location of the objects at all time, we treat the position of the objects as a function of time. A spatial network (*SN*) consists of a user *u* and a POI *p*. Both objects are statically or dynamically restricted within the edges of the *SN*. If *v* is the velocity of a moving object (user) at a location $l \in SN$, then the object position after *t* time can be represented as

$$l_u(t_i) = f(v * t) \qquad (14)$$

Consequently object (POI) location can also be represented as follows:

$$l_p(t_i) = f(v * t) \tag{15}$$

Where *I* in equations 14 and 15 represents the *i*-th location of a *u* and the *p* after time t_i, while *t* is the elapsed time.

Note that *v* is a vector and therefore has magnitude and direction. From equations 14 and 15 we can see that it is possible for the same POI to be the nearest neighbor to a user at different places at different time in the same segment $\overline{se}$.

We assume that a *u* and the *p* are contained in the same $\overline{se}$. If we know l_u and l_p at time t_i we will be able to determine the nearest POI to a user at all time in $\overline{se}$. This requires that server updates database information each time object location changes.

Modelling Distance

Finding distances between moving objects are challenging. A brute force approach can be used to calculate the distances. This approach is effective but inefficient. Work in [11] modelled distances of moving object with a directional weighted graph G(*V*,*E*), where *V* is the vertex and *E* is a single directed edge. If *E* is the empty set that contains no vertex and *R* is the set for all the single directed edges then all elements *E* ∈ *R* corresponds to a single directed edge. If we assign the edges weight $w(v_i, v_{i+1})$, the length of a path P ∀*E* ∈ *R* is

$$\mathrm{P} = \sum w(v_i, v_{i+1}) \tag{16}$$

We use this approach to model distances between objects. We represent vertices as nodes (POIs), and the edges are the distances between the POIs. The distance between two POIs is the number of edges in the shortest path connecting them. The shortest distance $d(n_i, n_j)$ between two nodes is therefore defined as min *w(*P*)* for all paths between $n_i$and n_j, where n_i and n_j are the *i*-th and j-th node respectively. If a user *u* is located between path n_i, n_j then, the nearest POI to the *u* is the POI with minimum *w(*P*)* from the user. The *w(*P*)* is represented as a time interval between the POIs.

Indexing Continuous Nearest Neighbor Query

As mentioned in [74], the result of a query q through a path contains a set $< S_i, T_i >$ tuples where $S_i = p_1, p_{2\ldots}p_k$ is the ordered list of the POI (*k* is the number of POI in the query path), T_i represents the time interval during which S_i is the nearest POI to a user through the path. POIs are ordered by their increasing distance to *u*, i.e., p_i is the closest POI to *u*, while p_{i+1} is the next closest POI to *u,* and p_k is the *k-th* POI closest to *u* in the interval T_i,T_{i+1} and T_k, respectively. The returned sets through the path in the interval or segment $\overline{se}$ satisfy the conditions:

$$\cup_i T_i = [t_s, t_e]$$

and

$$T_i \cap T_j = \emptyset, \forall i \neq j$$

where $[t_s, t_e]$ is the query interval divided into non-intersecting sub-intervals *n* in the intervals $T_i = [t_s, t_1]$ and $T_n = [t_{n-1}, t_e]$ for $(t_s < t_1 < t_{n-1} < t_e)$. The time stamps $(t_1, t_2 \cdots t_{n-1})$ are used for the split or the transition time interval where the nearest POI to *u* will change.

Voronoi Partition

To help understand the proximity relationship between points in a plane, we provide an overview of Voronoi diagram. Voronoi diagram shown in Figure 25 is a way of dividing space into regions. It is a special kind of decomposition of a metric space determined by distances to a specified discrete set of objects in space. A set of spots is specified beforehand and for each spot there will be a corresponding region consisting of all points closer to that spot than to any other [15]. It is one of the most fundamental constructs defined by a discrete set of points [75] and [76]. Given a number of points in a plane Voronoi diagram divides the plane according to the nearest-neighbor rule, and each point is associated with the region of the plane closest to it.

Aurenhammer in [77] defined Voronoi diagram in terms of *dominance* of two distinct points *x* and *y*. Let *S* denote a set of *n* points in a plane, for two distinct points x and $y \in S$ the *dominance* of x over y is defined as the subset of the plane which is at least as close to x as to y, i.e.

$dom(x, y) = \{r \in R^2 | \delta(r, x) \leq \delta(r, y)\}$, where δ is the distance function.

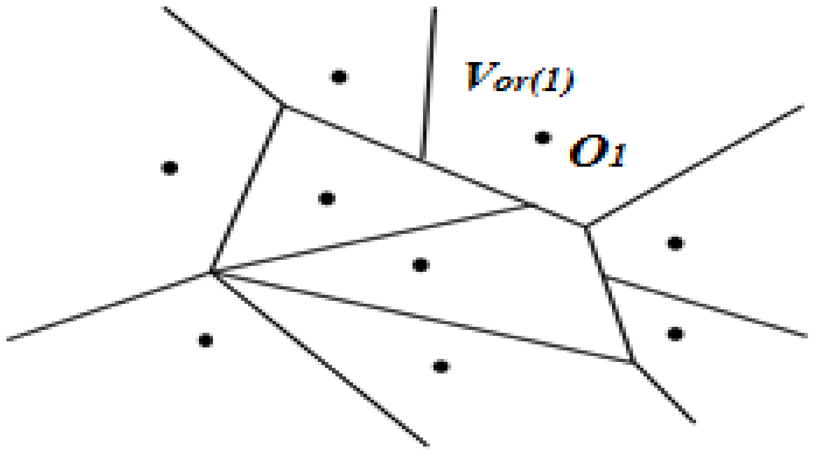


Figure 25 Voronoi diagram showing 9 points of interest on a plane

Therefore, $dom(x, y)$ is a closed half plane bounded by a perpendicular bisector of x and y. All points of the plane closer to x are separated from the points in x by this bisector termed the *separator*. The region of a point $x \in S$ is the portion of the plane lying in all of the *dominances* of x over the remaining points in *S*, i.e.

$$reg(x) = \bigcap_{y \in S - x\}} dom(x, y)$$

As [77] stated, since the regions are coming from intersecting $n - 1$ half planes they are convex polygons. Therefore, the boundary of a region consists of at most $n - 1$ edge(s) (maximal open straight-line segments) and vertices (their endpoints). Each point on an edge is equidistant from exactly two points, and each vertex is equidistant from at least three points. As a result, the regions are edge to edge and vertex to vertex, i.e., they form a polygonal partition of the plane. Therefore, $reg(x)$ cannot be empty since it contains all points of the plane at least as close to x as to any other points in *S*, therefore $x \in reg(x)$.

Applying Voronoi Diagram

We modelled our design after Network Voronoi Diagram (NVD). NVD as described in [75] and used in [43] has been extensively used to evaluate various spatial proximity queries [78]. Kolahdouzan in [79] implemented NVD similar to our work based on network distance where the results set were generated incrementally assuming all object to be static as noted in [11]. Okabe et al. in [75] defined NVD as specialized Voronoi diagram where objects location are restricted to the links that connect the nodes of the graph and the distance between objects are defined as the length of the shortest distance (shortest path or shortest time) in the network distance rather than Euclidean. In our work, we modelled the network as weighted graphs where the intersections are represented by nodes of the graph, and the roads are represented by the links connecting the nodes. The weights are the distances between the nodes that represent the time it takes to travel between the nodes assuming a constant velocity. We assume that all the POIs in a segment are pre-determined. For example, all the gambling centers from town A to town B are pre-determined.

We map an area to a $\sqrt{n} \times \sqrt{n}$ grid and super-impose Voronoi diagram over the grid. All the POIs in the area are bounded to a cell in the grid. The contents of the cells represent the two endpoints of a query line segment $\overline{se}$, where $\overline{se}$ contains $p_{1\rightarrow k}$ (k is the set of points in $\overline{se}$) and *dominance* of p_1 over p_2 and *dominance* of p_2 over p_1 exist, i.e.,

$dom(p_1, p_2) = \{x \in R^2 | \delta(x, p_1) \leq \delta(x, p_2)\}$ and

$dom(p_2, p_1) = \{x \in R^2 | \delta(x, p_2) \leq \delta(x, p_1)\}$ are true.

Each cell c now contains k POIs which is the answer set returned after all the points that intersect a Voronoi cell are bounded to *c*. This leads to the heuristics from [16] and stated below.

Heuristic HC2: For a given query segment$\overline{se}$ $kNN(s) = \{O_{s_i}, i \in [1,k]\}$ and $kNN(e) =$ $\{O_{e_i}, i \in [1,k]\}$, then $\{O_{s_i}, O_{e_i}, i \in [1,k]\} \subseteq CkNN(\overline{se})$.

Since *s* and *e* are the two endpoints in the query line segment, the *k* points between these line segments are part of the answer set.

Heuristic HC3: For a given query segment $\overline{se}$ if

$kNN(s) = kNN(e) = \{O_i, i \in [1,k]\}$, then $kNN(\overline{se}) = \{O_i, i \in [1,k]\}$.

Proof: Following [80], for *k*=1 HC3 can be shown with Voronoi diagram. If *Vor*(O_i) represents the Voronoi cell for object O_i for any query point the object must be located within one *Vor*. *Vor*(O_i) and object O_i must be the nearest neighbor to that query point. As in Figure 25, the Voronoi diagram partitions the space into 9 parts based on the object positions. The area *Vor*(1) is the corresponding Voronoi cell for object O_1, i.e., $O_1$is the only nearest neighbor to any query point inside *Vor(*1*)*. Based on the computational geometry in [80] *Vors* are convex. Since $kNN(s) = kNN(e) = \ O_1$ for *k*=1 both endpoints and the entire query line segment ($\overline{se}$) lie inside the *Vor* of object O_1. Therefore, $O_1$is the nearest neighbor to any query point along the query line segment. For every cell *c* of the grid server determines all Voronoi cell intersecting it and adds the corresponding object to *c*. Cell *c* contains all the potential nearest neighbor to a query through a path which we denote as set *X*. The cells are padded if necessary to create cells of equal size. All objects in M have equal bits (M is the matrix representation of the database information). The number of bits can be of any size so long as they are consistent in M to avoid inference of the requested object by the server based on the number of bits transferred.

Hilbert Space Filling Curve

Hilbert curve order is used to map all the POIs in a *c*. It is a continuous fractal space filling curve that preserves spatial proximity [81]. If $I = \{t|0 \leq t \leq 1\}$ represent the unit interval on a space and $Q = \{(x, y)|0 \leq x \leq 1, 0 \leq y \leq 1\}$ is the unit square as in [82], for each positive integer *n* the interval *I* is partitioned into 4^n subintervals of lengths 4^{-n} and the square Q into 4^n sub-squares of sides 2^{-n}. A one to one correspondence between the subintervals of *I* and sub-squares of Q are constructed to meet the following conditions:

Adjacency, adjacent subintervals correspond to adjacent sub-squares while sharing a common edge.

Nesting, if at the *n*-th partition the subinterval I_{n_k} corresponds to sub-square Q_{n_k} then at the (n+1)-st partition the 4 subintervals of I_{n_k} must correspond to the 4 sub-squares of Q_{n_k}.

Hilbert curve as shown in Figure 26 visits every point in *n*-dimensional grid space exactly once without crossing itself [73]. It is used for mapping multi-dimensional space to one-dimensional space while still preserving locality [43]. For instance, if *x* and *y* are the coordinates of a point within the unit square and *d* is the distance along the curve when it reaches that point, the points that have nearby *d* values will also have nearby (*x*, *y*) values [83]. That is, if two points are close in the 2-D space they are likely to be close in the Hilbert ordering as well. Analysis by [84] shows that Hilbert curve is good in preserving proximity.

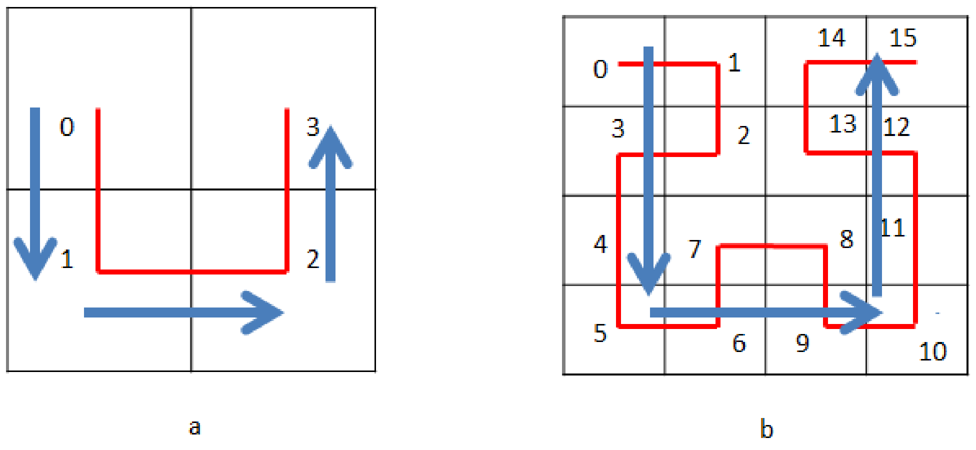


Figure 26 Hilbert Curve of (a) order 1 and (b) order 2

Applying Hilbert Type Space Filling Curve

After we bind all POIs to *c* using Voronoi diagram, we map all POIs in *c* using Hilbert curve order. All the points $p_i \in X$ for *i* from 1 to *k* in the intervals *I* are mapped to Hilbert Curve order as in [43] and partially in [85] such that a one to one correspondence between the subintervals of *I* and sub-squares of *Q* satisfy adjacency and nesting conditions which determines a continuous function w_f that maps *I* onto *Q* [82].

Following [82], a one way function $w_f \;\; I \to Q$ is stated as follows:

1. If there is $t \in I$ which is not the endpoint of any subintervals of *I* then there is a *t* which is part of a closed nested subintervals$\{J_u\}_1^\infty$ one from each partition with lengths approaching zero. A unique point $p = \{p_1, p_2\} \in Q$ is determined by the corresponding sequence of closed nested squares $\{S_u\}_1^\infty$ with a diameter that approaches zero. Define a function $w_f(t) = p$ where *t* =0 and *t* = 1 are similar.

2. If for some value u, for instance, u=v, $t \in I$ is common to two adjacent intervals J_u, J'_u, then $\forall u \geq v$, t is common to two adjacent intervals J_u, J'_u and therefore belong to two nested sequences $\{ J_u \}_v^\infty$ and $\{J'_u\}_1^\infty$. The corresponding sequences of squares $\{S_u\}_v^\infty$ and $\{S'_u\}_v^\infty$ determine the same point $p \in Q$ since the squares are adjacent and their diagonals approach zero [82]. For such t we define $w_f(t) = p$. To show that the function w_f is continuous, if $|t_1 - t_2| \leq 4^{-u}$ then $t_1$and t_2 must lie on the same interval or in two adjacent intervals of the u-th partition. The corresponding images should be in the adjacent squares that form a rectangle with sides 2^{-u} and $2 \cdot 2^{-u}$.

Therefore,

$$\left\| w_f(t_1) - w_f(t_2) \right\| \leq \sqrt{5} \cdot 2^{-n} \tag{17}$$

Since there is a correspondence between points in the intervals and squares, all the points are assigned Hilbert values. Remember that all the points in a path are contained in one cell of the grid. For a path with 9 POIs *c* will contain 9 POIs, and all POIs are assigned Hilbert value *H* based on their distance to one another. For example, points p_1, p_2, p_3, p_4 have *H* values of 5, 15, 20, and 10 respectively. Points with closer values are closer in space.

Database Design

We transform the H values which represents set of points in a multi-dimensional space into records in a database akin to [86] . Queries on the records are queries on the sets of points which are represented by their Hilbert values. We represent the database *D* as a $\sqrt{n} \times \sqrt{n}$ matrix

M, and we use R-tree data structure for the geometric data storage. R-tree is the most widely used geometric data structure suitable for storage on a disk [87]. The idea is to group nearby objects and represent them with minimum bounding rectangle (MBR) in the next higher level of the tree and have nodes of high degree that fit exactly into one block. The bounding rectangle is used to determine whether or not to search inside a sub-tree. Figure 27 depicts one-level R-tree with nodes of size 3. Each key value is greater or equal to the values in the left node which are Hilbert's. The index key is the maximum Hilbert value in the segment concatenated with the cell id so that when a user identifies its cell the user can then use the value to determine which query set belongs to its path. R-tree performs very well and has a linear worst case scenario. It was shown in [88] that R-tree on n-points in d-dimensional space will visit $\Omega((n/B)^{1-1/d} + g/B)$ blocks, where g is the number of query answers. A version shown in [89] called PR tree was said to achieve the same bound when the stored objects and query objects are axis-parallel hyper-rectangles [80]. Readers interested in R-tree structure can read [90].

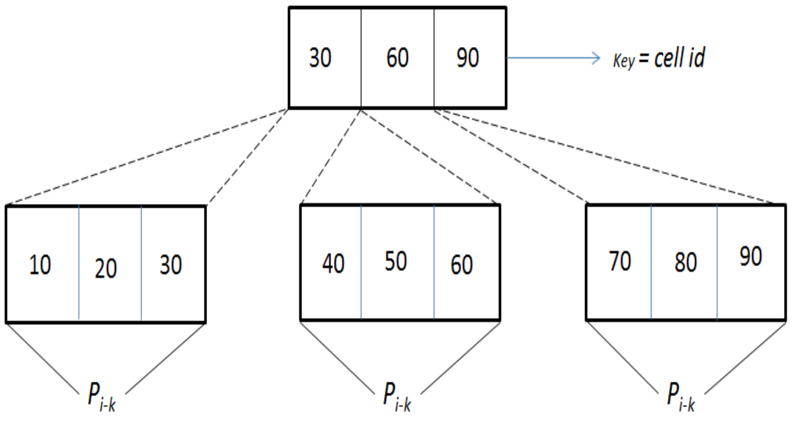


Figure 27 R-Tree representation of a database for k=3

The cells are identified by the cell id, and each cell contains k POI in the segment $\overline{se}$ representing the path of travel by a user. Each cell contains equal k and is padded if necessary to prevent server inference from number of k transmitted. Stored in the database with the POIs are the weights w which represent the average distance between adjacent POIs in $\overline{se}$ which will be used to determine the split or the transition point which will indicate the point where the nearest POI to a user will change.

Processing MQSO-CNNQ

The following steps will be followed to answer MQSO-CNNQ:

Step 1: Following the process described above server creates location data record in the database represented with R-tree geometric data storage where the keys are the maximum Hilbert value for the segment.

Step 2: User in location l initiates a query to obtain the nearest POI along a path of interest. Server sends the grid, the key, k (number of POI available for a segment as stored in the database), and the average distance d_{avg} between the POIs. Each segment $\overline{se}$ has its own d_{avg}. Note: It is assumed that the POIs are approximately equidistance from one another, and that the time interval where the nearest neighbor change occurs can be averaged to a uniform value such that the average time interval t_{avg} represents the transition time for all p from $p_1, p_2, \cdots, p_k$ in each segment represented in the grid for a known constant velocity v. If d_i is the distance of travel before the nearest neighbor changes from p_i to p_j, for $i \leq j \leq k$ server computes the average distance as

$$d_{avg} = \frac{\sum_i^{k-1} d_i}{k-1} \tag{18}$$

and the average time for object travelling at constant velocity *v* becomes

$$t_{avg} = \frac{\sum_i^{k-1} d_i / k-1}{v} = \frac{d_{avg}}{v} \tag{19}$$

where *k* is the number of objects in the path.

Step 3: User receives the grid together with the parameters and identifies the cell it belongs to. Focusing on that cell the transition time t_t at any velocity *v* is then calculated by the user as $t_t = \frac{d_{avg}}{v}$ as shown in algorithm of Figure 28. Using the key user will then request all the POI contained in the set left of the key. For instance, if the key of interest to a user is 60 the user will request the values 40, 50 and 60, Figure 27. Following the same principle as in snapshot query with a database $s = \sqrt{n}$ user will generate query message $y = ([y_1, y_2, \cdots, y_s]^k)$ and $x = ([x_1, x_2, \cdots, x_s]^k)$ each targeting the k^{th} element in the segment $\overline{se}$, where *k* is the number of POI contained in the path such that

$$y_b \in QNR, and\ \forall j \neq b, y_j \in QR$$

and

$$x_a \in QNR, and\ \forall r \neq a, x_r \in QR$$

User sends the query message together with the transition time t_t to the server.

```
(* Creating query message and computing transition time *)
Input: grid G,k,keys and $d_{avg}$
Output: $t_t$,$y_{1\to k}$,$x_{1\to k}$
Procedure $createQuery(G, k, keys, avgdistance)$
1.    for each $c \in G$
2.        if $u \in c$
3.            (* Compute transition time *)
4.            do $t_t = d_{avg}/v$;
5.      do generate $y_{1\to k}$ and $x_{1\to k}$;
6.    return $t_t$,$y_{1\to k}$ and $x_{1\to k}$;
```

Figure 28 User computing the transition time and creating query message

Step 4: Using algorithm of Figure 29 server will run PIR protocol on user request and creates a size *k* FIFO queue data structure of Z_α. The server adds an item to the queue in the order represented on the tree, i.e., the closest neighbor to the user from the point of origin (closest Hilbert value) is added first in that order until the farthest neighbor. Server returns the first Z_α to the user. Using the same method as in snapshot query user decrypts server cryptic response to reveal its nearest neighbor. At each t_t interval server will update user with the next value in the queue until the end of the segment or user terminates the query. Upon receipt of a nearest neighbor user determines the validity of the nearest neighbor by using its position and a flag with a value of true or false. For instance, when server updates the user with the nearest neighbor at the user specified interval, in order for it to be the nearest neighbor user must reach the transition point which will result in setting the flag to false, otherwise, if the flag is true (meaning user has not reached the transition point, previous POI is still valid) user caches the nearest neighbor without decrypting it and use the previous nearest neighbor since it has not reached the transition point where the nearest neighbor will change. User only decrypts server response if the flag is set to false. The flag will ensure that any deviation by the user from the expected travel pattern will not result in false positive.

```
(* Server running PIR protocol *)
Input: $n$, $R_{tree}$, $t_t$, $y_{1\to k}$, $x_{1\to k}$
Output: $arrayZ_\alpha[k]$
Procedure $PirProtocol(query)$
1.    for $i \leftarrow 1$ to $\sqrt{n}$;
2.      for $j \leftarrow 1$ to $\sqrt{n}$;
3.        for $s \leftarrow 1$ to $k$;
4.          do $Z_{ijs} = \prod W_{ijs}$;
5.      for $i \leftarrow 1$ to $k$;
6.        do $arrayZ_\alpha = Z_{ijs}$;
7.        for $i \leftarrow 1$ to $k$;
8.          if time=$t_t$;
9.            return $arrayZ_\alpha[i]$;
```

Figure 29 Server running PIR protocol and responding to user request.

Processing MQMO-CNNQ

We modified our design to address MQMO-CNNQ. In MQSO-CNNQ we used R -tree indexing to help speed up searching and improve performance since objects location is static throughout the life of a query and therefore updates are deemed infrequent. This condition does not hold for moving object where updates are frequent due to changes in object locations. Moving object database need to accommodate frequent updates while simultaneously allowing efficient query processing [91]. A study was done in [92], design issues were considered for efficient retrieval of moving objects with frequent updates. It concluded that most techniques are complimentary and can be used in parallel. Query index approach rather than object index was used in [93] to avoid the constraints imposed on the speed or path of the moving object. Time parameterized R-tree was used in [94] which were based on using current and anticipated future positions of moving objects. Similar method based on B+ -tree was used in [95] where the indexed data in the tree are no longer points, but rather linear functions and the time of update. In the method, updates are only performed if the function predicted position is considered inaccurate [96]. While it is possible to get by with predicted object position for a short duration of a query life, it may not suffice for the entire query existence.

We implement B+ -tree similar to [95]. We use B+ -tree to allow efficient update and access to object information. Because of its support for linearization and indexing [91] stated that B+ -tree is more appropriate to model moving object. If at time *T* object is at location l_0 and the same object is at location l'_0 at time *T'* then the distance moved by the object can be modelled as

$$d_m = f(v * t) \tag{20}$$

where $v \geq 0$ is the velocity of the moving object, while *t* is the elapsed time. Figure 30 shows partial B+ -tree index structure and the matrix $\mathrm{M} = \sqrt{n} \times \sqrt{n}$ representation of database *D* for *k*=4. The values in the leaf node shown in the figure are the Hilbert values for the objects.

The process is similar to static object query. Like in the static object query we create Voronoi diagram using the set of objects and super-impose a regular grid of $\sqrt{n} \times \sqrt{n}$ on top of the Voronoi diagram. Using Hilbert Curve ordering we create a one to one correspondence between the subintervals of *I* and sub-squares and then assigns Hilbert value to the objects based on their distance to one another. Geometric database storage of the objects using B+-tree is created. The index key is the Hilbert value concatenated with the cell id. All the objects in $\overline{se}$ are therefore within the range of minimum and maximum Hilbert value for the $\overline{se}$. The objects are modelled as a linear function of time and therefore the data to be indexed are no longer point representation of Hilbert value, but rather linear functions. The internal nodes serve as a directory, and each has a pointer to the right sibling. Stored in the leaf node are the velocity, the location, and id of the object. All objects are assumed to have constant velocity. Objects location from time $T \rightarrow T'$ is determined based on the stored velocity. Objects update their velocity when a change in velocity occurs; otherwise, no updates are required. This will prevent unnecessary updates.

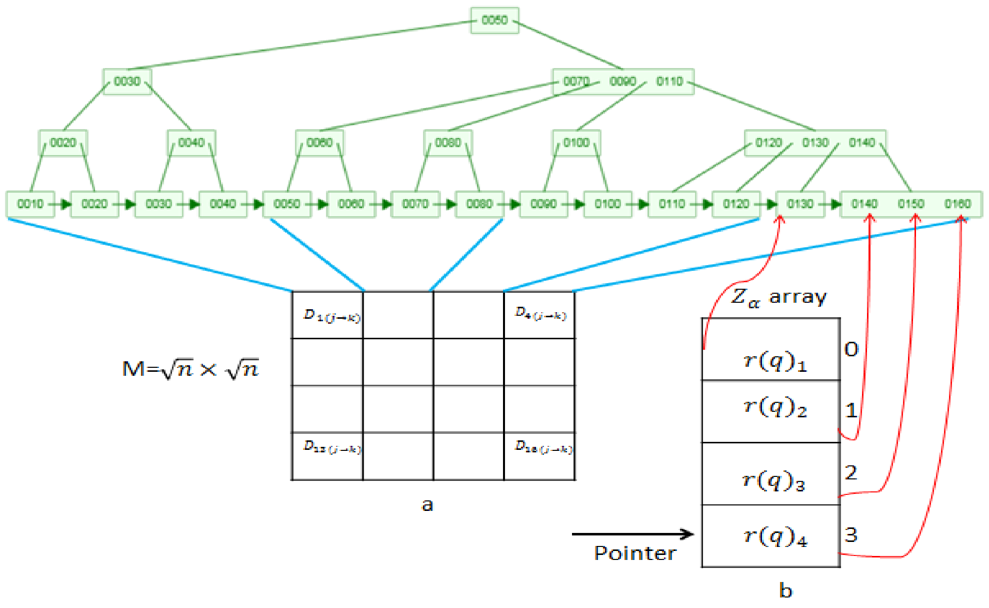


Figure 30 B+ tree index representation of database matrix and $Z_\alpha$array for k=4 with pointer at $Z_\alpha[3]$.

As in static object query, transition time is determined. This is the time or point where the nearest neighbor to a query will change. We partition the time axis into different time duration, where each time tick is the update time t_u, i.e., the time the server updates database information. At each $t_u \cong t_t$ server computes l'_0. If there is no Δv there will be no update from user. Server will use the stored v to determine object location.

The following steps will be followed to answer MQMO-CNNQ.

Step 1: Server creates nearest neighbor record following the same procedure as in MQSO and builds B+ -tree geometric database storage of the POIs.

Step 2: User initiates a query to obtain nearest neighbor along its path. Server sends the grid, the key, k (number of POI available for a segment as stored in the database), and the average distance d_{avg} between the POIs. We also assume that no new object enters the segment after query initiation, therefore k does not change.

Step 3: User receives the grid together with the parameters and identifies the grid it belongs to. Focusing on the grid it belongs to user finds the transition time using its velocity and d_{avg}. User will then request all the objects contained in the range up to key maximum. Following the same principle used in the MQSO with a database $s = \sqrt{n}$ user generates query message $y = ([y_1, y_2, \cdots, y_s]^k)$ and $x = ([x_1, x_2, \cdots, x_s]^k)$ each targeting the k^{th} element in the segment $\overline{se}$, where k is the number of objects contained in the path such that

$$y_b \in QNR, and\ \forall j \neq b, y_j \in QR$$

and

$$x_a \in QNR, and\ \forall r \neq a, x_r \in QR$$

User sends t_t to the server together with the query message.

Step 4: Server runs PIR protocol on user request and with equations 3 and 4 computes Z_α. It then creates array of size k for Z_α in the order shown in Figure 30b. Initially, the pointer points to j=0. Server sends to the user the value at j=0. At each subsequent $t_u \cong t_t$, server updates the database with current order of object location for the $\overline{se}$. Server will again create array of size k for Z_α and shifts the pointer to j+1 and send the value at j+1 to the user. If the flag is set to false which indicates that the transition point has been reached the user will again decrypt the message, otherwise, user caches the server response. Server continues the same process of recreating array of Z_α at each t_u and each time shifting the pointer by 1 until the last k object is sent to the user or the user terminates the query.

Analysis and Results

The process for MQSO and MQMO are the same for the most part, therefore, our experimental evaluation represents the analysis of both schemes. We present the evaluation with respect to transmission time and complexity. For the server transmission time we simulate and compare different *k* values, were *k* is the number of POI in a path. We also simulate and compare different *n* (*n* is the size of the grid) in order to understand optimal parameters. Next, we compare user transmission time while changing *k* and *n.* We also vary the size of modulus used in the PIR protocol. We then implement the design of [15] and compare the response time with our technique. Finally, we conduct analysis on the impact of large prime on PIR protocol.

Experimental Setup

The application was developed in C++. The server and client experiment was conducted on ns-3 simulator running on Ubuntu 9.04 Linux variant operating system with intel-quad-core processor operating at 2.66GHz with 256MB RAM. The server communication with the access point runs a CSMA protocol with data rate of 100Mbps and 6500ns channel delay. Client communication is wifi 802.11a standard with a stream data rate of 54Mbps and a default constant speed propagation delay representing the propagation speed of light in the vacuum. Maximum packet size is 2304 bytes including the headers. The actual payload is 1450 bytes. The access point (AP) share the same type of Physical level and channel attributes as the client wifi device which we defaulted to ns3 values. The AP is stationary while the client is mobile and randomly wonders around at a random speed in a bounded box defined by x and y coordinates. We varied the grid size from 16 up to 225. For the objects we use Sequoia points in [97] which contain 62,556 California place names. We use 768 and 1,536 bits for the modulus. Shown in Table 6 are the rest of the parameter settings.

Transmission Time

First, we simulate the time it takes the server to transmit the packets using different values of k as n changes from 16 upwards to 225. The result is shown in Figure 31. The experiment shows that for small n such as $n = 16$, k has no significant impact on the transmission cost. However, as n increases the impact of k becomes noticeable. When $k =5$ the increase in the transmission time seem to be proportional to increase in n until $n = 144$ when the transmission cost took a sharp increase. When *k* = 10 and 15 the sharp increase in the transmission time occurs sooner at *n* = 64. What this means is that when *k* =5 increase in *n* up to 144 will produce a more desirable result, while for *k* = 10 and 15 increase in *n* up to 64 will produce better result.

Table 6 Parameter Settings

Parameters	Settings
Packet Transmission Interval	1 second
Spatial Object Size	80 - 3375
Object Size	10kB
Query Size	100 Bytes

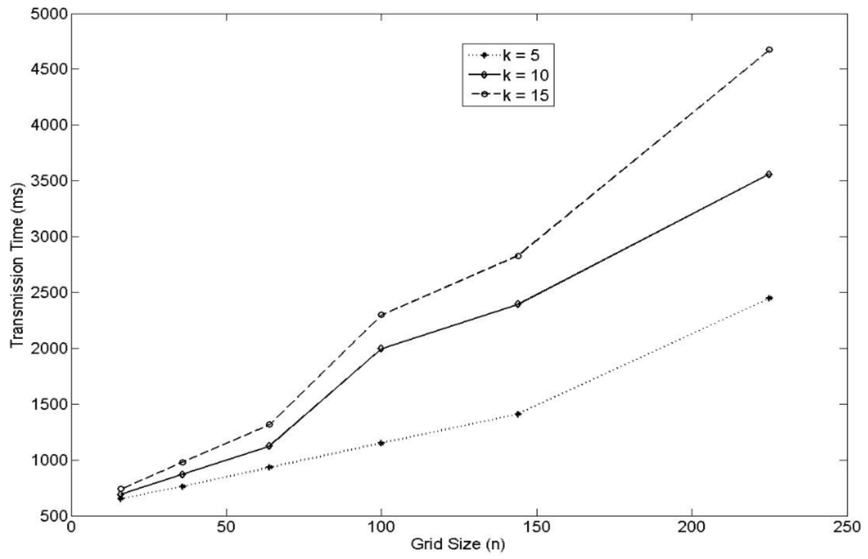


Figure 31 Server Transmission time for three different *k* as *n* changes from 16 up to 225

Figure 32 shows the time for the server to transmit its packets for the three different values of *k* and *n*. From the experiment we made the following observations: First, when *n* is small such as *n* = 16 the impact to transmission time is less significant as *k* increases. Second, when *n* increases to 64 the cost for transmitting the packets takes a sharp increase at *k* = 10. Third, when *n* increases to144 the transmission time increase is very minimal. What this means is that in the first observation, for an area with high concentration of *k* we can create a bigger cell to accommodate more *k* which will result in reducing the size of *n* in order to have little or no impact to the transmission cost. For the second observation, when *n* increases up to 64, *k* > 10 will have significant increase in transmission cost. Therefore it will be more desirable to reduce the size of *n*, for instance *n* = 16 and *k* >10 will be more desirable. In the third observation, for *n* = 144 the proportionality of the increase is less when *k* > 10 than when *k* < 10. This means that the design will still maintain acceptable performance when *n* is large and *k* > 10.

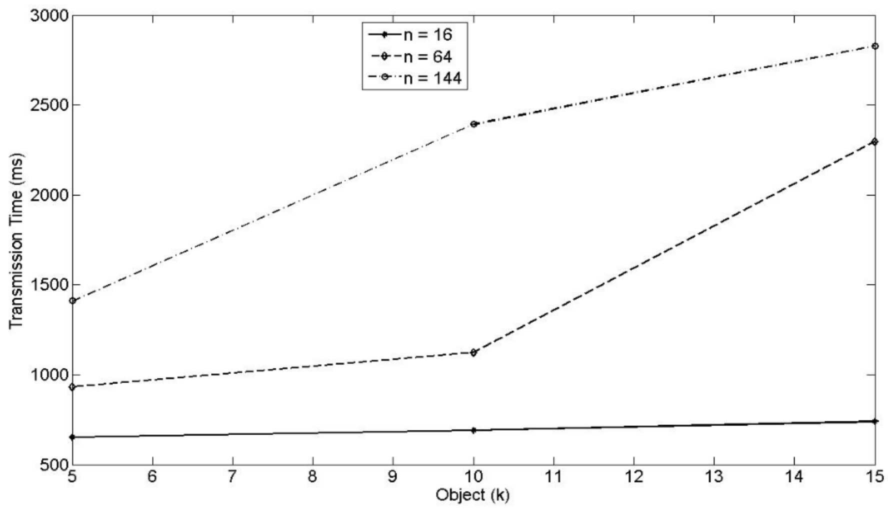


Figure 32 Server Transmission time for k = 5, 10, 15 and n = 16, 64 and 144

We conduct experiments to determine the cost for the user to transmit the request to the server. Figure 33 shows the result for three values of *k* (5, 10, and 15) and *n* (16, 64, and 144) with modulus *N* set to 768 bits. We observe negligible increase in transmission cost for n = 16 for all three *k* values. We also simulate and compare the time increase for the three *k* values when *n* is set to 64 and 144, and we observe a larger proportion of time increase when *n* is set to144 than when *n* is 64. The observation here is that it will be more desirable to make *k* as small as possible when *n* is inevitably large. For an area with large concentration of *k* we can make *n* as small as possible while making *k* large to achieve better result.

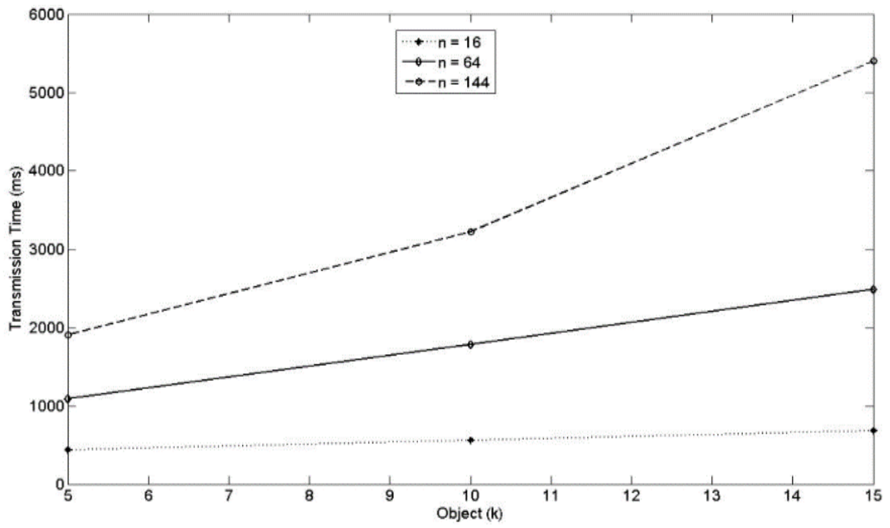


Figure 33 Client Transmission time with 96 Bytes Modulus and k = 5, 10, and 15

We also compare the cost for a user to transmit its request to the server using different *k* and *n* values. As the results in Figure 34 shows, the transmission cost impact is minimal when *n* is small for three different *k* values. However, we start seeing significant difference as *n* increases. The difference is more noticeable when *n* is greater than 144. The significance of this is on how best to scale *k* and *n* for better performance. The experiment shows that it is desirable to keep *n* small when *k* is large, and when *k* is small *n* can be large without adding significant cost to a user.

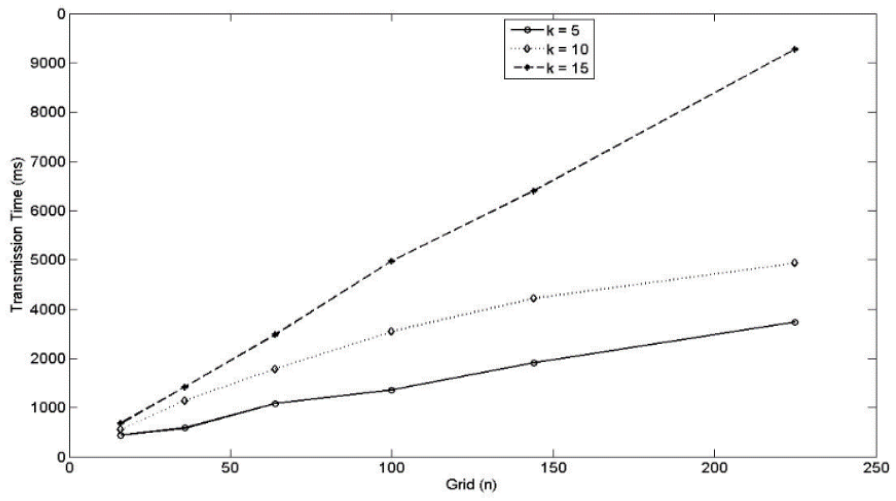


Figure 34 Client Transmission time with 96 Bytes Modulus and n from 16 to 225

Also, as an alternative and for added security we increase the modulus *N* to 1536 bits and the results are shown Figure 35. We used it on three values of *k* (5, 10, 15) and *n* (16, 64 144) and compare the transmission cost with 768 bit modulus. We did not observe any significant increase in the transmission cost for small sizes of *n* and *k*. For instance, when *n* is 16 and *k* is set to 5 the cost is 437ms for 768 bits, and 558ms for 1536 bits. When *k* increases to 15 and *n* stays the same the cost is 679ms for 768 bits, and 1327ms for 1536 bits. The difference between 768 bits and 1536 bits for $k = 5$ and $k = 15$ starts to narrow as *n* gets larger. Nevertheless, the size of *N* together with *k* and *n* has impact on the cost. This offers privacy and efficiency tradeoff opportunity for a user. For instance, a user searching for nearest gas station may have minimal privacy requirement and therefore for efficiency choose smaller *N* than a user looking for a special need medical facility which may prioritize privacy over efficiency and therefore chooses larger *N*.

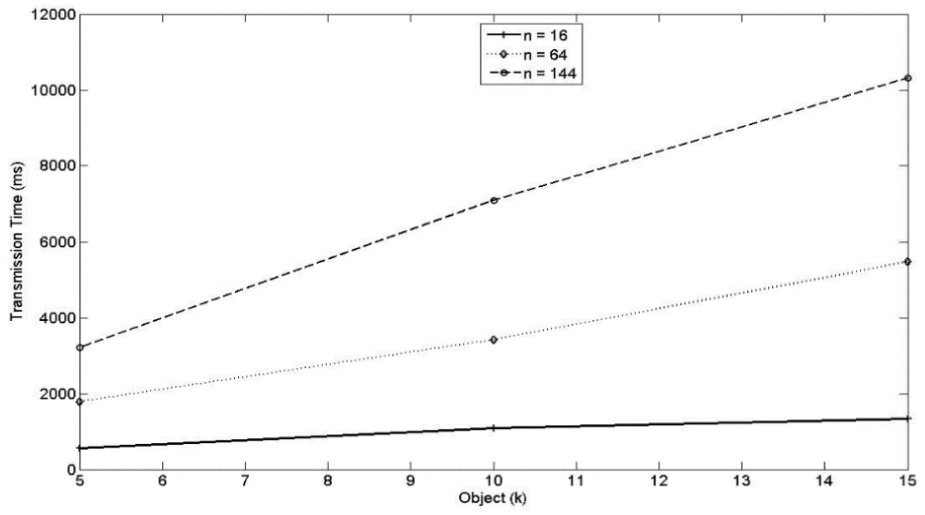


Figure 35 Client Transmission time showing impact of 192 Bytes Modulus and k = 5, 10, and 15

Complexity

We analyze server and user computation complexity and compare with [5]. The results are shown in Table 7. As already stated the database is of size $\sqrt{n}\, x\sqrt{n}$, and each cell content is *k***m* bit long. The length of the modulus is P bits, and *f*(P) is the computation function for the P bit. Our scheme shows improvement over [5] in the user communication when $\sqrt{n} > k$. Our technique has downside in the server computation which is caused by the double PIR. Nevertheless, the reduction in the server transmission and user computation cost due to the double PIR makes up for the downside. Since $\sqrt{n}$ is likely to be greater than *k* our scheme will always perform better.

Table 7 Computation complexity for Server and Client

Type	Our Scheme	Previous Scheme
User Communication	$O(2Pkm)$	$O(Pm\sqrt{n})$
User Computation	$O(f(P)km)$	$O(f(P)m\sqrt{n})$
Server Communication	$O(Pkm)$	$O(Pm\sqrt{n})$
Server Computation	$O(2(f(P)km\sqrt{n}))$	$O(f(P)km\sqrt{n})$

Comparison with Previous Technique

We conduct experiments to compare our design with the design of [15]. Table 8 shows the steps involved in the two techniques. We did not consider offline activities, i.e., events that happens prior to a user declaring her intention for the nearest neighbor. Such events includes: creating a square grid, super-imposing Voronoi diagram over the grid, and mapping all the POI to Hilbert value. In the case of [15] this activity will be using Voronoi to divide road network into cells with centered Central Server (CS). Both designs were implemented in ns3 simulator. We implemented [15] based on author's description, the algorithm provided, and as we understood it. The design as described in [15] involves a three-way process. Using Voronoi graph the authors partitions the entire area into cells each centered with a CS. A user receives several broadcasting messages from neighboring CSs and determines the cell she is heading to. The user computes the distances between each CS and herself and registers to the CS with minimum distance from her. Registration information includes user location, destination, and history velocity. After successful registration to a CS the user will report the location periodically to the CS and drops broadcasting messages from the CS until the user gets out of the cell or registers to the next cell. CS makes moving direction prediction for the user using user history velocity. It then organizes *k* users to form a cooperative *k*-anonymity group according to the user's moving trend. CS takes its own location and a neighbor CS location which the user is heading to as two elementary anchors and chooses other continuous anchors in the line segment between them. CS picks an anchor from anchor sequence to replace user's location and send a snapshot query of a continuous query request with *k* queries to LBS Provider (LSP). The LSP uses the anchor point to perform incremental nearest neighbor search (INN) and returns its results to the CS. The CS then refines the result and then returns *k*-POI to the user.

For comparison we measure the response time (i.e., from initialization to the time user receives the POI). We use the same travelling path in simulating the two designs. We compare the time for the initial response and then subsequent responses until query terminates. For the client and server applications in ns3 we use same parameters as in our previous experiment described earlier in this chapter. We compare the work of [15] with three different sizes of modulus that we used for encryption in our technique. Table 9 shows the results. As the results show, initially, as the size of modulus increases, [15] performs better. However, our design performs better throughout the life of the query as the user continuously receives nearest POI. We believe that one of the contributing factors to the high cost of [15] is due to the fact that the design is basically a snapshot query rolled into a continuous query. Almost similar operations are performed throughout the segment by both client and server. In our design after the initial operation server queues the *k*-POI and sends a single POI to the user at every transition time interval without the user issuing another query. Also, another downside of [15] is the *k*-anonymity requirement. There are chances that a query may never be answered if it did not satisfy *k*-anonymity. Also user privacy is at the mercy of the CS.

Table 8 Steps in CNNQ for the compared technique

Our Design	Previous Design
User initiates a query	CSs sends out broadcasting messages
Server sends the grid, the key, *k,* and d_{avg} to the user	User receives several neighbor CS broadcasting messages, and determines the cell she is heading to, and then computes the distance between each CS and herself, then register to CS with minimum distance from her
User identifies the cell it belongs to and uses its *v* to find t_t where the nearest neighbor change will occur. Using the key user will generate query messages each targeting the k^{th} element in the path.	CS computes all the neighboring cell's weight for a user using history velocity of the user and stores them in prediction set in ascending order
Server runs PIR protocol on user request, and then creates a size *k* FIFO queue data structure of Z_α. The server adds items to the queue in the order it was represented in the tree. Closest Hilbert value is added first in that order until the farthest neighbor. Server returns the first Z_α to the user.	CS takes its own location and a neighbor CS location which the user is heading to as two elementary anchors and chooses other continuous anchors in the line segment between them. CS organizes *k* users to form a cooperative *k*-anonymity group according to their moving trends and sends *k* queries to LSP.
At each t_t interval server will update user with the next value in the queue until the end of the segment or user terminates the query	CS sends each snapshot query of continuous query request with an anchor which the user has not passed yet.
User then determine Z_α value by computing equation 6	CS performs Singoes algorithm to compute results for a user according to candidate sets returned from LSP

Table 9 Comparing response time using different modulus in our work with previous technique

Size of *k,* n and N	Our Design Initial Response Time (s)	Our Design Subsequent Response Time (s)	Previous Design Initial Response Time (s)	Previous Design Subsequent Response Time (s)
k=10, n=144, N=768bits	3.536	0246	4.283	4.283
k=10, n=144, N=1024bits	5.130	0360	4.283	4.283
k=10, n=144, N=2048bits	9.104	0448	4.283	4.283

Analysis of Large Prime on PIR Protocol

Private information retrieval protocol is based on the computational intractability of a large prime. Two large prime *p* and *q* are multiplied to produce modulus *N*. Some of the existing proposal for PIR encryption in LBS is performed with 768 bits modulus. However, the discovery in December of 2009 renders 768 bits encryption obsolete. A team of mathematicians, computer scientists, and cryptographers were able to deduce the two prime numbers that when multiplied produces 768 bits (232 digits) number using number field sieve (NFS) (identifying appropriate integers). The polynomial selection took half a year on 80 processors which represents 3% of the task. The sieving took more than two years on several hundreds of machines with more than 10^{20} operations (equivalent to 1500 years of computing on a single core 2.2GHz AMD Opteron) [98]. The previous record was the May 2005 factorization of 663-bit 200-digit number [99]. Though not many individual will have access to these types of clusters; however, this may be trivial for many corporations. The team stated that the overall effort is so low that 768 bit is no longer recommended for the protection of even data of little value. They also stated that factoring 768 bit is about several thousand times harder than 512 bit, and they expect factoring 1024 bit to be about a thousand times harder than 768 bit. There are so many prime numbers of the same size and below in a given key length. Therefore, factoring the key is directly proportional to the key length. ECRYPT II [100] recommendation on key length shown in Table 10 is based on desired security level and expected life-span. We conduct experiment using different key sizes to analyze the computational impact of the size of the keys.

Table 10 Recommended minimal key sizes based on security level

Protection	Modulus (N) Bits
Smallest general purpose level (2015-2015)	1248
Legacy standard level (2015-2020)	1776
Medium term protection (2015-2030)	2432
Long term protection (2015-2040)	3248
Foreseeable future	15424

Experimental Results

In this section, we present the results of the experiment conducted in LBS nearest neighbor query using different key sizes. The algorithm was implemented in c/c++ languages running on Window 7 machine with win64 OpenSSL version 1.0.2. We examine the computation cost on the user and the server to understand the tradeoff between performance and security. For the experiment we use five different modulus 512 bits (78 digits), 768 bits (116 digits), 1024 bits (155 digits), 2048 bits (309 digits) and 4096 bits (617 digits). We use the *BN_generate_prime_ex()* of the openssl to generate the prime numbers. For measuring time we averaged 100 runs. We first examine the impact on the user. We measure the computation time for the user to generate the modulus N which it then sends to the server together with the query message so that we can observe the impact of different key sizes. The result is shown in Figure 36. The figure shows increase in execution time as the size of the key increases, and increases exponentially as the size of the key gets larger.

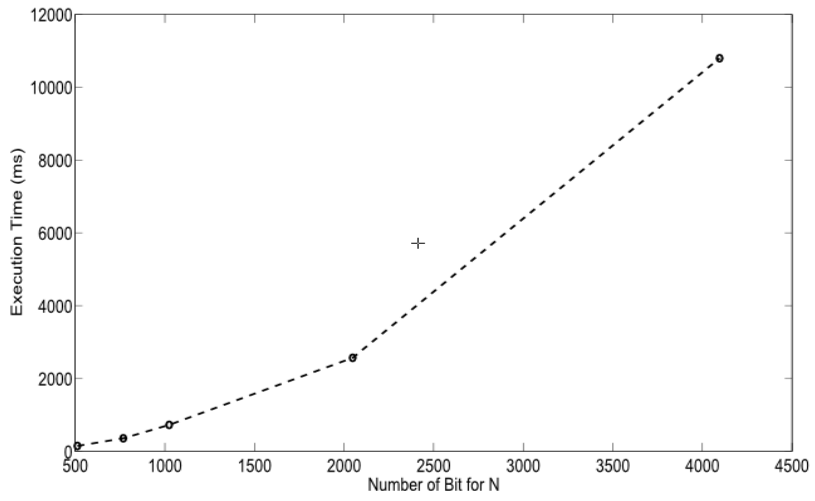


Figure 36 User creating a query message with modulus N

We also conduct experiment to determine the execution time on the server for running PIR protocol on different sizes of *n* when the modulus N is set to 512 bits. The observation shown in Figure 37 shows the impact of *n* on PIR protocol. The result shows that when *n* is small the impact on the execution time is minimal, but the execution time increases as *n* increases.

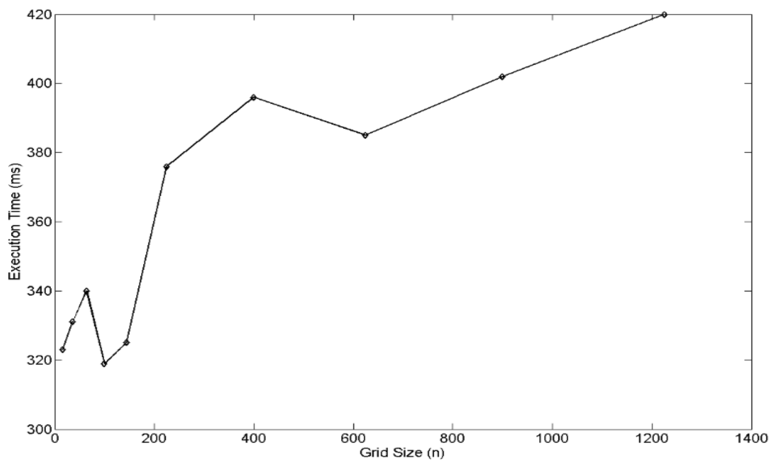


Figure 37 Server running PIR protocol with constant modulus and varying n

We also conduct experiment to determine the computation cost on the server for running PIR protocol on user query using different sizes of *N*. As Figure 38 shows, we did not observe significant increase in the execution time for the same value of *n* as N increases. The size of *n* definitely has impact on the cost especially as *n* increases, however, for the same *n*, the impact is minimal. We also observe that increasing the size of the key has more impact on the user than the server. Based on this observation we recommend that key size should be chosen based on user privacy requirement.

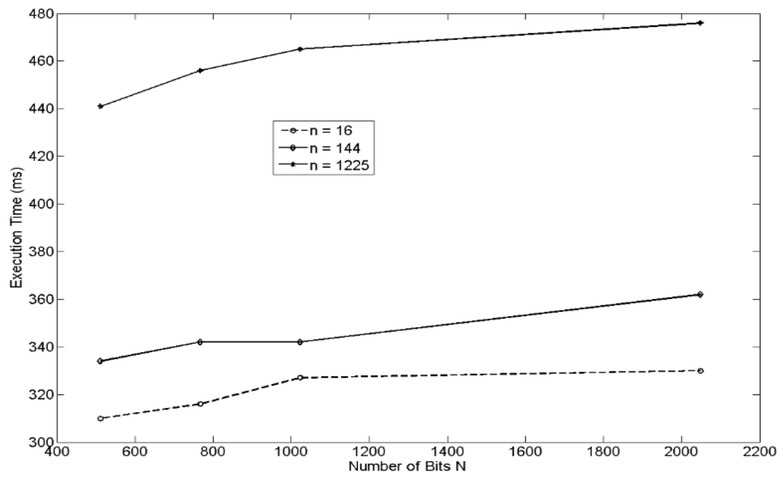


Figure 38 Server running PIR protocol with different key size and three different n

CHAPTER SEVEN CONCLUSION AND FUTURE WORK

Conclusion

In this dissertation, we explore techniques to protect user private information in location-based services nearest neighbor query. We provide solutions for snapshot nearest neighbor query and continuous nearest neighbor query models. In each model, we present theoretical concepts and the design implementations, and we perform experiments to show the applicability. In SNNQ we use space mapping technique to create neighborhood for each point of interest in an area by using the concept of sets in a topological space. The design allows a user in a location contained in an area to relate the location with the position of the information in the database which is represented as a square matrix. We provide database that scales with the size of the objects in the area which allows a user to relate the position with location of information in the database and for scalability. A user requests from the server a specific POI based on the location. Server runs private information retrieval protocol twice on the data to prevent database from learning user requested information, and the server releases to the user the only requested information. The server is not able to unmask the user personal information due to computational intractability of a large prime. We provide statistical analysis of our design with respect to complexity and execution time. Our experimental analysis shows an efficient scheme that provides great architecture for scaling.

To address the slow response time of the CPU observed in our snapshot query with the increase in the size of the grid and objects in the area, we adopt parallel implementation of our algorithm using GPU and CUDA interface, and we provide analysis of the implementation. We compare the execution time of the CPU serial implementation against GPU-CUDA parallel implementation. GPU shows speedup over CPU. Improvement obtained in the GPU shows the impact of the massive parallelism achievable in GPU. We study parameter tuning and its impact on run time and provide analysis that can be leveraged by researchers of parallel programming using GPU and CUDA API. We provide analysis for choosing thread and block size. Finally, we provide a model for predicting GPU execution time.

As an alternative, we explore the viability of a hardware in providing data privacy by using secure coprocessor in processing private location-based services snapshot nearest neighbor query. We create data structures and applications in the coprocessor to allow for oblivious computation. We then utilize ORAM to access external memory. We perform experiment by emulating the secure coprocessor to determine network transmission cost and complexity, and we compare the design with the software approach. Hardware approach shows an improvement over software approach as the size of the object and grid increases. The cost for the secure coprocessor remains constant in all cases while in the software approach the cost is related to the size of the object and grid.

We then extend our privacy solution to continuous nearest neighbor query by addressing moving query static object and moving query moving object types of queries. For our design, we combine Voronoi diagram and Hilbert curve order to create object relationships and proximity. The technique allows a user to determine the transition point at which to receive the next nearest neighbor POI from the server based on the velocity. It also allows a user to authenticate server response using a flag with true or false value. We conduct evaluation on the user and the server transmission time, and the results show efficient and scalable design. We conduct experiments on the response time and compare with similar technique. The results show that our design reduces the response time. We also perform complexity analysis and compare with similar design as applicable. The results for the most part show improvement over similar design. Finally, we conduct analysis on effect of large prime on PIR protocol. The results show that increasing the size of the key has more impact on the user than the server; nevertheless, size of the grid has impact on the server since we observe increase in response time as the grid size increases.

Ways for increasing computational capabilities of machines continue to evolve, a single approach alone may no longer be sufficient to offer information protection. Hybrid approach that combines hardware and software may provide better alternative. Advance in computing technology will always help the good and the bad guys. Cat and Mouse game will always exist between cryptographers and adversaries. What might be secure today may no longer be secure tomorrow; therefore, any security design is time relative.

Future Work

In a moving query moving object continuous nearest neighbor query, we encounter frequent updates due to change in object position. The constant change in position results in frequent computation to determine current nearest neighbor to a user. One can improve on this to reduce the response time by introducing parallelization through GPU-CUDA interface to the continuous nearest neighbor query. A design that will allow each thread to independently handle individual update and computation can be explored. Voronoi diagram creation and Hilbert curve mapping are some of the task that can be explored with GPU computation. Each scalar processor can be assigned subsets of the grid, and the threads can be created to handle individual object computation.

Another direction for future work will be to incorporate into location-based services nearest neighbor query the four basics primitives of cryptography [101], i.e., integrity (access only to authorized users), confidentiality (detecting tampered information on transit), authentication (Ensuring message origin), and non-repudiation (un-deniability of action).

In the protocol designed for the secure processor we use RSA encryption. The security of RSA keys rests on the difficulty of factoring the product of two large prime. However, though extremely difficult, breakthrough in the past few years has shown that products of two large prime can be determined through number field sieve. Also RSA is said to be vulnerable to Shor's algorithm. One can explore NTRU as an alternative to RSA. It is based on the algebraic structure of polynomial rings and it is said to be safe from Shor's algorithm attack. The hard problem is based on the short vector problem (finding a short vector in a lattice) [102]. To create the key, two chosen polynomials must satisfy the additional requirement that it has inverses modulo q and modulo p. NTRU technique can be used to generate the key. Anybody interested in NTRU can read [102].

Another area worth exploration is the power consumption. Mobile device has limited power supply at any given time, possible directions for future work will include energy consumption by the mobile device. Energy measurement methodology can be applied to determine variations in energy consumption due to the design, and protocol like TailEnder [103] can be used to handle energy consumption issues.

LIST OF REFERENCES

[1] Goodrich Ryan, "Location-Based Services: Definition & Examples," [Online]. Available: http://www.businessnewsdaily.com/5386-location-based-services.html. [Accessed 16 October 2013].

[2] Jensen Cory, "Location Based Services (LBS)," [Online]. Available: http://www.techopedia.com/definition/12888/location-based-services-lbs. [Accessed 10 August 2014].

[3] L. Liu and B. Gedik, "Protecting location privacy with personalized k-anonymity: Architecture and algorithms," *In Proceedings of the IEEE Transactions on Mobile Computing*, vol. 7, pp. 1-18, January 2008.

[4] L. Liu and B. Gedik, "Location privacy in mobile systems: A personalized anonymization model," *In Proceedings of the ICDCS International Conference on Distributed Computing Systems*, pp. 620-629, 6-10 June 2005.

[5] G. Ghinita, P. Kalnis, A. Khoshgozaran, C. Khoshgozaran and K. Tan, "Privacy queries in location based services: Anonymizers are not necessary," *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 121-132, 10-12 June 2008.

[6] P. P. Halkarnikar, A. P. Chougale, H. P. Khandagale and P. P. Kulkarni, "Parallel K-Nearest Neighbor Implementation on Multicore Processors," *In Proceedings of the 2012*

International Conference on Radar, Communication and Computing (ICRCC), pp. 221-223, 21-22 December 2012.

[7] S. W. Smith and D. Safford, "Practical server privacy with secure coprocessors," *IBM Systems Journal - End-to-end security*, vol. 40, no. 3, pp. 683 - 695 , March 2001.

[8] D. Grawrock, "The Intel Safer Computing Initiative Building Blocks for Trusted Computing," D. B. Spencer, Ed., Richard Bowles Intel Press, 2006, p. 293.

[9] G. E. Suh, D. Clarke, B. Gassend, M. V. Dijk and S. Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," *In proceedings of the 17th annual International Conference on Supercomputing*, pp. 160-171, 2003.

[10] U. Hengartner and D. R. Cheriton, "Hiding Location Information from Location-Based Services," *In Proceedings of the IEEE 14th International Conference on Mobile Data Management*, pp. 268 - 272, May 2007.

[11] X. Yin, Z. Ding and J. Li, "Moving Continuous K Nearest Neighbor Queries in Spatial Network Databases," *In Proceedings of the WRI World Congress on Computer Science and Information Engineering*, vol. 4, pp. 535-541, April 2009.

[12] H. G. Elmongui, M. F. Mokbel and W. G. Aref, "Continuous aggregate nearest neighbor queries," *Journal on Advances of Computer Science for Geographic Information Systems*, vol. 17, pp. 63-95, January 2013.

[13] Y. Tao, D. Apadias and Q. Shen, "Continuous nearest neighbor search," *In Proceedings of the 28th international conference on Very Large Data VLDB,* pp. 287-298, August 2002.

[14] J.-L. Huang and C.-C. Huang, "A Proxy-Based Approach to Continuous Location-Based Spatial Queries in Mobile Environments," *In Proceedings of the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING,* vol. 25, no. 2, pp. 260-273, February 2013.

[15] C. Ma, C. Zhou and S. Yang, "A Voronoi-Based Location Privacy-Preserving Method for Continuous Query in LBS," *International Journal of Distributed Sensor Networks,* 7 October 2014.

[16] B. Zheng, W.-C. Lee and D. L. Lee, "On Searching Continuous k Nearest Neighbors in Wireless Data Broadcast Systems," *In Proceedings of the IEEE Transactions on Mobile Computing,* vol. 6, no. 7, March 2007.

[17] G. Ghinita, "Understanding the privacy efficiency trade-off in location based queries," *In Proceedings of the SIGSPATIAL ACM International Workshop on Security and Privacy in GIS and LBS,* pp. 1-5, November 2008.

[18] H. Shin, V. Atluri and J. Vaidya, "A profile anonymization model for privacy in a personalized location based service environment," *In Proceedings of the 9th International Conference on Mobile Data Management,* pp. 73-80, 27-30 April 2008.

[19] L. Liu and B. Gedik, "A customizable k-anonymity model for protecting location privacy," *In Proceedings of the ICDCS International Conference on Distributed Computing Systems,* pp. 620-629, April 2004.

[20] F. Olumofin, P. K. Tysowski, I. Goldberg and U. Hengartner, "Achieving Efficient Query Privacy for Location Based Services," *In Proceedings of the PETS 10th International Conference on Privacy Enhancing Technologies,* vol. 6205, pp. 93-110, 21-23 July 2010.

[21] M. F. Mokbel, C.-Y. Chow and W. G. Aref, "The New Casper: Query Processing for Location Services without Compromising Privacy," *In Proceedings of the 32nd VLDB International Conference on VeryLlarge Data Bases,* pp. 763-774, January 2006.

[22] P. Kalnis, G. Ghinita, K. Mouratidis and D. Papadias, "Preventing location-based identity inference in anonymous spatial queries," *In Proceedings of the IEEE Transactions on Knowledge and Data Engineering,* vol. 19, pp. 1719-1733, December 2007.

[23] M. Gruteser and D. Grunwald, "Anonymous usage of location-based services through spatial and temporal cloaking," *In Proc. ICMSAS 1st International Conference on Mobile Systems, Applications and Services Mobisys'03,* p. 31–42, 2003.

[24] M. Kohlweiss, S. Faust, L. Fritsch, B. Gedrojc and B. Preneel, "Efficient oblivious augmented maps: Location-based services with a payment broker," *In Proceedings of the ACM 7th International Conference on Privacy Enhancng Technologies,* pp. 77-94, 20-22 June 2007.

[25] F. Z. R. Cheng, "An improved privacy protocol in location based service," *In the Proceedings of the International Conference on Information Engineering and Computer Science*, pp. 1-4, 19-20 December 2009.

[26] R. Vishwanathan, *EXPLORING PRIVACY IN LOCATION-BASED SERVICES USING CRYPTOGRAPHIC PROTOCOLS,* PhD Thesis, Universty of North Texas, 2011.

[27] E. Kushilevitz and R. Ostrovsky, "Replication Is Not Needed: Single Database, Computationally-Private Information Retrieval," *In Proceedings of the 38th Annual Symposium on Foundations of Computer Science,* pp. 364-373, 20-22 October 1997.

[28] A. Papadopoulos and Y. Manolopoulos, "Parallel Processing of Nearest Neighbor Queriesin Declustered Spatial Data," *In Proceedings of the 4th ACM international workshop on Advances in geographic information systems*, pp. 35-43, Novemeber 1996.

[29] J. Pan, C. Lauterbach and D. Manocha, "Efficient Nearest-Neighbor Computation for GPU-based Motion Planning," *In Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS),* pp. 2243 - 2248, 18-22 October 2010.

[30] V. Garcia, E. Debreuve and M. Barlaud, "Fast k Nearest Neighbor Search using GPU," *In Proceedings of the IEEE Computer Society International Conference on Computer Vision and Pattern Recognition Workshops,* pp. 1 - 6, 23-28 June 2008.

[31] A. Iliev and S. Smith, "PRIVATE INFORMATION STORAGE WITH LOGARITHMIC-SPACE SECURE HARDWARE," *In Proceedings of the 3rd Working Conference on Privacy and Anonymity in Networked and Distributed Systems*, pp. 201-216, 2004.

[32] A. Iliev and S. W. Smith, "Protecting Client Privacy with Trusted Computing at the Server," *Journal of the IEEE Security and Privacy*, vol. 3, pp. 20-28, April 2005.

[33] X. Yu, C. W. Fletcher, L. Ren, M. V. Dijk and S. Devadas, "Efficient Private Information Retrieval Using Secure Hardware," MIT CSAIL CSG Technical Memo 509, April 2013.

[34] S. Papadopoulos, S. Bakiras and D. Papadias, "Nearest neighbor search with strong location privacy," *Journal Proceedings of the VLDB Endowment* , vol. 3, no. 1-2, pp. 619-629 , September 2010.

[35] A. Khoshgozaran, H. Shirani-Mehr and C. Shahabi, "SPIRAL:A Scalable Private Information Retrieval Approach to Location Privacy," *In Proceedings of the 9th International Conference on Mobile Data Management Workshops*, pp. 55 - 62, 27-30 April 2008.

[36] E. Stefanov, M. V. Dijk, H. T.-H. Chan, E. Shi, C. Fletcher, L. Ren, X. Yu and S. Devadas, "Path ORAM: An Extremely Simple Oblivious RAM Protocol," *In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 299-310, January 2014.

[37] C.-Y. Chow, M. F. Mokbel and W. G. Agref, "Casper* Query Processing for Location Services without Compromising Privacy," *In Proceedings of the ACM Transactions on Database Systems (TODS),* vol. 34, no. 4, pp. 1-45, December 2009.

[38] C.-Y. CHow and M. F. Mokbel, "Enabling Private Continuous Queries For Revealed User Locations," *In Proceedings of the 10th international conference on Advances in spatial and temporal databases,* pp. 258-273, 16-18 July 2007.

[39] A. Yamamura and T. Saito , "Private information retrieval based on the Subgroup Membership Problem," *In Proceedings of the 6th Australasian Conference on Information Security and Privacy* , vol. 2119, pp. 206-220, 2001.

[40] S. Goldwasser and M. Bellare, "Introduction to Modern Cryptography Lecture Notes," July 2008. [Online]. Available: http://cseweb.ucsd.edu/~mihir/papers/gb.pdf. [Accessed 22 August 2013].

[41] E. Bach and J. Shallit, Algorithmic Number Theory, vol. 1 Efficient Algorithms, MIT Press, 1996, p. 512.

[42] K. S. McCurley, "Odds and Ends from Cryptology and Computational Number Theory. in Cryptography and Computational Number Theory," *In Proceedings of the Symposia in Applied Mathematics,* vol. 42, pp. 145-166, August 1990.

[43] C. Asanya and R. Guha, "Anonymous retrieval of k-nn poi in location based services (LBS)," *In Proc. WORLDCOMP International Conference on Security and Management,*

SAM '13, pp. 294-300, 22-25 July 2013.

[44] USBGN, "Geographic names information system domestic and antarctic names state and topical gazetteer," October 2013. [Online]. Available: http://geonames.usgs.gov/domestic/download_data.htm. [Accessed 5 May 2014].

[45] C. o. Boston., September 2013. [Online]. Available: http://www.cityofboston.gov/neighborhoods/. [Accessed 5 May 2014].

[46] C. o. Boston, "Data boston: Active food establishment. Electronic Publication," 2013. [Online]. Available: https://data.cityofboston.gov/Permitting/Active-Food-. [Accessed 5 May 2014].

[47] R. W. Sinnot, "Virtues of the Haversine: Sky and Telescope," vol. 68, 1984.

[48] NVIDIA, "Nvidia's next generation cuda compute architecture:kepler tm gk110. 1.0.," NVIDIA, [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA NVIDIAKepler-. [Accessed 6 April 2013].

[49] F. Gieseke, J. Heinermann, C. Oancea and C. Igel, "Buffer k-d Trees: Processing Massive Nearest Neighbor Queries on GPUs," *In Proceedings of the 31 st International Conference on Machine Learning,* vol. 32, pp. 172-180, June 2014.

[50] S. Liang, Y. Liu, C. Wang and L. Jian, "Design and Evaluation of a Parallel K-Nearest Neighbor Algorithm on CUDA-enabled GPU," *In Proceedings of the IEEE 2nd*

Symposium on Web Society, pp. 53-60, 16-17 August 2010.

[51] C. Silvestri, F. Lettich, S. Orlando and C. S. Jensen, "GPU-based Computing of Repeated Range Queries over Moving Objects," *In Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing,* pp. 640-647, 12-14 February 2014.

[52] J. L. Rodengen and J. VanZile, "Amdahl Law," in *The Legend of Amdahl* , 1 ed., Write Stuff Syndicate, 2000, p. 144.

[53] S. Cook, A Developer's Guide To Parallel computing with GPUs, Waltham, MA: Morgan Kaufman, 2013.

[54] J. Cazalas, *Efficient and Scalable Evaluation of Continuous Spatio-Temporal Queries in Mobile Computing Environments,* Orlando, Florida: PhD Thesis, University of Central Florida, 2012.

[55] J. D. Owens and Y. Zhang, "A quantitative performance analysis model for gpu architectures," *In Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA),* p. 382–393, 12-16 February 2011.

[56] J. Yang, Y. Zhang and L. Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering," *In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture,* pp. 351 - 360, December 2003.

[57] T. Gilmont, J. Legat and J. -J. Quisquater, "Enhancing security in the memory management unit," *In Proceedings of the 25th EUROMICRO International Conference,* vol. 1, pp. 449 - 456, 8-10 September 1999.

[58] M. Kuhn, "The TrustNo1 Cryptoprocessor Concept," Purdue University, April, 1997.

[59] B. Yee, *Using Secure Coprocessors*, Pittsburgh, PA: School of Computer Science Carnegie Mellon University, 1994.

[60] D. Boneh, R. A. DeMillo and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," *In Proceedings of the International Conference on the Theory and Application of Cryptographic,* vol. 1233, pp. 37-51, 11-15 May 1997.

[61] J. Kong, O. Aciicmez, J.-P. Seifert and H. Zhou, "Deconstructing new cache designs for thwarting software cache-based side channel attacks," *In the Proceedings of the 2nd ACM workshop on Computer security architectures,* pp. 25-34, 31 October 2008.

[62] J. Kong, O. Aciicmez, J.-P. Seifert and H. Zhou, "Architecting against Software Cache-Based Side-Channel Attacks," *Journal of IEEE Transactions on Computers,* vol. 62, no. 7, pp. 1276-1288, 24 May 2013.

[63] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh and D. Ponomarev, "Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks," *Journal of ACM Transactions on Architecture and Code Optimization,* vol. 8, no. 4, p. 21,

1 January 2012.

[64] O. Goldreich, "Towards a Theory of Software Protection and Simulation by Oblivious RAMS," *In Proceedings of the 19th Annual ACM Symposium on Theory of Computing,* pp. 182-194, 1 January 1987.

[65] R. Ostrovsky , "Efficient computation on oblivious RAMs," *In Proceedings of the 22nd Annual ACM Symposium on Theory of Computing,* pp. 514-523, 1990.

[66] O. Goldreich and R. M. Ostrovsky, "Software protection and simulation on oblivious RAMs," *Journal of Association for Computing Machinery,* vol. 43, pp. 431-473, 1 May 1996.

[67] E. Shi, T. -H. Hubert Chan, E. Stefanov and M. Li, "Oblivious RAM with O((logN)3) Worst-Case Cost," *In Proceedings of the 17th international conference on The Theory and Application of Cryptology and Information Security*, vol. 7073, pp. 197-214, 4 December 2011.

[68] E. Stefanov, E. Shi and D. Song, "Towards Practical Oblivious RAM," *In Proceedings of Network and Distributed System Security Symposium,* 18 Dec 2012.

[69] D. Boneh, D. Mazieres and R. A. Popa, "Remote Oblivious Storage: Making Oblivious RAM Practical," MIT, Cambridge, MA., 2011.

[70] M. T. Goodrich, M. D. Mitzenmacher, O. Ohrimenko and R. Ohrimenko, "Oblivious RAM

simulation with efficient worst-case access overhead," *In Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop,* pp. 95-100, 25 July 2011.

[71] C. Fletcher, M. V. Dijk and S. Devadas, "A Secure Processor Architecture for Encrypted Computation on Untrusted Programs," *In Proceedings of the 7th ACM Workshop on Scalable Trusted Computing,* pp. 3-8, 5 October 2012.

[72] K. Scarfone, M. Souppaya and M. Sexton, "Guide to Storage Encryption Technologies for End User Devices," National Institute of Standards and Technology, Gaithersburg, MD 20899, November 2007.

[73] B. Zheng, W.-C. Lee and D. L. Lee, "Search Continuous Nearest Neighbors On the Air," *In Proceedings of the 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services,* pp. 236 - 245, 22-26 August 2004.

[74] Q. Zhao, Y. Lu and Y. Zhang, "Predictive Continuous Nearest-Neighbor Query Processing in Moving-Object Databases," *In Proceedings of the International Conference on Wireless Communications, Networking and Mobile Computing,* pp. 3019 - 3022, 21-25 September 2007.

[75] A. Okabe, B. Boots, K. Sugihara and S. N. Chiu, Spatial Tessellations: Concepts and Applications of Voronoi Diagrams, 2 ed., John Wiley and Sons Ltd., 2000, p. 696.

[76] G. Zhao, K. Xuan, W. Rahayu, D. Taniar, M. Safar, M. L. Gavrilova and B. Srinivasan, "Voronoi-Based Continuous k Nearest Neighbor Search in Mobile Navigation," *In*

Proceedings of the IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS, vol. 58, no. 6, pp. 2247-2257, 12 May 2011.

[77] F. Aurenhammer, "Voronoi Diagrams —A Survey of a Fundamental Geometric Data Structure," *Journal of ACM Computing Surveys,* vol. 23, no. 3, pp. 345-405, September 1991.

[78] U. Demiryurek and C. Shahabi, "Indexing Network Voronoi Diagrams," *In Proceedings of the 17th international conference on Database Systems for Advanced Applications,* pp. 526-543, 15 April 2012.

[79] M. Kolahdouzan and C. Shahabi, "Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases," *In Proceedings of the 13th International Conference on Very Large Data Bases,* vol. 30, pp. 840-851, 31 August 2004.

[80] M. Berg, M. Kreveld, M. Overmars and O. Cheong, Computational Geometry: Algorithms and Applications, 3 ed., Springer-Verlag, 2008, p. 386.

[81] H.-l. Chen and Y.-i. Chang, "Neighbor-Finding Based on Space-Filling Curves," *Journal of Information Systems,* vol. 30, no. 3, pp. 205-226, 1 May 2005.

[82] N. J. Rose, "Hilbert-Type Space-Filling Curves," North Carolina State University, Chicago, 2001.

[83] "Applications and mapping algorithms," 18 October 2014. [Online]. Available:

http://en.wikipedia.org/wiki/Hilbert_curve. [Accessed 16 October 2014].

[84] B. Moon, H. V. Jagadish, C. Faloutsos and J. H. Saltz, "Analysis of the Clustering Properties of the Hilbert Space-Filling Curve," *Journal of IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 1, pp. 124-141, January 2001.

[85] W.-S. Ku, L. Hu, C. Shahabi and H. Wang, "Query Integrity Assurance of Location-Based Services Accessing Outsourced Spatial Databases," in *Advances in Spatial and Temporal Databases*, vol. 5644, Aalborg, Denmark, Springer Berlin Heidelberg, 2009, pp. 80-97.

[86] T. Theen-Theen, L. Davis and R. Thurimella, "One Dimensional Index for Nearest Neighbor Search," Dept. of Computer Science, University of Maryland, College Park MD 20742, USA, 1999.

[87] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," *In Proceedings of the ACM SIGMOD International Conference of Managament of Data*, vol. 14, no. 2, pp. 47-57, June 1984.

[88] P. K. Agarwal, M. de Berg, J. Gudmundsson, M. Hammar and H. J. Haverkort, "Box-Trees and R-Trees with Near-Optimal Query Time," *In Proceedings of the 17th Annual Symposium on Computational Geometry*, pp. 124-133, 1 June 2001.

[89] L. Arge, M. De Berg, H. Haverkort and K. Yi , "The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree," *Journal of ACM Transactions on Algorithms*

(TALG), vol. 4, no. 1, pp. 9:1-30, March 2008.

[90] H. Samet, Foundations of Multidimensional and Metric Data Structures, San Mateo, CA.: Morgan Kaufmann, 2006, p. 1024.

[91] D. Lin, *Indexing and Querying Moving Objects Databases,* Singapore: National university of Singapore, 2006.

[92] B. C. Ooi, K. L. Tan and C. Yu, "Frequent Update and Efficient Retrieval: an Oxymoron on Moving Object Indexes?," *In Proceedings of the 3rd International Conference on Web Information Systems Engineering (Workshops),* pp. 3-12, 11 December 2002.

[93] D. V. Kalashnikov, S. Prabhakar, W. G. Aref and S. E. Hambrusch, "Efficient Evaluation of Continuous Range Queries on Moving objects," *In Proceedings of the 13th International Conference on Database and Expert Systems Applications DEXA,* pp. 731-740, 2002.

[94] S. Saltenis, C. S. Jensen, S. T. Leutenegger and M. A. Lopez, "Indexing the Positions of Continuously Moving Objects," *In Proceedings of the ACM SIGMOD International Conference on Management of Data,* vol. 29, no. 2, pp. 331-342, June 2000.

[95] C. S. Jensen, D. Lin and B. C. Ooi, "Query and Update Efficient B+-Tree Based Indexing of Moving Objects," *In Proceedings of the 30th International Conference on Very Large Data Bases,* vol. 30, pp. 768-779, August 2004.

[96] A. Civilis, C. S. Jensen, J. Nenortaite and S. Pakalnis, "Efficient tracking of moving objects with precision guarantees.," *In Proceedings of the 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services*, pp. 164 - 173, 22-26 August 2004.

[97] Devsaran, "Sequoia Points," Chorochronos Groups, 2015. [Online]. Available: http://www.chorochronos.org/?q=node/58. [Accessed 15 January 2015].

[98] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. t. Riele, A. Timofeev and P. Zimmermann, "Factorization of a 768-Bit RSA Modulus," *In Proceedings of the 30th Annual Conference on Advances in Cryptology*, p. 333–350, 15-19 August 2010.

[99] F. Bahr, M. Boehm, J. Franke and T. Kleinjung, "Factorization of RSA-200," May 2005. [Online]. Available: http://www.loria.fr/~zimmerma/records/rsa200. [Accessed 10 January 2015].

[100] D. Giry, "Cryptographic Key Length Recommendation," ECRYPT II, 26 February 2015. [Online]. Available: http://www.keylength.com/en/3/. [Accessed 6 March 2015].

[101] C. Sheedy, *Privacy Enhanced Protocols using Pairing Based Cryptography*, Dublin, Ireland: Dublin City University, January, 2010.

[102] J. Hoffstein, D. Lieman, J. Pipher and J. H. Silverman, "NTRU: A Public Key Cryptosystem," Protocols from other Families of Public-Key Algorithms, Technical

Report, October 1999.

[103] N. Balasubramanian, A. Balasubramanian and A. Venkataramani, "Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications," *In Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, pp. 280-293, 4-6 November 2009.

[104] A. Papadopoulos and Y. Manalopoulos, "Parallel Processing of Nearest Neighbor Queries," in *GIS '96 Proceedings of the 4th ACM international workshop on Advances in geographic information systems* , 1996.

[105] "Private information retrieval," 2012. [Online]. Available: http://en.wikipedia.org/wiki/Private_information_retrieval.

[106] Y. Huang and R. Vishwanathan, "Privacy preserving group nearest neighbour queries in location-based services using cryptographic techniques," pp. 1-5, 6-10 Decemeber 2010.

[107] A. R. Butz, "Alternative Algorithm for Hilbert's Space-Filling Curve," in *IEEE Transactions on Computers,*, April 1971.

[108] J. K. Lawder, "Calculation of Mappings Between One and n-dimensional Values Using the Hilbert Space-," August 15, 2000.

[109] R. Sion and C. Bogdan, "On the Computational Practicality of Private Information Retrieval," in *In Proceedings of the Network and Distributed Systems Security*

Symposium,2007. Stony Brook Network Security and Applied Cryptography Lab Tech Report, 2007.

[110] A. Beimel, Y. Ishai and T. Malkin, "Reducing the Servers' Computation in Private Information Retrieval: PIR with Preprocessing," *Journal of Cryptology*, vol. 17, no. 2, pp. 125-151, March 2004 .

[111] J. L. Kelley, General Topology, vol. 27, Springer Science & Business Media, June, 1975, p. 298.