This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2456021, IEEE Transactions on Parallel and Distributed Systems

On Acceleration and Scalability of Number Theoretic Private Information Retrieval

Ecem Ünal and Erkay Savaş, *Member, IEEE*

***Abstract*—We present scalable and parallel versions of Lipmaa's computationally-private information retrieval (CPIR) scheme [20], which provides log-squared communication complexity. In the proposed schemes, instead of binary decision diagrams utilized in the original CPIR, we employ an octal tree based approach, in which non-sink nodes have eight child nodes. Using octal trees offers two advantages: i) a serial implementation of the proposed scheme in software is faster than the original scheme and ii) its bandwidth usage becomes less than the original scheme when the number of items in the data set is moderately high (e.g., 4,096 for 80-bit security level using Damgård-Jurik cryptosystem). In addition, we present a highly-optimized parallel algorithm for shared-memory multi-core/processor architectures, which minimizes the number of synchronization points between the cores. We show that the parallel implementation is about 50 times faster than the serial implementation for a data set with 4,096 items on an eight-core machine. Finally, we propose a hybrid algorithm that scales the CPIR scheme to larger data sets with small overhead in bandwidth complexity. We demonstrate that the hybrid scheme based on octal trees can lead to more than two orders of magnitude faster parallel implementations than serial implementations based on binary trees. Comparison with the original as well as the other schemes in the literature reveals that our scheme is the best in terms of bandwidth requirement.**

***Index Terms*—Number Theoretic Private Information Retrieval, Security, Privacy, Parallel Algorithms**

I. Introduction

A private information retrieval (PIR) scheme is a protocol that allows a user to access any data item, f_x, in a remotely stored database $\mathcal{F}$ (i.e., $f_x \in \mathcal{F}$), without revealing to the database server the data item being accessed; primarily its index x is not revealed to the server as data items are not necessarily stored in encrypted form. PIR was first introduced in [7] and has recently enjoyed considerably high attention as a result of the raised awareness of privacy concerns pertinent in outsourcing and cloud computing practices. In particular, PIR can help a cloud computing user hide *access patterns* to his data, which is demonstrated to reveal sensitive information.

In computational PIR (CPIR) [8], the difficulty of the server (or any other third party) finding out the index x of the data item during an access can be reduced to a computationally difficult problem. Lipmaa's computationally-private information retrieval (CPIR) protocol [20] suggests using additively-homomorphic encryption algorithm by Damgård and Jurick [9], whose security depends on the well-known decisional composite residuosity assumption while other schemes in the literature depend on relatively less studied lattice problems as in [1, 2]. There are also other recent schemes based on fully homomorphic encryption techniques such as the one in [11].

Naturally, the user can always download the entire database to obtain the requested data item (i.e., *the trivial solution*). Therefore, the amount of data exchanged between the user and the server during a PIR protocol run must be much smaller than the database size. More formally, the exchanged data amount must be sublinear to the size of the database. While several techniques [1, 2, 11] successfully accelerate the server-side computations, their bandwidth performance are not always acceptable. The Lipmaa's scheme (also known as BddCpir due to its use of binary decision diagrams) is known to offer superior bandwidth performance due to its log-squared asymptotic communication complexity.

On the other hand, the BddCpir scheme is not one of the best schemes in the literature in terms of computational complexity. Therefore, the primary aim of this paper is to accelerate the Lipmaa's original scheme. The proposed schemes can also improve its communication complexity for sufficiently large databases.

a) Our Contribution: Firstly, we provide new, improved versions of the original BddCpir [20] using octal trees, which provide a significant improvement in the computational complexity as well as a limited improvement in the bandwidth performance. Secondly, we propose a non-trivial, efficient parallel algorithm for server-side computations. The new parallel algorithm (i.e., Algorithm 2), relying on partitioning of the database into subtrees, proves to be faster than Algorithm 1 in [29]. Thirdly, we give an in-depth theoretical analysis of the proposed parallel algorithm to estimate the speedup values that can be obtained for various database sizes when different number of processor cores are used. In our analysis, we show that the proposed parallel algorithm reduces the number of synchronization points, which allows the processor cores to work independently. Since the computational complexity increases with the number of data items, the scheme becomes impractical for large databases. We present, therefore, a slightly different method (i.e., the hybrid approach in Section V) to scale the CPIR scheme to work with relatively large database sizes. While the scheme in [29] provides implementation results only for a database with 512 items on a four-core processor, we provide experimental results on computing platforms featuring as many as 30 cores for databases with up to 262,144 items. Finally, we give a comparison of the bandwidth requirements of the proposed technique with those in the literature, and show that the proposed technique is the best due to its log-squared asymptotic communication

A preliminary version of this paper appeared in [29].

E. Ünal and E. Savaş are with the Faculty of Engineering and Natural Sciences, Sabancı University, İstanbul TURKEY, e-mail: ({ecemunal, erkays}@sabanciuniv.edu).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2456021, IEEE Transactions on Parallel and Distributed Systems

complexity.

II. Background

In the literature, there is a plethora of publications on private information retrieval (PIR) protocols. One major category is information theoretically secure PIR schemes (IT-PIR) [18]. A recent IT-PIR scheme by Goldberg [15] uses multi-server approach with threshold cryptography, where privacy is information theoretically guaranteed as long as no more than t (i.e., the threshold) servers collude. Other category of PIR schemes includes computationally secure solutions, referred as CPIR. A CPIR scheme bases its security on hard problems such as integer factorization, decisional composite residuosity problem, and lattice problems. The works in [1, 2, 4, 6–8, 11, 20, 22–24] are some of the prominent examples of CPIR in the literature. A recent work [10] proposes a hybrid scheme that combines IT-PIR and CPIR. Most of the PIR schemes suffer from high communicational complexity as many PIR schemes have $O(\sqrt(n))$ complexity, where n is the number of items in the database. Two exceptions are the works by Doröz et al. [11] and Lipmaa [20], which provide log and log-squared asymptotic complexities for communication, respectively. An in-depth comparison of different schemes in terms of computation and communication complexities are provided in Section VII.

Due to its superior communicational complexity, we use Lipmaa's (n, 1) - CPIR protocol, BddCpir [20], as a starting point. Binary decision diagrams (*BDD*) and an additively-homomorphic public-key cryptosystem [9] are two important building blocks in the BddCpir [20] scheme. In this section, we provide the basics of the BDD and the Damgård-Jurik cryptosystem [9], which is an additively homomorphic encryption scheme.

A. Binary Decision Diagrams and Homomorphic Encryption

A binary decision diagram is similar to binary trees, where each node has at most two children except for leaf (sink or terminal) nodes.

Properties of a BDD: In a binary decision diagram, non-sink nodes are labeled as $R_{i,j}$, where i and j denotes the level (the root is in the highest level) and the position of the node in a level, respectively. Also, two outgoing edges of the internal nodes are labeled as 0 or 1. The sink nodes, however, are data items, whose indices are m-bit strings, representing the route taken from the root node to a sink node, where m is the depth of the tree. In other words it is the concatenation of the labels of the edges that are visited while reaching the sink node from the root node.

In BddCpir, data items in sink nodes are privately retrieved on user inputs. Thus, the labels of the sink nodes are used as indices to retrieve data items. More precisely, to retrieve the data item f_x, stored in the sink node with the label x of length m-bit, the client includes x in his query to the server. In Figure 1, the binary decision tree of depth two represents a database with four data items, namely f_0, f_1, f_2, f_3.

In CPIR, to access a data item, its index is encrypted using an additively-homomorphic public-key cryptosystem before it is sent to the server as will be explained in subsequent sections. Similarly, the requested data item is returned also in homomorphically encrypted form. An additively-homomorphic public key cryptography algorithm satisfies the following important homomorphic properties over encryption operation $E(x_1) \cdot E(x_2) = E(x_1 + x_2)$ and $E(x_1)^c = E(c \cdot x_1)$, where x_1 and x_2 are plaintext messages, and c is a constant.

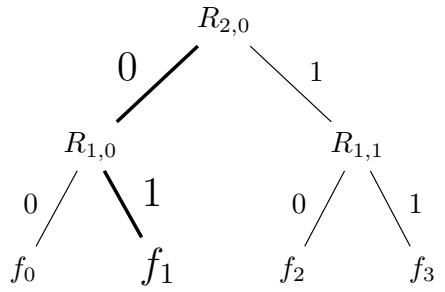

Figure 1. An example BDD constructed by the server, shows the case where the client query is $x = 01$ to access file f_1.

Octal Trees: For performance reasons, instead of the binary decision diagrams, we propose using octal trees in our protocol. This new type of tree has essentially the same properties as the binary trees, except that internal nodes have eight children. In fact, one can also use other trees with different number of child nodes such as quadratic trees with four child nodes as used in [29]. However, since the bandwidth advantage of quadratic trees is rather marginal and octal trees offer significant computational improvement, we focus our attention on octal trees. In an octal tree, the edges of the internal nodes are labeled by three-bit strings, namely $\{000, 001, \ldots, 111\}$ and hence the labels of the sink nodes have $3m$-bit strings, where m again represents the depth of the tree.

B. (n, 1) - CPIR

In this section, we first explain the (2,1) CPIR sub-protocol, which is the base of the (n, 1) CPIR scheme in [20]. Then, we show how it is extended to a database with n items.

(2, 1) - CPIR: In (2,1)-CPIR protocol, the server holds only two data items, namely $\{f_0, f_1\}$; therefore the client's input x is either 0 or 1 since it only requests one of $\{f_0, f_1\}$. The PIR protocol ensures that the client receives f_x while the server obtains no information about x. In PIR, the files themselves are not necessarily stored in encrypted form, while the client receives the response in encrypted form, which requires a final decryption by the client. $(2, 1)$ - CPIR [20] works in four steps:

1) Client generates private and public keys (sk, pk); not necessarily repeated for every execution of PIR.
2) Client computes $c = E_{pk}(x)$ (simply $E(x)$ henceforth since encryption is always performed using the public key, pk) and sends (pk, c) to the server.
3) Server computes $R = E(f_0) \cdot c^{f_1 - f_0}$ and sends R to the client.
4) Client computes $D_{sk}(R)$ (simply $D(R)$ henceforth) to find f_x.

Since the cryptosystem used for encryption and decryption is additively homomorphic we can prove that the client will

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2456021, IEEE Transactions on Parallel and Distributed Systems

get f_x at the end of the protocol as

$$R = E(f_0) \cdot c^{f_1 - f_0} = E(f_0) \cdot E(x)^{f_1 - f_0} = E(f_0 + x(f_1 - f_0)) = E(f_x).$$

Extending (2, 1)-CPIR to (n, 1)-CPIR: The (2, 1)-CPIR protocol can be used to construct the (n, 1) - CPIR protocol for databases with n data items in the general case. Assuming n being a power of two for sake of simplicity, the PIR computation starts with sink nodes paired in two. Then, the computation continues upward and stops at the root node. In the end, the requested data item is obtained, which is encrypted as many times as the depth of the tree.

Example 1: The server computation of the (4, 1)-CPIR protocol for data items $\{f_0, f_1, f_2, f_3\}$ is implemented for the user input $x = (x_1, x_0)$ in two steps as follows. In the first step, we calculate

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

where $c_0 = E(x_0)$. In the second step, we work with the ciphertexts obtained from the previous step as

$$\begin{aligned} R_{1,0} &= E(R_{1,0}) \cdot c_1^{R_{1,1} - R_{1,0}} \\ &= E(R_{1,0} + c_1 \cdot (R_{1,1} - R_{1,0})) \\ &= E(E(f_{0x_0}) + c_1 \cdot (E(f_{1x_0}) - E(f_{0x_0}))), \end{aligned}$$

where $c_1 = E^{(2)}(x_1)$. Therefore, we obtain the double encryption of f_x, namely $E^{(2)}(f_x)$, which is sent to the user. In the general case, the client receives $E^{(m)}(f_x)$, where m is the depth of the binary tree. Note that $c_i = E^{(i+1)}(x_i)$ for $i = 0, \ldots, m-1$.

C. Damgård - Jurik Cryptosystem

The Lipmaa's CPIR [20] scheme suggests using an additively-homomorphic public key encryption, which allows multiple encryptions. The Paillier public key cryptosystem [25] provides additive homomorphic property, however, it cannot be used directly in (n, 1)-CPIR since it does not allow encryption of a ciphertext. Therefore, the Damgård-Jurick public key cryptosystem [9], which is a generalization of the Paillier scheme and provides multiple encryption property, is used in the proposed scheme.

The Damgård - Jurik cryptosystem uses a setting similar to the RSA algorithm, where we work in a ring of integers Z_N^*, and N is the product of two large prime numbers, p and q. The security of the Damgård Jurik cryptosystem relies on the decisional composite residuosity assumption [25], which is also used in the Paillier cryptosystem. The encryption algorithm, which is the most-time consuming part of the proposed scheme, is performed as follows.

Given a plaintext $m \in Z_{N^s}$, we choose a random number $r \in_R Z_{N^{s+1}}^*$ and compute the ciphertext as $E(m, r) = g^m r^{N^s} \bmod N^{s+1}$, where $g = N + 1$. For the key generation and decryption operations, one can refer to [9].

The natural number s in the encryption operation changes from 1 to m during the PIR computations. The encryptions in the sink nodes are performed with $s = 1$, those in the second level will be done with $s = 2$, and so on. Since s increments as we move to upper nodes in the tree, the PIR calculations become more costly in terms of computation.

Example 2: For a tree with 2^m data items in its sink nodes, the encrypted index values are formed as

$$c_i = g^{x_i} r_i^{N^{i+1}} \bmod N^{i+2},$$

where $r_i \in_R Z_{N^{i+2}}^*$ for $i \in \{0, \ldots, m-1\}$.
Considering the quadratic complexity of the Damgård-Jurik encryption operation with respect to the modulus size, the time complexity of the CPIR scheme will be prohibitively high even for databases with moderately high number of data items. The continuous message expansion with multiple encryptions hinders the scalability of the CPIR scheme.

III. Requirements of Efficient PIR

PIR protocols are designed to reduce bandwidth requirements which must be sublinear to the database size. In oblivious transfer protocols [26, 30] the user is allowed to retrieve at most one of the database items, which usually results in that the number of bits exchanged between the server and the user become larger than the database itself. Removing this additional privacy requirement, PIR protocols can offer more bandwidth efficient solutions.

The following performance metrics can be used to evaluate the efficiency of a PIR protocol:

- **Bandwidth Efficiency:** The total size of a PIR query and its response must be significantly smaller than the database size. Thus, a good metric is the ratio of the total size of the exchanged data to the database size. An efficient PIR scheme should aim to optimize both query and response sizes.
- **Computational Efficiency:** PIR protocols usually require expensive cryptographic operations. Computational efficiency is expressed usually from the *throughput* perspective; namely the number of data items or database size processed in a unit time. However, as users tolerate waiting for only a limited amount of time for a query processing, the latency is also important.
- **Scalability:** A PIR should scheme remain applicable as the number of data items and/or database size increase. PIR Schemes suitable to parallel implementations will be advantageous for scalability. In this work, we propose algorithms that benefit parallel implementations.

In the next section, we briefly introduce our key technique that outperforms the original BddCpir scheme in terms of both computational and bandwidth complexities.

IV. Our Approach

We utilize three techniques to improve the scalability, computational and bandwidth efficiency of the CPIR scheme. The first technique involves using octal trees, which decreases the tree depth. The second technique is the utilization of parallel algorithms that take advantage of shared-memory multi-core processors. And finally, the third technique is a hybrid scheme which scales PIR to large databases with a relatively small increase in the bandwidth requirement.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2456021, IEEE Transactions on Parallel and Distributed Systems

Using other trees, such as quadratic and hexadecimal with four and 16 child nodes, respectively, can also considered. For instance, quadratic trees, given in Appendix A and analyzed in [29] are slightly better than octal trees in terms of bandwidth requirement (see Figure 2), but not as fast as octal trees. Hexadecimal trees can offer computational advantages, but suffer from high bandwidth requirement. We will focus on octal tree-based solution, which provides a good balance between the computational and communication complexity, and give results for the other trees occasionally for comparison purposes.

A. (n,1)-CPIR using Octal Tree

Octal tree decreases the depth, which helps improve the complexity of the overall system; particularly the complexity of cryptographic operations when the number of data items is high. While the number of indices sent by the user is increased, using octal trees, in fact, improves the overall bandwidth even for relatively large databases due to their shallow depth.

Assuming that the number of nodes is a power of 8, namely $n = 8^m$, the client sets his private and public keys (sk, pk) and computes

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

for $s = 1, \ldots, m$ and sends them and pk to the server. The details of the server computation are given in Algorithm 5 in Appendix B. The server finally obtains $R_{m,0}$ and sends it to the client. The client performs the decryption $D^{(m)}(R_{m,0})$ to retrieve f_x.

Example 3: For an octal tree with a total of eight sink nodes (i.e., data items), the client sends $c_i = E(x_i)$, for $i \in \{0, 1, 2\}$, $c_{0,1} = E(x_0 \cdot x_1)$, $c_{0,2} = E(x_0 \cdot x_2)$, $c_{1,2} = E(x_1 \cdot x_2)$, and $c_{0,1,2} = E(x_0 \cdot x_1 \cdot x_2)$ to the server that computes the following:

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

B. A Parallel Algorithm for Server-Side Computation

In this section, we introduce a highly efficient parallel algorithm for server side computations of the (n,1) CPIR scheme. The client can encrypt selection bits in parallel as their encryptions are independently performed. Therefore, client side computations can be accelerated by a trivial parallel algorithm when multiple cores are available. On the other hand, server-side computations, composed of both parallel and sequential operations, can lead to different parallel algorithms.

A serial algorithm for server-side computations using a binary tree-based (n, 1)-CPIR is given in Algorithm 1. As can be observed from the serial algorithm, the operations in a level in the decision tree are independent from each other and can be performed in parallel. In addition, the homomorphic encryption operation in each level of the tree consists of two modular exponentiation operations (i.e., $g^m \bmod N^{s+1}$ and $r^{N^s} \bmod N^{s+1}$) that can also be calculated in parallel.

Algorithm 1 SerCPIR($\mathcal{C}$, $\mathcal{F}$): Serial algorithm for server-side computation for binary tree-based (n,1)-CPIR

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

```
for j ← 0 to 2^m − 1 do
    R_{0,j} ← f_j
end for
for s ← 1 to m do
    for j ← 0 to 2^{m−s} − 1 do
        f_0 ← R_{s−1,2j}
        f_1 ← R_{s−1,2j+1}
        R_{s,j} ← E^(s)(f_0) · c_{s−1}^{f_1−f_0} mod N^{s+1}
    end for
end for
return R_{m,0}
```

For parallelization of server-side computations, all independent operations in a level can be performed concurrently starting from the leaf nodes as proposed in Algorithm 1 in [29]. However, this approach suffers from increased number of synchronization points; namely the cores have to communicate more frequently, and therefore block waiting for each other.

The proposed parallel algorithm, depicted in Algorithm 2 on the other hand, divides the binary tree into as many subtrees as the number of available cores assuming that the number of cores is a power of 2 (i.e., 2^κ) and less than the number of data items (i.e., 2^m) in the database. Each processor core, then, works on the assigned subtree in isolation (*cf.* Step 5 of Algorithm 2). When each core finishes its portion of the task, the cores synchronize and start working with the upper levels of the tree concurrently (*cf.* Steps 7–17 of Algorithm 2). As can be observed from Steps 7–17 of the algorithm, there are two levels of parallelism. First, the computation of $R_{s,j}$ values are distributed among the available cores in the first level while the two exponentiations needed for each $R_{s,j}$ are also distributed in the second level (*cf.* Steps 11–14 of Algorithm 2). This implies a finer parallelization in the second level, which is more meaningful for the computations in the top levels of the tree. In [29], this technique is used for the entire tree, which results in a slower implementation. Algorithm 2 reduces the number of the synchronization points as will be discussed in detail in Section IV-C.

C. Analysis of Computational Complexity

In this section, we explain the advantages of octal tree implementations when compared with binary tree implementations. The theoretical analysis in this section clearly shows that server-side computations can be accelerated considerably when implemented using Algorithm 2. Furthermore, the theoretical analysis yields expected speedup values, which are

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2456021, IEEE Transactions on Parallel and Distributed Systems

Algorithm 2 Parallel algorithm for server-side computation for binary tree based (n,1)-CPIR

Require: $\mathcal{C} = \{c_0, \ldots, c_{m-1}\}$, $\mathcal{F} = \{f_0, \ldots, f_{2^m-1}\}$, and $\kappa < m$
Ensure: $R_{m,0}$
1: $\delta \leftarrow 2^{m-\kappa}$
2: $\mathcal{C}_0^{m-\kappa-1} \leftarrow \{c_0, \ldots, c_{m-\kappa-1}\}$
3: **for** $j \leftarrow 0$ to $2^\kappa - 1$ **in parallel do**
4: $\quad \mathcal{F}_{j\delta}^{j\delta+\delta-1} \leftarrow \{f_{j\delta}, \ldots, f_{j\delta+\delta-1}\}$
5: $\quad R_{m-\kappa,j} \leftarrow$ SerCPIR($\mathcal{C}_0^{m-\kappa-1}$, $\mathcal{F}_{j\delta}^{j\delta+\delta-1}$)
6: **end parallel for**
7: **for** $s \leftarrow m-\kappa+1$ to m **do**
8: $\quad$ **for** $j \leftarrow 0$ to $2^{m-s}-1$ **in parallel do**
9: $\quad\quad f_0 \leftarrow R_{s-1,2j}$
10: $\quad\quad f_1 \leftarrow R_{s-1,2j+1}$
11: $\quad\quad$ **in parallel do**
12: $\quad\quad\quad u_0 \leftarrow E^{(s)}(f_0)$
13: $\quad\quad\quad u_1 \leftarrow c_{s-1}^{f_1-f_0} \bmod N^{s+1}$
14: $\quad\quad$ **sync**
15: $\quad\quad R_{s,j} \leftarrow u_1 \cdot u_0 \bmod N^{s+1}$
16: $\quad$ **end parallel for**
17: **end for**
18: **return** $R_{m,0}$

not exact, but useful to provide an upper bound for the actual speedup values. Section VI provides the actual implementation results.

When Algorithm 2 is inspected, one can observe that two fundamental operations are performed: the Damgård-Jurik encryption and modular exponentiation. Also, the Damgård-Jurik encryption consists of two modular exponentiations. Therefore, Algorithm 2 consists of many modular exponentiation operation, whose complexity is quadratic with the bit length of the modulus (i.e., N^{s+1}).

If we assume that a 1,024-bit modular exponentiation takes τ seconds (i.e., $\lceil \log_2(N) \rceil = 1024$), then we can conclude that an exponentiation with $s = 1$ (i.e., in the sink nodes) are expected to take time proportional to $\tau_1 = 4\tau$ seconds since we work with modulo N^2. And the cost of exponentiation can be assumed to increase quadratically as we move to the upper level nodes in the tree. For the general case, in the sth level of the tree, one exponentiation takes expectedly $\tau_s = (s+1)^2 \cdot \tau$ if we strictly use the quadratic complexity assumption.

Three and nine exponentiations are performed for every node in binary and octal trees, respectively. For a node in the s^{th} level, we can adopt the following formulas for the computation complexity, $t_s^b = 3 \cdot \tau_s$ and $t_s^o = 9 \cdot \tau_s$, respectively for binary and octal trees. Then, the overall time complexity of binary and octal trees can be estimated using the following formulas

$$\begin{aligned} T_{2^m} &= \sum_{s=1}^{m} 2^{m-s} t_s^b \text{ for } m \geq 1 \\ T_{8^m} &= \sum_{s=1}^{m} 8^{m-s} t_s^o \text{ for } m \geq 1, \end{aligned} \quad (1)$$

where m is the depth of the corresponding tree. Relying on the assumption of the quadratic complexity of modular exponentiation operation with respect to the bit length of the modulus in homomorphic encryption, we can compute the expected speedup values between different tree implementations.

Example 4: For $n = 512$ (e.g., $m = 9$ and $m = 3$ for binary and octal trees, respectively), using the quadratic complexity assumption, the exponentiations in different levels of the tree take $\tau_1 = 4\tau$, $\tau_2 = 9\tau$, ..., $\tau_9 = 300\tau$ seconds. Using the formulas in Eq. 1, we calculate $T_{2^m} = 16,458\tau$ s and $T_{8^m} = 3,096\tau$ s. Thus, the octal tree implementation is expected to achieve a speedup of about $\frac{16,458\tau}{3,096\tau} \approx 5.32$ over the binary tree implementation.

As we will show in Section VI, the actual speedup for this case will be over 10. There are two reasons for this discrepancy. Firstly, in our estimations we use the asymptotic complexity of modular exponentiations which does not exactly reflect 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. See Figure 7 in Appendix C for the actual timing values of exponentiations with different modulus lengths.

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 I. As can be observed from the table, the expected speedup of using octal tree is about 11.76 when the number of items is 32,768. Note that the estimated timings listed in Table I are sufficiently close to the actual timings enumerated in Table VI, which shows the accuracy of our timing model, captured in Eq. 1.

Number of Items	Server Computation (ms)	
	binary	octal
2	4.2	-
4	26.4	-
8	97.8	12.6
16	279.6	-
32	703.2	-
64	1,628	154.8
128	3,566	-
256	7,564	-
512	15,700	1,373
4,096	131,490	11,239
32,768	1,062,800	90,346

Table I
ESTIMATED TIMINGS OF SERVER-SIDE COMPUTATION (USING GMP LIBRARY ON AN INTEL XEON CPU E1650@3.50 GHZ)

1) Complexity of Parallel Implementation of Binary Tree: In this and the next sections, we provide a theoretical analysis for the complexity of the proposed parallel algorithms for two 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 costs of synchronization points. For the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2456021, IEEE Transactions on Parallel and Distributed Systems

expected execution time we use the timing model introduced in Eq. 1 while we do not take into account the time spent in synchronization points, which is practically impossible to model in real systems. Therefore, our estimations for expected execution times in this section will be always less than the actual timing values in Section VI. Nevertheless, the estimations can be profitably utilized to predict speedups gained through parallelization; in the worst case we can predict if there is a speedup using the estimations with certain accuracy and compare different parallel algorithms.

The second metric is the number and costs 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 14 of Algorithm 2 indicates that two cores computing Steps 12 and 13 have to synchronize since in Step 15, the multiplication operation needs both u_0 and u_1 that are computed by two different cores. For example, one core sends u_1 to the other core that has u_0 and can now perform the multiplication in Step 15 of Algorithm 2. Therefore, we need to count these and similar other synchronization points in our analysis.

The cost of a synchronization point, not considered as a separate metric, 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 can access the same address space, each core works with data in its own level-1 cache most of the time. Thus, a cache coherency protocol [16] 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 the 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 14 of Algorithm 2 requires the transfer of u_1 (or u_0) from one core to the other and the size u_1 depends on the current level of the tree, namely, s. For example, if we use a 1,024-bit modulus in our Damgård-Jurik algorithm with $|N|$-bit modulus, the size of u_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 2 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 have 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 estimated 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 Eqs. 2, 3, 4, we can calculate the estimated expected execution times and the total number and the total cost of synchronization points for different tree sizes and for different number of process cores. Translating the numbers and costs of synchronization points into actual time estimations is not attempted since the underlying processor technologies adopt different architectures, techniques and algorithms for cache coherency, which renders any estimation inaccurate. The results are given in Table II, where the execution times are in milliseconds. The timing estimations in Table II should be taken into account along with the number and costs 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 an excessively high number of cores benefits only very large trees.

2) 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 cost of synchronization points in the 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

$$T^p_{8^m} = \left\lceil \frac{8^\lambda}{c} \right\rceil T_{8^\sigma} + \sum_{s=\sigma+1}^{m} \left\lceil \frac{8^{m-s} \cdot 9}{c} \right\rceil \tau_s, \tag{5}$$

where

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2456021, IEEE Transactions on Parallel and Distributed Systems

No. of Items	Perf. Metrics	No. of Cores (l) 1	4	8	16	32
	no. of synch.	-	7	17	37	77
64	Time (ms)	1,628	450	276	206	181
	syn. cost	-	48	132	320	720
128	Time (ms)	3,566	954	553	379	309
	syn. cost	-	56	156	384	880
256	Time (ms)	7,564	1,978	1,098	697	523
	syn. cost	-	64	180	448	1,040
512	Time (ms)	15,700	4,045	2,169	1,289	888
	syn. cost	-	72	204	512	1,200
4,096	Time (ms)	131,490	33,119	16,870	8,873	4,972
	syn. cost	-	96	276	704	1,680

Table II
ESTIMATION OF TIMING VALUES FOR SERIAL (WITH ONE CORE) AND PARALLEL IMPLEMENTATIONS WITH DIFFERENT NUMBER OF PROCESSOR CORES (WITH 4, 8, 16 AND 32 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)

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

$$S_{8^m} = \begin{cases} 2 \cdot \vartheta + 15 \cdot \sum_{s=1}^{\lambda-1} 8^{\lambda-1-s} & m \geq \lambda \text{ and } c > 0 \\ 15 \cdot 8^{m-1} + 15 \cdot \sum_{s=1}^{m-1} 8^{m-1-s} & m < \lambda, \end{cases} \tag{6}$$

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

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

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

Using Eqs. 5, 6, 7, the estimated expected execution times, and the total number and the total cost of synchronization points for different tree sizes and different number of process cores are calculated and tabulated in Table III.

D. Analysis of Communication Complexity

In this section, we analyze the bandwidth requirements of the proposed scheme based on octal trees and explain why it can be better than the binary tree-based approach. A bandwidth-efficient PIR scheme requires exchanging of much less data between the user and the server compared to the database size. Formally speaking, the bandwidth of a PIR scheme must be sublinear to the size of the database. The bandwidth of the PIR scheme [20], independent of the tree type used, has log-square complexity. However, actual implementations using different trees have different bandwidth requirements, which turns out to be important in practice.

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)	1,373	355	185	95	58
	syn. cost	-	42	49	213	293
4,096	Est. Time (ms)	11,239	2,831	1,429	722	383
	syn. cost	-	54	63	292	404

Table III
ESTIMATION OF TIMING VALUES FOR SERIAL (WITH ONE CORE) AND PARALLEL IMPLEMENTATIONS WITH DIFFERENT NUMBER OF PROCESSOR CORES (WITH 4, 8, 16 AND 32 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)

In the PIR protocol, the client sends encrypted selection bits to the server in the first phase and receives the encrypted data item in the second phase. In binary decision tree, the number of selection bits is $\lceil \log_2 n \rceil$, 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 encrypted 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 more general case, the selection bit for the s^{th}-level, $c_{s-1} = E^{(s)}(x_1)$ will be $(s+1)|N|$-bit long.

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

The binary and octal trees have $\log_2 n$, and $\log_8 n$ levels, respectively. In the most general case, for a tree in which each non-sink node has 2^x child nodes the bandwidth requirements for the encrypted selection bits can be written as

$$[(2^x - 1) \cdot (2 + 3 + \ldots + (\lceil \log_{2^x} n \rceil + 1))] \cdot |N|.$$

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 requirement of the response message sent by the server to the user is $m + 1 \cdot |N|$ bits, where $m = \lceil \log_{2^x} n \rceil$ is the depth of the corresponding tree.

The overall communication cost, namely the total size of encrypted indexes and the response of a PIR query, is tabulated in Table IV for different database sizes. The quadratic tree implementation always provide the best bandwidth performance for all database sizes. The octal tree implementation is only slightly worse than the binary case for the database sizes in Table IV. However, the octal tree will eventually be better than

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2456021, IEEE Transactions on Parallel and Distributed Systems

the binary tree as the database size increases. For instance, for a database with $n = 4,096$ data items, where each data item is 1 Kbit in length, the number of bits exchanged will be the same, namely 105,472 bits, for both cases. But, for a larger database with $n = 32,768$ data items where the size of items is 1 Kbit, the bandwidth performance of the octal tree is better than the binary tree. Trees with higher number of child nodes

n	Database size	binary	quadratic	octal
2	2,048	4,096	-	-
4	4,096	8,192	8,192	-
8	8,192	13,312	-	16,384
16	16,384	19,456	18,432	-
32	32,768	26,624	-	-
64	65,536	34,816	31,744	38,912
128	131,072	44,032	-	-
256	262,144	54,272	48,128	-
512	524,288	65,536	-	68,608

Table IV
ACTUAL COSTS OF OVERALL COMMUNICATION FOR DIFFERENT DATABASE SIZES (IN NUMBER OF BITS) AND $\lceil \log_2(N) \rceil = 1024$.

suffer from high communication rates as discussed below.

We enumerate the total number of exchanged bits for larger number of data items and depicted the results in Figure 2. The figure demonstrates that quadratic and octal trees are superior than the binary in terms of bandwidth requirements for the number of items of practical interest (from 4,096 to 2^{36} data items). We also depict the bandwidth requirements of hexadecimal trees, in which non-sink nodes have 16 children, in Figure 2. As can be observed from the figure, using trees with larger number of child nodes will be inferior to binary tree implementation in terms of bandwidth requirements. Therefore, we focus our investigations on octal trees only.

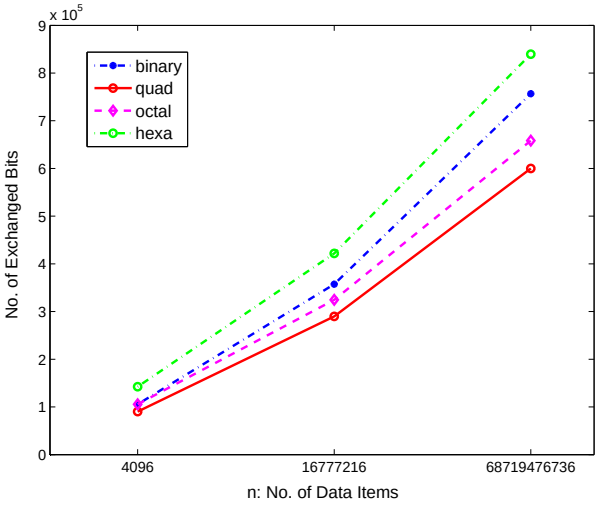

Figure 2. Total number of exchanged bits in different types of trees

V. HYBRID CPIR FOR PARALLEL IMPLEMENTATIONS

In this section, we present a hybrid method that is advantageous for larger databases. As can be understood from the previous discussions, the server-side computation becomes very complicated as a result of the increasing difficulty of performing encryptions in higher levels of the tree as the tree gets deeper. In addition, the parallel processing is of no significant assistance since the computation proceeds sequentially from the bottom to the top of the tree. Therefore, we propose a modified version of CPIR that takes advantage of parallel processing, and allows the scheme to scale to a large number of data items provided that many-core processors are available.

The new scheme, based on partitioning the tree into smaller, equal size, subtrees, is described in Algorithms 3 and 4. In the algorithms, 2^m and 2^l stand for the numbers of data items in the entire tree and in each of the subtrees, respectively. The number of subtrees, $\mu = 2^{m-l}$, is determined depending on the performance requirements.

In Algorithm 3, the client-side operations are described. The user encrypts a separate selection bit (ς_j in Algorithm 3) for each subtree, and the selection bit is 1 for the subtree that contains the requested data item, and 0 otherwise. Note that the number of index bits is now $l < m$. The user sends the homomorphically encrypted index and selection bits to the server, which in turn uses Algorithm 4 to retrieve the encrypted version of the requested data item, $R_{l,0}$. For appropriate values of m and l, the new method can be beneficial as we will show later in this section and Section VI.

Algorithm 3 Client-side computation for binary tree-based Hybrid CPIR

Require: m, l, and $x = x_{l-1} \dots x_1, x_0$
Ensure: $\{c_1, \dots, c_{l-1}\}$ and $\{\varsigma_0, \dots, \varsigma_{2^{m-l}-1}\}$

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 $\{\varsigma_0, \dots, \varsigma_{\mu-1}\}$

The first phase of Algorithm 4, in which server-side operations are described, collapses the subtrees to a single subtree as described in Steps 1–10. As a result, we obtain a single subtree whose depth is l. Then, the resulting subtree, which contains the desired data item, is divided among the processor cores, which process their portions of the subtree in isolation (*cf.* Steps 11–24). Finally, the processor cores join and start computing the upper part of the subtree collaboratively (*cf.* Steps 25–35). An example of the hybrid CPIR scheme is provided in Appendix D.

A. Communication Complexity of the Hybrid CPIR

The new hybrid CPIR incurs an overhead in communication complexity due to the additional selection bits that are sent to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2456021, IEEE Transactions on Parallel and Distributed Systems

Algorithm 4 Server-side computation for binary tree-based Hybrid 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 and $\kappa < l$
Ensure: $R_{m,0}$

▷ Collapsing subtrees into one subtree
$\mu = 2^{m-l}$ ▷ No. of subtrees
$\gamma = 2^{l-\kappa}$
for $j \leftarrow 0$ to $2^\kappa - 1$ **in parallel do**
  for $i \leftarrow 0$ to $\gamma - 1$ **do**
    $R_{0,j\gamma+i} = E(\varsigma_0)^{f_{j\gamma}}$
    for $k \leftarrow 0$ to $\mu - 1$ **do**
      $R_{0,j\gamma+i} \leftarrow R_{0,j\gamma+i} \cdot E(\varsigma_k)^{f_{j\gamma+(\mu-1)2^l}} \bmod N^2$
    end for
  end for
end parallel for

▷ Cores computing in the collapsed subtree in isolation
$\delta \leftarrow 2^{l-\kappa}$ ▷ Number of data items assigned to a core
for $j \leftarrow 0$ to $2^\kappa - 1$ **in parallel do**
  for $i \leftarrow 0$ to $\delta - 1$ **do**
    $\tilde{R}_{0,i} \leftarrow R_{0,j\delta+i}$
  end for
  for $s \leftarrow 1$ to $l - \kappa$ **do**
    for $i \leftarrow 0$ to $2^{l-s} - 1$ **do**
      $f_0 \leftarrow \tilde{R}_{s-1,2i}$
      $f_1 \leftarrow \tilde{R}_{s-1,2i+1}$
      $\tilde{R}_{s,j} \leftarrow E^{(s+1)}(f_0) \times c_{s-1}^{f_1-f_0} \bmod N^{s+2}$
    end for
  end for
  $R_{l-\kappa,j} \leftarrow \tilde{R}_{l-\kappa,0}$
end parallel for

▷ Cores join
for $s \leftarrow l - \kappa + 1$ to l **do**
  for $j \leftarrow 0$ to $2^{l-s} - 1$ **in parallel do**
    $f_0 \leftarrow R_{s-1,2j}$
    $f_1 \leftarrow R_{s-1,2j+1}$
    in parallel do
      $t_0 \leftarrow E^{(s+1)}(f_0)$
      $t_1 \leftarrow c_{s-1}^{f_1-f_0} \bmod N^{s+2}$
    sync
    $R_{s,j} \leftarrow t_1 \cdot t_0 \bmod N^{s+2}$
  end parallel for
end for
return $R_{l,0}$

the server (in addition to the index bits). The formula for the number of bits sent to the server by the user can be given as

$$\text{BW} = (2\mu + (2^x - 1) \cdot (3 + 4 + \ldots + (l + 2)))|N|,$$

where BW stands for the bandwidth, l is the depth of the corresponding subtree, $\mu = 2^{m-x \cdot l}$ is the number of data items in each subtree ($l < m$, where 2^m is the number of data items in the entire tree). Finally, x is 1 and 3 for binary and octal trees, 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 $(l + 2) \cdot |N|$.

The effects of the octal version of the hybrid solution in the bandwidth requirements are illustrated in Figure 3. From the figure, we conclude that the effect can be made negligible if the size of the subtrees 2^l for a given m is selected carefully. We also observe a similar effect on binary and quadratic trees.

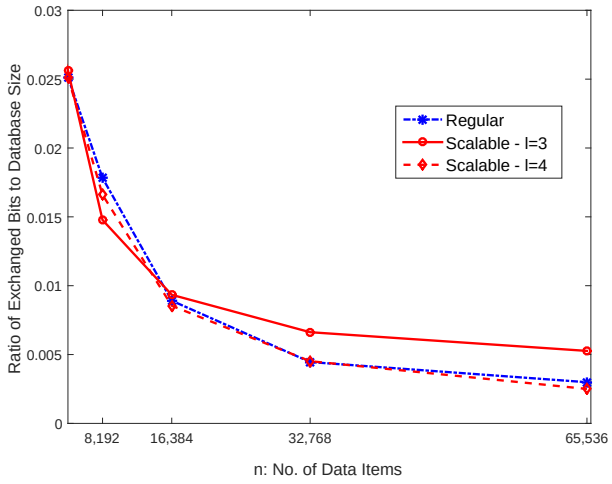

Figure 3. Octal Tree Case: Ratio of exchanged number of bits to database size for $|N| = 1024$, $l = 3, 4$.

Appendix E provides our estimates for the expected execution times of the hybrid scheme based on the time model introduced in Section IV-C. The actual execution times are given in Sections VI-B3 and VI-B4.

VI. Implementation Results

We implemented both the serial and the parallel versions of all CPIR schemes based on binary, quadratic, and octal trees using C++ programming language with the GMP library optimized for big number arithmetic. For parallel implementations we used the OpenMP API that allows shared-memory multiprocessing programming. In the first platform used in the implementations, which is a computer featuring six cores, with hyper-threading support running 64-bit Ubuntu Linux 12.04 operating system, we utilize a maximum of four parallel threads in our implementations for the first set of experiments. Each core is an Intel Xeon CPU E1650 operating at 3.50 GHz. Finally, we used a 1,024-bit modulus, providing 80-bit equivalent security. We also included the timing results on two separate computing platforms that feature 16 and 30 cores, respectively to demonstrate that our claims for the scalability of the proposed schemes are justified.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2456021, IEEE Transactions on Parallel and Distributed Systems

A. Client-Side Computations

The client performs encryption operations for building the secure indexes (i.e., encrypted selection bits) and one decryption operation to retrieve the requested data item. Encryptions are parallelized while the decryption, which is relatively a simple operation, is performed in serial. For the three cases, the results are given in Table V. As can be observed, the CPIR implementations based on quadratic and octal trees offer an obvious advantage over the binary tree implementation as far as the client-side computations are concerned.

No. of Items	Client Encryption (ms)			Client Decryption (ms)		
	binary	quad	oct	binary	quad	oct
2	2	-	-	2	-	-
4	7	2	-	5	2	-
8	19	-	5	11	-	2
16	34	10	-	19	5	-
32	55	-	-	30	-	-
64	78	24	19	41	11	5
128	114	-	-	58	-	-
256	151	45	-	78	18	-
512	200	-	48	102	-	10
1,024	257	99	-	130	28	-
2,048	324	-	-	163	-	-
4,096	416	155	93	200	41	18
32,768	-	-	197	-	-	28

Table V
TIMINGS OF CLIENT'S SELECTION BIT ENCRYPTIONS AND THE DECRYPTION OF THE FINAL RESULT

B. Server-Side Computations

The server-side computations constitute the most time- and resource-consuming part of all CPIR schemes since all data items have to be processed before the requested one is selected out. Therefore, the computation complexity is directly a function of the database size. On the other hand, some of the involved operations are often independent and therefore, can be performed in parallel. In what follows, we present the timing results for both serial (*cf.* Algorithm 1) and parallel (*cf.* Algorithm 2) implementations and demonstrate that the proposed CPIR schemes take advantage of parallel processing.

1) Serial Case: In serial implementation based on Algorithm 1, a single thread is used to implement server side computations of the three CPIR schemes and the results are enumerated in the left hand side of Table VI. The table shows that we can achieve speedups of up to $\frac{16167}{1580} = 10.23$ for a database with 512 data items. As the number of data items increases one should expect similar speedup values. For instance, for a database of 4,096 items, the speedup will be $\frac{135,249}{12,052} = 11.22$.

2) Parallel Case: We implemented Algorithm 2 and tabulated the results in the right hand side of Table VI. Obviously, parallel computation on shared-memory multicore computing platforms benefits all CPIR schemes and the benefit of parallelization is more pronounced when the number of data items is high. For example, with 512 data items, we can achieve a speedup of $1,580/407 = 3.88$ for the octal tree CPIR and $16,167/4,551 = 3.55$ for binary tree case, whereas in 4,096-item database case, the speedup of the octal tree method is $12,052/3,199 = 3.77$, and that of binary version is $135,249/36,039 = 3.75$.

No. of items	Server Computation (ms) Sequential			Server Computation (ms) Parallel		
	binary	quad	oct	binary	quad	oct
2	5	-	-	5	-	-
4	28	8	-	23	4	-
8	102	-	16	61	-	6
16	292	67	-	138	27	-
32	730	-	-	289	-	-
64	1,682	353	182	566	111	58
128	3,683	-	-	1,138	-	-
256	7,786	1,566	-	2,282	407	-
512	16,167	-	1,580	4,551	-	407
1,024	32,053	6,025	-	9,063	1,621	-
2,048	67,141	-	-	18,076	-	-
4,096	135,249	24,432	12,052	36,039	6,459	3,199
32,768	-	-	96,966	-	-	25,409

Table VI
TIMINGS OF SERVER COMPUTATION

With the 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.

From the binary tree serial implementation to the octal tree parallel implementation, the achieved speedup is $\frac{135,249}{3,199} = 42.28$ for a database with 4,096 items. In Table VI, for 4096 items, the decrease in times between binary and quadratic trees is much more important then the decrease between quadratic and octal tree based approaches. This is expected since from binary to quadratic tree the depth is halved (from 12 to 6) while from quadratic to octal tree the decrease in the depth is more modest (from 6 to 4).

If we use eight threads to better utilize all the computational power of the underlying computing platform (which can support up to 12 threads on its hyperthreaded six-core architecture), we can further accelerate the computations. For instance, for a database with 4,096 data items, the execution time decreases from 3,199 ms to 2,955 ms. Therefore, from the binary tree serial implementation to the octal tree parallel implementation with eight threads, the achieved speedup will be $\frac{135,249}{2,955} = 45.16$. This is an important improvement that brings CPIR schemes one step closer to practical usage.

Our preliminary theoretical analysis shows that the proposed schemes show weak scalability in parallel implementations. Namely, using more computational power (i.e., higher number of processor cores) benefits larger databases with more data items. On the other hand, a higher number of cores can also be beneficial for databases with moderately small sizes.

3) Hybrid CPIR with Octal Trees: We obtained the actual timing results of the hybrid CPIR introduced in Section V. For a subtree size of 512 (i.e., $l = 3$), server-side computations take 2,850 and 17,300 ms for databases with 4,096 and 32,768 items, respectively. When the size of subtree is 4,096 (i.e., $l = 4$), server-side computation take 22,600 ms for a database with 32,768 data items.

From these timing results, we can observe that smaller subtrees produce the best results for big databases. However, there is a trade off between the computational and communi-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2456021, IEEE Transactions on Parallel and Distributed Systems

cation costs, whereby smaller subtrees increase the bandwidth requirements as shown in Section V. In short, for large octal trees, when the hybrid CPIR is used, we can achieve a speedup up to $\frac{25,409}{17,330} = 1.47$ over the parallel implementation with a relatively slight increase in the bandwidth requirement.

4) Implementation Results on 16 and 30-Cores Shared-Memory Computers with Octal Trees: In order to demonstrate that the proposed parallel algorithms scale well with the increasing number of data items, we ported our implementations to two computer platforms with 16 and 30-cores, respectively.

The first 16-core computing platform features two Intel Sandybridge-EP CPUs clocked at 2.0 GHz and 256 GB of RAM memory split across two NUMA domains. NUMA stands for non-uniform memory access, and it is a memory design for multi-processor computing platforms where the access time depends on the memory location relative to the processor, from which the access is originated. Each CPU has eight-cores and HyperThreading is enabled. Each core has its own 32 KB L1 and 256 KB L2 caches. The 8 cores on a CPU share a 20 MB L3 cache. The machine runs 64-bit Debian with Linux kernel.We run the parallel and the hybrid versions of the octal tree-based implementations and enumerated the execution times in Tables VII and VIII, respectively.

No. of Items	**No. of Cores**				
	1	2	4	8	16
8	20	11	7	5	8
64	245	128	71	44	55
512	2,165	1,100	570	330	262
4,096	17,740	8,914	4,508	2,545	1,492
32,768	142,564	71,660	35,985	20,150	10,774

Table VII
EXECUTION TIMES OF THE PARALLEL IMPLEMENTATION IN MILLISECONDS FOR VARIOUS NUMBER OF DATA ITEMS AND NUMBER OF CORES ON A 16-CORE COMPUTER PLATFORM.

Table VII shows that the execution times decrease as the number of cores utilized on 16-core machine increases. The speedup values are very close to the number of cores employed in the calculation of CPIR when the number of items is sufficiently high. For instance, we are able to obtain speedup values of 7.07 and 13.23 for 8 and 16-core implementations, respectively, when the number of items is 32,768. Although the computing platform is a multiprocessor systems, in which inter-core communication can be more expensive than single-chip multi-core platforms, we are still able to obtain speedup values that commensurate with the number of cores employed. This is due to the fact that the proposed algorithms are designed to decrease to the number of synchronization points as explained in Section IV-C.

In Table VIII, the time performance of the hybrid implementation for various subtree sizes is listed. As can be observed, with a small increase in the bandwidth usage we can obtain significant improvements over server-side execution times.

The second computing platform features 30 cores running 64 bit CentOS 6.5, where each core is an Intel Xeon CPU E7-4870 v2 clocked at 2.30 GHz. Each CPU has 15 cores and HyperThreading is enabled. Each core has its own 32 KB L1 and 256 KB L2 caches. The 15 cores on a CPU share a 30 MB L3 cache. A total of 64 GB of RAM memory split across two NUMA domains. We also increased the number of items in the database to 2^{18} and tabulated the timing results in Table IX. The timing results show that we obtain significant speedup values for 30 cores over 16 cores.

No. of Items	**Subtree Size**	**No. of Cores**				
		1	2	4	8	16
4096	64	9.63	4.87	2.46	1.38	0.84
	512	12.52	6.55	3.21	1.81	1.16
32768	64	74.31	37.32	18.68	10.45	5.57
	512	77.23	38.79	19.44	10.89	5.88
	4,096	100.52	50.58	25.37	14.22	7.74

Table VIII
EXECUTION TIMES OF THE HYBRID METHOD IN SECONDS FOR VARIOUS NUMBER OF DATA ITEMS, SUBTREE SIZES, AND NUMBER OF CORES ON A 16-CORE COMPUTER PLATFORM.

No. of Items	**Subtree Size**	**No. of Cores**					
		1	2	4	8	16	30
4,096	64	9.14	4.63	2.31	1.17	0.67	0.58
	512	11.88	6.02	3.03	1.54	0.88	0.76
32,768	64	70.44	35.46	17.68	8.86	4.51	3.20
	512	73.26	36.82	18.45	9.23	4.72	3.36
	4096	95.44	48.03	24.02	12.04	6.17	4.39
262,144	64	562.96	282.83	140.29	70.33	35.14	24.81
	512	565.61	282.16	141.06	70.93	35.61	23.02
	4,096	588.16	293.12	147.32	73.41	36.81	25.37
	32,768	763.08	383.80	204.01	95.98	48.04	33.22

Table IX
EXECUTION TIMES OF THE HYBRID METHOD IN SECONDS FOR VARIOUS NUMBER OF DATA ITEMS, SUBTREE SIZES, AND NUMBER OF CORES ON A 30-CORE COMPUTER PLATFORM.

VII. LITERATURE ON PIR SCHEMES AND COMPARISON

There is a relatively high academic interest in efficient PIR schemes [1–4, 6–8, 11, 12, 18, 19, 24, 27]. 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 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. [4] 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 [4], each product term is logical AND of two Boolean variables. The scheme can use additive

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2456021, IEEE Transactions on Parallel and Distributed Systems

homomorphic encryption scheme proposed by Paillier [25]. In the scheme, the users sends $O(n^{1/3})$ ciphertexts as the query and receives $O(n^{1/3})$ 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].

The scheme in [11] uses a somewhat fully homomorphic encryption scheme (SWFHE), a topic of high interest in the cryptographic community in recent years [5, 13, 14, 21]. 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 the comparison circuit is very 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 [28]. One nice property of the PIR scheme in [11], 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 possibly different users.

In all three schemes [4, 11, 19], query sizes are reasonably low (see Table X). Especially, the SWFHE scheme in the bundled case [11] offers extremely small-sized queries. However, the bandwidth complexity has two components, namely query and response sizes and all three schemes suffer from very high response sizes per one bit of the requested data item as shown in Table X. In the proposed scheme, the response is several orders of magnitude smaller in size than in any other scheme in Table X. As can be observed from the table, for even small-sized data items (see the columns 6-9 of Table X), the response sizes dominate the bandwidth complexity and query sizes become negligible in comparison.

Since the schemes in [4, 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 [1, 2, 11], both of which utilize lattice-based cryptography. The former lattice-based scheme introduced in [1, 2], claims computational efficiency while the latter [11], which utilizes SWFHE, claims superior bandwidth performance over the former while accepting that 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 [11], but worse than that in [1, 2]. However, we also show that the scheme in [1, 2] can have such a poor bandwidth performance that it is sometimes better to download the entire database in many circumstances.

The CPIR schemes that rely on decisional trees use the Damgård-Jurik cryptosystem that is based on the decisional composite residuosity assumption [25], which is a relatively well studied classical problem in comparison with those security arguments used in lattice-based solutions, especially the one in [1, 2].

We compare the bandwidth requirements of the proposed method with octal tree and the two other techniques in Figures 4, 5, 6. In Figure 4, assuming that each data item in our data base is 1,024-bit in size we change the number of data items from 512 to 65,536 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 4, the proposed scheme always offers the best bandwidth performance.

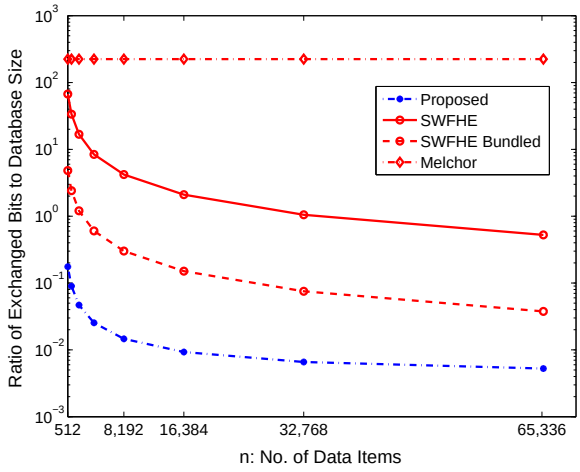

Figure 4. Bandwidth comparison of three schemes when the data item size is 1,024-bit; Melchor's scheme in [1, 2], SWFHE and SWFHE - bundled in [11]

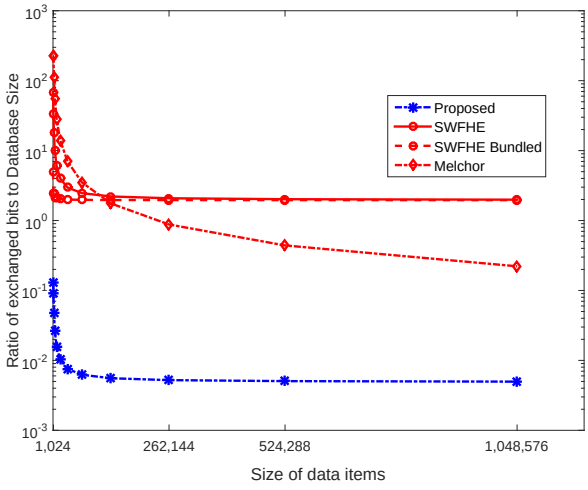

Figure 5. Bandwidth comparison of three schemes with variable data item size and $n = 1,024$; Melchor's scheme in [1, 2], SWFHE and SWFHE - bundled in [11]

Figure 5 illustrates the bandwidth performances of the three schemes, when the number of data items is fixed to $n = 1,024$ and the data item sizes are changed from 1,024-bit to 1 million bit. Figure 6 is similar except that we now use a database with higher number of items, namely $n = 65,536$. As pointed out earlier in our discussions regarding the bandwidth performance values in Table X, the response size dominates when the size of each data item increases. This is apparent in Figures 5 and 6 as the bandwidth performance of the bundled case of the SWFHE scheme is almost the same as the regular SWFHE scheme.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2456021, IEEE Transactions on Parallel and Distributed Systems

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)
KO [19]	32	8	2^{15}	2^{23}	2^{18}	2^{16}	2^{24}	2^{22}
BGN [4]	20	0.8	645	13004	5161	102	330, 280	6502
SWFHE [11]	4,000	32	250	784	2,000	6.125	128,128	392
SWFHE - Bundled [11]	6.35	0.0313	250	784	2,000	6.125	128,128	392
Proposed	42.5	32	0.625	0.875	5	≈ 0.068	320	0.4375

Table X

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 DATABASES WITH 2^{16} AND 2^{32} ITEMS, RESPECTIVELY.

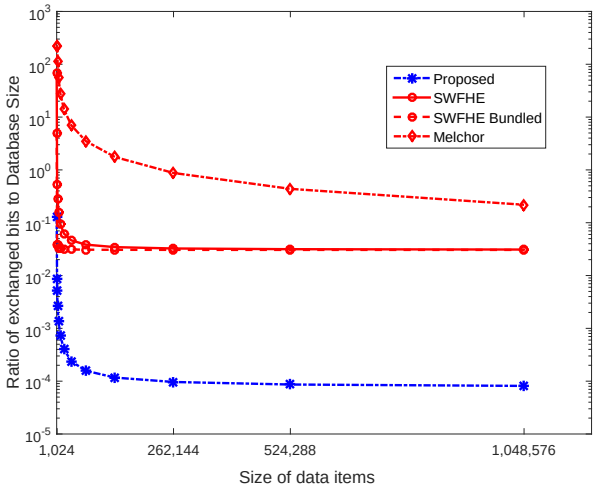

Figure 6. Bandwidth comparison of three schemes with variable data item size and $n = 2^{16}$; Melchor's scheme in [1, 2], SWFHE and SWFHE - bundled in [11]

Data item size (# of bits)	database size (# of bits)	[1, 2]	[11]	Proposed method
1 K	512 K	224	4.91	0.131
16 K	8 M	14.01	4.07	0.016
128 K	64 M	1.76	3.93	0.009
256 K	128 M	0.88	3.92	0.008
2 M	1 G	0.12	3.91	0.008

Table XI

RATIO OF EXCHANGED INFORMATION TO DATABASE IN DIFFERENT PIR SCHEMES $n = 512$ AND $|N| = 1,024$

In Figures 4, 5, 6, the lattice-based scheme in [1, 2] demonstrates a poor bandwidth performance. To give a better insight we tabulate the ratios of exchanged information to the database size in each scheme in Table XI when $n = 512$ and $|N| = 1,024$ bit. As can be observed in the table, the proposed method always results in superior bandwidth performance. The lattice-based scheme in [1, 2] 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 1,024-bit, the ratio of exchanged data to database size in the SWFHE-based PIR scheme is 0.0384, while it is only 0.0053 in the proposed scheme for the same setting.

For server-side computations, the lattice based scheme [1, 2] 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 1.23 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. For instance, we observed that our implementations on the 30-core computing platform can achieve as high as about 10 Mbit/s throughput for database of 32,768 items with subtree size of 64.

The SWFHE-based PIR scheme [11] reports two time performance metrics: i) throughput 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 1,024-bit long each, the time spent for processing a data item is given as 0.89 ms while it is 0.79 ms for 512 item database in our scheme. On the other hand, for the latency metric indicating the waiting time for a user, the time spent for processing a data item of 1024-bit is estimated to be around 16.93 ms [11]. For instance, for a database of 512 items each of which is 1,024-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 about 0.4 s.

Furthermore, if we use the scalable CPIR with eight threads on the six-core architecture and tolerate a slight increase in bandwidth requirements, we can achieve a better performance. For instance, we can achieve a throughput of 1.40 Mbit/s for a database of 4,096 items when octal subtrees with $l = 3$ 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 [11] in terms of throughput as well. In a case in our implementation on the 30-core computer with 262,144 items and subtree size of 64, the time spent for processing a data item can be reduced to as low as $0.095\ ms$.

VIII. CONCLUSION

We improved the CPIR protocol introduced by Lipmaa in several ways. First, we proposed to use octal trees, instead of binary trees in the original scheme, which results in an order of magnitude faster implementation than the original scheme for a database of 512 data items. We showed that the speedup is likely to increase for higher number of data items. We then introduced a highly-optimized parallel algorithm for shared-memory multi-processor computing platforms. The speedup

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2015.2456021, IEEE Transactions on Parallel and Distributed Systems

achieved via the proposed parallel algorithm is shown to be proportional to the number of cores. We also proposed a hybrid method to accelerate the CPIR protocol at the expense of a relatively small increase in the bandwidth. Our implementations on 16-core and 30-core computing platforms for large databases show that the execution time of the new scheme is more than two orders of magnitude faster than the original scheme with a straightforward serial implementation.

We compared the proposed scheme with the schemes in the literature in terms of bandwidth requirement, measured in terms of the total number of bits exchanged between the client and the server. Due to log-squared communication complexity of the original scheme, the bandwidth efficiencies of the proposed schemes are better 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 schemes.

IX. Acknowledgments

This work is supported by Turk Telekom under Grant Number 3014-07. We thank Gamze Tillem and Kamer Kaya for their support for the implementation results.

References

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

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

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

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

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

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

[7] 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, Oct 1995.

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

[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] Devet, C. and Goldberg, I., "The Best of Both Worlds: Combining Information-Theoretic and Computational PIR for Communication Efficiency", *Privacy Enhancing Technologies*, pp. 63–82, 2014.

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

[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., "Fully homomorphic encryption using ideal lattices", In *Symposium on the Theory of Computing (STOC)*, pp. 169–178, 2009.

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

[15] Goldberg, I., "Improving the Robustness of Private Information Retrieval", In *2007 IEEE Symposium on Security and Privacy*, (2007) 131148.

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

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

[21] 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.

[22] Lueks, W., Goldberg, I., "Sublinear Scaling for Multi-Client Private Information Retrieval", In *19th International Conference on Financial Cryptography and Data Security*, January 2015.

[23] 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.

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

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

[26] Rabin, M. O., "How to exchange secrets by oblivious transfer", *Technical Report TR-81*, Aiken Computation Laboratory, Harvard University, 1981. available at http://eprint.iacr.org/2005/187.

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

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

[29] Ünal, E. and Savaş, E. "Bandwidth-Optimized Parallel Private Information Retrieval", Proceedings of the 6th International Conference on Security of Information and Networks, SIN '14, to appear, 9-11 September 2014, Glasgow, UK.

[30] Wiesner, S., "Conjugate coding", *Sigact News*, vol. 15, no. 1, pp. 78–88, 1983.

Ecem Ünal received the BS (2012) degree in computer science from the Computer Science Department at Bilkent University. She completed the MS degree in the Department of Computer Science and Engineering at Sabancı University in 2014. Her research interests include applied cryptography, data and communication security, and distributed systems.

Erkay Savaş received the BS (1990) and MS (1994) degrees in electrical engineering from the Electronics and Communications Engineering Department at İstanbul Technical University. He completed the PhD degree in the Department of Electrical and Computer Engineering (ECE) at Oregon State University in June 2000. He has been a faculty member at Sabancı University since 2002. His research interests include applied cryptography, data and communication security, security and privacy in data mining applications, embedded systems security, and distributed systems. He is a member of IEEE, ACM, the IEEE Computer Society, and the International Association of Cryptologic Research (IACR).