Fully Homomorphic Encryption Based On the Parallel Computing

Delin Tan[1,2], Huajun Wang[1]

[1] College of Geophysics, Chengdu University of Technology, No.1, Dong san Road, Er xian qiao, Chenghua District, Chengdu, China

[e-mail:hjwang@sdut.edu.cn]

[2] Sichuan Normal University, No.5, Jing'an Road, Jinjiang District, Chengdu, China

[e-mail:tdltcl@126.com]

*corresponding author: Huajun Wang

Received April 29, 2017; revised August 10, 2017; accepted September 9, 2017; published January 31, 2018

Abstract

Fully homomorphic encryption(FHE) scheme may be the best method to solve the privacy leakage problem in the untrusted servers because of its ciphertext calculability. However, the existing FHE schemes are still not being put into the practical applications due to their low efficiency. Therefore, it is imperative to find a more efficient FHE scheme or to optimize the existing FHE schemes so that they can be put into the practical applications. In this paper, we optimize GSW scheme by using the parallel computing, and finally we get a high-performance FHE scheme, namely PGSW scheme. Experimental results show that the time overhead of the homomorphic operations in new FHE scheme will be reduced manyfold with the increasing of processing units number. Therefore, our scheme can greatly reduce the running time of homomorphic operations and improve the performance of FHE scheme through sacrificing hardware resources. It can be seen that our FHE scheme can catalyze the development of FHE.

Keywords: FHE, privacy leakage, untrusted server, ciphertext calculability, parallel computing

This research was supported by the project of education department of SiChuan province with No.17ZB0352 and by the the Natural Science Foundation of China with No. 61373162.

http://doi.org/10.3837/tiis.2018.01.024 **ISSN : 1976-7277**

1. Introduction

The privacy leakage problem on the untrusted servers has become a serious hazard problem in information security field. For example, the security of the cloud computing platform depends on the situation of solving the privacy leakage problem [1], and meanwhile its solution is also impacts the popularity in the industry. To solve the privacy leakage problem, people mainly use traditional cryptographic methods to encrypt user's privacy data stored on the cloud computing platform, such as the proxy re-encryption algorithm, the property encryption algorithm and so on. Unfortunately, the traditional cryptographic methods don't support the ciphertext calculation, so they need to decrypt the ciphertext firstly when they want to handle user's privacy data stored on the cloud computing platform and encrypted with the traditional cryptographic methods. This not only increases the complexity of operating user's privacy data, but also easily causes the privacy leakage problem during decrypting the ciphertext of user's privacy data. Therefore, it is necessary to find a cryptographic scheme which supports ciphertext calculation.

As is known to all, fully homomorphic encryption(FHE) scheme supports ciphertext calculation, so it satisfies the security requirement of the untrusted servers such as the cloud computing platform. The security of the untrusted servers will be greatly improved if they get the security support of FHE scheme.

At present, since Gentry constructed the first FHE scheme in 2009, more and more FHE schemes have been proposed and improved [2][3][4][5]. However, the existing FHE schemes can't be put into the practical applications because of their low efficiency [6]. For example, it will spend more than 30 minutes when the homomorphic operations of BGV scheme is run, although BGV scheme is the best efficient scheme during the existing FHE schemes. So it is easy to see that the running time can't satisfy the demand of the practical applications.

1.1 Introduction to the mian FHE schemes

At present, the best performance of FHE scheme is BGV(Fully Homomorphic Encryption without Bootstrpping), whose time complexity is $\tilde{O}(\lambda^{3})$ [7]. Besides BGV, there are several FHE schemes as follows.

(1) Fully homomorphic encryption over the integers, namely DGHV, with the time complexity $O(\lambda^{14})$ [8].

(2) Fully homomorphic encryption without modulus switching from classical gapsvp, namely Bra12, with the time complexity $\tilde{O}(\lambda^6)$ [9].

(3)Homomorphic encryption of approximate eigenvector: conceptually-simpler, asymptotically-faster, attributed-based, namely GSW, with the time complexity of per gate $O(N^{\omega})$ [10], where $\omega < 2.3727$.

(4) FHEW: Bootstrapping Homomorphic Encryption in less than a second [11], namely FHEW. It mainly optimizes bootstrapping technique with PPT algorithm, so its performance is mainly embodied in the running time of bootstrapping technique, and you can know by title.

Moreover, there are many FHE schemes [12-17] which all are optimized and improved on the above schemes. Meanwhile, they have been proved that they still can't be put into the practical applications due to their low efficiency.

1.2 Our motivation

The essence of low efficiency for the existing FHE schemes is due to the noise, which strengthens the security of FHE schemes, but also leds many complex and expensive operations into FHE schemes, such as relinearization technique, modulus switching technique, dimension reduction technique and so on. If we remove the noise, FHE schemes will be unsafety. It is obvious that insecure FHE schemes are useless, so the noise can't be removed. However, if FHE schemes can't be put into the practical applications, they are still useless. So it is an urgent problem to put FHE schemes into the practical applications. According to scientists' predictions, the study of FHE schemes will be a long process.

In order to solve the dilemma that FHE schemes can't be put into the practical applications, we optimize FHE schemes by the parallel computing. Our purpose is to reduce the time complexity of homomorphic operations in FHE scheme by the parallel computing. The main principle is to improve the performance of homomorphic operations by sacrificing hardware resources. The process is shown below: Firstly, we will decompose the homomorphic operations into a number of independent sub-operations in accordance with the principle of the parallel computing. Secondly, we assemble a number of processing units into a high-performance computing system which is a 2-dimensional grid. Thirdly, assign one or a group of processing units to these independent sub-operations. Namely, these sub-operations will be processed on the provided processing units respectively, so these sub-operations are running at the same time.

Through the above steps, the homomorphic operations will be optimized by the parallel computing, the running time of the homomorphic operations will be reduced, and the efficiency of the FHE scheme will be improved.

1.3 Roadmap

Next, we will introduce the relevant knowledge of the parallel computing in section 2, including the fundamental of Cannon algorithm. In section 3, we will introduce our proposed schemes, namely the parallelized GSW(PGSW) scheme. In section 4, we will analyze the performance of PGSW scheme. In section 5, we will introduce the corresponding experiments. Finally, we will analyze and summarize the whole paper in section 6.

2. Preliminaries

Below we usually denote the ordinary parameters by lowercase Greek letters, such as security parameter λ, dimension parameter m, n and κ, modulus parameter q and so on. In addition, we use $\vec{v}$ to represent the vector, use uppercase letters M to represent the matrix, and use Z_n to present the integer ring. Next, we introduce the basic principles of the parallel computing and Cannon algorithm, as shown below.

2.1 The parallel computing [18]

The parallel computing which is evolved from the serial computing is an important research direction in computer science. In simple terms, the parallel computing is a supercomputing implemented on the high-performance computing system. There are many high-performance computing systems available, such as the parallel computer cluster, the distributed computer system and so on. Their basic principles all are to use multiple processors to cooperatively solve the sameone problem. The workflow is as follow: Firstly, decompose the problem which is need to process into many independent parts. Secondly, assign an independent processing unit of the high-performance computing system for each individual part. Thirdly, run all the independent processing units of the high performance computing system at the same time. Then, each independent processing unit of the high performance computing system will produce a operation result respectively. According to the principle of the parallel computing, we merge all the operation results into a final result which exactly we need. Since all the

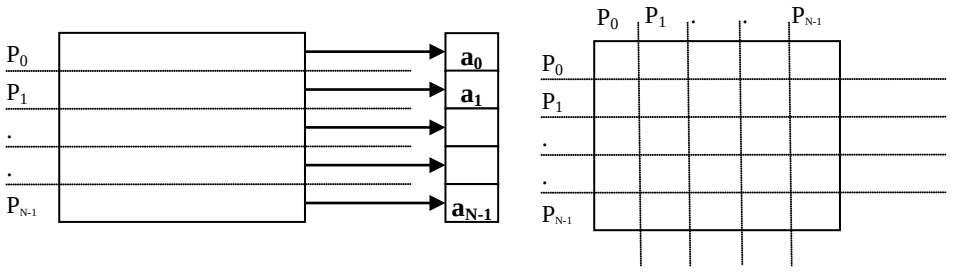
independent processing units of the high performance computing system run simultaneously, so the processing time of the whole problem based on the parallel computing is shortened, and correspondingly improve the performance of solving the problem.

According to GSW, due to the presence of noise, it needs to introduce a number of operations to achieve the FHE scheme, such as $BitDecomp()$, $BitDecomp^{-1}()$, $Powersof\,2()$, matrix addition, matrix multiplication and so on. Practice has proved that these operations are complex and time-consuming, especially matrix multiplication. At present, the optimal time complexity of matrix multiplication is $O(N^{\omega})$ ($\omega = 2.3727$) achieved by Williams [19] under the serial computing. After analyzing we can see that it is difficult to furtherly optimize the matrix multiplication under the serial computing, so we consider to optimize matrix multiplication under the parallel computing. Now, there are many parallel computing algorithms to optimize matrix multiplication, such as DNS algorithm [20], Cannon algorithm [21], Fox algorithm [22], Systolic algorithm [23] and so on. In this paper, we choose Cannon algorithm as the parallel computing algorithm for matrix multiplication, and we will introduce Cannon algorithm in the next section.

2.2 Cannon Algorithm

Cannon algorithm [24] was proposed by Alpatov etc. in 1997. Its main idea is to divide the matrix into many blocks, and then assign a processing unit of the high-performance computing system for each block respectively. Next, we will introduce the specific implementation process of matrix multiplication based on Cannon algorithm. For example, there are two matrices, namely matrix A and matrix B, and we use $A \cdot B$ to represent the product of matrix A and matrix B. Firstly, divide matrix A and matrix B into $p \times p$ blocks which form a 2-dimensional grid whose size is $p \times p$, and the number of each block is decided by its row numbers and column numbers in the 2-dimensional grid. For example, there is a block A_{ij} $(0 \le i, j < p)$, where i and j represents the i-th row and the j-th column in the 2-dimensional grid. Meanwhile, there is a computer cluster which forms a 2-dimensional grid whose size is $p \times p$, and the number of each computer is decided by the number of its row numbers and column numbers in the computer cluster. For example, There is a computer, namely

$D_{i,j}$ $(0 \le i, j < p)$, where i and j represents the i-th row and the j-th column in the computer cluster.



(a) striped partitioning method (b) checkerboard partitioning method

Fig. 1. two kinds of partitioning methods for matrix

According to Cannon algorithm, there are two kinds of partitioning methods for matrix. Respectively, striped partitioning method and checkerboard partitioning method. As shown in **Fig. 1**, they are the sample graph of two kinds of partitioning methods for matrix.

According to different demands, we take the different partition method. Here we mainly use the checkerboard partitioning method, and the algorithm implementation process is shown below.

(1) Assign blocks A_{ij} and B_{ij} to D_{ij} $(0 \le i, j < p)$ which is a processing unit of the computer cluster, and then compute the product $C_{i,j}$ of A_{ij} and B_{ij} on D_{ij}.

(2) Move the block A_{ij} $(0 \le i, j < p)$ loop to the left by i steps, and move the block B_{ij} $(0 \le i, j < p)$ cyclic up by j steps.

(3) Execute the multiplication-addition computation on D_{ij}, and finally add all the results to $C_{i,j}$. Move the block $A_{ij}(0 \le i, j < p)$ loop to the left by 1 step, meanwhile move the block $B_{ij}(0 \le i, j < p)$ cyclic up by 1 step.

(4) Repeat (3), execute the multiplication-addition on D_{ij} for p times altogether, and execute single step cycle on the block A_{ij} and the block B_{ij} for p times respectively.

From the above we can see, the parallel computing time of Cannon algorithm is $T_p = \frac{n^3}{p^2} + 4pt_s + 4t_w\frac{n^2}{p}$ in the 2-dimensional grid, where p is the dimension of 2-dimensional grid, t_s is the startup time, and t_w is the text-transmission time. Because t_s and t_w all are "small" numbers, so the parallel computing time of Cannon algorithm is mainly determined by n and p. It should be noted that the minimum dimension of the sub-matrix is 2, namely, $p = \frac{1}{2}n$, then the time complexity of Cannon algorithm is $O(n)$. It can be seen that the time complexity of matrix multiplication can be linear complexity when selecting the appropriate parameters in the high-performance computing system.

3. The proposed scheme(PGSW)

In this section, we will introduce the proposed scheme, parallelized GSW(PGSW), which is based on the GSW. The reason why we propose a new FHE scheme based on GSW is mainly because GSW scheme has many advantages that other FHE schemes don't have. Below we introduce the unique advantages of GSW scheme, as shown below.

(1) Solved the ciphertext dimension expansion problem caused by the homomorphic operations.

(2) Removed many complex and expensive operations, such as relinearization, modulus switching, dimension reduction and so on.

(3) The current FHE schemes are achieved basically on the ring. So its homomorphic operations mainly include addtion and multiplication. At present, only the homomorphic addition and multiplication of GSW scheme is in accordance with the operational rule of common arithmetic operations, and the other homomorphic operations need to redefine the operation symbols, so GSW scheme is the most natural FHE scheme.

So here we choose GSW scheme as our basis, and optimize it by the parallel

computing. The main idea of the optimization is to optimize the complex operations in GSW scheme. After optimizing, we get PGSW scheme finally. Next we introduce the optimization process of the basic operations, encryption module, decryption module etc. of GSW scheme.

3.1 Some Basic Operations and Their Optimization

There are many basic operations in GSW scheme, and some of these operations are not homomorphic operations, but they are very important. For example, $BitDecomp()$, $BitDecomp^{-1}()$, $Powersof2()$, $Flatten()$ and so on. Their main function is to ensure GSW scheme B-strongly-bounded, so that GSW scheme can evaluate a circuit of polynomial depth rather than merely polynomial degree. However, most of them are complex operations which have high time complexity. That is, these operations are extremely time-consuming operations under the serial computing so that GSW scheme is low efficient. In order to improve the efficiency of GSW scheme, we need to optimize these operations. As following, we introduce the optimization process of the above operations based on the parallel computing.

$PBitDecomp():BitDecomp()$ is an important part of $Flatten()$ which can keep the ciphertexts B-strongly-bounded. The primary mission of $BitDecomp()$ is to convert decimal matrices or vectors into binary matrices or vectors. When the operand of $BitDecomp()$ is vector $\vec{a}=(a_1,a_2,...a_k)$, then according to GSW scheme, we get:

$$BitDecomp(\vec{a})=(a_{1,0},...,a_{1,l-1},...,a_{k,0},...,a_{k,l-1}) \tag{1}$$

where $l*k=N$, and $a_{i,j}$ is the j-th bit in the binary representation of a_i. According to the definition of $BitDecomp()$, we can divide the above operation into two steps.

Step 1, convert all the decimal elements of vectors into the corresponding binary bits. Since all matrices and vectors are B- strongly-bounded, so the decimal elements are “small”, then we can build a conversion table between the decimal elements and the binary bits. When we need to convert the decimal elements into the binary bits, we can quickly query the conversion table and then get the corresponding results. Because the value range of decimal elements in matrices or vectors is “small”, the size of the conversion table is much smaller than the dimension N of matrix or vector, so the time complexity of quering operation for each decimal element through conversion table is

$O(1)$. Since there are κ decimal elments, so the time complexity of step 1 is $O(\kappa)$.

Step 2, assemble all the l-length binary bits into a N-dimensional binary vector. It is obvious that the time complexity of step 2 is $O(\kappa)$.

So the time complexity of *BitDecomp*() is $O(\kappa) * O(\kappa)) = O(\kappa^2)$ when the operand is vector under the serial computing. It is obvious that the time complexity of *BitDecomp*() is $O(N\kappa^2)$ when the operand is matrix under the serial computing. Although their time complexity are not high, we use *BitDecomp*() with other operations together, then the composite operations have high time complexity which hinders GSW from being put into the practical applications, so they need to be optimized by the parallel computing, and the following is the optimization process.

In step 1, the conversion of κ decimal elements are mutually independent, so we can use the parallel computing to optimize them. The main idea is to use κ processing units of the high performance computing system to do the conversion of κ decimal elements respectively, then the conversion of κ decimal elements will work at the same time, so the time complexity of the conversion of κ decimal elements is $O(1)$, and the time complexity of step 1 is also $O(1)$.

In step 2, we use κ processing units of the high performance computing system to do the assembling of κl-length binary bits, namely, each processing unit does one assembling respectively, and then the κ processing units work at the same time, so the time complexity of step 2 is also $O(1)$. Thus we can see that the time complexity of *BitDecomp*() is $O(1)$ when the operand is vector under the parallel computing.

When the operand is matrix, we can compute each row of the matrix respectively, so we also can optimize it with the parallel computing. Namely, compute each row on one processing unit, and then all the processing units work at the same time. Then the time complexity of computing N rows of matrix is $O(1)$ under the parallel computing, so the time complexity of *BitDecomp*() is also $O(1)$ when its operand is matrix under the parallel computing.

Here, we call the optimization result of *BitDecomp*() as *PBitDecomp*(), whose time complexity is $O(1)$ under the parallel computing no matter the operand is matrix or vector. The following are their computation results when the operand are vector and matrix respectively.

$$PBitDecomp(\vec{a}) = (a_{1,0},...,a_{1,l-1},...,a_{k,0},...,a_{k,l-1}) \tag{2}$$

$$PBitDecomp(A) = \begin{bmatrix} a_{11,0}\ a_{11,1}...a_{11,l-1}.....a_{1k,0}\ a_{1k,1}...a_{1\kappa,l-1} \\ a_{21,0}\ a_{21,1}...a_{21,l-1}.....a_{2k,0}\ a_{2k,1}...a_{2k,l-1} \\ ..... \\ a_{k1,0}\ a_{k1,1}...a_{k1,l-1}.....a_{kk,0}\ a_{kk,1}...a_{kk,l-1} \end{bmatrix} \tag{3}$$

$PBitDecomp^{-1}()$: $BitDecomp^{-1}()$ is the inverse process of $BitDecomp()$, and its main function is to convert the binary matrices or vectors into the decimal matrices or vectors. When the operand is vector, for example, $\vec{b} = (b_{1,0},...,b_{1,l-1},...,b_{k,0},...,b_{k,l-1})$, then have:

$$BitDecomp^{-1}(\vec{b}) = (\sum_{j=0}^{l-1} 2^j.b_{1,j},...,\sum 2^j.b_{k,j}) \tag{4}$$

Similarly to $BitDecomp()$, $BitDecomp^{-1}()$ can also be divided into two steps.

Step 1, convert l - length binary bits into the corresponding decimal element from left to right respectively. From the above analysis we can see, we can quickly query the conversion table for converting the l - length binary bits into a decimal elments. According to the analysis of the previous section, we can see that the time complexity of quering operation for the l - length binary bits is $O(1)$, correspondingly the time complexity of step 1 is $O(\kappa)$.

Step 2, assemble all κ decimal elements into a κ -dimensional dicimal vector. Obviously, the time complexity of the above operation is $O(\kappa)$.

So the time complexity of $BitDecomp^{-1}()$ is $O(\kappa) * O(\kappa) = O(\kappa^2)$ when the operand is vector under the serial computing.

When the operand is a matrix, it needs to execute the $BitDecomp^{-1}()$ on each row of the matrix separately. According to the analysis of the previous section, we can see that the time complexity of $BitDecomp^{-1}()$ is $O(N\kappa^2)$ when the operand is a matrix under the serial computing.

The same reason as $BitDecomp()$, $BitDecomp^{-1}()$ also needs to be optimized by the parallel computing, and the optimization process of $BitDecomp^{-1}()$ is similar to $BitDecomp()$, here we will no longer introduce it. Finally, the optimization result of $BitDecomp^{-1}()$ is called as $PBitDecomp^{-1}()$, whose time complexity is $O(1)$ under the parallel computing whether the operand is a matrix or vector.

$PPowersof\,2()$: $Powersof\,2()$ can ensure that the secret key $\vec{v}$ has at least one "big" coefficient, so it is an important operation in GSW scheme, and its operand is mainly vector. For example, there is a κ-dimensional vector $\vec{b} = (b_1, b_2, ...b_\kappa)$, then we have:

$$Powersof\,2(\vec{b}) = (b_1, 2b_1, ..., 2^{l-1}b_1, ..., b_k, 2b_k, ..., 2^{l-1}b_k) \tag{5}$$

So the main function of $Powersof\,2()$ is to extend the k-dimensional vector into the N-dimensional vector. According to GSW, this operation can also be divided into two steps.

(1) Convert each element of the operand vector into a l-dimensional vector whose coefficients is 2^0 to 2^{l-1} separately, and the time complexity of this conversion for each element is $O(l)$, so the time complexity of this conversion for κ elements is $O(\kappa * l) = O(N)$ under the serial computing.

(2) Assemble all the l-dimensional vectors into a N-dimensional vector according to the order of the elments in the operand vector. Obviously the time complexity of this operation is $O(\kappa)$.

From the above we can see, the time complexity of $Powersof\,2()$ is $O(N * \kappa)$ under the serial computing.

According to the principle of the parallel computing, we can optimize (1) by the parallel computing, and (2) don't need to be optimized. Finally we get a optimization result which we call it as $PPowersof\,2()$. In (1), when convert all elements of the operand vector into l-dimensional vector whose coefficients is from 2^0 to 2^{l-1}. Because we can make the convertion of κ elements of the oprand vector happened at the same time on κ processing units of the high-performance computing system, then its time complexity is $O(l)$. Thus we can see that the time complexity of (1) is $O(l)$ under the parallel computing.

When combine all the l-dimensional vectors into a N-dimensional vector according to the order of the elments in the operand vector, the time complexity of (2) is $\mathrm{O}(k)$. So the time complexity of $PPowersof2(\vec{b})$ is $\mathrm{O}(\kappa \cdot l) = O(N)$ when the operand is vector under the parallel computing.

When the operand of $PPowersof2()$ is matrix M, it needs to execute the $PPowersof2(\vec{b})$ on each row of the matrix separately. It is obviously that $PPowersof2(M)$ can be optimized by the parallel computing. Namely, let one processing unit to do $PPowersof2(\vec{b}_i)$ $(0 \le i < N)$, where $\vec{b}_i$ is the i-th row of matrix M. So we let N processing unit to do $PPowersof2(\vec{b}_i)$ at the same time. Then the time complexity of $PPowersof2(M)$ is also $\mathrm{O}(N)$ under the parallel computing.

$PFlatten()$: $Flatten()$ can ensure all the vectors and matrices are B-strongly-bounded in GSW scheme, and then we can obtain a leveled FHE scheme that can evaluate a circuit of polynomial depth without bootstrapping, relinearization and modulus switching and so on. In fact, $Flatten()$ consists of $BitDecomp()$ and $BitDecomp^{-1}()$. By GSW, we have:

$$Flatten() = BitDecomp(BitDecomp^{-1}()) \tag{6}$$

So we don't need to specifically optimize it because its members have been optimized. Here we call the optimization result of $Flatten()$ to be $PFlatten()$, and we directly replace the corresponding operation with their optimization value. So we have:

$$PFlatten() = PBitDecomp(PBitDecomp^{-1}()) \tag{7}$$

From the above formula we can see, the time complexity of $PFlatten()$ is the product of the time complexity of $PBitDecomp()$ and $PBitDecomp^{-1}()$, since the time complexity of $PBitDecomp()$ and $PBitDecomp^{-1}()$ all are O(1) under the parallel computing, so the time complexity of $PFlatten()$ is also O(1) under the parallel computing.

3.2 Some Homomorphic Operations and Their Optimization

There are many homomorphic operations besides the above basic operations, such as $Mult()$, $add()$, $MultConst()$ and so on. They all are important operations in GSW scheme, and also have high time complexity, so they are needed to be optimized. The optimization principle is similar to the above optimization process. Namely, we use the principle of the parallel computing to optimize these operations. Next, we introduce their optimization process.

$PMatrixMult(C_1, C_2)$: In GSW scheme, $Mult()$ is used to do the matrix multiplication. According to GSW, we have:

$$Mult(C_1, C_2) = Flatten(C_1 \cdot C_2) \tag{8}$$

Because $Flatten(C)$ can be optimized into $PFlatten(C)$ which has a constant time complexity under the parallel computing, thus the time complexity of $Mult(C_1, C_2)$ is mainly dominated by the matrix multiplications. So far the optimal time complexity of matrix multiplications is $O(N^{2.3727})$ which is acchived by williams [18] under the serial computing. Accroding to the principle of the parallel computing, we can optimize matrix multiplication by Cannon algorithm whose working principle has been presented in section 2.2. Thus it is not necessary to introduce its theoretical knowledge here, we only demonstrate the algorithm through the following example.

$$A=\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,N-2} & a_{0,N-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,N-2} & a_{1,N-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{N-2,0} & a_{N-2,1} & \cdots & a_{N-2,N-2} & a_{N-2,N-1} \\ a_{N-1,0} & a_{N-1,1} & \cdots & a_{N-1,N-2} & a_{N-1,N-1} \end{bmatrix} B=\begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,N-2} & b_{0,N-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,N-2} & b_{1,N-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ b_{N-2,0} & b_{N-2,1} & \cdots & b_{N-2,N-2} & b_{N-2,N-1} \\ b_{N-1,0} & b_{N-1,1} & \cdots & b_{N-1,N-2} & b_{N-1,N-1} \end{bmatrix} P_{p\times p}=\begin{bmatrix} P_{0,0} & P_{0,1} & \cdots & P_{0,p-2} & P_{0,p-1} \\ P_{1,0} & P_{1,1} & \cdots & P_{1,p-2} & P_{1,p-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ P_{p-2,0} & P_{p-2,1} & \cdots & P_{p-2,p-2} & P_{p-2,p-1} \\ P_{p-1,0} & P_{p-1,1} & \cdots & P_{p-1,p-2} & P_{p-1,p-1} \end{bmatrix} \tag{9}$$

There are two $N \times N$-dimensional matrices A and B. According to Cannon algorithm, our high-performance computing system is a computer cluster of 2-dimensional grid which consists of $p \times p$ computers. Moreover, matrix A, matrix B and the computer cluster P are shown as in equation (9).

Next we will introduce the process to optimize the matrix multiplication with Cannon algorithm, and we can divide the above operation into four steps.

Step 1, divide matrix A and matrix B into p^2 blocks, each sub-block is

represented by A_{ij} and B_{ij} $(0 \le i < p, 0 \le j < p)$. Easy to calculate, the size of each sub-block is $(N / p) \times (N / p)$. Next, these sub-blocks are assigned to p^2 processing units, namely, assign the block A_{ij} and B_{ij} to the processing unit $P_{i,j}$ $(0 \le i, j < p)$, and then the corresponding sub-results $C_{i,j}$ can be calculated on the corresponding processing unit $P_{i,j}$, here $C_{i,j}$ is the product of the sub-blocks A_{ij} and B_{ij}.

Step 2, move the block A_{ij} $(0 \le i, j < p)$ loop to the left by i steps, and move the block B_{ij} $(0 \le i, j < p)$ cyclic up by j steps.

Step 3, execute the multiplication-addition computation on $P_{i,j}$, then we obtain a sub-result $C_{i,j}$, and there are $C_{i,j} = A_{ij} \times B_{ij} + C_{ij}$. Move A_{ij} $(0 \le i, j < p)$ loop to the left by 1 step, meanwhile move B_{ij} $(0 \le i, j < p)$ cyclic up by 1 step.

Step 4, repeat step 3, and execute the multiplication-addition on $P_{i,j}$ for p times altogether, execute single step cyclic on the blocks A_{ij} and B_{ij} for p times respectively.

Finally, we get matrix C which is the product of matrix A and matrix B, the result is shown in the following formula.

$$C = \begin{bmatrix} C_{0,0}\ C_{0,1} \ldots\ldots\ldots\ldots C_{0,p-2}\ C_{0,p-1} \\ C_{1,0}\ C_{1,1} \ldots\ldots\ldots\ldots C_{1,p-2}\ C_{1,p-1} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ C_{p-2,0} C_{p-2,1} \ldots C_{p-2,p-2} C_{p-2,p-1} \\ C_{p-1,0} C_{p-1,1} \ldots C_{p-1,p-2} C_{p-1,p-1} \end{bmatrix} \tag{10}$$

According to Cannon algorithm, the running time is $T_p = \frac{N^3}{p^2} + 4pt_s + 4t_w \frac{N^2}{p}$ on the computer cluster, where p is the dimension of computer cluster of 2-dimensional grid, t_s is the startup time, and t_w is the text-transmission time. Since both t_s and t_w are

"small", so the time complexity of Cannon algorithm is mainly dominated by N and p. If $p=\frac{1}{2}N$, then the time complexity of Cannon algorithm is $O(N)$. So the optimal time complexity of matrix multiplication under Cannon algorithm is $O(N)$ when taking the appropriate parameters.

$PMatrixAdd(C_1, C_2)$: $Add()$ is used to do the matrix addition in GSW scheme. According to GSW, we have:

$$Add(C_1, C_2) = Flatten(C_1 + C_2) \tag{11}$$

where C_1, $C_2 \in Z_q^{N\times N}$ are $N \times N$ dimension matrix. Because $Flatten()$ can be optimized by the parallel computing, and its optimization result is called as $PFlatten()$ whose time complexity is $O(1)$. So the time complexity of $Add(C_1, C_2)$ is mainly dominated by the matrix addition. The time complexity of matrix addition is $O(N^2)$ under the serial computing, so we can optimize it by the parallel computing. Below we will introduce the process to optimize the matrix addtion with the paralle computing.

For example, there are two $N \times N$-dimensional matrices A and B, and we will compute their sum. According to the parallel computing theory, the matrix addition can also be optimized by the computer cluster of 2-dimensional grid, which consists of $p \times p$ computers. Different from matrix multiplication, the optimization of matrix addition needs more computers. Matrix A, matrix B and the computer cluster are shown as below.

$$A=\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,N-2} & a_{0,N-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,N-2} & a_{1,N-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{N-2,0} & a_{N-2,1} & \cdots & a_{N-2,N-2} & a_{N-2,N-1} \\ a_{N-1,0} & a_{N-1,1} & \cdots & a_{N-1,N-2} & a_{N-1,N-1} \end{bmatrix} B=\begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,N-2} & b_{0,N-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,N-2} & b_{1,N-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ b_{N-2,0} & b_{N-2,1} & \cdots & b_{N-2,N-2} & b_{N-2,N-1} \\ b_{N-1,0} & b_{N-1,1} & \cdots & b_{N-1,N-2} & b_{N-1,N-1} \end{bmatrix} P=\begin{bmatrix} P_{0,0} & P_{0,1} & \cdots & P_{0,N-2} & P_{0,N-1} \\ P_{1,0} & P_{1,1} & \cdots & P_{1,N-2} & P_{1,N-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ P_{N-2,0} & P_{N-2,1} & \cdots & P_{N-2,N-2} & P_{N-2,N-1} \\ P_{N-1,0} & P_{N-1,1} & \cdots & P_{N-1,N-2} & P_{N-1,N-1} \end{bmatrix} \tag{12}$$

Firstly, divide the matrices A and B into $N \times N$ blocks, namely A_{ij} and B_{ij} $(0 \le i, j < N)$ and each block only contains one element. Assign the blocks A_{ij} and B_{ij} to the processing unit $P_{i,j}$ $(0 \le i, j < N)$, and then compute the result $C_{i,j}$ on $P_{i,j}$,

where $C_{i,j}$ is the sum of the blocks A_{ij} and B_{ij}. Then we get C which is a $N \times N$ matrix, and its element is $C_{i,j}$ $(0 \le i, j < N)$. In fact, Matrix C is the sum of the matrix A and matrix B.

It is obvious that the time complexity of matrix addition is O(1) under the parallel computing because all the computations work at the same time. We call the optimization value of $Add(C_1, C_2)$ to be $PMatrixAdd()$.

$PMultConst(C, \alpha)$: $MultConst(C, \alpha)$ executes the matrix-constant-multiplication, where $C \in Z_q^{N \times N}$ is a ciphertext, $\alpha \in Z_q$ is a constant number. According to GSW we have:

$$M_\alpha \leftarrow Flatten(\alpha . I_N) \tag{13}$$

$$MultConst(C, \alpha) = Flatten(M_\alpha \cdot C) \tag{14}$$

Since $Flatten()$ has a optimization value under the parallel computing, namely $PFlatten()$ whose time complexity is O(1), and the time complexity of $M_\alpha \cdot C$ is Matrix-Multiplication whose time complexity is $O(N)$ under the parallel computing, so the time complexity of $MultConst(C, \alpha)$ is $O(N)$ under the parallel computing, and we call $PMultConst(C, \alpha)$ as the optimization value of $MultConst(C, \alpha)$.

From the above analysis we can see that the time complexity have been greatly improved when using the parallel computing to optimize the basic operations and homomorphic operations of GSW scheme. Below we compare the time complexity of them before and after optimization, as shown in **Table 1**.

Table 1. The time complexity comparisons before and after optimization

$BitDecomp()$		$PBitDecomp()$		$BitDecomp^{-1}()$		$PBitDecomp^{-1}()$		$Powersof\ 2()$	
V	$O(k^2)$	V	O(1)	V	$O(k^2)$	V	O(1)	V	$O(N*k)$
M	$O(Nk^2)$	M	O(1)	M	$O(Nk^2)$	M	O(1)	M	$O(N^2*k)$

PPowersof 2()		*Flatten*()		*PFlatten*()		*Mult*()	*PMatrixMult*()
V	$O(N)$	V	$O(k^4)$	V	O(1)		
M	$O(N)$	M	$O(N^2k^4)$	M	O(1)	$O(N^{\omega+2}\cdot\kappa^4)$	$O(N)$
Add()		*PMatrixAdd*()		*MultConst*()		*PMultConst*()	
$O(N^4\cdot\kappa^4)$		O(1)		$O(N^{4+\omega}\cdot\kappa^8)$		$O(N)$	

It should be noted in **Table 1** that V represents the vector and M represents the matrix. Moreover in **Table 1**, the original time complexity of some operations is significantly higher than the existing optimal time complexity, such as $Mult()$, namely matrix multiplication whose the optimal time complexity is $O(N^{\omega})$, but here is $O(N^{\omega+2}\cdot\kappa^4)$. Why? In order to ensure the result ciphertexts is also B-strongly-bounded, it needs additional operations. For example, in GSW, there is:

$$Mult(C_1, C_2) = Flatten(C_1 \cdot C_2) \tag{15}$$

In the above formula, it introduces the additional operation, namely $Flatten()$ whose time complexity is $O(N^2k^4)$. Thus, the time complexity of the original operation will become larger, through introducing the additional operation.

3.3 The Encryption Model and Its Optimization

GSW scheme is called to be the approximate eigenvector method. Its main idea is to encrypt the plaintext μ into the ciphertext matrix C by the approximate eigenvector $\vec{v}$ and fully homomorphic encryption algorithm, where μ is a "small" integer, ciphertext C is a $N\times N$ dimension matrix over $\mathbb{Z}_q$, the key $\vec{v}$ is a N-dimensional vector over $\mathbb{Z}_q$ and $\vec{e}$ is a small error vector. From GSW we can see that the

encryption equation is:

$$C.\vec{v} = \mu.\vec{v} + \vec{e} \tag{16}$$

As shown in the above equation, key $\vec{v}$ is an approximate eigenvector of the ciphertext matrix C, and the plaintext μ is the approximate eigenvalue of the ciphertext matrix C, $\vec{e}$ is the error vector which is B-bounded and its main function is to ensure the security of the cryptographic scheme. The main security principle of this cryptographic scheme is the LWE problem introduced by Regev [25].

Note that the ciphertext of this cryptographic scheme is matrix, the result of ciphertext calculation is also the same dimension matrix, so this way can eliminate the ciphertext dimension expansion problem, and also furtherly eliminate many complex and expensive operations, such as modulus switching, relinearization and so on.

However, this scheme can only evaluate polynomials of polynomial degree in N, namely, this scheme is only somewhat homomorphic encryption scheme. In order to obtain a leveled fully homomorphic encryption scheme, it requires the plaintext μ and the ciphertext matrix C are B-strongly-bounded, and the error vector $\vec{e}$ is B-bounded. Then the encryption scheme is shown as the following formula according to GSW scheme.

$$C = Flatten(\mu \cdot I_N + BitDecomp(R \cdot A)) \tag{17}$$

However, in order to improve the efficiency of the FHE scheme, we use the parallel computing to optimize the complex operation in GSW scheme. From the above formula we can see, there are some basic operations which have been optimized in the previous section, such as $Flatten()$, $BitDecomp()$, matrix addtion, matrix multiplication and so on. In the above formula, it mainly involves three operations, such as $MatrixAdd()$, $Flatten()$, $BitDecomp()$ and so on. These operations have been optimized in the previous section, so we can directly use the optimization results to replace the corresponding operations in the above formula. Finally, we get the optimization results, as shown below:

$$C = PFlatten(PMatrixAdd(\mu \cdot I_N, PBitDecomp(PMatrixMult(R, A)))) \tag{18}$$

Where μ is the plaintext, C is the ciphertext corresponding to μ, I_N is a N-dimensional unit matrix, R is a random $N \times m$ matrix with 0/1 elements, A is $m \times (n+1)$ dimension matrix over $\mathbb{Z}_q$. Note that both R and A are generated by

KeyGen() introduced in GSW, so we no longer introduce it in details.

Next, we analyze the performance of this encryption model. From the above we know, both *PFlatten*() and *PBitDecomp*() have constant time complexity, so the main overhead of the encryption model is the product of $R \cdot A$, the product of $\mu \cdot I_N$ and the matrix addition. Because the matrix multiplication and the matrix addition have been optimized under the parallel computing, and their time complexity are $O(N)$ and $O(1)$ respectively. Although R and A are not N-dimensional matrices, in fact, their dimension is less than N, so the time complexity of $R \cdot A$ is less than the time complexity of the matrix multiplication whose dimension is N. For convenience, we take the time complexity of $R \cdot A$ for $O(N)$. From the above, we can see that the time complexity of matrix addtion and the matrix-constant-multiplication are $O(1)$ and $O(N)$. So the time complexity of encryption model is $O(N)$ under the parallel computing .

Because GSW scheme is called to be the FHE scheme of the approximate eigenvector method, so we verify that the encryption module is consistent with the approximate eigenvector attributes under the under parallel computing. That is, verify that whether the new FHE scheme is correct. Next, we verify that the optimized encryption formula still conforms to the properties of the approximate eigenvector method, then we have:

$$C.\vec{v} = PFlatten(\mu.I_N + PBitDecomp(R.A)).\vec{v} \tag{19}$$

Accroding to GSW, we know that:

$$< Flatten(\vec{a}), Powersof\ 2(\vec{b}) >=< \vec{a}, Powersof\ 2(\vec{b}) > \tag{20}$$

Correspondingly, we have:

$$< PFlatten(\vec{a}), PPowersof\ 2(\vec{b}) >=< \vec{a}, PPowersof\ 2(\vec{b}) > \tag{21}$$

and $\vec{v} = PPowersof\ 2(\vec{s})$, so we have:

$$\begin{aligned} &PFlatten(\mu \cdot I_N + PBitDecomp(R \cdot A)) \cdot \vec{v} = (\mu \cdot I_N + PBitDecomp(R \cdot A)) \cdot \vec{v} = \\ &\mu \cdot \vec{v} + PBitDecomp(R \cdot A) \cdot \vec{v} = \mu \cdot \vec{v} + R \cdot A \cdot \vec{s} = \mu \cdot \vec{v} + small \end{aligned} \tag{22}$$

So the optimized encryption formula still satisfies the properties of the approximate eigenvector method, it can be seen that our optimization is correct.

3.4 Decryption Model and Its Optimization

The main function of decryption model is to decrypt the ciphertext obtained by the homomorphic encryption or homomorphic operations, and this decryption operation is shown as follows according to GSW:

$$x \leftarrow < C_i, \vec{v} >= \mu \cdot \upsilon_i + e_i \tag{23}$$

$$\mu = \lceil x / \upsilon_i \rfloor \tag{24}$$

where C_i is the i-th row of the ciphertext matrix C, v_i is the i-th element of vector $\vec{v}$, and e_i is the i-th element of the error vector $\vec{e}$. In order to correctly decrypt it, there is a big coefficient in $\vec{v}$ at least, and this can be guaranteed by $PPowersof2()$. From the above formula, we know that the time complexity of decryption model is $O(1)$, so it is not necessary to be optimized.

3.5 Circuit Model and Its Optimization

In order to meet the plaintext {0,1} space range, GSW scheme uses boolean circuit to realize the FHE scheme. According to De Morgan theorem, we can see that a cryptographic scheme can be called as fully homomorphic encryption scheme when it just supports one of the following homomorphic operations, such as *NAND* homomorphic operation, *AND* and *XOR* homomorphic operation, or *NOR* homomorphic operation. It uses *NAND* homomorphic operation in GSW scheme, so we only optimize *NAND* circuit here.

$PNAND(C_1, C_2)$: Accroding to GSW scheme, we can use *NAND* gates to construct a leveled FHE scheme whose depth of the arithmetic circuit is L. In fact, it uses $NAND(C_1, C_2)$ to express the ciphertext of $NAND(\mu_1, \mu_2)$ in GSW, where μ_1 and μ_2 all are the binary bits, C_1 and C_2 are the ciphertext of μ_1 and μ_2. In order to optimize the *NAND* gates, we can only optimize the $NAND(C_1, C_2)$.

Accroding to GSW, we know that:

$$NAND(C_1, C_2) = Flatten(I_N - C_1.C_2) \tag{25}$$

From the above formula we can see that there are three basic operations, namely $Flatten()$, matrix subtraction and matrix multiplication. Here $Flatten()$ and matrix multiplication have been optimized in the previous section, so we can use its optimization results directly, namely $PFlatten()$ and $PMatrixMult()$. Because matrix subtraction is the opposite operation of the matrix addition, so we can use the optimization value of matrix addition to replace the optimization value of matrix subtraction according to the relationship between addition and subtraction. So we can directly use the optimization results to replace the corresponding operation in the above formula, and the result is shown as the following formula:

$$PNAND(C_1, C_2) = PFlatten(PMatrixAdd(I_N, -PMatrixMult(C_1, C_2))) \tag{26}$$

From the above formula we can see, the time complexity of $PFlatten()$ and $PMatrixAdd()$ all are $O(1)$, and the time complexity of $PMatrixMult()$ is $O(N)$. Hence, the time complexity of $PNAND(C_1, C_2)$ is $O(N)$ under the parallel computing.

4.The Performance of PGSW

According to GSW scheme, we obtain a leveled fully homomorphic encryption scheme which is a circuit of depth-L with $NAND$ gates. For the dimension parameter N and the depth parameter L, GSW scheme evaluates depth-L circuits of $NAND$ gates with $O(N^{4+\omega} \cdot \kappa^4)$ field operations for per gate, where $\omega < 2.3727$; In PGSW scheme, the field operations of per gate is $O(N)$, so the time complexity for evaluating depth-L circuits of $NAND$ gates is $O(NL)$ while GSW scheme is $O(N^{4+\omega} \cdot \kappa^4 \cdot L)$.

Moreover, although the decryption operation in the original scheme does not need to be optimized, the encryption operation in the original scheme is still optimized by the parallel computing. It can be seen that the time complexity of the encryption operation before being optimized is $O(N^{\omega+5} \cdot \kappa^6)$, and the time complexity of the encryption operation after being optimized is $O(N)$. The reason why the time complexity of encryption operation is greatly improved is that its encryption operation is composed of

many complex operations, and the time complexity of the encryption operation is the product of the time complexitys of these complex operations. Because these complex operations can be optimized greatly by the parallel computing, so the time complexity of the encryption operation can also been optimized greatly.

It can be seen that the time complexity of PGSW has been greatly improved. There are the performance comparisons of the usual FHE schemes showed as in **Table 2** [25] .

Table 2. The performance comparisons for the usual FHE schemes

Scheme	**DGHV**	**BGV**	**Bra12**	**GSW**	**PGSW**
Performance	$O(\lambda^{14})$	$\tilde{O}(\lambda^{2})$	$\tilde{O}(\lambda^{6})$	$O(N^{4+\omega}\cdot\kappa^{4})$	$O(N)$

5.Experiments

In this section, we take some experiments to verify the performance of our scheme. Becasuse the main overhead of this scheme is matrix multiplication, and the time complexity of other operations can not exceed the time complexity of the matrix multiplication, so we mainly implement Cannon algorithm based on MPI. Experimental environment is as follows: the operating system platform is win7, Cpu is i5-3337U@1.80GHz, the development kit is visual C++ 6.0 and the mpich2-1.4p1-win-ia32. The experimental results are shown as in **Table 3.**

Table 3. the experimental results basing on Cannon algorithm

Experiment 1		**Experiment 2**		**Experiment 3**	
200	1	200	4	200	8
3.853772s		1.530456s		0.745632s	
Experiment 4		**Experiment 5**		**Experiment 5**	
400	1	400	4	400	8
5.772368s		3.163587s		1.465911s	

As can be seen from **Table 3**, when the dimension of matrix is 200 and the number of the processing unit is 1, the overhead time is 3.853772s through running Cannon algorithm which is implemented by MPI; When the dimension of matrix is 200 and the number of the processing unit is 4, the overhead time is 1.530456s through running

Cannon algorithm which is implemented by MPI; When the dimension of matrix is 200 and the number of the processing unit is 8, the overhead time is 0.745632s through running Cannon algorithm which is implemented by MPI; When the dimension of matrix is 400 and the number of the processing unit is 1, the overhead time is 5.772368s through running Cannon algorithm which is implemented by MPI; When the dimension of matrix is 400 and the number of the processing unit is 4, the overhead time is 3.163587s through running Cannon algorithm which is implemented by MPI; When the dimension of matrix is 400 and the number of the processing unit is 8, the overhead time is 1.465911s through running Cannon algorithm which is implemented by MPI;

From the **Table 3**, we know that the overhead time of Cannon algorithm will be reduced when the number of the processing unit increases. According to Cannon algorithm, the largest number of processing unit is $\frac{N}{2}$, then we can predict that its overhead time of Cannon algorithm will reach microseconds. It can be seen that the performance of the FHE scheme based on the parallel computing has been greatly improved, and it promotes the developement of FHE scheme.

6. Conclusions

With the development of cloud computing, more and more user's privacy datas are stored on the cloud servers which are untrusted platform, so the leakage problem of user's privacy datas will become an unavoidable problem, and then urgently need to be solved. Accroding to the attributes of FHE, it is the best way to solve the leakage problem of privacy data on untrusted servers. However, the current FHE schemes aren't suitable for the practical applications because of their inefficiency, so we urgently need to improve the efficiency of the existing FHE schemes.

In this paper, we optimize GSW scheme by the parallel computing, and then gain PGSW scheme. From the analysis we know that the performance of the new scheme has been greatly improved, as shown in **Table 2**. Through the experiment we known that the time overhead of matrix multiplication which is the highest time complexity of GSW scheme will reach the level of microsecond when we select appropriate amount of the processing units of high performance computing system. Although PGSW scheme takes up more hardware resources, and also increases the cost of FHE scheme, it play an important role in putting FHE scheme into practical applications.

References

[1] R. W. Huang, X. L. Gui, S. Yu, "Design of Privacy-Preserving Cloud Storage Framework," in *Proc. of the Ninth International Conference on Grid and Cloud Computing*, pp. 128-132, November 1-5, 2010. Article (CrossRef Link).

[2] J. Alperin-Sheriff, C. Peikert, "Faster bootstrapping with polynomial error,". in *Proc. of the International Cryptology Conference*, pp. 297–314, August 17-21, 2014. Article (CrossRef Link).

[3] K. Myungsun, H. T. Lee, S. Ling, H. X. Wang, "On the Efficiency of FHE-based Private Queries," *IEEE Transactions on Dependable & Secure Computing*, vol. 1, no.99, pp.1176-1189, 2016. Article (CrossRef Link).

[4] H. S. Wang, Q. Tang, "Efficient Homomorphic Integer Polynomial Evaluation based on GSW FHE," *Cryptology ePrint Archive*, Report 2016/488, pp.488-505, 2016. Article (CrossRef Link).

[5] J. H. Cheon, K. Han, D. Kim, "Faster Bootstrapping of FHE over the Integers," *Cryptology ePrint Archive*, Report 2017/079, pp.79-91,2017. Article (CrossRef Link).

[6] S. Halevi, V. Shoup, "Bootstrapping for helib," in *Proc. of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 641–670, April 23-30, 2015. Article (CrossRef Link).

[7] Z. Brakerski, C. Gentry, V. Vaikuntanathan, "(Leveled)fully homomorphic encryption without bootstrapping," *in Proc. of the 3rd Innovations in Theoretical Computer Science Conference*, pp. 309-325, January 8-10, 2012. Article (CrossRef Link).

[8] M. V. Dijk, C. Gentry, S. Halevi, "Fully homomorphic encryption over the integers," in *Proc. of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 24-43, May 30-June 3, 2010. Article (CrossRef Link).

[9] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *Proc. of the 32nd Annual Cryptology Conference*, pp.868-886, August 19-23, 2012. Article (CrossRef Link).

[10] C. Genry, A. Sahai, B. Water, "Homomorphic encryption from learning with errors: conceptually-simpler, asymptotically-faster, attribute-based," in *Proc. of the 33rd Annual Cryptology Conference Advances in Cryptology*, pp.75-92, August 18-22, 2013. Article (CrossRef Link).

[11] L. Ducas and et al, "FHEW:Bootstrapping Homomorphic Encryption in less than a second," in *Proc. of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp.617-640, April 23-30, 2015. Article (CrossRef Link).

[12] Z. Brakerski, V. Vaikuntanathan, "Efficient fully homomor-phic encryption from (standard) LWE," in *Proc. of IEEE 52nd Annual Symposium on Foundations of Computer Science*, pp. 97-106, October 22-25, 2011. Article (CrossRef Link).

[13] C. Gentry, S. Halevi, N. Smart, "Homomorphic evaluation of the AES circuit," in *Proc. of the 32nd Annual Cryptology Conference*, pp. 850-867, August 19-23, 2012. Article (CrossRef Link).

[14] Z. Brakerski, C. Gentry, S. Halevi, "Packed ciphertexts in LWE-based homomorphic encryption," in *Proc. of the 16th International Conference on Practice and Theory in Public-Key Cryptography*, pp. 1-13, February 26 – March 1, 2013. Article (CrossRef Link).

[15] R. Hiromasa, M. Abe and et al, "Packing Messages and Optimizing Bootstrapping in GSW-FHE," in *Proc. of IACR International Workshop on Public Key Cryptography*, pp. 699–715, March 30-April 1, 2015. Article (CrossRef Link).

[16] J.Biasse, L.Ruiz, "FHEW with efficient multibit bootstrapping," in *Proc. of the International Conference on Cryptology and Information Security in Latin America*, pp. 119–135, August 23-26, 2015. Article (CrossRef Link).

[17] I. Chillotti, N. Gama and et al, " Faster Fully Homomorphic Encryption:Bootstrapping in less than 0.1 Seconds," in *Proc. of the International Conference on the Theory and Application of Cryptology and Information Security*, pp. 3-33, December 4-8, 2016. Article (CrossRef Link).

[18] S. Nicola, "Design and Analysis of Distributed Algorithms, 1st Edtion," Wiley, New York, 2006. Article (CrossRef Link).

[19] V. V. Williams, "Multiplying matrices faster than coppersmith-winograd," in *Proc. of the forty-fourth annual ACM symposium on Theory of computing*, pp. 887–898, May 19-22, 2012. Article (CrossRef Link).

[20] E. Dekel, D. Nassimi, S. Sahni, "Parallel matrix and graph algorithms," *SIAM J. Comput*, vol.10, no.4, pp.657-675, November, 1981. Article (CrossRef Link).

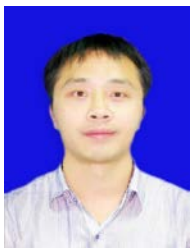
[21] D. G. R. A. Van, J. Watts, "SUMMA: scalable universal matrix multiplication algorithm," *Concurrency & Computation Practice & Experience*, vol.9, no.4, pp. 29-29, April, 1997. Article (CrossRef Link).

[22] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, "A three-dimensional approach to paralle matrix multiplication," *IBM Journal of Research and Development*, vol. 39, no. 5, pp. 575-582, September, 1995. Article (CrossRef Link).

[23] D. J. Evans, G. M. Megson, "A systolic simplex algorithm, 1st Edition," *International Journal of Computer Mathematics, Berkshire*, 1991. Article (CrossRef Link).

[24] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *in Proc. of the 37th Annual ACM Symposium on Theory of Computing*, ACM, pp. 84-93, May 22-24, 2005. Article (CrossRef Link).

[25] Z. G. Chen, J. Wang, X. X. Song, "Research of fully homomorphic encryption," *Application and Research About Computer Journal*, vol.31, no.6, pp. 1624-1630, April, 2014. Article (CrossRef Link).



Delin Tan(1981-) is a lecturer in SiChuan Normal University, China. He is currently pursuing a doctoral degree in the School of College of Geophysics,Chengdu University of Technology, China. His research interests include fully homomorphic encryption, cloud computing etc.

Huajun Wang(1964-) is a Professor and Ph.D of Chengdu University of Technology. His main research interests are sensor networks, network security, embedded and so on.

.