PARALLEL, SCALABLE AND BANDWIDTH-OPTIMIZED COMPUTATIONAL PRIVATE INFORMATION RETRIEVAL

Ecem Ünal

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of
Master of Science

Sabancı University

August, 2014

PARALLEL, SCALABLE AND BANDWIDTH-OPTIMIZED COMPUTATIONAL PRIVATE INFORMATION RETRIEVAL

APPROVED BY:

Assoc. Prof. Dr. Erkay Savaş (Thesis Supervisor)	....
Assoc. Prof. Dr. Cem Güneri	....
Asst. Prof. Dr. Hüsnü Yenigün	....

DATE OF APPROVAL: ....

© Ecem Ünal 2014
All Rights Reserved

PARALLEL, SCALABLE AND BANDWIDTH-OPTIMIZED COMPUTATIONAL PRIVATE INFORMATION RETRIEVAL

Ecem Ünal

Computer Science and Engineering, Master's Thesis, 2014

Thesis Supervisor: Erkay Savaş

Abstract

With the current increase of interest in cloud computing, the security of user data stored in remote servers has become an important concern. Hiding access patterns of clients can be crucial in particular applications such as stock market or patent databases. Private Information Retrieval (PIR) is proposed to enable a client to retrieve a file stored in a cloud server without revealing the queried file to the server. In this work, we offer improvements to BddCpir, which is a PIR protocol proposed by Lipmaa. The original BddCpir uses Binary Decision Diagrams (BDD) as the data structure, where data items are stored at the sink nodes of the tree. First of all, we offer the usage of quadratic and octal trees instead, where every non-sink node has four and eight child nodes, respectively, to reduce the depth of the tree. By adopting more shallow trees, we obtain an improved server implementation which is an order of magnitude faster than the original scheme, without changing the asymptotic complexity. Secondly, we suggest a non-trivial parallelization method that takes advantage of the shared-memory multi-core architectures to further decrease server computation latencies. Finally, we show how to scale the PIR scheme for larger database sizes with only a small overhead in bandwidth complexity, with the utilization of shared-memory many-core processors. Consequently, we show how our scheme is bandwidth-efficient in terms of the data being exchanged in a run of the CPIR protocol, in proportion to the database size.

PARALEL, ÖLÇEKLENEBİLİR VE AĞ KULLANIMI İÇİN OPTİMİZE EDİLMİŞ HESABA DAYALI MAHREMİYET-KORUMALI BİLGİ ERİŞİMİ

Ecem Ünal

Bilgisayar Bilimleri ve Mühendisliği, Yüksek Lisans, 2014

Tez Danışmanı: Erkay Savaş

Özet

Bulut bilişime ilginin artmasıyla birlikte, uzak sunucularda saklanan kullanıcı bilgilerinin güvenliği önemli bir sorun haline gelmiştir. İstemcilerin erişim modellerini gizlemek, özellikle borsa veya patent veritabanı gibi uygulamalarda elzem olabilmektedir. Mahremiyet-Korumalı Bilgi Erişimi (PIR), bir istemcinin bulut sunucuda saklanan bir veri öğesini (örneğin bir dosya) sunucuya hangisine eriştiğini söylemeden elde etmesini sağlamak için tasarlanmış bir protokoldr. Bu tezde, Lipmaa tarafından önerilen bir PIR protokolü olan BddCpir üzerine iyileştirmeler sunulmuştur. Orijinal BddCpir, veri yapısı olarak, veri öğelerini uç düğümlerde depolayan İkili Karar Diyagramlarını (BDD) kullanmaktadır. Öncelikle, veri yapısı olarak BDD yerine dörtlü ve sekizli ağaçların kullanımını önerilmiştir. Bu tür ağaçlarda uç olmayan her düğümün sırasıyla dört ve sekiz alt düğümü olduğu için, daha az derinliği olan ağaçlar elde edilerek, sunucu performansı orijinal asimptotik karmaşıklığı değişmeden bir mertebe iyileştirilebilmektedir. İkinci olarak, sunucu işlem gecikmesini daha da azaltabilmek için paylaşımlı bellek kullanan çok çekirdekli işlemciler için tasarlanmış bir paralelleştirme yöntemi sunulmuştur. Üçüncü olarak da, bu tezde önerilen PIR protokolünün bant genişliğine yalnızca ufak bir ek yük ekleyerek nasıl ölçeklenebileceği gösterilmiştir. Son olarak, önerilen protokolün bir çalışmasında harcadığı bant genişliği bakımından, veri tabanı boyutuna oranla, ne kadar verimli olduğunun analizi yapılmaktadır.

to my beloved family...

Acknowledgements

This thesis would not have been possible without the support of my supervisor, committee, friends and family.

Foremost, I would like to express the deepest gratitude to my thesis supervisor Assoc. Prof. Erkay Savaş. The presented work existed and developed with the help of his ideas, immense knowledge as well as his guidance and encouragement. I also would like to thank my thesis jury, Asst. Prof. Dr. Hüsnü Yenigün and Assoc. Prof. Dr. Cem Güneri for their valuable suggestions and inquiries.

I am thankful to all the members of our Cryptography and Information Security Lab for the great environment they provided in terms of both research and friendship. Every one of them is important to me, but Naim Alperen Pulur has a special place among them. I am beyond grateful to his presence when I needed motivation the most; his unconditional support and help aided me during my writing process. In addition, I would like to thank my roommate Saime Burçe Özler, since she was always there for me during both good and rough times.

Last, but not least, I would like express my special appreciation and thanks to my parents and my brother. I would not be here without the unlimited love and support they provided throughout my life.

Contents

1 Introduction **1**

2 Background Work **4**

2.1 Cryptographic Properties . . . 5

2.1.1 Homomorphic Encryption . . . 5

2.1.2 Damgård-Jurik Cryptosystem . . . 6

2.2 Binary Decision Diagrams . . . 8

2.2.1 Properties of a BDD . . . 8

2.2.2 Quadratic and Octal Trees . . . 9

2.3 (2, 1) CPIR . . . 9

2.4 (n, 1) CPIR . . . 10

3 Problem Statement **14**

4 CPIR using Quadratic and Octal Trees **16**

4.1 Utilizing Quadratic Trees in CPIR . . . 16

4.1.1 (4, 1) CPIR . . . 17

4.1.2 (n, 1) CPIR with Quadratic Trees . . . 18

4.2 Utilizing Octal Trees . . . 20

4.2.1 (8, 1) CPIR . . . 20

4.2.2 (n, 1) CPIR with Octal Trees . . . 21

5 Parallelization of CPIR **24**

5.1 Client Side Parallelization . . . 24

5.2 Server Side Trivial Parallelization Algorithm . . . 26

5.3 Server Side Two-Degree Parallelization Algorithm . . . 28

5.4 Server Side Core-Isolated Parallelization . . . 31

6 Scalable CPIR for Parallel Implementations **33**

7 Communication and Computation Analysis **40**

7.1 Analysis of Communication Complexity 40

7.2 Analysis of Computational Complexity 42

7.2.1 Complexity of Parallel Implementation of Binary Tree 44

7.2.2 Complexity of Parallel Implementation of Quadratic Tree 47

7.2.3 Complexity of Parallel Implementation of Octal Tree 50

7.3 Analysis of Scalable CPIR 52

7.3.1 Communication Complexity 52

7.3.2 Computational Complexity 54

8 Implementation Results **56**

8.1 Client-Side Computations 56

8.2 Server-Side Computations 58

8.2.1 Serial Case 58

8.2.2 Parallel Case 59

8.2.3 Scalable CPIR 60

9 Comparison **61**

10 Conclusion **67**

List of Algorithms

1 Parallel client side computation for binary tree based $(n, 1)$ CPIR . . . 25
2 Parallel client side computation for quadratic tree based $(n, 1)$ CPIR . 25
3 Parallel client side computation for octal tree based $(n, 1)$ CPIR 26
4 Parallel server computation for binary $(n,1)$ CPIR v1 27
5 Parallel server computation for quadratic $(n,1)$ CPIR v1 27
6 Parallel server computation for octal $(n,1)$ CPIR v1 27
7 Parallel server computation for binary $(n,1)$ CPIR v2 28
8 Parallel server computation for quadratic $(n,1)$ CPIR v2 29
9 Parallel server computation for octal $(n,1)$ CPIR v2 30
10 Parallel server computation for binary $(n,1)$ CPIR v3 32
11 Client-side computation for binary tree-based Scalable CPIR 35
12 Server-side computation for binary tree-based Scalable CPIR 36

List of Figures

1 An example BDD constructed by server 9

2 A depth-2 quadratic tree implementing (16,1)-CPIR 19

3 Collapsing four subtrees into one tree 37

4 Modular exponentiation timings 44

5 Communication complexity of Scalable CPIR - Binary Tree Case 52

6 Communication complexity of Scalable CPIR - Quadratic Tree Case . . 53

7 Communication complexity of Scalable CPIR - Octal Tree Case 54

8 Bandwidth comparison, with 1024-bit data items 64

9 Bandwidth comparison, with $n = 1024$ variable sized data items 65

10 Bandwidth comparison, with $n = 2^{16}$ variable sized data items 66

List of Tables

1 The bandwidth requirements of the selection bits 41

2 Actual bandwidth costs of overall communication 42

3 Estimated timings of server-side computation 45

4 Estimation of timing values for serial and parallel implementations with different number of processor cores and number of synchronization points and their associated costs - Binary tree case (using GMP library on an Intel Xeon CPU E1650@3.50 GHz) 48

5 Estimation of timing values for serial and parallel implementations with different number of processor cores and number of synchronization points and their associated costs - Quadratic tree case (using GMP library on an Intel Xeon CPU E1650@3.50 GHz) 49

6 Estimation of timing values for serial and parallel implementations with different number of processor cores and number of synchronization points and their associated costs - Octal tree case (using GMP library on an Intel Xeon CPU E1650@3.50 GHz) 51

7 Estimated execution times of the hybrid method for various number of data items, number of cores, and speedup values over the normal parallel implementation; $l = 3$. (using GMP library on an Intel Xeon CPU E1650@3.50 GHz) 55

8 Estimated execution times of the hybrid method for various number of data items, number of cores and speedup values over the normal parallel implementation; $l = 4$. (using GMP library on an Intel Xeon CPU E1650@3.50 GHz) 55

9 Timings of client's selection bit encryptions 57

10 Timings of client's decryption of the final result 57

11 Timings of server computation - sequential 58

12 Timings of server computation - parallel 59

13 Timings of server computation - scalable 60

14 Comparison of bandwidth requirements 62
15 Ratio of exchanged information to database in different PIR schemes . 65

1 Introduction

In this age of big data, cloud computing has gained a significant importance. Instead of setting up their own servers, which is costly in terms of money and time, people are now renting cloud servers for their immense computation and storage capabilities. Although they are useful and easy to maintain, outsourcing to cloud servers arises the security concerns for the data stored in these cloud-powered systems. The cloud computing users would want not only the secrecy and integrity of their data guaranteed, but also their access patterns to be hidden. For instance, if a stock-market database is queried many times for the value of a certain stock, knowing the access frequencies may inadvertently affect their prices, which is an undesirable outcome. Hence, Private Information Retrieval (PIR) is introduced as a solution to this problem. PIR essentially enables the user to access one of its files without the server learning the requested file. Formally, a client that wants to retrieve f_x from a remote server storing a database $\mathcal{F} = (f_0, f_1, \ldots, f_{n-1})$, $f_x \in \mathcal{F}$, can accomplish this without revealing neither x nor f_x to the server using a PIR protocol.

The trivial solution to this problem would be the client downloading the whole database and selecting f_x among them. This would not be possible if the user had restrictions about the files it could access, which is the case for oblivious transfer, a similar concept in cryptographic literature [30]. Therefore, the fundamental requirement for an efficient PIR is a sublinear communication rate. In other words, the data exchanged between the client and the server must be asymptotically less than the database size.

The concept of private information retrieval first introduced by Chor et. al. in 1995 [6], and received serious attention. Afterwards, Computational PIR (CPIR), which bases the security of the protocol on a computationally difficult problem, is presented

in 1997 again by Chor [7]. There is also Information-Theoretic PIR (itPIR), which preserves the security of the client against computationally unbound servers. However, Chor et. al. proved that if the database is stored only in one server without any replication, the best itPIR protocol is the trivial one [6]. Therefore information theoretic security can only be achieved efficiently if there are more than one non-communicating servers. Contrarily, CPIR does not require such a replication as proved by Kushilevitz and Ostrovsky [19]. On the grounds of this information, this thesis is mainly interested in efficient single-server computational PIR protocols, thus PIR will imply CPIR henceforth.

CPIR protocols generally rely on the security of the underlying encryption scheme, therefore each of them employs a different computationally-difficult problem. In 1997, Kushilevitz and Ostrovsky suggested a CPIR scheme [19], utilizing Goldwasser-Micali public key cryptosystem [16], thus depending on the intractability of quadratic residuosity problem. Later, in 1999, the first polylogarithmic communication rated CPIR is presented by Cachin et. al., based on the number theoretic ϕ-hiding assumption, which is also introduced in the same paper [5]. There exist several other schemes based on lattice problems such as the ones constructed by Aguilar-Melchor and Gaborit [23, 24], or NTRU based protocol by Doroz, Sunar and Hammouri [10]. Furthermore, with the current interest and development in fully homomorphic encryption systems, there are some recent PIR schemes based on them [13, 35]. In addition to all these protocols, Lipmaa presented a scheme that combines a non-cryptographic data type, binary decision diagrams, and a probabilistic, additively homomorphic public key cryptosystem, Damgård-Jurik, into a bandwidth efficient protocol called BddCpir [20]. The security of BddCpir is also based on the same security assumption as the Damgård-Jurik cryptosystem, namely the complexity of the well studied decisional composite residuosity problem [9]. Many of the aforementioned schemes provide efficient techniques to speed up the server computation, but fail to provide a reasonable bandwidth performance [10, 23, 24]. On the other hand, Lipmaa's BddCpir is not one of the best schemes in terms of computational complexity.

Therefore, we offer the application of quadratic and octal trees, instead of binary

ones, to improve the BddCpir protocol in terms of computational complexity, while preserving the bandwidth efficiency. Afterwards, we define some non-trivial parallelization algorithms to utilize modern multi-core processors for further enhancement in server-side computations.

In particular, this work first starts by defining preliminary information such as homomorphic encryption, binary decision diagrams, Damgård-Jurik cryptosystem and Lipmaa's BddCpir in Chapter 2. Then, in Chapter 3, the properties that we aim to achieve in our improved methods are listed, thus stating the problem definition. Chapter 4 explains how quadratic and octal trees can be utilized in a PIR protocol, and shows the client is still able to correctly retrieve its requested data item. After defining the necessary protocols, Chapter 5 illustrates how they can utilize parallelization techniques to improve the overall computational complexity. In Chapter 6, a scalable CPIR is presented for databases with high number of data items. Once our methods are proposed, their analysis is presented in Chapter 7 in terms of both communication and computational complexities. To support our claims in the analysis part, Chapter 8 presents the implementation results and actual execution times of both our methods and BddCpir. Lastly, we compare the proposed schemes with similar protocols in the literature in Chapter 9 and conclude the thesis in Chapter 10.

2 Background Work

As it has been introduced in the first section, our proposed PIR scheme is based on Lipmaa's BddCpir protocol [21]. Therefore, in order to start defining our improvements, we first need to explain this protocol. BddCpir enables the client to query a server with a database of n files and be able to privately retrieve 1 file out of n. Therefore, $(n, 1)$ CPIR notation is also employed for this scheme and it will be more frequently used throughout this document.

$(n, 1)$ CPIR is based on Binary Decision Diagrams (often abbreviated as BDD), utilizes a more primitive (2, 1) CPIR scheme and requires a cryptosystem with specific properties. Particularly, the requirements state that it should be an additively homomorphic, length-flexible public key cryptosystem with randomized key generation and encryption algorithms [21]. Therefore, we will start by defining homomorphic ecnryption and then we will move on to Damgård-Jurik cryptosystem which satisfies the specified conditions.

After outlining the cryptosystem, we will continue with BDDs and demonstrate how they are used to store data in a server. In that subsection the preliminaries of our quadratic and octal tree methods are also given.

Once the preliminary data structures and encryption system are described, we can continue with (2, 1) CPIR, the basic scheme that is used to retrieve 1 file out of 2 files that are stored in the server. Since there are only 2 files in this case, the client will send 1 (encrypted) selection bit to select one of the two files and we will show how the server returns the selected file correctly without decrypting the selection bit. After that, we will show how to extend the (2, 1) CPIR into a generic $(n, 1)$ scheme while still using the same structures and protocols as the building blocks.

2.1 Cryptographic Properties

BddCpir protocol and our improved version of it both function because of the underlying properties of the cryptosystem used. Both BddCpir and our scheme share the same probabilistic public key cryptographic protocol, proposed by Damgård and Jurik [9], because of its multiple encryption and additive homomorphism properties. Therefore we will start by defining homomorphic encryption. After this definition, Damgård-Jurik cryptosystem, its key generation, encryption and decryption operations will follow. In addition, there will be a proof of how Damgård-Jurik satisfies the additive homomorphism requirement.

2.1.1 Homomorphic Encryption

Encryption systems that allow operations to be performed on encrypted data (cipher text) without decrypting it are said to be homomorphic cryptosystems. In this way, a user does not need to know the private key to be able to perform calculations on encrypted data. This allows us to make use of powerful but not fully trusted systems (e.g. cloud servers) to compute costly operations on our data instead of client computers with limited resources.

More formally, an encryption is homomorphic if using known $E(x)$ and $E(y)$ it is possible to compute $E(f(x, y))$ without using private key [33]. In this context E is the encryption function and f can be $+$, $\times$ or $\oplus$. If f is an addition function, in other words, if the cryptosystem allows summation over encrypted text, then the algorithm is called additive homomorphic encryption. Examples of such cryptosystems include Paillier [29], Goldwasser-Micali [16] and Damgård-Jurik [9]. Similarly, if multiplication can be calculated using ciphertext, thenf the algorithm is referred as multiplicative homomorphic encryption. RSA [1] and ElGamal [11] are among the examples of such systems. There are also fully homomorphic cryptosystems that allow both addition and multiplication over the ciphertext.

2.1.2 Damgård-Jurik Cryptosystem

As we defined in our cryptographic requirements, additive homomorphism is a must have property. The example cryptosystems that are given in the previous section, such as Paillier, can be used in basic (2, 1) BddCpir construction which includes only one encryption [20,29]. However using Paillier, we cannot extend the protocol to generalized (n, 1) case since Paillier does not allow to adjust the block length of the scheme after the public key has been generated. Therefore, Damgård-Jurik, which is a generalization of Paillier scheme [9], is the cryptosystem of choice for our protocols.

Damgård-Jurik cryptosystem uses the RSA setting, where the modulo arithmetic is employed with a modulus N, which is the product of two sufficiently large prime numbers, p and q. However it differs from RSA in its security principal; RSA relies on the computational difficulty of factorization of large integers, whereas the security of Damgård-Jurik is based on the decisional composite residuosity problem, which is also used in the original Paillier cryptosystem [29].

A very important part of this cryptosystem is the natural number s. First of all, the Paillier scheme is a special case of Damgård-Jurik where s is set to 1. Therefore incrementing s will allow the block length of the scheme to be changed, thus allowing us to encrypt the same data more than once. In other words, in Damgård-Jurik, encryption of an already encrypted file is possible by altering the s value. In the BddCpir protocols, we will start by setting s to 1 at the lowest level of the tree, and we will increment it by one as we advance upwards in the tree.

Key generation In order to generate the keys, the security parameter k needs to be set first.

N of length k bits is an RSA modulus and it is generated as $N = pq$ where p and q are two large primes.

The other public key, also referred as the base, $g \in Z^*_{N^{s+1}}$ is chosen such that $g = (1+N)^j x \mod N^{s+1}$ with a known j that is relatively prime to N and $x \in H$ where H is isomorphic to Z^*_N. In our implementation, we use the simplification suggested by the creators of the cryptosystem [9] and take g as simply $N + 1$.

For private key, first λ, the least common multiple of $p-1$ and $q-1$ is computed: $\lambda = \text{lcm}(p-1, q-1)$. Then using Chinese Remainder Theorem (CRT), the private key d is chosen such that

$$d = 1 \bmod N^s \text{ and } d = 0 \bmod \lambda.$$

Using the above procedures, public keys N, g and private key d are generated.

Encryption Given a plaintext $m \in Z_{N^s}$; random $r \in Z^*_{N^{s+1}}$ is chosen and ciphertext is computed as

$$E(m,r) = g^m r^{N^s} \bmod N^{s+1}.$$

Decryption Given a ciphertext c, first $c^d \bmod N^{s+1}$ is computed. Then by using the algorithm defined by [9], we can obtain m. More detail about the algorithm and decryption process in general can be found in [9].

Additive homomorphism Given ciphertexts $E(m_1)$ and $E(m_2)$,

$$\begin{aligned} E(m_1) \cdot E(m_2) &= g^{m_1} r_1^{N^s} \cdot g_2^m r_2^{N^s} \bmod N^{s+1} \\ &= g^{(m_1+m_2)} (r_1 r_2)^{N^s} \bmod N^{s+1} \\ &= g^{(m_1+m_2)} r^{N^s} \bmod N^{s+1} \\ E(m_1) \cdot E(m_2) &= E(m_1 + m_2) \end{aligned}$$

We can safely say that the above homomorphic property holds since $r_1 r_2$ is equialent to another random number $r \in Z^*_{N^{s+1}}$. Similarly, Damgård-Jurik also satisfies the following equation provided that c is a natural number:

$$E(m)^c = E(m \cdot c)$$

Because of the properties given above, Damgård-Jurik is an additively homomorphic encryption system.

2.2 Binary Decision Diagrams

A binary decision diagram is a directed acyclic graph where each node of the diagram can have at most two outgoing transitions as in binary tree. The underlying graphs of the decision diagrams that we use in our protocol always have tree properties, therefore in this context BDDs can also be thought as trees.

2.2.1 Properties of a BDD

In a binary decision diagram, non-sink (also called non-terminal) nodes are labeled as $R_{i,j}$ where i denotes the level in the tree and j denotes the position of the node in a level. The initial value of index i is 0 at the terminal nodes and it increases as we approach the root node (in upwards direction). Likewise, j index starts with 0 at leftmost node and increases while going right at a level. Besides nodes, the two outgoing edges of the internal nodes are also labeled as 0 and 1, respectively.

The sink nodes can either be represented with $R_{0,j}$ or f_j, since in BddCpir protocol, those nodes hold the actual data items (files) of the database. In this work, we employ both of the notations as appropriate for the context. The index j of f_j (or $R_{0,j}$) has the bit length of m, representing the route taken from the root node to that sink node. In other words, the indices of the data items are the concatenation of the labels of the edges that are visited while reaching the sink node from the root node. Therefore their bit length, m, is equal to the depth of the tree. Since illustrating the indices as bit strings requires more space and they are harder to handle, we use their decimal equivalents in j index for convenience. Figure 1 illustrates the aforementioned properties on a binary decision diagram with 4 sink nodes and thus having a depth of 2.

As mentioned, in BddCpir protocol, the sink nodes represent the data items stored in the server to be privately retrieved by the client. Thus, the labels of the sink nodes are used to identify the indices of data items. Therefore if the client queries the server with a binary input x of bit length m, the server returns the data item f_x, stored in the sink node with the label x. While processing the user input to return the requested data item, the server stores the intermediate values at non-sink nodes $R_{i,j}$, where $i > 0$.

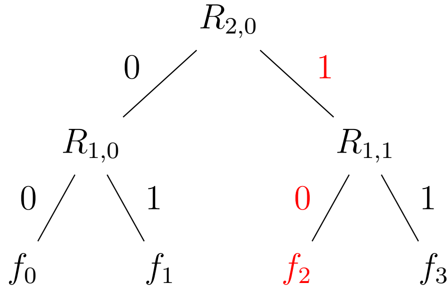

Figure 1: An example BDD constructed by server, shows the case where the client queries the database with binary input $x = 10$, to retrieve file f_2.

2.2.2 Quadratic and Octal Trees

For performance reasons, which will be explained in depth later in subsequent sections, we propose using quadratic and octal trees instead of binary decision diagrams. These types of trees essentially have the same properties as the binary ones except their child count.

Quadratic Trees If the non-sink nodes of a tree has 4 children, it is called quadratic tree or occasionally *quadtree*. The outgoing edges of the internal (non-sink) nodes in a quadratic tree are labeled as {00, 01, 10, 11}, therefore the labels of the sink nodes have $2m$ bit strings where m is the depth of the tree.

Octal Trees Octal Trees, which are sometimes called octrees, have 8 children in their non-sink nodes. The outgoing edges of those nodes are labeled by 3-bit strings {000, 001, 010, . . . , 111}, hence the sink nodes' label strings have bit length of $3m$, where m is again the depth of the tree.

2.3 (2, 1) CPIR

In 2005, Lipmaa proposed a communication-effective (2, 1) CPIR protocol [20], which is a basic cryptographic primitive that only allows 1 file to be retrieved from a 2-file setting. In this 1-out-of-2 protocol, the server has a database $\mathcal{F} = (f_0, f_1)$ where files have ℓ bit length, more formally $f_i \in \{0, 1\}^\ell$. Since there are only two files, to

retrieve f_x from the server, a client should input either 0 or 1, so $x \in \{0,1\}$. The protocol works in three steps:

1. Client generates public and secret keys (pk, sk), computes $c = E_{pk}(x)$ and sends (pk, c) to the server.
2. Server computes $R = E_{pk}(f_0) \cdot c^{f_1 - f_0}$ and sends R to the client.
3. Client computes $D_{sk}(R)$ to find f_x.

$E_{pk}(x)$ will be referenced as simply $E(x)$ and $D_{sk}(R)$ as $D(R)$ henceforth, since encryption and decryption are always performed using public and private keys, respectively.

Proof. Since we have already shown that our cryptosystem is additively-homomorphic, we can also show that client will get f_x after decryption as follows

$$\begin{aligned} R &= E\left(f_0\right) \cdot c^{\,f_1 - f_0} \\ &= E\left(f_0\right) \cdot E\left(x\right)^{\,f_1 - f_0} \\ &= E\left(f_0 + x\left(f_1 - f_0\right)\right) \\ &= E\left(f_x\right). \end{aligned}$$

□

2.4 $(n,\ 1)$ CPIR

Again in [20], Lipmaa proposes a more generalized $(n,\ 1)$ CPIR using (2, 1) CPIR and binary decision diagrams as building blocks. To extend the primitive protocol to n-file databases, (2, 1) CPIR must be repeatedly applied to 2-file subtrees. Specifically, the protocol will start processing from the sink nodes, continue in a bottom-up manner and stop at the root node. While going up in the tree, two data items are processed into one by using the second step of (2, 1) CPIR described in Section 2.3, and the result of this calculation is stored in an upper level node. When all the items in a level are processed, the protocol continues with the elements in the proceeding level until there is no upper level. After the calculation is finished, the ciphertext stored in the root

node of the tree must be sent to the client that will decrypt it to reach the content of the file it requested.

In this 1-out-of-n protocol, the server has a database $\mathcal{F} = (f_0, f_1, ..., f_{n-1})$ with n ℓ-bit files, $f_i \in \{0,1\}^\ell$, $f_i \in \mathcal{F}$. To retrieve a file f_x from the database $\mathcal{F}$, the client sends encrypted version of the input x. Namely, for input $x = (x_0, \ldots, x_{m-1})$, $x_i \in \{0,1\}$, the client sends $\mathcal{C} = (c_0, \ldots, c_{m-1})$, where each $c_i = E(x_i)$, and m is the depth of the tree, $m = \lceil \log_2(n) \rceil$. At the end of the protocol, the client gets f_x by decrypting the ciphertext m times.

Example 1. To illustrate, let us consider a case where the server has 4 files to be chosen from and these files are stored in the sink nodes of a binary decision diagram. Data items are $\mathcal{F} = \{f_0, f_1, f_2, f_3\}$ and client inputs are $x = (x_0, x_1)$. First, client computes and sends $c_0 = E(x_0), c_1 = E(x_1)$. Upon receiving those inputs, server computes the following on the first (lowermost) level:

$$R_{1,0} = E\left(f_0\right) \cdot c_0^{f_1 - f_0},$$
$$R_{1,1} = E\left(f_2\right) \cdot c_0^{f_3 - f_2}$$

As described in Section 2.2.1 and illustrated in Figure 1, $R_{1,0}$ and $R_{1,1}$ are second-level nodes of the tree. After processing the first level, server then starts to work with the ciphertexts obtained from the previous step as

$$R_{2,0} = E\left(R_0\right) \cdot c_1^{R_1 - R_0}.$$

Different from the previous step, the other selection bit c_1 is used, as appropriate for the level. The computation of the server stops at this point and sends $R_{2,0}$ to be decrypted by the client. Upon receiving the ciphertext, client needs to perform the decryption operation twice in order to obtain f_x since $R_{2,0}$ contains a double encryption as shown below

$$\begin{aligned} R_{2,0} &= E\left(R_0\right) \cdot c_1^{R_1 - R_0} \\ &= E\left(R_0 + c_1 \cdot \left(R_1 - R_0\right)\right) \\ &= E\left(E\left(f_{0x_0}\right) + c_1 \cdot \left(E\left(f_{1x_0}\right) - E\left(f_{0x_0}\right)\right)\right) \\ &= E\left(E\left(f_{x_1 x_0}\right)\right) \end{aligned}$$

□

The important point in this protocol is we need to make sure that every encryption, exponentiation and multiplication operation is calculated on the correct modulus. At the beginning, while starting from the raw data on the lowest level, the natural number s used in Damgård-Jurik cryptosystem must be set to 1 since this will be the first encryption. After that, in each level this s value will be incremented by 1, allowing multiple encryptions. Besides encryption, all the other operations will also use $N^{(s+1)}$ as their modulus, specified according to their level. Therefore the c_i inputs sent by the client also need to be computed on the correct modulus. Specifically, the least significant bit of the input string should be encrypted with $s = 1$ (in other words, using modulus N^2), and the encryption of most significant bit should use $s = m$ (i.e. $\bmod N^{m+1}$) assuming the input is m bits long.

Example 2. In an 8-file binary tree system, the input bits will be formed by the user as

$$
\begin{aligned}
c_0 &= g^{x_0} r_0^{n} \bmod N^2 \\
c_1 &= g^{x_1} r_1^{N^2} \bmod N^3 \\
c_2 &= g^{x_2} r_2^{N^3} \bmod N^4,
\end{aligned}
$$

where $r_0 \in_R Z^*_{N^2}$, $r_1 \in_R Z^*_{N^3}$, and $r_2 \in_R Z^*_{N^3}$ and $x = (x_2, x_1, x_0)$ is the index of the desired data item. The same moduli used by the client will also be used by the server in the respective levels of the tree. □

Therefore, considering the quadratic complexity of Damgård-Jurik encryption operation, the computation latency will be inevitably high even for databases with moderately high number of items because of the constant increase in modulus. This continuous message expansion with multiple encryptions hinders the scalability of the CPIR scheme.

3 Problem Statement

PIR protocols, by definition, should have an efficient communication complexity compared to the trivial solution. This property differentiates PIR protocols from oblivious transfer schemes that have higher bandwidth requirements [30]. Since in oblivious transfer, the user is allowed to access only one item in the database, the removal of this requirement in PIR allows more communication-efficient protocols to be constructed.

However, communication is not the only restriction in PIR. The server-side computation must also be reasonable so that a user can prefer utilizing a PIR scheme instead of the naive solution of downloading the whole database. Because of these reasons, we aim to achieve two major performance measures to obtain an efficient PIR protocol:

- **Computational Efficiency and Scalability** Since at the core of the PIR protocols there lies particularly costly cryptographic operations, such as encryption, multiplication and exponentiation of both plaintext as well as encrypted data, computational complexity is an important measure for the PIR schemes. The efficiency is generally based on the throughput metric, expressed as the number of data items processed in a unit time. Besides that, the latency is also significant since the users would only tolerate waiting for a limited amount of time. Apart from the latency and throughput requirements, an efficient PIR protocol should also be scalable. Namely, even if the number of data items in the database grows, the scheme must remain applicable. PIR schemes with parallelizable methods will have an advantage for the scalability requirement, since they allow the distribution of the work onto different cores. Therefore, in this work we try to benefit from parallelization of costly computations.

- **Bandwidth Efficiency** As the requirement for any PIR scheme, the communication complexity must be strictly smaller than the database size. The communication cost consists of both query and response size, sent by the client and the server respectively. While some of the PIR schemes focus on minimizing the amount of bits in the query sent by the client to the server, others devote their efforts to decrease the response length sent from the server to the client. In this thesis, we are not separating them from each other and aim to optimize the total bandwidth exhausted by both the client and the server.

As a consequence, the main aim of this work is to outperform the original BddCpir in terms of both computational and bandwidth efficiency. In the subsequent chapters, we explain our methods to achieve this goal.

4 CPIR using Quadratic and Octal Trees

The underlying data structure of BddCpir has a significant effect on the computational complexity of the protocol because of the message expansion caused by multiple encryptions. Since we need to increase the natural number s on each level of the binary tree used in BddCpir, the modulus which we use in our modular arithmetic operations constantly increases, and consequently resulting in unacceptable latencies on databases with high number of files, as demonstrated by our experiments in Chapter 8. Considering the main factor in this increase, namely the depth of the tree, we focus on decreasing the depth of the tree while preserving the number of items in a database. For this purpose, we change the data structure used for storing the files in BddCpir from binary to quadratic and octal trees. With the increase in the number of children a node can have, the depth of the tree decreases, thus resulting in reduced computational complexity. In this section, we will explain how the CPIR protocols work with quadratic and octal trees comprehensively.

4.1 Utilizing Quadratic Trees in CPIR

In a quadratic tree, each non-sink node has four children as described in Section 2.2.2. Similar to the binary case, the files are stored in the sink nodes of the tree, and the protocol processes the tree in a bottom-up manner. Let us first define the primitive (4, 1) CPIR used with a quadratic tree and then proceed to the generalization of this basic scheme to $(n, 1)$ case.

4.1.1 (4, 1) CPIR

(4, 1) CPIR is a 1-out-of-4 protocol that uses a minimal quadratic tree with 4 sink nodes and a root node. In this scheme, the server holds a database of four files of bit length ℓ, $\mathcal{F} = (f_0, f_1, f_2, f_3)$, $f_i \in \{0,1\}^\ell$, one of which is to be picked for retrieval by the user. In order to retrieve f_x from the server, a client determines the input bits $x = (x_1 x_0)$ beforehand, and sends $E(x_1 \cdot x_0)$ in addition to $E(x_1)$ and $E(x_0)$. Although the additional encrypted index bit may seem to increase the communication complexity, this protocol achieves an improvement in overall bandwidth usage as it will be presented in Chapter 7 in detail.

Formally speaking, given a database $\mathcal{F}$ and input bits x, the protocol is executed as follows:

1. Client:
   - generates public and secret keys (pk, sk)
   - computes $\mathcal{C} = \{c_0, c_1, c_{0,1}\}$: $c_0 = E(x_0)$, $c_1 = E(x_1)$, $c_{0,1} = E(x_1 \cdot x_0)$
   - sends $(pk, \mathcal{C})$ to the server.
2. Server:
   - computes $R = E(f_0) \cdot c_0^{f_1 - f_0} \cdot c_1^{f_2 - f_0} \cdot c_{0,1}^{f_3 - f_2 - f_1 + f_0}$
   - sends R to the client.
3. Client computes $D_{sk}(R)$ to find $f_{x_1 x_0}$.

Proof. The following proof shows that client will obtain f_x after decrypting R, based on the fact that Damgård-Jurik is an additively-homomorphic encryption:

$$
\begin{aligned}
R &= E(f_0) \cdot c_0^{f_1 - f_0} \cdot c_1^{f_2 - f_0} \cdot c_{0,1}^{f_3 - f_2 - f_1 + f_0} \\
&= E(f_0) \cdot E(x_0)^{f_1 - f_0} \cdot E(x_1)^{f_2 - f_0} \cdot E(x_1 \cdot x_0)^{f_3 - f_2 - f_1 + f_0} \\
&= E(f_0 + x_0 \cdot (f_1 - f_0) + x_1 \cdot (f_2 - f_0) + x_1 \cdot x_0 \cdot (f_3 - f_2 - f_1 + f_0)) \\
&= E(x_1 \cdot x_0 \cdot \boldsymbol{f_3} + x_1 \cdot (1 - x_0) \cdot \boldsymbol{f_2} + x_0 \cdot (1 - x_1) \cdot \boldsymbol{f_1} + (1 - x_1) \cdot (1 - x_0) \cdot \boldsymbol{f_0}) \\
&= E(f_{x_1 x_0}) = E(f_x)
\end{aligned}
$$

□

4.1.2 $(n,\ 1)$ CPIR with Quadratic Trees

The new primitive (4, 1) CPIR can be generalized to n-file case using quadratic trees. The generalization process is similar to the one from (2, 1) to $(n,\ 1)$ case with binary trees: client sends encrypted input bits to retrieve any desired file, server constructs a tree from database that holds the files at its sink nodes, and processes the tree in a bottom-up manner using (4, 1) CPIR repeatedly, then returns the final ciphertext that is stored at the root node of the tree. Client accesses the requested file by decrypting the ciphertext for number of times equal to the depth of the tree.

Assuming that the number of data items n is an exact power of 4, i.e. $n = 4^m$, the quadratic tree will have a depth of m. In order to retrieve a file from this database, client has to decide $2m$ input bits $x = (x_0, x_1, x_2, \ldots, x_{2m-1})$. After determining the input bits, client computes $E(x_{2i})$, $E(x_{2i+1})$ and $E(x_{2i} \cdot x_{2i+1})$ for each level of the tree $i = 0, \ldots, m-1$. The significant factor in this operation is that the modulus used for each level of the tree should be different, namely, both client and server encryptions should be performed on $\bmod\ N^{s+1}$, where $s = i+1$ for level i of the tree. To imply the number s used in the modulus during encryptions, we use $E^{(s)}(x)$ notation for arbitrary x. If no s is present, $s = 1$, i.e. $\bmod\ N^2$ is presumed. In summary, given $\mathcal{F}$ of $n = 4^m$ files and input bits x, $(n,\ 1)$ CPIR protocol with quadratic trees works as follows:

1. Client:

   - sets public and secret keys (pk, sk)
   - computes $\mathcal{C}$:

     for $s = 1, \ldots, m$,

     $$c_{2s-2} = E^{(s)}(x_{2s-2}),\ c_{2s-1} = E^{(s)}(x_{2s-1}),\ c_{2s-2,2s-1} = E^{(s)}(x_{2s-2} \cdot x_{2s-1})$$

   - sends $(pk, \mathcal{C})$, to the server.

2. Server:

   - for $j = 0, 1, \ldots, 4^m - 1$, set $R_{0,j} = f_j$
   - for $s = 1, \ldots, m$ and $j = 0, 1, \ldots, 4^{m-s} - 1$

$$R_{s,j} = E^{(s)}(R_{s-1,4j}) \cdot (c_{2s-2})^{R_{s-1,4j+1} - R_{s-1,4j}} \cdot (c_{2s-1})^{R_{s-1,4j+2} - R_{s-1,4j}} \cdot (c_{2s-2,2s-1})^{R_{s-1,4j+3} - R_{s-1,4j+2} - R_{s-1,4j+1} + R_{s-1,4j}}$$

   - sends $R_{m,0}$ to the client.

3. Client computes $D(R_{m,0})$ m times in order to retrieve f_x.

In Figure 2, an example quadratic tree is shown, which is constructed by the server for a 16 file database. To illustrate, $R_{1,0}$ will hold the processed version of f_0, f_1, f_2, f_3 according to step 2.2 of the protocol with $s = 1$, namely on modulus N^2. Likewise, after calculating $R_{1,1}$, $R_{1,2}$ and $R_{1,3}$ in the same manner with respective files, $R_{2,0}$ will be calculated with $R_{1,0}$, $R_{1,1}$, $R_{1,2}$ and $R_{1,3}$ using the same formulation with $s = 2$ (using modulus N^3). When reached to the root of the tree, in this case the node labeled $R_{2,0}$, the server stops calculation and returns the ciphertext held by that node. Since the depth of this example tree is 2, upon receiving the ciphertext, the client needs to decrypt it twice: first by using $s = 2$ and then the resulting ciphertext with $s = 1$.

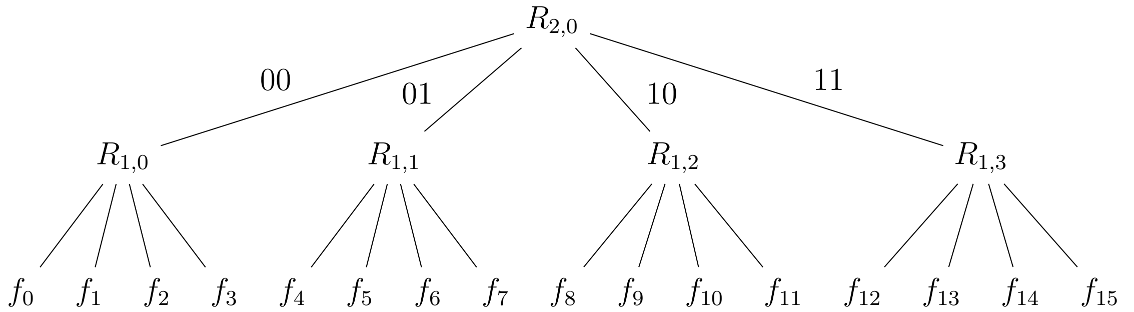

Figure 2: A depth-2 quadratic tree implementing (16,1)-CPIR

4.2 Utilizing Octal Trees

The non-sink nodes of the octal trees have 8 children as explained in Section 2.2.2. This property helps us to further reduce the depth of the tree for the same amount of files in a database, without adversely effecting the bandwidth usage. Similarly to the binary and quadratic case, the server again holds the files in the sink nodes of the tree and all the calculated intermediate values in the non-sink nodes of the tree. This chapter first defines the basic 1-out-of-8 CPIR protocol and then shows how it can be generalized into 1-out-of-n case using octal trees.

4.2.1 (8, 1) CPIR

This protocol is the equivalent of (2, 1) CPIR for the octal tree case. In this primitive scheme, we assume there are 8 files in the server and the client queries it to retrieve one of them. Specifically, the server keeps a database $\mathcal{F} = (f_0, f_1, \ldots, f_7)$, with each file having ℓ-bit length, $f_i \in \{0,1\}^\ell$, and the client wants to obtain the file f_x, where $x = (x_2x_1x_0)$ where $x_i \in \{0,1\}$. Given the database $\mathcal{F}$ and input bits x, the (8, 1) CPIR works as follows:

1. Client:
   - generates public and secret keys (pk, sk)
   - computes $\mathcal{C}$: $c_0 = E(x_0)$,$c_1 = E(x_1)$, $c_2 = E(x_2)$, $c_{0,1} = E(x_0 \cdot x_1)$, $c_{0,2} = E(x_0 \cdot x_2)$, $c_{1,2} = E(x_1 \cdot x_2)$, $c_{0,1,2} = E(x_0 \cdot x_1 \cdot x_2)$
   - sends (pk, $\mathcal{C}$) to the server.
2. Server computes

$$R = E(f_0) \cdot {c_0}^{f_1-f_0} \cdot {c_1}^{f_2-f_0} \cdot {c_2}^{f_4-f_0} \cdot {c_{0,1}}^{f_3-f_2-f_1+f_0} \cdot {c_{0,2}}^{f_5-f_4-f_1+f_0} \cdot {c_{1,2}}^{f_6-f_4-f_2+f_0} \cdot {c_{0,1,2}}^{f_7-f_6-f_5+f_4-f_3+f_2+f_1-f_0}$$

   and sends R to the client.
3. Client computes $D_{sk}(R)$ to find f_x, where $x = x_2x_1x_0$.

Proof. Utilizing the property of additive homomorphism in the underlying cryptosystem, we can show that the computation of R yields to the encryption of the client-requested file.

$$\begin{aligned}
R &= E(f_0) \cdot c_0^{f_1-f_0} \cdot c_1^{f_2-f_0} \cdot c_2^{f_4-f_0} \cdot c_{0,1}^{f_3-f_2-f_1+f_0} \cdot c_{0,2}^{f_5-f_4-f_1+f_0} \cdot c_{1,2}^{f_6-f_4-f_2+f_0} \\
&\quad \cdot c_{0,1,2}^{f_7-f_6-f_5+f_4-f_3+f_2+f_1-f_0} \\
&= E(f_0) \cdot E(x_0)^{f_1-f_0} \cdot E(x_1)^{f_2-f_0} \cdot E(x_2)^{f_4-f_0} \\
&\quad \cdot E(x_1 \cdot x_0)^{f_3-f_2-f_1+f_0} \cdot E(x_2 \cdot x_0)^{f_5-f_4-f_1+f_0} \cdot E(x_2 \cdot x_1)^{f_6-f_4-f_2+f_0} \\
&\quad \cdot E(x_2 \cdot x_1 \cdot x_0)^{f_7-f_6-f_5+f_4-f_3+f_2+f_1-f_0} \\
&= E(f_0 + x_0 \cdot (f_1 - f_0) + x_1 \cdot (f_2 - f_0) + x_2 \cdot (f_4 - f_0) \\
&\quad + x_1 \cdot x_0 \cdot (f_3 - f_2 - f_1 + f_0) + x_2 \cdot x_0 \cdot (f_5 - f_4 - f_1 + f_0) \\
&\quad + x_2 \cdot x_1 \cdot (f_6 - f_4 - f_2 + f_0) \\
&\quad + x_2 \cdot x_1 \cdot x_0 \cdot (f_7 - f_6 - f_5 + f_4 - f_3 + f_2 + f_1 - f_0)) \\
&= E(x_2 \cdot x_1 \cdot x_0 \cdot \boldsymbol{f_7} + x_2 \cdot x_1 \cdot (1 - x_0) \cdot \boldsymbol{f_6} + x_2 \cdot (1 - x_1) \cdot x_0 \cdot \boldsymbol{f_5} \\
&\quad + x_2 \cdot (1 - x_1) \cdot (1 - x_0) \cdot \boldsymbol{f_4} + (1 - x_2) \cdot x_1 \cdot x_0 \cdot \boldsymbol{f_3} + (1 - x_2) \cdot x_1 \cdot (1 - x_0) \cdot \boldsymbol{f_2} \\
&\quad + (1 - x_2) \cdot (1 - x_1) \cdot x_0 \cdot \boldsymbol{f_1} + (1 - x_2) \cdot (1 - x_1) \cdot (1 - x_0) \cdot \boldsymbol{f_0}) \\
&= E(f_x) = E(f_{x_2 x_1 x_0}).
\end{aligned}$$

□

4.2.2 $(n, 1)$ CPIR with Octal Trees

The generalization of (8, 1) CPIR to $(n, 1)$ CPIR with octal trees is quite similar to those in the quadratic and binary cases. With a database of $n = 8^m$ files $\mathcal{F}$, the server constructs an octal tree of depth m. To query f_x, the client has to determine m input bits to be encrypted and sent. Different from the binary and quadratic cases, now every level of the tree requires 3 input bits to be chosen and their encryptions are not sufficient, particularly, the client has to obtain multiplication of their every combination other and encrypt these bit combinations too, as in step 1.2 of (8, 1) CPIR protocol. The need for sending 7 encrypted bits rather than 3 (which would be the case if we

used quadratic tree instead) is the cost of using octal trees for a reduced depth. This is also the reason why we stopped at 8-child trees instead of continuing with 16-child, 32-child, etc.. As it will be shown in Chapter 7 in detail, this is the highest number of children we can use in the database without exceeding the bandwidth usage of original BddCpir with binary trees for the database sizes we employed in our implementations.

Provided the database $\mathcal{F}$ of $n = 8^m$ files, and client input bits x, $(n,\ 1)$ CPIR protocol with octal trees will start processing the tree from bottom to up, and return the resulting ciphertext at the root node to the client as follows:

1. Client:

   - sets public and secret keys (pk, sk)
   - computes $\mathcal{C}$:

     for $s = 1, \ldots, m$

     $c_{3s-3} = E^{(s)}(x_{3s-3})$, $c_{3s-2} = E^{(s)}(x_{3s-2})$, $c_{3s-1} = E^{(s)}(x_{3s-1})$,

     $c_{3s-3,3s-2} = E^{(s)}(x_{3s-3} \cdot x_{3s-2})$, $c_{3s-3,3s-1} = E^{(s)}(x_{3s-3} \cdot x_{3s-1})$,

     $c_{3s-2,3s-1} = E^{(s)}(x_{3s-2} \cdot x_{3s-1})$, $c_{3s-3,3s-2,3s-1} = E^{(s)}(x_{3s-3} \cdot x_{3s-2} \cdot x_{3s-1})$
   - sends $(pk, \mathcal{C})$ to the server.

2. Server:

   - for $j = 0, 1, \ldots, 8^m - 1$, set $R_{0,j} = f_j$
   - for $s = 1, \ldots, m$ and $j = 0, 1, \ldots, 4^{m-s} - 1$

$$
\begin{aligned}
R_{s,j} = \ & E^{(s)}\left(R_{s-1,8j}\right) \cdot \left(c_{3s-3}\right)^{R_{s-1,8j+1}-R_{s-1,8j}} \cdot \left(c_{3s-2}\right)^{R_{s-1,8j+2}-R_{s-1,8j}} \\
& \cdot \left(c_{3s-1}\right)^{R_{s-1,8j+4}-R_{s-1,8j}} \cdot \left(c_{3s-3,3s-2}\right)^{R_{s-1,8j+3}-R_{s-1,8j+2}-R_{s-1,8j+1}+R_{s-1,8j}} \\
& \cdot \left(c_{3s-3,3s-1}\right)^{R_{s-1,8j+5}-R_{s-1,8j+4}-R_{s-1,8j+1}+R_{s-1,8j}} \\
& \cdot \left(c_{3s-2,3s-1}\right)^{R_{s-1,8j+6}-R_{s-1,8j+4}-R_{s-1,8j+2}+R_{s-1,8j}} \\
& \cdot c_{3s-3,3s-2,3s-1}^{R_{s-1,8j+7}-R_{s-1,8j+6}-R_{s-1,8j+5}+R_{s-1,8j+4}-R_{s-1,8j+3}+R_{s-1,8j+2}+R_{s-1,8j+1}-R_{s-1,8j}}
\end{aligned}
$$

   - sends $R_{m,0}$ to the client.

3. Client computes $D^{(s)}(R_{m,0})$ for $s = m, m-1, \ldots, 1$ to retrieve f_x.

5 Parallelization of CPIR

All of the protocols that we have defined in Chapter 4, or has been defined before by Lipmaa [21], have a substantial amount of repetitive computations that are mostly independent from each other. Both the client-side and server-side computations can benefit from parallelization since their costly encryption and modular exponentiation operations can be operated separately by different threads. Thus, in this chapter we specify how we utilize parallelization in CPIR protocols to improve computational complexity and outline the proposed parallel algorithms.

Parallelization of the client side computations is rather trivial as outlined in the Section 5.1. However, for the server side operations, we try three methods, where each method includes an improvement over the preceding ones. We list each of them in order to demonstrate our progress and explain our main parallelization method better.

5.1 Client Side Parallelization

The encryption of the input bits constitutes most of the client side computation. The remaining part, repetitive decrypting, is serial in nature since each decryption procedure works on the result of previous decryption. Therefore, we operate all the encryptions done by the client in different threads, hence distributing the computation onto all available cores.

Implementation Details The algorithm for client side parallelization is pretty much straightforward as can be observed in Algorithms 1, 2 and 3. There is only one minor detail of the implementation; the iterations of the `for` loops are independent from each other, however they do not consume the same amount of time since the encryptions on each iteration use a different s thus operating on a distinct modulus. Therefore, in order to optimize the utilization of processor cores and prevent them from being idle during the execution of the longest encryption, we use dynamic scheduling for the iterations of the `for` loop in step 1 of Algorithms 1, 2 and 3. OpenMP, the parallelization library we are using in our implementation, allows such dynamic allocations by assigning an iteration of the for loop to a thread as they become available, removing the need to wait for other threads to complete their executions [25]. Dynamic scheduling is especially useful for loops with iterations that have fluctuating amounts of work such as our client side encryptions. However, the parallelization of step 2 in Algorithms 2 and 3 should not be dynamic since the encryptions inside are expected to take up approximately the same amount of time.

Algorithm 1 Parallel client side computation for binary tree based $(n, 1)$ CPIR

Require: $x = (x_{m-1}x_{m-2}\dots x_0)$, pk
Ensure: $\mathcal{C}$
1: **for** $s \leftarrow 1$ to m **in parallel do**
2: $\quad c_{s-1} \leftarrow E^{(s)}(x_{s-1})$
3: **end parallel for**
4: **return** $\mathcal{C} = \{c_{m-1}, c_{m-2}, \dots, c_0\}$

Algorithm 2 Parallel client side computation for quadratic tree based $(n, 1)$ CPIR

Require: $x = (x_{2m-1}x_{2m-2}\dots x_0)$, pk
Ensure: $\mathcal{C}$
1: **for** $s \leftarrow 1$ to m **in parallel do**
2: $\quad$ **in parallel do**
3: $\quad\quad c_{2s-2} \leftarrow E^{(s)}(x_{2s-2})$
4: $\quad\quad c_{2s-1} \leftarrow E^{(s)}(x_{2s-1})$
5: $\quad\quad c_{2s-2,2s-1} \leftarrow E^{(s)}(x_{2s-2} \cdot x_{2s-1})$
6: $\quad$ **sync**
7: **end parallel for**
8: **return** $\mathcal{C}$

Algorithm 3 Parallel client side computation for octal tree based $(n,\, 1)$ CPIR

Require: $x = (x_{3m-1}x_{3m-2}\ldots x_0)$, pk
Ensure: $\mathcal{C}$
1: **for** $s \leftarrow 1$ to m **in parallel do**
2:     **in parallel do**
3:         $c_{3s-3} \leftarrow E^{(s)}(x_{3s-3})$
4:         $c_{3s-2} \leftarrow E^{(s)}(x_{3s-2})$
5:         $c_{3s-1} \leftarrow E^{(s)}(x_{3s-1})$
6:         $c_{3s-3,3s-2} \leftarrow E^{(s)}(x_{3s-3} \cdot x_{3s-2})$
7:         $c_{3s-3,3s-1} \leftarrow E^{(s)}(x_{3s-3} \cdot x_{3s-1})$
8:         $c_{3s-2,3s-1} \leftarrow E^{(s)}(x_{3s-2} \cdot x_{3s-1})$
9:         $c_{3s-3,3s-2,3s-1} \leftarrow E^{(s)}(x_{3s-3} \cdot x_{3s-2} \cdot x_{3s-1})$
10:     **sync**
11: **end parallel for**
12: **return** $\mathcal{C}$

5.2 Server Side Trivial Parallelization Algorithm

For the server side computations, the first parallelization method we try is the most straightforward one. Since all the base protocols executed on a level of the tree are independent from each other, their parallelization is almost *embarrassingly parallel* [34]. On the start of the processing of a level, we assign all the independent executions of primitive computations (e.g., encryprions and modular exponentations) to distinct threads, and wait for them to be completed. Note that in this method, all the threads spawned in a level have to be completely finished before we can proceed to the next level of the tree. Although all the protocols in a level will be operating on different files, they are expected to take approximately same time. Therefore provided that there are adequate number of cores to work on and the server has a reasonable workload, the idle time before proceeding to next level should be minimal.

There is no restriction about data structure to be used, in other words all binary, quadratic and octal tree implementations of $(n,\, 1)$ CPIR can be parallelized using this trivial method. The parallelization methods for binary, quadratic and octal based server systems are shown in the Algorithm 4, 5, 6 respectively.

Algorithm 4 Parallel server computation for binary $(n,1)$ CPIR v1

Require: $\mathcal{C}$: m encrypted input bits
Ensure: $R_{m,0}$

1: **for** $s \leftarrow 1$ to m **do**
2:     **for** $j \leftarrow 0$ to $2^{m-s} - 1$ **in parallel do**
3:         $t_0 \leftarrow R_{s-1,2j}$
4:         $t_1 \leftarrow R_{s-1,2j+1}$
5:         $R_{s,j} \leftarrow E^{(s)}(t_0) \cdot (c_{s-1})^{t_1 - t_0} \bmod N^{s+1}$
6:     **end parallel for**
7: **end for**
8: **return** $R_{m,0}$

Algorithm 5 Parallel server computation for quadratic $(n,1)$ CPIR v1

Require: $\mathcal{C}$: $3m$ encrypted input bits
Ensure: $R_{m,0}$

1: **for** $s \leftarrow 1$ to m **do**
2:     **for** $j \leftarrow 0$ to $4^{m-s} - 1$ **in parallel do**
3:         **for** $k \leftarrow 0$ to 3 **do**
4:             $t_k \leftarrow R_{s-1,2j+k}$
5:         **end for**
6:         $R_{s,j} \leftarrow E^{(s)}(t_0) \cdot (c_{2s-2})^{t_1 - t_0} \cdot (c_{2s-1})^{t_2 - t_0} \cdot (c_{2s-2,2s-1})^{t_3 - t_2 - t_1 + t_0} \bmod N^{s+1}$
7:     **end parallel for**
8: **end for**
9: **return** $R_{m,0}$

Algorithm 6 Parallel server computation for octal $(n,1)$ CPIR v1

Require: $\mathcal{C}$: $7m$ encrypted input bits
Ensure: $R_{m,0}$

1: **for** $s \leftarrow 1$ to m **do**
2:     **for** $j \leftarrow 0$ to $8^{m-s} - 1$ **in parallel do**
3:         **for** $k \leftarrow 0$ to 7 **do**
4:             $t_k \leftarrow R_{s-1,2j+k}$
5:         **end for**
6:         $R_{s,j} \leftarrow E^{(s)}(t_0) \cdot (c_{3s-3})^{t_1 - t_0} \cdot (c_{3s-2})^{t_2 - t_0} \cdot (c_{3s-1})^{t_4 - t_0} \cdot (c_{3s-3,3s-2})^{t_3 - t_2 - t_1 + t_0}$
$\cdot (c_{3s-3,3s-1})^{t_5 - t_4 - t_1 + t_0} \cdot (c_{3s-2,3s-1})^{t_6 - t_4 - t_2 + t_0}$
$\cdot (c_{3s-3,3s-2,3s-1})^{t_7 - t_6 - t_5 + t_4 - t_3 + t_2 + t_1 - t_0} \bmod N^{s+1}$
7:     **end parallel for**
8: **end for**
9: **return** $R_{m,0}$

5.3 Server Side Two-Degree Parallelization Algorithm

Two-degree parallelization method is the second algorithm we try as an improvement over the first one described in the previous section. Again, it relies on the independency of costly operations performed in a level of the tree being processed by the server. As it can be observed from the previous algorithms, the calculations for the upper node ciphertexts include an encryption and varying number of modular exponentiations depending on the tree used. Since each of these calculations are independent from each other, we can process all of them in different threads, and synchronize to calculate the upper tree node by multiplying them using the corresponding modulus of the level.

This method further divides the costly computations performed in a level and benefits multi-core systems in a greater extend. As in the previous method, the threads created at a level have to be completely finished before advancing on the next level. Although this method better splits the work done on a level into pieces, the synchronization cost will be higher since numerous threads will be created, especially at the lowermost levels of the tree.

Similar to the prior method, this parallelization can be applied to binary, quadratic and octal tree based $(n, 1)$ CPIR as shown in Algorithm 7, 8 and 9 respectively.

Algorithm 7 Parallel server computation for binary $(n,1)$ CPIR v2

Require: $\mathcal{C}$: m encrypted input bits
Ensure: $R_{m,0}$
1: **for** $s \leftarrow 1$ to m **do**
2:     **for** $j \leftarrow 0$ to $2^{m-s} - 1$ **in parallel do**
3:         $t_0 \leftarrow R_{s-1,2j}$
4:         $t_1 \leftarrow R_{s-1,2j+1}$
5:         **in parallel do**
6:             $q_0 \leftarrow E^{(s)}(t_0)$
7:             $q_1 \leftarrow (c_{s-1})^{t_1 - t_0} \bmod N^{s+1}$
8:         **sync**
9:         $R_{s,j} \leftarrow q_0 \cdot q_1 \bmod N^{s+1}$
10:     **end parallel for**
11: **end for**
12: **return** $R_{m,0}$

Algorithm 8 Parallel server computation for quadratic $(n,1)$ CPIR v2

Require: $\mathcal{C}$: $3m$ encrypted input bits
Ensure: $R_{m,0}$

1: **for** $s \leftarrow 1$ to m **do**
2:     **for** $j \leftarrow 0$ to $4^{m-s} - 1$ **in parallel do**
3:         **for** $k \leftarrow 0$ to 3 **do**
4:             $t_k \leftarrow R_{s-1,2j+k}$
5:         **end for**
6:         **in parallel do**
7:             $q_0 \leftarrow E^{(s)}(t_0)$
8:             $q_1 \leftarrow (c_{2s-2})^{t_1-t_0} \bmod N^{s+1}$
9:             $q_2 \leftarrow (c_{2s-1})^{t_2-t_0} \bmod N^{s+1}$
10:            $q_3 \leftarrow (c_{2s-2,2s-1})^{t_3-t_2-t_1+t_0} \bmod N^{s+1}$
11:        **sync**
12:        $R_{s,j} = q_0$
13:        **for** $k \leftarrow 1$ to 3 **do**
14:            $R_{s,j} \mathrel{*}= q_k \bmod N^{s+1}$
15:        **end for**
16:    **end parallel for**
17: **end for**
18: **return** $R_{m,0}$

Algorithm 9 Parallel server computation for octal $(n,1)$ CPIR v2

Require: $\mathcal{C}$: $7m$ encrypted input bits
Ensure: $R_{m,0}$

1: **for** $s \leftarrow 1$ to m **do**
2:     **for** $j \leftarrow 0$ to $8^{m-s} - 1$ **in parallel do**
3:         **for** $k \leftarrow 0$ to 7 **do**
4:             $t_k \leftarrow R_{s-1,2j+k}$
5:         **end for**
6:         **in parallel do**
7:             $q_0 \leftarrow E^{(s)}(t_0)$
8:             $q_1 \leftarrow (c_{3s-3})^{t_1-t_0} \bmod N^{s+1}$
9:             $q_2 \leftarrow (c_{3s-2})^{t_2-t_0} \bmod N^{s+1}$
10:             $q_3 \leftarrow (c_{3s-1})^{t_4-t_0} \bmod N^{s+1}$
11:             $q_4 \leftarrow (c_{3s-3,3s-2})^{t_3-t_2-t_1+t_0} \bmod N^{s+1}$
12:             $q_5 \leftarrow (c_{3s-3,3s-1})^{t_5-t_4-t_1+t_0} \bmod N^{s+1}$
13:             $q_6 \leftarrow (c_{3s-2,3s-1})^{t_6-t_4-t_2+t_0} \bmod N^{s+1}$
14:             $q_7 \leftarrow (c_{3s-3,3s-2,3s-1})^{t_7-t_6-t_5+t_4-t_3+t_2+t_1-t_0} \bmod N^{s+1}$
15:         **sync**
16:         $R_{s,j} = q_0$
17:         **for** $k \leftarrow 1$ to 7 **do**
18:             $R_{s,j} *= q_k \bmod N^{s+1}$
19:         **end for**
20:     **end parallel for**
21: **end for**
22: **return** $R_{m,0}$

5.4 Server Side Core-Isolated Parallelization

The previous methods for server side parallelization are level-bound, meaning that they have to synchronize the threads created on each level of the tree before continuing. This property both introduces high synchronization overheads and also brings the possibility for some cores to stay idle during the computation due unbalanced workload of each thread. For this reason, in order to reduce synchronization points between cores, we propose our main parallelization method, where we isolate the tree onto available cores.

Main principle of this method is dividing the tree into as many subtrees as the number of available cores and having them calculate their assigned subtrees separately. Naturally, after each core finishes its part, a synchronization is necessary. After the integration of their calculations via synchronization, the remaining part of the tree is processed as in second parallelization method depicted in Section 5.3.

For example, provided that there are 2^{κ} number of available cores and $n = 2^m$ files in a server implementing $(n, 1)$ CPIR with binary trees and $n > \kappa$, each core will have to process $2^{m-\kappa}$ files in isolation using the original scheme without any parallelization inside for a tree of $m - \kappa$ levels. Specifically, for $m - \kappa$ levels, the cores do not need to communicate in any manner. After the cores finish computing their portion of the tree, they have to synchronize and continue processing remaining κ levels concurrently. This algorithm is able to operate on quadtree and octree based $(n, 1)$ CPIR protocols as well, and the files in those trees will be separated into cores in a similar manner.

The algorithm implementing this method for binary $(n, 1)$ CPIR is described in Algorithm 10. The isolated work of the cores can be identified in pseudocode statements between the lines 2-8 whereas the concurrent work after synchronization lies between lines 11-21.

Algorithm 10 Parallel server computation for binary $(n,1)$ CPIR v3

Require: $\mathcal{C}$: m encrypted input bits, $\mathcal{F} = \{f_0, \ldots, f_{2^m-1}\}$ 2^κ: number of cores, $\kappa < m$
Ensure: $R_{m,0}$

1: **for** $p \leftarrow 0$ to $2^\kappa - 1$ **in parallel do** ▷ cores work in isolation
2: **for** $s \leftarrow 1$ to $m - \kappa$ **do**
3: **for** $j \leftarrow 0$ to $2^{m-s} - 1$ **do**
4: $t_0 \leftarrow R_{s-1,2\cdot j\cdot p}$
5: $t_1 \leftarrow R_{s-1,2\cdot j\cdot p+1}$
6: $R_{s,j\cdot p} \leftarrow E^{(s)}(t_0) \cdot (c_{s-1})^{t_1 - t_0} \bmod N^{s+1}$
7: **end for**
8: **end for**
9: **end parallel for**
▷ cores sync and continue with the rest of the tree concurrently
10: **for** $s \leftarrow m - \kappa + 1$ to m **do**
11: **for** $j \leftarrow 0$ to $2^{m-s} - 1$ **in parallel do**
12: $t_0 \leftarrow R_{s-1,2j}$
13: $t_1 \leftarrow R_{s-1,2j+1}$
14: **in parallel do**
15: $q_0 \leftarrow E^{(s)}(t_0)$
16: $q_1 \leftarrow (c_{s-1})^{t_1 - t_0} \bmod N^{s+1}$
17: **sync**
18: $R_{s,j} \leftarrow q_0 \cdot q_1 \bmod N^{s+1}$
19: **end parallel for**
20: **end for**
21: **return** $R_{m,0}$

6 Scalable CPIR for Parallel Implementations

The proposed parallelization method improves computation efficiency of the server notably, however, if the database starts to have higher number of files, then CPIR will not be able to handle those files efficiently due to the increased depth of the tree. In other words, the system will not be able to scale adequately, even with the help of the aforementioned parallelization approach, since with the increased number of files, the database tree will get deeper, increasing the size of the modulus and making the encryption and exponentiation processes more costly. Therefore, to achieve scalability, which is a must have property of an efficient CPIR as defined in Chapter 3, we propose a modified version of CPIR that takes advantage of parallel processing, and allows the scheme to scale to large number of data items provided that many-core processors are available.

The scalable method for CPIR is based on holding the whole database in separated, manageable-sized subtrees instead of one big tree, and collapsing them into one subtree upon receiving a request from the client and then operating on that subtree. For this reason, obviously the client has to send different number of selection bits from the normal CPIR schemes. Since a subtree will have fewer number of items than the database size, the depth of the tree will be reduced, giving us a considerable amount of bandwidth gain. However, in contrast, to choose between possible subtrees, the client will have to send additional encrypted selection bits. With careful selection of subtree sizes, we can obtain speedup without too much adverse affect on the bandwidth. The exact analysis of this method in terms of bandwidth and computation costs will also be given in Chapter 7.

The details of the scalable method are demonstrated in Algorithms 11 and 12.

Specifically, Algorithm 11 illustrates the client computations and Algorithm 12 describes the steps executed by the server process.

First of all, the subtree size and the number of subtrees must be decided and known by both server and client. Specifically, considering a binary-tree based database with $n = 2^m$ files, if the number of items in a subtree is 2^l, with $l < m$, that gives us the number of subtrees in a system as $\mu = 2^{m-l}$. Number of subtrees, μ, must be selected according to the performance requirements. The analyses at Chapter 7 and actual results at Chapter 8 provide an insight about the selection of l and μ, and show us how the selection affects the performance clearly.

After determining how many files a subtree will hold and calculating the number of subtrees, the client may begin to query the server to get a file f_x. In order to do so, for the scalable CPIR, the client must decide both which subtree holds the requested file and in that subtree, which file corresponds to f_x, differently from the previous schemes. The encrypted selection bits, denoted with ς_i in Algorithm 11, are used to indicate the selected subtree, whereas input bits, $c_j \in \mathcal{C}$ are typical input bits to select the file within a subtree, similar with the previous schemes. Specifically, if a subtree contains the desired file, the client will encrypt 1 using the homomorphic Damgård-Jurik cryptosystem, and 0 otherwise. Since the depth of the tree is now reduced, the client will have to encrypt l regular input bits for the binary tree case as shown in Algorithm 11.

Upon receiving the selection bits ς_i and input bits c_j, the server starts the process by collapsing all the subtrees into one. As shown in the steps between 1-10 in Algorithm 12, the server uses ς_i to collapse the subtrees into one; and after the merge, it works on the collapsed subtree as a regular tree.

The collapsing process includes a modular exponentiation for each file in the database, as seen on the line 7 of Algorithm 12. After all the files have been raised to a power of corresponding ς, they are all multiplied using the same modulus (with $s = 1$). Since now the collapsed subtree contains encrypted files at the bottom, our regular, parallel bottom-to-up processing will start with $s = 2$, and increments it while going up to higher levels. The server splits the subtree into smaller parts, assigning each of them to

Algorithm 11 Client-side computation for binary tree-based Scalable CPIR

Require: m, l, and $x = x_{l-1} \dots x_1, x_0$
Ensure: $\{c_1, \dots, c_{l-1}\}$ and $\{\varsigma_i, \dots, \varsigma_{2^{m-l}-1}\}$

1: $\mu \leftarrow 2^{m-l}$
2: $\zeta \leftarrow x_{m-1}, \dots, x_l$
3: **for** $i \leftarrow 0$ to $\mu - 1$ **do**
4:     **if** $i \neq \zeta$ **then**
5:         $\varsigma_i \leftarrow E(0)$
6:     **else**
7:         $\varsigma_i \leftarrow E(1)$
8:     **end if**
9: **end for**
10: **for** $s \leftarrow 1$ to l **do**
11:     $c_{s-1} \leftarrow E^{(s+1)}(x_{s-1})$
12: **end for**
13: **return** $\{c_0, \dots, c_{l-1}\}$ and $\{\varsigma_0, \dots, \varsigma_{\mu-1}\}$

a different core to work in isolation as shown in lines 11-23. Finally, in the rest of the algorithm, the server collects the results from processor cores and continues the CPIR process for the remaining part of the tree.

Algorithm 12 Server-side computation for binary tree-based Scalable CPIR

Require: $m, \mathcal{C} = \{c_0, \ldots, c_{m-1}\}, \mathcal{F} = \{f_0, \ldots, f_{2^m-1}\}, \{\varsigma_0, \ldots, \varsigma_{2^{m-l}}\}, l < m$ and $\kappa < l$
Ensure: $R_{m,0}$

▷ Collapsing subtrees into one subtree
1: $\mu = 2^{m-l}$ ▷ Number of subtrees
2: $\delta = 2^{l-\kappa}$ ▷ Number of data items assigned to a core
3: **for** $j \leftarrow 0$ to $2^\kappa - 1$ **in parallel do**
4: **for** $i \leftarrow 0$ to $\delta - 1$ **do**
5: $R_{0,j\delta+i} = 1$
6: **for** $k \leftarrow 0$ to $\mu - 1$ **do**
7: $R_{0,j\delta+i} \leftarrow R_{0,j\delta+i} \cdot \varsigma_k^{f_{j\delta+k(2^l)}} \bmod N^2$
8: **end for**
9: **end for**
10: **end parallel for**

▷ Cores computing in the collapsed subtree in isolation
11: **for** $j \leftarrow 0$ to $2^\kappa - 1$ **in parallel do**
12: **for** $i \leftarrow 0$ to $\delta - 1$ **do**
13: $\tilde{R}_{0,i} \leftarrow R_{0,j\delta+i}$
14: **end for**
15: **for** $s \leftarrow 1$ to $l - \kappa$ **do**
16: **for** $i \leftarrow 0$ to $2^{l-s} - 1$ **do**
17: $t_0 \leftarrow \tilde{R}_{s-1,2i}$
18: $t_1 \leftarrow \tilde{R}_{s-1,2i+1}$
19: $\tilde{R}_{s,j} \leftarrow E^{(s+1)}(t_0) \times c_{s-1}^{t_1 - t_0} \bmod N^{s+2}$
20: **end for**
21: **end for**
22: $R_{l-\kappa,j} \leftarrow \tilde{R}_{l-\kappa,0}$
23: **end parallel for**

▷ Cores join
24: **for** $s \leftarrow l - \kappa + 1$ to l **do**
25: **for** $j \leftarrow 0$ to $2^{l-s} - 1$ **in parallel do**
26: $t_0 \leftarrow R_{s-1,2j}$
27: $t_1 \leftarrow R_{s-1,2j+1}$
28: **in parallel do**
29: $q_0 \leftarrow E^{(s+1)}(t_0)$
30: $q_1 \leftarrow c_{s-1}^{t_1 - t_0} \bmod N^{s+2}$
31: **sync**
32: $R_{s,j} \leftarrow q_1 \cdot q_0 \bmod N^{s+2}$
33: **end parallel for**
34: **end for**
35: **return** $R_{l,0}$

Figure 3: Collapsing four subtrees into one tree

Example 3. Consider a binary tree having 4 levels for a database with $2^m = 16$ files as illustrated in Figure 3. Assuming that there are $2^\kappa = 2$ processor cores available and we choose the subtree size as $2^l = 4$. The chosen values, $m = 4, l = 2, \kappa = 1$ are proper for a scalable CPIR since $l < m$ and $\kappa < l$. Since we select a subtree to hold 4 files, there will be $16/4 = 4$ subtrees in our selection (i.e., two selection bits are needed in addition to the index bits).

Suppose that the client is interested in file f_{11}, marked with red in Figure 3. Normally, the input bits would be $x = 1011$ and client would encrypt all of them using appropriate moduli for each bit and send them to server. However in scalable CPIR case, client separates $m - l$ bits for subtree selection and remaining l of them for index bits. In this example setting, the client will separate 10 from x as the selection bits to compute $\varsigma_2 = E(1)$ for the subtree starting with $R_{2,2}$ and $\varsigma_i = E(0), i = 0, 1, 3$ for the remaining subtrees. The rest of the input bits, 11, are encrypted for each level of the subtree, so, the client prepares the encrypted input bits as $c_0 = E^{(2)}(1)$, $(x_0 = 1)$, $c_1 = E^{(3)}(1)$, $(x_1 = 1)$. The modulus for the bottom level of the tree is no longer N^2, but N^3 since, as indicated in line 7 of Algorithm 12, now the leaf nodes of the tree hold encrypted data items instead of plaintext files. This means that we already use a

modulus with $s = 1$ for the encryption of the files in the selection operation and in order to continue encrypting them, we need to increment s. Therefore, in the scalable CPIR, we start performing the modular arithmetic operations with mod N^3 and increment s as the level increases.

After the server receives ς_i, $i = 0, 1, 2, 3$ and c_j, $j = 0, 1$ calculated by the client, it starts collapsing the subtrees into one using ς_i as depicted in lines 1-10 of Algorithm 12. In the example case, μ is calculated as $2^{4-2} = 4$ and $\delta = 2^{2-1} = 2$. The subtree collapsing operation is also performed in parallel; therefore we assign the calculation of each data item that will be on the subtree after collapsing to a specific core. That core is responsible for retrieving the required files from each subtree, raising them to the corresponding ς_i and multiplying them with each other. Therefore, to collapse our 4 subtrees of 4 files using 2 cores in parallel, each core will be responsible for 2 files. Precisely, one core will compute $R_{0,0}$, $R_{0,1}$ of the new subtree, and the other core will calculate $R_{0,2}$, $R_{0,3}$ as follows

$$R_{0,j} = \prod_{i=j}^{12+j} (\ \varsigma_{\lfloor i/4 \rfloor}\ )^{f_i} \mod N^2, \quad i \mathrel{+}= 4.$$

Due to the homomorphic encryption by the Damgård-Jurik cryptosystem, the operation will be a homomorphic multiplication of the files in the unwanted subtrees with 0 since those subtrees do not contain the requested file, therefore $\varsigma = E(0)$. Consequently we have, through additive homomorphism, $E(0)^f = E(0 \cdot f) = E(0)$. Similarly, for the subtree that contains the desired file, $\varsigma = E(1)$. Therefore, $E(1)^f = E(1 \cdot f) = E(f)$. Again due to the homomorphic properties, multiplying an ecnrypted file with a corresponding file in another subtree will result in $E(0) \cdot E(f) = E(0 + f) = E(f)$. In brief, at the end of collapsing procedure, $R_{0,j}$ will hold $E(f_{j+8})$, $j = 0, 1, 2, 3$ since $f_8, f_9, f_{10},$ andf_{11} are contained in the selected subtree considering the example database in Figure 3.

Now, we have a tree with 4 encrypted files at its sink nodes, the remaining process is similar to previous schemes, except we will start modulus variable s from 2 instead

of 1. The calculations done in the new subtree for this example proceed as follows:

$$
\begin{aligned}
R_{1,0} &= E^{(2)}(R_{0,0}) \cdot c_0^{R_{0,1}-R_{0,0}} \\
R_{1,1} &= E^{(2)}(R_{0,2}) \cdot c_0^{R_{0,3}-R_{0,2}} \\
R_{2,0} &= E^{(3)}(R_{1,0}) \cdot c_1^{R_{1,1}-R_{1,0}}
\end{aligned}
$$

At the end, $R_{2,0}$ is sent to the client to be decrypted 3 times since $R_{2,0}$ is calculated with $s = 3$.

7 Communication and Computation Analysis

This chapter gives the analyses of all the schemes proposed so far, in terms of communication an computational complexity. Firstly, it will start by the bandwidth usages of new quadratic and octal trees along with the original binary one, then continue with the computation requirements of new trees and their parallelization also in comparison with the original BddCpir. Lastly, the performance of the scalable CPIR with octal trees is evaluated, examining the case when database tree grows larger.

7.1 Analysis of Communication Complexity

A practical PIR scheme should be more efficient than the user downloading all the database (*the trivial solution*) in terms of the amount of information exchanged between the user and the server. Formally speaking, the bandwidth requirements of a PIR scheme must be sublinear to the size of the database. The bandwidth of the original $(n, 1)$-CPIR scheme based on binary decision trees has a logarithmic complexity. The proposed schemes based on quadratic and octal trees also have logarithmic complexities. However, the actual implementations of these three CPIR schemes have different bandwidth requirements, which are important in practice.

In PIR protocol, the client sends encrypted selection bits to the server in the first stage and receives the encrypted data item in the second stage. For a binary decision tree, the number of selection bits is $\log_2 n$, where n is the number of data items in the database. Assuming $f_i < N$ for all data items and $|N|$ is the size of the modulus N, the size of the selection bit for the lowest level of the tree, $c_0 = E(x_0)$, is $2|N|$-bit due to message expansion property of the Damgård-Jurik encryption. The selection bit for

the second level $c_1 = E^{(2)}(x_1)$, therefore, will be $3|N|$-bit long. In general, the selection bit for the level s, $c_{s-1} = E^{(s)}(x_1)$ will be $(s+1)|N|$-bit long.

The proposed CPIR schemes based on quadratic and octal trees require 3 and 7 selection bits for each level of the tree, respectively. This is less efficient than BddCpir, which requires only a single bit for one level. On the other hand, quadratic and octal trees are shallower than binary trees; thus it is not immediately clear as to which scheme offers the best communicative efficiency. This calls for a more detailed inspection of bandwidth requirements of each scheme.

In an n file database, the binary, quadratic and octal trees would have $\log_2 n$, $\log_4 n$, and $\log_8 n$ levels, respectively. According to this, the bandwidth requirements of the encrypted selection bits for each type of the tree are given as in Table 1.

	Client $\rightarrow$ Server (# of bits)
Binary Tree	$[2 + 3 + \ldots + (log_2 n + 1)] \cdot \lvert N \rvert$
Quadtree	$[3 \cdot (2 + 3 + \ldots + (log_4 n + 1))] \cdot \lvert N \rvert$
Octree	$[7 \cdot (2 + 3 + \ldots + (log_8 n + 1))] \cdot \lvert N \rvert$

Table 1: The bandwidth requirements of the selection bits in different tree implementations

The size of the response, which contains the requested data item in encrypted form, is also important since this is a part of the exchanged messages. The bandwidth requirements of the response message sent by the server to the user are $[log_2 n + 1] \cdot |N|$, $[log_4 n + 1] \cdot |N|$, and $[log_8 n + 1] \cdot |N|$ for binary, quadratic, and octal trees, respectively.

The overall communication cost sums up the number of bits exchanged for the selection bits and the response, which is tabulated in Table 2 for different database sizes. The quadratic tree always results in the minimum bandwidth requirements. The binary case is slightly better than octal tree for database sizes given in Table 2. However, the octal tree will eventually be better than the binary tree as the database size increases. For instance, for a database with $n = 4096$ data items, where each data item is 1 Kbit in length, the number of bits exchanged will be the same, namely 105472 bits, for both cases. The octal tree implementation will result in a better communication complexity for a database of more than $n = 4096$ data items.

n	Database size	binary	quadratic	octal
2	2048	4096	-	-
4	4096	8192	7168	-
8	8192	13312	-	16384
16	16384	19456	12288	-
32	32768	26624	-	-
64	65536	34816	31744	38912
128	131072	44032	-	-
256	262144	54272	48128	-
512	524288	65536	-	68608

Table 2: Actual bandwidth costs of overall communication for different database sizes (in number of bits)

7.2 Analysis of Computational Complexity

In this section, we explain why the quadratic and octal tree implementations are better than the binary tree implementation in terms of the efficiency of server-side computations. We provide a theoretical analysis showing that we should expect a speedup in server-side computations. On the other hand, the theoretical analysis fails to give an exact value for the actual speedup, for which we provide the actual implementation results in Chapter 8.

The most fundamental operation of the Damgård-Jurik encryption, on which an overwhelming proportion of server-side computations is spent, is modular exponentiation operation, which has quadratic complexity with the bit length of the modulus (i.e., N^{s+1}). Suppose that a 1024-bit modular exponentiation takes τ seconds (i.e., N is a 1024-bit number). The first exponentiations performed for the lowest level non-sink nodes ($R_{1,j}$) then are expected to take time which is proportional to $\tau_2 = 4\tau$ seconds each since we work with modulo N^2. And the cost of exponentiation increases in a similar manner as we go up in the tree.

For every node of the binary tree, three exponentiations are performed. In quadratic and octal trees, we need five and nine exponentiations, respectively, for a node. For a node in the sth level, we can adopt the following formulas for the computational complexity, $t_s^b = 3 \cdot \tau_s$, $t_s^q = 5 \cdot \tau_s$, and $t_s^o = 9 \cdot \tau_s$, respectively for binary, quadratic and octal trees. Then, the overall time complexity of binary, quadratic, and octal trees can

be estimated using the following formulas

$$
\begin{aligned}
T_{2^m} &= \sum_{s=1}^{m} 2^{m-s} t_s \text{ for } m \geq 1 \\
T_{4^m} &= \sum_{s=1}^{m} 4^{m-s} t_s \text{ for } m \geq 1 \\
T_{8^m} &= \sum_{s=1}^{m} 8^{m-s} t_s \text{ for } m \geq 1,
\end{aligned} \tag{1}
$$

where m is the number of levels in the corresponding tree. Employing the assumptions on the quadratic complexity of modular exponentiation operation with respect to the bit length of the modulus in homomorphic encryption, we can compute an expected speedup values between different tree implementations. For instance, for $n = 512$ (e.g., $m = 9$ and $m = 3$ for binary and octal trees, respectively), the octal tree implementation is expected to achieve a speedup of about 5.32 over a binary tree implementation. As we will show in Chapter 8, the actual speedup for this case will be over 10. There are two reasons for this discrepancy. Firstly, we use asymptotic complexity of modular exponentiations which does not exactly give the actual execution time of the modular exponentiation for a specific operand length. Secondly, the big integer libraries employ specific optimization techniques based on architectural properties of the underlying microprocessor for relatively low bit sizes. As the bit size increases, it becomes difficult to use the same optimization techniques. For example, we incur a severe memory latency due to the fact that we cannot keep the operands in registers when the operands become large.

To obtain the actual timing values of modular exponentiation operations for various bit lengths, we use one of the most optimized big integer libraries available, GMP on an Intel Xeon CPU E1650 processor operating at 3.50 GHz and depict the results along with expected timing results according to asymptotic complexity in Figure 4. In the figure, we take the actual timing value for $s = 1$ and $\log_2(N) = 1024$ as the starting point and compute the asymptotic timings using the quadratic complexity assumption. For instance, the asymptotic timing value for $s = 3$ (corresponding to the case where we work with modulo-N^4) is taken as 8 ms, which is four times the execution time of actual

exponentiation time with modulo-N^2. As can be observed from Figure 4, the actual exponentiation times increase faster than expected, which will benefit quadratic and octal trees. Note that the asymptotic complexity, which is actually quadratic, seems to be linear in Figure 4 when it is depicted along with the actual timing values, which increase much faster.

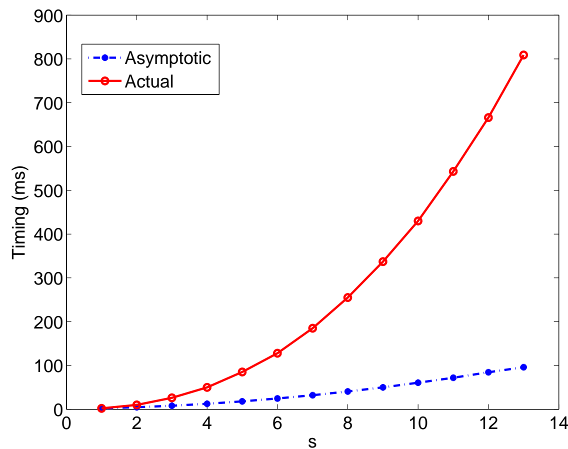

Figure 4: Asymptotic and actual timings of modular exponentiation for different values of s when $\log_2(N) = 1024$.

Using the actual timing values for exponentiation operations on an Intel Xeon CPU E1650 processor operating at 3.50 GHz, we estimate the execution times of server-side computations for various number of data items, and enumerate the results in Table 3. As can be observed from the table, the expected speedup of using octal tree is up to about 11.7 when the number of items is 4096. Note that the estimated speedup values listed in Table 3 are sufficiently close to the actual values given in Table 11, which shows the accuracy of our timing model, given in Equations 1.

7.2.1 Complexity of Parallel Implementation of Binary Tree

In this section, we provide a theoretical analysis for the complexity of the proposed parallel algorithms for three different tree-based CPIRs. We will use two metrics to evaluate the efficiency of the parallel algorithms: i) expected execution time excluding the synchronization overhead and ii) the number and cost of synchronization points.

Number of	Server Computation (ms)		
Items	binary	quadratic	octal
2	4.2	-	-
4	26.4	7	-
8	97.8	-	12.6
16	279.6	58	-
32	703.2	-	-
64	1628	307	154.8
128	3566	-	-
256	7564	1368	-
512	15700	-	1373
4096	131490	23218	11239
32768	1062800	-	90346

Table 3: Estimated timings of server-side computation

The expected execution time uses the time model introduced in Equations 1 while it does not take into account the time spent in synchronization points, which is practically impossible to measure in real systems. Therefore, our estimations for expected execution times in this section will be always less than the actual timing values in Chapter 8. Nevertheless, the estimations can be profitably used to predict the speedup gained through parallelization.

The second metric is the number and cost of synchronization points, during which the processor cores synchronize and possibly exchange data. Naturally, an efficient parallel algorithm minimizes the number and costs of the synchronization points. For instance, Step 18 of Algorithm 10 indicates that two cores computing Step 15 and Step 16 have to synchronize beforehand (i.e., **sync** in Step 17) since in Step 18, the multiplication operation needs both q_0 and q_1 that are computed by two different cores. For example, one core sends q_1 to the other core that has q_0 and can now perform the multiplication in Step 18 of Algorithm 10. Therefore, we need to count these and similar other synchronization points in our analysis.

The other metric is the cost of synchronization point, which is related to the amount of data transferred from one core to the other(s) in a synchronization point. While it is true that multicore processors use a shared-memory model whereby cores share the address space, each core works with data in its own level-1 cache. Thus, a cache

coherency protocol [8] transfers data between the caches of cores when a core needs the data generated by another core. Since the transfer takes place in a system bus at a certain bandwidth, the amount of transferred data affects the time spent in the synchronization point. In CPIR protocol, as computation proceeds to the upper levels of the tree, the amount of data transferred in each synchronization point increases as well. For instance, the synchronization operation in Step 17 of Algorithm 10 requires the transfer of q_1 (or q_0) from one core to the other and the size of q_1 depends on the level of in the tree, namely, s. For example, if we use a 1024-bit modulus in our Damgård-Jurik algorithm with $|N|$-bit modulus, the size of q_1 is $|2N|$-bit and $|3N|$-bit for $s = 1$ and $s = 2$, respectively. We quantify the cost of each synchronization points by the value of s in our analysis.

We start our analysis by estimating the execution time of Algorithm 10 for the binary tree. Assuming that 2^κ is the number of cores and $m \geq \kappa \geq 0$, where 2^m is the number of data items, we can obtain the following formula for the time model of the operations at the server side

$$T^p_{2^m} = T_{2^\sigma} + \sum_{s=\sigma+1}^{m} \lceil 2^{\sigma-s} \cdot 3 \rceil \tau_s, \tag{2}$$

where

$$\sigma = \begin{cases} m - \kappa & m \geq \kappa \\ 0 & \text{otherwise.} \end{cases}$$

For the number of synchronization points, we can derive the following formula

$$S_{2^m} = \begin{cases} 2^\kappa + \sum_{s=1}^{\kappa-1} 2^{\kappa-1-s} + \sum_{s=1}^{\kappa-1} 2^{\kappa-s} & m > \kappa \\ 2^m + \sum_{s=2}^{m} 2^{m-s} + \sum_{s=2}^{m} 2^{m-s+1} & m = \kappa \\ \sum_{s=1}^{m} 2^{m-s} + \sum_{s=1}^{m} 2^{m-s+1} & m < \kappa. \end{cases} \tag{3}$$

The total cost of synchronization points can be computed using the formula

$$CS_{2^m} = \begin{cases} 2^{\kappa-1} \cdot \varkappa + 2^{\kappa-1} \cdot (\varkappa + 1) + \\ \sum_{s=1}^{\kappa-1} 2^{\kappa-1-s}(\varkappa + s) + \\ \sum_{s=1}^{\kappa-1} 2^{\kappa-s}(\varkappa + 1 + s) & m > \kappa \\ 2^{m-1} + 2^{m-1} \cdot 2 + \sum_{s=2}^{m} 2^{m-s} \cdot s + \\ \sum_{s=2}^{m} 2^{m-s+1} \cdot (s+1) & m = \kappa \\ \sum_{s=1}^{m} 2^{m-s} \cdot s + \sum_{s=1}^{m} 2^{m-s+1} \cdot (s+1) & m < \kappa, \end{cases} \tag{4}$$

where $\varkappa = m - \kappa + 1$.

Using Equations 2, 3, 4, we can calculate the estimated expected execution times and precisely calculate the total number and the total cost of synchronization points for different tree sizes and for different number of process cores. The results are given in Table 4, where execution times are in seconds. The timing estimations in Table 4 should be taken into account along with the number and cost of synchronization points. For example, when the number of items is only 64, the gain in the expected execution times diminishes with the number of cores while the number and costs of synchronization points grow very fast. Consequently, we can conclude that using more cores can deteriorate the performance if the number of items is not too high. Using more and more cores benefits only very large trees.

7.2.2 Complexity of Parallel Implementation of Quadratic Tree

In this section we perform the same analysis for the quadratic tree implementation as the one in Chapter 7.2.1 for the binary tree case. Suppose that c is the number of cores and $\lambda = \lceil \log_4 c \rceil$. Then we have

$$T^p_{4^m} = \left\lceil \frac{4^\lambda}{c} \right\rceil T_{4^\sigma} + \sum_{s=\sigma+1}^{m} \left\lceil \frac{4^{m-s} \cdot 5}{c} \right\rceil \tau_s, \tag{5}$$

No. of Items	Perf. Metrics	No. of Cores (c)				
		1	4	8	16	32
	no. of synch.	-	7	17	37	77
64	Time (ms)	1628	450	276	206	181
	syn. cost	-	48	132	320	720
128	Time (ms)	3566	954	553	379	309
	syn. cost	-	56	156	384	880
256	Time (ms)	7564	1978	1098	697	523
	syn. cost	-	64	180	448	1040
512	Time (ms)	15700	4045	2169	1289	888
	syn. cost	-	72	204	512	1200
4096	Time (ms)	131490	33119	16870	8873	4972
	syn. cost	-	96	276	704	1680

Table 4: Estimation of timing values for serial and parallel implementations with different number of processor cores and number of synchronization points and their associated costs - Binary tree case (using GMP library on an Intel Xeon CPU E1650@3.50 GHz)

where

$$\sigma = \begin{cases} m - \lambda & m \geq \lambda \\ 0 & \text{otherwise.} \end{cases}$$

For the number of synchronization points, we can derive the following formula

$$S_{4^m} = \begin{cases} 2 \cdot \varphi + 7 \cdot \sum_{s=1}^{\lambda-1} 4^{\lambda-1-s} & m \geq \lambda \text{ and } c > 0 \\ 7 \cdot 4^{m-1} + 7 \cdot \sum_{s=1}^{m-1} 4^{m-1-s} & m < \lambda, \end{cases} \tag{6}$$

where $\varphi = 4^{\lambda-1} \cdot (2 - \rho) \cdot (2^{\rho+1} - 1)$ and $\rho = \lfloor \frac{c}{4^\lambda} \rfloor$.

The total cost of synchronization points can be computed using the formula

$$CS_{4^m} = \begin{cases} \varphi \cdot \varrho + \varphi \cdot (\varrho + 1) + \\ 3 \cdot \sum_{s=1}^{\lambda-1} 4^{\lambda-1-s} \cdot (\varrho + s) + \\ 4 \cdot \sum_{s=1}^{\lambda-1} 4^{\lambda-1-s} \cdot (\varrho + 1 + s) & m \geq \lambda \text{ and } c > 0 \\ 11 \cdot 4^{m-1} + \\ 3 \cdot \sum_{s=1}^{m-1} 4^{m-1-s} \cdot (s + 1) + \\ 4 \cdot \sum_{s=1}^{m-1} 4^{m-1-s} \cdot (s + 2) & m < \lambda, \end{cases} \quad (7)$$

where $\varrho = m - \lambda + 1$.

Using Equations 5, 6, 7, we can calculate the estimated expected execution times and precisely calculate the total number and the total cost of synchronization points for different tree sizes and for different number of process cores. The results are given in Table 5.

No. of Items	Perf. Metrics	No. of Cores (c)				
		1	4	8	16	32
	no. of synch.	-	6	23	31	99
64	Est. Time (ms)	307	88	47	34	25
	syn. cost	-	21	65	85	193
256	Est. Time (ms)	1368	363	189	116	75
	syn. cost	-	27	88	116	292
1024	Est. Time (ms)	5712	1464	746	411	237
	syn. cost	-	33	111	147	391
4096	Est. Time (ms)	23218	5860	2954	1538	820
	syn. cost	-	39	134	178	490

Table 5: Estimation of timing values for serial and parallel implementations with different number of processor cores and number of synchronization points and their associated costs - Quadratic tree case (using GMP library on an Intel Xeon CPU E1650@3.50 GHz)

7.2.3 Complexity of Parallel Implementation of Octal Tree

In this section, we provide our analysis for the expected execution time, the total number and the total costs of synchronization points in octal tree case. Suppose that 8^m is the number of data items, c is the number of cores and $\lambda = \lceil \log_8 c \rceil$. Then we have

$$T^p_{8^m} = \left\lceil \frac{8^\lambda}{c} \right\rceil T_{8^\sigma} + \sum_{s=\sigma+1}^{m} \left\lceil \frac{8^{m-s} \cdot 9}{c} \right\rceil \tau_s, \tag{8}$$

where

$$\sigma = \begin{cases} m - \lambda & m \geq \lambda \\ 0 & \text{otherwise.} \end{cases}$$

For the number of synchronization points, we can derive the following formula

$$S_{8^m} = \begin{cases} 2 \cdot \vartheta + 15 \cdot \sum_{s=1}^{\lambda-1} 8^{\lambda-1-s} & m \geq \lambda \text{ and } c > 0 \\ 15 \cdot 8^{m-1} + 15 \cdot \sum_{s=1}^{m-1} 8^{m-1-s} & m < \lambda, \end{cases} \tag{9}$$

where $\vartheta = 8^{\lambda-1} \cdot \alpha \cdot (2^\beta - 1)$, $\alpha = \dfrac{8^\lambda}{c}$, and $\beta = \log_2 \dfrac{c}{8^{\lambda-1}}$.

The total cost of synchronization points can be computed using the formula

$$CS_{8^m} = \begin{cases} \vartheta \cdot \varrho + \vartheta \cdot (\varrho + 1) + \\ 7 \cdot \sum_{s=1}^{\lambda-1} 8^{\lambda-1-s} \cdot (\varrho + s) \\ 8 \cdot \sum_{s=1}^{\lambda-1} 8^{\lambda-1-s} \cdot (\varrho + 1 + s) & m \geq \lambda \text{ and } c > 0 \\ 23 \cdot 8^{m-1} + \\ 7 \cdot \sum_{s=1}^{m-1} 8^{m-1-s} \cdot (s + 1) \\ 8 \cdot \sum_{s=1}^{m-1} 8^{m-1-s} \cdot (s + 2) & m < \lambda, \end{cases} \tag{10}$$

where $\varrho = m - \lambda + 1$.

Using Equations 8, 9, 10, we can estimate the expected execution times and precisely calculate the total number and the total cost of synchronization points for different tree sizes and for different number of process cores. The results are given in Table 6.

No. of Items	Perf. Metrics	No. of Cores (c)				
		1	4	8	16	32
	no. of synch.	-	12	14	79	111
64	Est. Time (ms)	155	43	25	13	10
	syn. cost	-	30	35	134	182
512	Est. Time (ms)	1373	355	185	95	58
	syn. cost	-	42	49	213	293
4096	Est. Time (ms)	11239	2831	1429	722	383
	syn. cost	-	54	63	292	404

Table 6: Estimation of timing values for serial and parallel implementations with different number of processor cores and number of synchronization points and their associated costs - Octal tree case (using GMP library on an Intel Xeon CPU E1650@3.50 GHz)

7.3 Analysis of Scalable CPIR

7.3.1 Communication Complexity

The new scalable CPIR incurs an overhead in communication complexity due to the selection bits that are sent to the server. The formula for the number of bits sent to the server by the user for the binary case can be given as

$$\mathrm{BW}_b = (2\mu + 3 + 4 + \ldots + (l + 2))|N|$$

where 2^m, 2^l, and $\mu = 2^{m-l}$ are the number of data items in the entire tree, the number of data items in each subtree ($l < m$), and the number of subtrees, respectively, while N is the modulus used in the Damgård-Jurik cryptosystem. On the other hand, the number of bits sent by the server to the user will be $(l + 2) \cdot |N|$. See Figure 5 for the bandwidth requirements of the scalable solution for the binary case. In the figure, the bandwidth requirements of the scalable solution is compared with the original CPIR scheme (*regular* in the figure) for different number of data items. As can be observed from the figure, there is no significant change (for the worse) in the ratio of the number of exchanged bits between the client and the server to the database size.

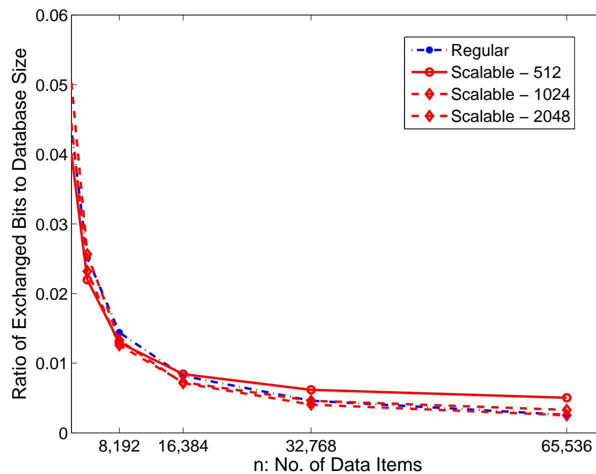

Figure 5: Binary Tree Case: Ratio of exchanged number of bits to database size for $|N| = 1024$, $l = 9, 10, 11$.

Similarly, we can obtain the following equations for the total bandwidth usage for quadratic and octal trees

$$\mathrm{BW}_q = (2\mu_q + 3 + 4 + \ldots + (l_q + 2))|N|$$

and

$$\mathrm{BW}_o = (2\mu_o + 3 + 4 + \ldots + (l_o + 2))|N|$$

where $\mu_q = 2^{m-2l}$, $\mu_o = 2^{m-3l}$, and l_q and l_o represent the depth of the corresponding tree. The effects of the quadratic and octal versions of the scalable solution in the bandwidth requirements are illustrated in Figures 6 and 7, respectively. From both figures, we can conclude that the effect is negligible if the size of the subtrees 2^l for a given m is selected carefully.

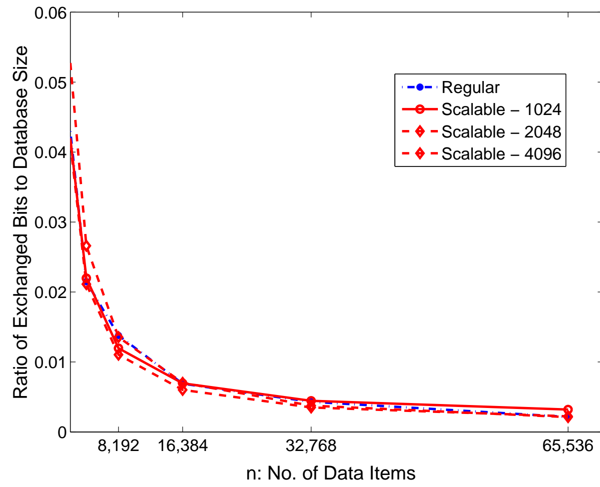

Figure 6: Quadratic Tree Case: Ratio of exchanged number of bits to database size for $|N| = 1024$, $l = 5, 6$.

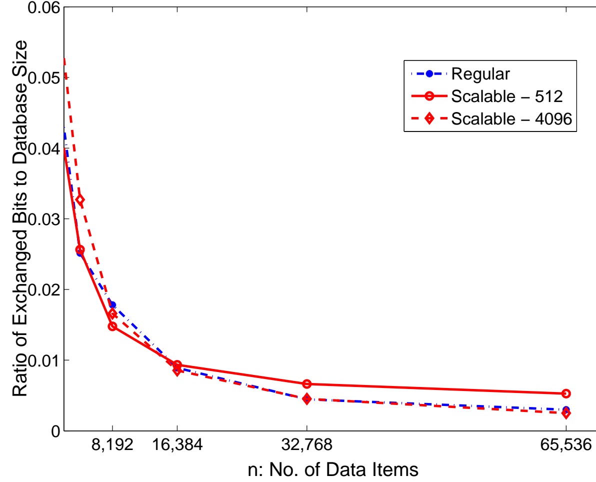

Figure 7: Octal Tree Case: Ratio of exchanged number of bits to database size for $|N| = 1024$, $l = 3, 4$.

7.3.2 Computational Complexity

We also provide our estimations for the expected execution times of the scalable solution in comparison with our earlier parallel implementation for the octal tree case in Tables 7 and 8. We tabulate the execution times only for the octal tree case due to the fact that it has the best time performance. We obtain similar results for the binary and quadratic tree implementations. Note that the figures in Tables 7 and 8 are estimated lower bounds for the expected execution times and do not take into account the time lost due to synchronization points. As pointed out earlier, the estimations are valuable in predicting the speedup figures we can obtain.

In Table 7, we use a subtree with 512 data items and change the total number of data items from 4096 to 65536, where each data item is 1024 bit. As can be observed from the table, we can obtain significant speedups when the number of data items increases, which proves our claim for scalability. The jumps in the speedup values in the table are due to the fact that the octal tree grows by eight when the depth increments by one.

We repeat the same calculations for a larger subtree with 4096 data items in Table 8 and obtain again significant speedup values. Note that having a larger subtree has a bandwidth advantage over a smaller subtree.

no. of cores	Imp.	Exec. Times for No. of Items (s)				
		4 K	8 K	16 K	32 K	64 K
4	parallel	2.83	22.62	22.62	22.62	180.91
	hybrid	2.65	4.09	6.95	9.03	24.16
	speedup	1.07	5.54	3.25	1.78	7.49
8	parallel	1.43	11.34	11.34	11.34	90.49
	hybrid	1.34	2.06	3.49	6.36	12.09
	speedup	1.07	5.51	3.25	1.78	7.48
16	parallel	0.72	5.68	5.68	5.68	45.27
	hybrid	0.68	1.04	1.75	3.19	6.05
	speedup	1.07	5.49	3.24	1.78	7.48
32	parallel	0.38	2.88	2.88	2.88	22.70
	hybrid	0.36	0.54	0.90	1.61	3.05
	speedup	1.06	5.34	3.21	1.78	7.45
64	parallel	0.21	1.48	1.48	1.48	11.41
	hybrid	0.20	0.29	0.47	0.83	1.55
	speedup	1.06	5.07	3.14	1.78	7.38

Table 7: Estimated execution times of the hybrid method for various number of data items, number of cores, and speedup values over the normal parallel implementation; $l = 3$. (using GMP library on an Intel Xeon CPU E1650@3.50 GHz)

no. of cores	Imp.	Exec. Times for No. of Items (s)			
		8 K	16 K	32 K	64 K
4	hybrid	12.59	15.45	21.19	32.66
	speedup	1.80	1.46	1.07	5.54
8	hybrid	6.32	7.75	10.62	16.35
	speedup	1.79	1.46	1.07	5.53
16	hybrid	3.17	3.89	5.32	8.19
	speedup	1.79	1.46	1.07	5.53
32	hybrid	1.62	1.98	2.70	4.13
	speedup	1.77	1.45	1.07	5.49
64	hybrid	0.85	1.03	1.39	2.10
	speedup	1.74	1.44	1.06	5.42

Table 8: Estimated execution times of the hybrid method for various number of data items, number of cores and speedup values over the normal parallel implementation; $l = 4$. (using GMP library on an Intel Xeon CPU E1650@3.50 GHz)

8 Implementation Results

The implementation consists of all the schemes presented in this work, in other words, quadratic and octal tree implementations, their parallelization methods and scalable CPIR are implemented, alongside with the original BddCpir in order to provide a reference point for our results and support the improvements we claimed. Both client and server side timings are provided in terms of computational latency (time to gather a result). For client side secure index calculations (i.e. encrypting selection bits), we always used the parallelized version. For server side calculations, serial, parallel and scalable versions are separately presented, as shown in the subsequent sections.

All the protocols are implemented in C++, using GMP, The GNU Multiple Precision Arithmetic Library, which provides very efficient assembly-level optimizations for big number calculations [15]. For parallelization and threading purposes, OpenMP is utilized since it provides high-level parallelism for C++ codes running on shared-memory multi-core processors [27]. The timings that will be presented in this chapter are obtained on a machine running 64 bit Ubuntu 12.04 LTS. The processor is a 6-core Intel Xeon CPU E5-1650 v2 operating at 3.50 GHz with hyper-threading support. Finally, cryptographic operations are calculated on a 1024-bit modulus, providing 80-bit equivalent security, which is sufficient for PIR applications.

8.1 Client-Side Computations

The client performs encryption operations to build secure indices (i.e. encrypted index bits), and decryption operations to retrieve the requested data item. The encryptions are parallelized using algorithms proposed in Section 5.1, however decryptions are sequential by nature. Table 9 presents parallel client side encryption timings for

binary, quadratic and octal tree implementations, and Table 10 enumerates decryption timings. As the results indicate, quadratic and octal tree based CPIR implementations offer an obvious advantage over the binary tree implementation as far as the client side computation is concerned.

Number of Items	Client Encryption (ms)		
	Binary Tree	Quadratic Tree	Octal Tree
2	2	-	-
4	9	3	-
8	21	-	4
16	34	10	-
32	52	-	-
64	79	24	16
128	113	-	-
256	158	46	-
512	207	-	37
1024	265	82	-
2048	326	-	-
4096	741	130	78
32768	-	-	143

Table 9: Timings of client's selection bit encryptions

Number of Items	Client Decryption (ms)		
	Binary Tree	Quadratic Tree	Octal Tree
2	2	-	-
4	5	2	-
8	10	-	2
16	19	5	-
32	28	-	-
64	40	10	5
128	57	-	-
256	77	19	-
512	101	-	10
1024	129	28	-
2048	161	-	-
4096	360	40	19
32768	-	-	28

Table 10: Timings of client's decryption of the final result

8.2 Server-Side Computations

The server-side computations are the most time and resource consuming part of the CPIR protocols since all the items in the database should be processed in order to reply the client with the appropriate file. Therefore, computational complexity is directly a function of the database. On the other hand, the involved operations are often independent thus allow parallelization methods to be applied as defined in the previous chapters. In the following parts, we present the timing results for both serial and parallel implementations of CPIR and demonstrate our improvements.

8.2.1 Serial Case

The serial CPIR implementation, which corresponds to original BddCpir in binary tree case, utilizes only one core of the processor and performs calculations sequentially. The timings for this case are presented in Table 11.

Number of Items	Server Computation (ms)		
	Binary	Quadratic	Octal
2	5	-	-
4	31	8	-
8	111	-	15
16	315	67	-
32	718	-	-
64	1,659	352	180
128	3,640	-	-
256	7,707	1,440	-
512	15,900	-	1,468
1024	32,750	5,950	-
2048	66,370	-	-
4096	141,370	24,240	11,930
32768	-	-	95,803

Table 11: Timings of server computation - sequential

As observed in the table, with the help of octal trees, we can achieve speedups up to $15900/1468 = 10.83$ for a database with 512 data items. As the number of items increases, the speedup values also increase as expected: for a 4096-file database, the speedup is $141370/11930 = 11.85$.

8.2.2 Parallel Case

Among the three server-side parallelization methods presented in Chapter 5, we provide the timings of the non-trivial one, namely the algorithm introduced in Section 5.4, in Table 12. Obviously, the parallelization on shared memory multi-core processors benefits all the CPIR schemes regardless of the tree type.

Number of Items	Server Computation (ms)		
	Binary	Quadratic	Octal
2	4	-	-
4	22	4	-
8	68	-	6
16	157	28	-
32	332	-	-
64	680	129	64
128	1,390	-	-
256	2,792	504	-
512	5,556	-	505
1024	11,144	2,027	-
2048	22,249	-	-
4096	44,371	8,158	4,033
32768	-	-	32,250

Table 12: Timings of server computation - parallel

The benefit of parallelization is more pronounced when the number of data items is high. For example, with 512 files, we can achieve a speedup of $1468/505 = 2.90$ for the octal tree CPIR and $15900/5556 = 2.86$ for binary tree case, whereas in 4096-file database case, the speedup of octal tree method is $11930/4033 = 2.96$, and binary version is $141370/44371 = 3.19$. As the speedup values indicate, the advantage of the parallel methods are best observed in databases with high number of files.

These results show that using octal tree in the CPIR scheme does not negatively affect the parallelism in the server-side computation. With CPIR schemes we cannot achieve the ideal speedup, which is equal to the number of cores in the computing platform, since the parallelism becomes weaker in the topmost levels of the decision tree, where the encryption operation is the hardest.

Finally, from the binary tree serial implementation to the octal tree parallel implementation the achieved speedup is

$$\frac{141370}{4033} = 35.05.$$

This is an important improvement that brings CPIR schemes one step closer to practical usage.

8.2.3 Scalable CPIR

The results in Table 13 show the actual timings of the Scalable CPIR introduced in Chapter 6. Since the method is designed to be effective on the databases with higher number of items, we only experimented the octal tree based CPIR with depths 4 and 5, in other words, databases with $n = 4096$ and $n = 32768$ number of items are used. For 4096-file case, subtrees with $l = 2$ and $l = 3$ are chosen, and for 32768-file setting, $l = 2$, $l = 3$ and $l = 4$ are used.

Number of Items	Size of a Subtree	Server Computation (ms)
		Octal
4096	64	2,185
	512	2,850
32768	64	16,640
	512	17,330
	4096	22,660

Table 13: Timings of server computation - scalable

As can be observed in the Table 13, subtrees with smaller number of items produce the best results for big databases. However, the cost of getting better computation latencies result in also an increased bandwidth requirement, as demonstrated in the analysis chapter. In short, for large octal trees, when scalable CPIR is used instead of parallelized CPIR, we can achieve

$$\frac{32250}{16640} = 1.94$$

speedup with the cost of increased bandwidth usage.

9 Comparison

There is a relatively high academic interest in efficient PIR schemes [2, 3, 5–7, 10, 12, 18, 19, 23, 24, 28, 31]. One of the earliest proposals are due to Kushilevitz and Ostrovsky [19], which uses a partially homomorphic scheme based on the difficulty of quadratic non-residue problem. The database is arranged as a square matrix of $(\sqrt{n} \times \sqrt{n})$, $\mathcal{D}$, where n is the number of data items. The user sends a homomorphically encrypted bit for each column of the matrix $\mathcal{D}$, where all the bits are 0 except for the bit corresponding to the column that contains the requested data item, which is 1. The database server, then, performs homomorphic computations for each row of $\mathcal{D}$, and sends the resulting ciphertexts back to the user. The user decrypts the ciphertext corresponding to the row that contains the requested data. Overall, the user sends $\beta \cdot \sqrt{n}$ bits to the server that sends back $\beta \cdot \sqrt{n}$ bits for each bit of the requested data item as a response, where β is the size of the ciphertext used in homomorphic encryption scheme.

Another scheme by Boneh et al. [3] uses additive homomorphic computation of two-disjunctive normal form (2-DNF) of polynomials. Disjunctive normal form is also known as sum of products expressions of logical functions in Boolean algebra, which basically means applying logical-OR operation on the product terms obtained by logical AND operation on Boolean variables. In 2-DNF in [3], each product term is logical AND of two Boolean variables. The scheme can use additive homomorphic encryption system proposed by Paillier [29]. In the scheme, the users sends $2 \cdot n^{1/3}$ ciphertexts as the query and receives $n^{1/3}$ ciphertexts as the response. Therefore, the bandwidth complexity is reduced to $O(n^{1/3})$ from $O(n^{1/2})$ of Kushilevitz and Ostrovsky in [19].

Scheme	Query Size		Resp. Size per bit		Resp. Size per KB		Resp. Size per 64 KB	
	2^{16} (KB)	2^{32} (MB)	2^{16} (B)	2^{32} (B)	2^{16} (KB)	2^{32} (MB)	2^{16} (KB)	2^{32} (MB)
[19]	32	1	2^{15}	2^{23}	2^{18}	2^{16}	2^{24}	2^{22}
[3]	32	1	16	512	128	4	8192	256
SWFHE [10]	249	32	250	784	2000	6.125	128159	392
SWFHE - Bundled [10]	0.396	0.03125	250	784	2000	6.125	128159	392
Proposed	42.5	32	0.5	0.75	4	≈ 0.06	256	0.375

Table 14: Comparison of bandwidth requirements in terms of Query and Response Sizes; for the proposed scheme $l = 2^9$ and $l = 2^{15}$ are chosen for the database sizes of 2^{16} and 2^{32}, respectively.

The scheme in [10] uses a somewhat fully homomorphic encryption system (SWFHE), a topic of high interest in the cryptographic community in recent years [4, 13, 14, 22]. Once a fully homomorphic computation is possible (and practical), the selection of the requested data item is reduced to homomorphic comparison of index bits used to address the data items. As comparison circuit is highly simple, a SWFHE based on a variant of NTRU [17] encryption scheme becomes almost practical for PIR implementation. For the security assumptions of the NTRU encryptions schemes, see [32]. One nice property of the PIR scheme in [10], the index bits of the requested data items from different queries can be packed or bundled into a single query which is the homomorphic encryption of these index bits. The bundled case can be especially useful when many queries are generated (perhaps by different users) to amortize the bandwidth overhead of the PIR scheme. This, however, requires a trusted proxy server to collect and bundle queries from different users.

In all three schemes [3, 10, 19], query sizes are reasonably low (see Table 14). Especially, the SWFHE scheme in the bundled case [10] offers extremely small-sized queries. However, the bandwidth complexity has two components query and response size and all three schemes suffer from very high response sizes per one bit of the requested data item as shown in Table 14. In the proposed scheme, the response is at least more than one order of magnitude smaller in size than in any other scheme in Table 14. As can be observed from the table, for even small-sized data items (see the columns 6-9 of Table 14), the response sizes dominate the bandwidth complexity and query sizes become negligible in comparison.

Since the schemes in [3, 19] are not well known for their computational and bandwidth efficiencies we provide a more detailed comparison of the proposed schemes against two more recent schemes in the literature [10, 23, 24], both of which utilize lattice-based cryptography. The former lattice-based scheme introduced in [23, 24], claims computational efficiency while the latter [10], which utilizes SWFHE, claims superior bandwidth performance over the former while accepting the former is computationally much more efficient. We demonstrate that our proposed scheme is always superior so far as the bandwidth efficiency is concerned while the computational efficiency of our scheme is comparable to or better than that in [10], but worse than that in [23, 24]. However, we also show that the scheme in [23, 24] can have such a poor bandwidth performance that it is sometimes better to download the entire database in many circumstances, as also pointed out in [26].

The CPIR schemes that rely on decisional trees use the Damgård-Jurik cryptosystem that is based on the decisional composite residuosity assumption [29], which is a relatively well studied classical problem in comparison with those security arguments used in lattice-based solutions, especially the one in [23, 24].

We compare the bandwidth requirements of the proposed method with octal tree and the two other technique in Figures 8, 9, 10. In Figure 8, assuming that each data item in our database is 1024-bit in size we change the number of data items from 512 to 65536 and illustrate the ratio of exchanged bits (i.e., query + response sizes) to the size of the entire data base. As observed from the graphs given in logarithmic scale in Figure 8, the proposed scheme always offers the best bandwidth performance.

Figure 9 illustrates the bandwidth performances of the three scheme, when the number of data items is fixed to $n = 1024$ and the data items sizes are changed from 1024-bit to 1 million bit. Figure 10 is similar except that $n = 65536$. As pointed out earlier in our discussions regarding the bandwidth performance values in Table 14, the response size dominates when the size of each data item increases. This is apparent in Figures 9 and 10 as the bandwidth performance of the bundled case of the SWFHE scheme is almost same as the regular SWFHE scheme.

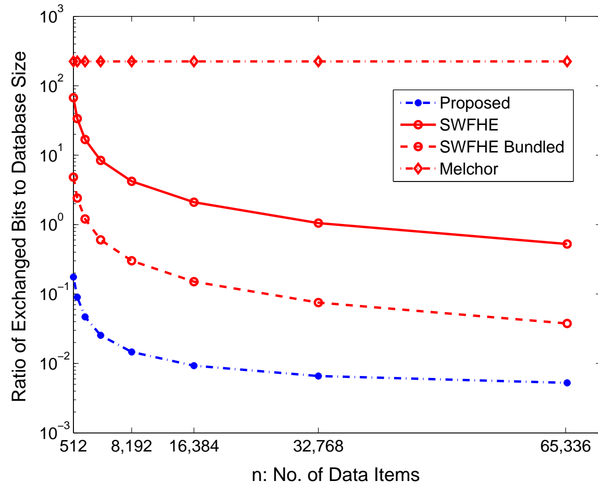

Figure 8: Bandwidth comparison of three schemes when the data item size is 1024-bit; Melchor's scheme in [23, 24], SWFHE and SWFHE - bundled in [10]

In Figures 8, 9, 10, the lattice-based scheme in [23, 24] demonstrates a very poor bandwidth performance. To give a better insight we tabulate the ratios of exchanged information to the database size in each scheme in Table 15 when $n = 1024$ and $|N| = 1024$. As can be observed in the table, the proposed method always results in superior bandwidth performance. The lattice-based scheme in [23, 24] requires the transmission of fewer number of bits than the database size only after the size of the database reaches 128 Mbit. The scheme based on the SWFHE never offers better performance than transmitting the entire database in this setting. The SWFHE-based scheme bandwidth requirements will be acceptable only for databases with many data items. For instance, for a database with 2^{16} items where each data item is 1024-bit, the ratio of exchanged data to database size in the SWFHE-based PIR scheme is 0.53, while it is only 0.03 in the proposed scheme for the same setting.

For server-side computations, the lattice based scheme [23, 24] is reported to offer 230 Mbit/s for a database with only 12 data items, each of which is 3 MB. The proposed method offers about 1 Mbit/s for a database with 512 data items when the parallel implementation of octal tree is used. When more cores are used it is possible to increase the throughput of the server-side computations.

SWFHE-based PIR scheme [10] reports two time performance metrics: i) through-

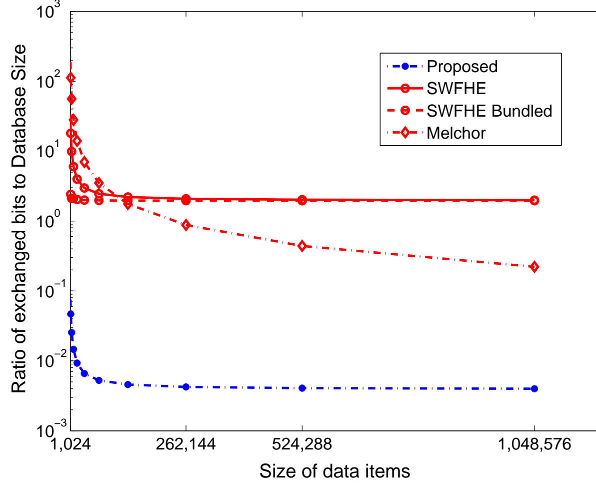

Figure 9: Bandwidth comparison of three schemes with variable data item size and $n = 1024$; Melchor's scheme in [23, 24], SWFHE and SWFHE - bundled in [10]

Data item size (number of bits)	Database size (number of bits)	[23, 24]	[10]	Proposed method
1 K	512 K	224	67.21	0.135
32 K	8 M	14.01	7.96	0.016
128 K	64 M	1.76	4.42	0.009
256 K	128 M	0.88	4.16	0.008
2 M	1 G	0.11	3.94	0.008

Table 15: Ratio of exchanged information to database in different PIR schemes $n = 1024$ and $|N| = 1024$

put when multiple requests are bundled into a single query, hence the *bundled* case, and ii) latency when a request is sent alone (*single* case). In the bundled case for data items of 1024-bit long each, the time spent for processing a data item is given as 0.89 ms while it is 1.00 ms in our scheme. On the other hand, for the latency metric indicating the waiting time for a user, (which is what matters most for the user) the time spent for processing a data item is 16.93 ms. For instance, for a database of 512 items each of which is 1024-bit long, a user has to wait for a query response for about 8.7 s in the SWFHE-based PIR scheme while in our scheme he has to wait for only 0.5 s, which corresponds to more than one order of magnitude improvement on behalf of our scheme.

Furthermore, if we use the scalable CPIR and tolerate a slight increase in bandwidth requirements, we can achieve a better performance. For instance, we can achieve a

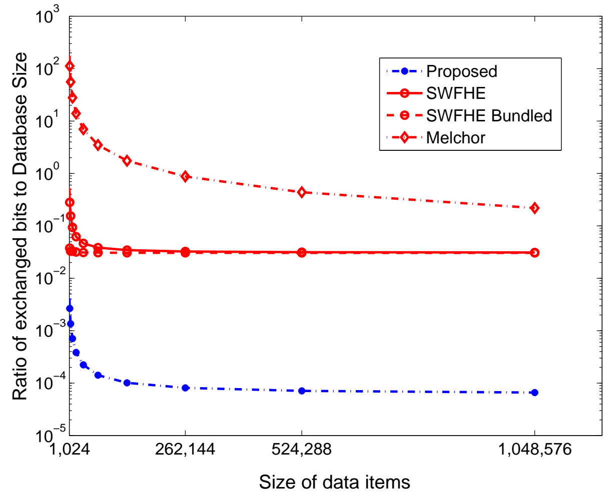

Figure 10: Bandwidth comparison of three schemes with variable data item size and $n = 2^{16}$; Melchor's scheme in [23, 24], SWFHE and SWFHE - bundled in [10]

throughput of 1.44 Mbit/s for a database of 4096 items when octal subtrees with $l = 8$ are used. This will decrease the time spent for processing a data item to about 0.7 ms, whereby the proposed scheme outperforms the SWFHE-based PIR scheme [10] in terms of throughput as well.

10 Conclusion

With the increased usage of cloud servers, this thesis aimed to provide reasonable bandwidth and latency requirements for PIR schemes to protect user privacy. The basis protocol was Lipmaa's BddCpir, which relies on the additively-homomorphic encryption system Damgård-Jurik and binary decision diagrams. We experimented with the data structure and observed the effects of using trees with nodes that have more number of children. As a consequence of our experimentations, we discovered that using quadratic and octal trees improves the performance of server side calculations without adversely effecting the bandwidth efficiency. Going more than 8-child trees, however, starts to increase the bandwidth requirements, thus we limited our interest to quadratic and octal trees. As our implementation results demonstrate, in a 4096-file database with 1024 bit files, using octal trees instead of binary ones decreases the server latency by 11.85 times.

After changing the data structure of original BddCpir and obtaining substantial improvements, we continued to investigate the ways to further decrease server-side latency. The observation of the independent modular exponentiations in the protocols helped us to develop efficient parallelization algorithms for both server and client sides. We implemented the parallelized versions of CPIR with all tree types, and subsequently, achieved a speedup of 35.05 from serial implementation of binary CPIR to parallel CPIR with octal trees for a 4096 Kbit database.

For moderately small database sizes, the parallelization method with octal trees provide a practical server operation time (505 ms for a 512 Kbit database), however, when the number of items or file sizes increase, the scheme again loses its feasibility, for instance, if the server stores a 32 Mbit database, client has to wait for 32.25 s

to receive the reply for a PIR query, even with parallelized octal CPIR. Since this is far beyond practical, we offered a scalable CPIR, to provide applicability of CPIR to bigger databases. With the new octal scalable CPIR method, 32 Mbit databases can be processed within 17 s, which is 1.94 times faster than the parallel octal CPIR. Since we performed our experiments on a machine with 4 cores, this impractical execution time can be improved greatly using computing platforms that feature higher number of cores. Since these type of processors are not rather common, we leave the verification of our claims concerning the scalability of the proposed schemes as future work.

Lastly, we compared the proposed scheme with the schemes in the literature in terms of bandwidth requirements and found out that the new scheme provides bandwidth efficiencies, which are better from than those of the other schemes by one to three orders of magnitude. Also, the adopted security assumption in our scheme is well studied in comparison with the alternative schemes; another reason for further interest in the proposed scheme.

References

[1] Adleman, L. M., Rivest, R. L., and Shamir, A., "Cryptographic communications system and method" *U.S. Patent No. 4,405,829*, 1983.

[2] Ambainis, A., "Upper bound on the communication complexity of private information retrieval", In *Proc. of the 24th ICALP*, 1997.

[3] Boneh, D., Goh, E.-J., Nissim, K., "Evaluating 2-DNF Formulas on Ciphertexts", In *Theory of Cryptography*, LNCS 3378 pp. 325-341, 2005.

[4] Brakerski, Z., Gentry, C., Vaikuntanathan, V., "Fully homomorphic encryption without bootstrapping" In *ITCS* pp. 309-325, 2012.

[5] Cachin, C., Micali, S., Stadler, M., "Computationally Private Information Retrieval with Polylogarithmic Communication", In *EUROCRYPT 99*, pp. 402-414, 1999.

[6] Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M., "Private Information Retrieval", In *FOCS 95: Proceedings of the 36th Annual Symposium on the Foundations of Computer Science*, pp. 41-50, 1995.

[7] Chor, B., Gilboa, N., "Computationally Private Information Retrieval", In *29th STOC*, pp. 304-313, 1997.

[8] Hennessy, J. L., Patterson, D. A., "Computer Architecture, Fifth Edition: A Quantitative Approach", *Morgan Kaufmann Publishers Inc.*, San Francisco, CA, USA, 2011.

[9] Damgård, I., and Jurik, M., "A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System", In *Public Key Cryptography*, pp. 119-136. Springer Berlin Heidelberg, 2001.

[10] Doröz, Y., Sunar, B., and Hammouri, G., "Bandwidth Efficient PIR from NTRU", *Workshop on Applied Homomorphic Cryptography and Encrypted Computing, WHAC'14*, 2014.

[11] ElGamal, T, "A public key cryptosystem and a signature scheme based on discrete logarithms", In *Advances in Cryptology* pp. 10-18, 1985.

[12] Gentry, C., Ramzan, Z., "Single-Database Private Information Retrieval with Constant Communication Rate", In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, pp. 803-815, 2005.

[13] Gentry, C., "A fully homomorphic encryption scheme". *Doctoral Dissertation, Stanford University* 2009.

[14] Gentry, C., Halevi, S., "Implementing Gentrys fully-homomorphic encryption scheme", In *Advances in CryptologyEUROCRYPT*, pp. 129-148, 2011.

[15] GMP, "The GNU MP Bignum Library", Free Software Foundation, `https://gmplib.org/`, 2014.

[16] Goldwasser, S., and Micali, S., "Probabilistic encryption", In *Journal of computer and system sciences 28.2*, pp. 270-299, 1984.

[17] Hoffstein, J., Pipher, J., Silverman, J., "NTRU: A ring-based public key cryptosystem", In *Algorithmic number theory* pp. 267-288, 1998.

[18] Ishai, Y., Kushilevitz, E., "Improved upper bounds on information-theoretic private information retrieval", In *Proc. of the 31th ACM Sym. on TC*, 1999.

[19] Kushilevitz, E., Ostrovsky, R., "Replication Is Not Needed: Single Database, Computationally-Private Information Retrieval", *FOCS '97*, 1997.

[20] Lipmaa, H., "An oblivious transfer protocol with log-squared communication." In *Information Security, Springer Berlin Heidelberg*, pp. 314-328, 2005.

[21] Lipmaa, H., "First CPIR protocol with data-dependent computation", In *Information, Security and Cryptology ICISC 2009*, pp. 193-210. Springer Berlin Heidelberg, 2010.

[22] Lopez-Alt. A., Tromer, E., Vaikuntanathan, V., "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption", In *Proceedings of the 44th symposium on Theory of Computing*, pp. 1219-1234. ACM, 2012.

[23] Aguilar-Melchor, C., Gaborit, P. "A Lattice-Based Computationally-Efficient Private Information Retrieval Protocol", In *WEWORC 2007*, 2007.

[24] Aguilar-Melchor, C., Crespin, B., Gaborit, P., Jolivet, V., Rousseau, P. "High-Speed PIR Computation on GPU", In *SECURWARE'08*, pp. 263-272, 2008.

[25] MSDN, "OpenMP - Using the Schedule Clause." Microsoft Developer Network, `http://msdn.microsoft.com/en-us/library/9w1x7937.aspx`, 2014.

[26] Olumofin, F., and Goldberg, I., "Revisiting the computational practicality of private information retrieval", In *Proceedings of the 15th international conference on Financial Cryptography and Data Security*, pp. 158-172, 2012.

[27] OpenMP, "The OpenMP API specification for parallel programming", The OpenMP ARB, `http://openmp.org/wp/`, 2014.

[28] Ostrovsky, R., Shoup, V., "Private Information Storage", In *29th STOC*, pp. 294-303, 1997.

[29] Paillier, P., "Public-key cryptosystems based on composite degree residuosity classes", In *Advances in cryptology, EUROCRYPT'99*, pp. 223-238. Springer Berlin Heidelberg, 1999.

[30] Rabin, M. O., "How to exchange secrets by oblivious transfer", *Technical Report TR-81*, Aiken Computation Laboratory, Harvard University, 1981.

[31] Sion, R., Carbunar, B., "On the Computational Practicality of Private Information Retrieval", In *NDSS07*, 2007.

[32] Stehlé, D., Steinfeld, R., "Making NTRU as secure as worst-case problems over ideal lattices", In *Advances in Cryptology EUROCRYPT* pp. 27-47, 2011.

[33] Tebaa, M., El Hajji, S., and El Ghazi, A., "Homomorphic encryption applied to the cloud computing security", In *Proceedings of the World Congress on Engineering (Vol. 1)* pp. 4-6, 2012.

[34] Wilkinson, B., Allen, M. *Parallel programming* Vol. 999, New Jersey: Prentice hall, 1999.

[35] Yi, X., Kaosar, M. G., Paulet, R., and Bertino, E. "Single-database private information retrieval from fully homomorphic encryption", *Knowledge and Data Engineering, IEEE Transactions on 25.5*, 1125-1134, 2013.