Middle East Technical University
Department of Computer Engineering
Wireless Systems, Networks and Cybersecurity (WINS) Laboratory

Find the Number of Islands

CENG519 Network Security
2021-2022 Spring
Term Project Report

Prepared by
Berker Acır
Student ID: 2098697
berker.acir@metu.edu.tr
Computer Engineering
18 June 2022

Abstract

With Fully Homomorphic Encryption (FHE), cloud computing services can do computations over encrypted data without requiring decryption and this emerges privacy-preserving applications. In this term project, one of the graph problem is studied and solved using FHE. The problem is the variation of the standard problem, counting the number of connected components in an undirected graph: Finding the number of islands. For solving the problem with FHE, different approaches are developed and their implementation details are explained in this term project report. Implementations are compared extensively with each other in terms of compile, key generation, encryption, execution, decryption and reference execution times, total running times and approximation rates.

Table of Contents

List of Figures

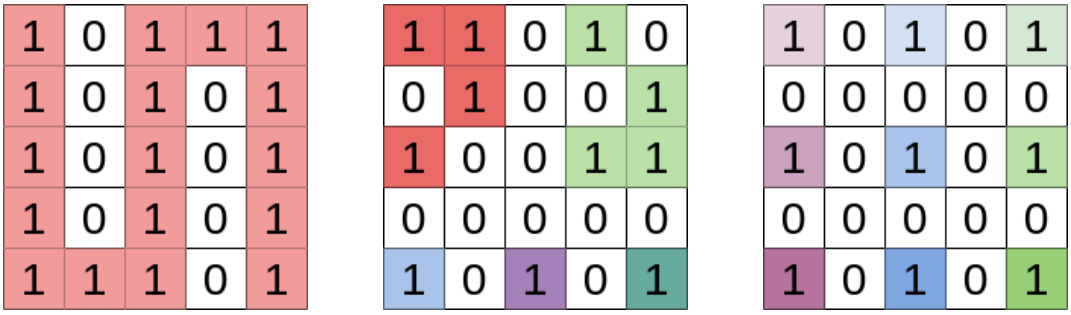
List of Tables

1 The Problem: Find the Number of Islands

The problem is to find the number of islands in the given boolean (or 0-1) 2D matrix. A group of connected 1s forms an island and a cell in the matrix can be connected to 8 neighboring cells. This problem is also the variation of the standard problem: **Counting the number of connected components in an undirected graph**. A connected component of an undirected graph is a subgraph in which every two vertices are connected to each other by at least one path, and there are no such vertices that is connected to no other vertices outside of the subgraph.

The problem can be easily solved by applying Depth First Search (DFS) on each component. In each DFS call, a connected component or subgraph is visited (or explored). Then, DFS is called on the next unvisited component (or node) until there are none left. The number of DFS calls gives the number of connected components (or islands in this problem). Also, Breadth First Search (BFS) can be used instead of DFS. Time complexity of the solution with DFS calls is $O(n.m)$ where n is the number of rows and m is the number of columns in the adjacency matrix. Also, the problem can be solved directly with **Connected-Component Labeling (CCL)** algorithms.

In the Figure 1 , there are three different 2D matrices consisted of only 0s and 1s together with their island counts.



(a) 1 island (b) 5 islands (c) 9 islands

Figure 1 . 2D matrices and their island counts

This term project report is organized as follows. Section 2 contains brief information about Homomorphic Encryption and gives information about libraries used in the project. Section 3 summarize the approach and its steps for solving this problem with FHE in details. Section 4 describes the methodoligies used in the project, presents results of the implementation simulations and discusses the project in the light of methodoligies and simulation results. Finally, Section 5 concludes the project and shows its future directions.

2 Homomorphic Encryption

Most of the encryption schemes include the following three functionalities: *key generation*, *encryption*, and *decryption*. **Symmetric-key** encryption schemes use the same secret key for both encryption and decryption, whereas **public-key** encryption schemes use two keys separately: a public key for encryption, a secret key for decryption. Therefore, public-key encryption schemes allow anyone who knows the public key to encrypt data, but only those who know the secret key can decrypt and read the data. Symmetric-key encryption can be

used for efficiently encrypting very large amounts of data, and enables secure outsourced cloud storage.

Traditional symmetric-key and public-key encryption schemes can be used for secure storage and communication; however, any outsourced computation has to decrypt the encrypted data first before any computations can take place. Therefore, cloud services providing outsourced computation capabilities must have access to the secret keys. Even though those services implement access policies in order to prevent unauthorized employees from accessing to these keys, it still does not guarantee data privacy to the data owners in the presence of both internal (malicious employee) and external (outside attacker) threats [1].

Homomorphic encryption can offer data privacy on cloud computing services since it allows to do computations directly on the encrypted data without accessing to the secret key, and the results of such encrypted computations remain encrypted which can be only decrypted with the secret key (by the data owner) [2]. Homomorphic refers to homomorphism in the algebra where each function in the plaintext space has an equivalent one in the ciphertext space and vice verse. Although homomorphic encryption has first proposed in 1970s [3], its implementation became practical with the Craig Gentry's breakthrough in 2009 [4]. After that, multiple homomorphic encryption schemes with different capabilities and trade-offs have been invented and most of these are public-key encryption schemes.

Homomorphic encryption includes multiple types of encryption schemes that can perform different classes of computations over encrypted data [2]. Some common types of homomorphic encryption are:

- ***Partially homomorphic encryption*** encompasses schemes that support the evaluation of circuits consisting of only one type of gate (e.g., addition or multiplication).
- ***Somewhat homomorphic encryption*** schemes can evaluate two types of gates, but only for a subset of circuits.
- ***Leveled fully homomorphic encryption*** supports the evaluation of arbitrary circuits composed of multiple types of gates of bounded (pre-determined) depth.
- ***Fully homomorphic encryption* (FHE)** allows the evaluation of arbitrary circuits composed of multiple types of gates of unbounded depth.

Even though homomorphic encryption is promising technology for privacy-preserving cloud computing, it has its own limitations and drawbacks. Only some computations on encrypted data are possible and this strictly restricts its use case scenarios. For example, it is impossible to draw insights from encrypted data. Also, homomorphic encryption comes with a significant performance overhead such that it is infeasible to run very large and complex algorithm homomorphically. Apart from its performance overhead, it has also memory overhead as data encrypted with homomorphic encryption is many times larger than unencrypted data. Hence, it may not make sense to encrypt entire large databases with this technology. Instead, meaningful use-cases are in scenarios where strict privacy requirements prohibit unencrypted cloud computation altogether.

For the majority of homomorphic encryption schemes, the multiplicative depth of circuits is the main practical limitation in performing computations over encrypted data. Homomorphic encryption schemes are inherently malleable[1]. In terms of malleability, homomorphic

[1]Malleability is a property of some cryptographic algorithms. An encryption algorithm is malleable if it is possible to transform a ciphertext into another ciphertext which decrypts to a related plaintext. For example, given an encryption of a plaintext m, it is possible to generate another ciphertext which decrypts to $f(m)$, for a known function f, without necessarily knowing or learning m [5].

encryption schemes have weaker security properties than non-homomorphic schemes.

2.1 The Simple Encrypted Arithmetic Library (SEAL)

Microsoft SEAL [6], first released in 2015, is a homomorphic encryption library that allows some operations (i.e. additions, multiplications, shift, negate etc.) to be performed on encrypted integers and real numbers. Other operations, such as encrypted comparison, or sorting, are not feasible to evaluate on encrypted data using this technology. Therefore, only specific privacy-critical cloud computation parts of programs can be implemented with this library.

Microsoft SEAL comes with different homomorphic encryption schemes with very different properties. The Brakerski-Gentry-Vaikuntanathan (BGV) [7] and the Brakerski/Fan-Vercauteren (BFV) [8] schemes allow modular arithmetic to be performed on encrypted integers. Also, the Cheon, Kim, Kim and Song (CKKS) [9] scheme allows additions and multiplications on encrypted real or complex numbers, but yields only approximate results. In applications such as summing up encrypted real numbers, evaluating machine learning models on encrypted data, or computing distances of encrypted locations, using CKKS scheme will be more suitable. But, for applications where exact values are needed, the BFV and BGV schemes are much more appropriate.

Microsoft SEAL offers leveled FHE operations and does not implement bootstrapping[2] for implemented schemes. The source code is available under an MIT license, is well documented, and includes a wide range of examples for all implemented schemes.

2.2 The Encrypted Vector Arithmetic Language and Compiler (EVA)

EVA [10] is an input language explicitly designed for vector arithmetic and targets arithmetic circuits in CKKS using the Microsoft SEAL library. It can be thought as a compiler for homomorphic encryption, that automates away the parts that require cryptographic expertise such as parameter selection, ciphertext maintenance etc. It gives the users a simple way to write programs that operate on encrypted data without encryption parameter selection, rescaling insertion, relinearization, having access to the secret key. The EVA program is converted into a term graph, and during multiple passes, graph rewriting rules transform it. However, EVA does not consider depth-reducing transformations. While EVA can be used for any (vectorized) application, the main focus is primarily on neural network inference.

3 FHE Friendly Approach

Even though the problem can be solved with a bounded (at most number of nodes) number of DFS (or BFS) calls, it cannot be solved using DFS with FHE since FHE limits the computational operations such as branching on encrypted values (i.e. comparison). In DFS (or BFS), it is required to check the value of the cell in the input matrix. For example, the algorithm needs to check the neighboring cells' values in order to decide whether include that cells in the connected component or not. However, checking the cell's value is not possible in FHE since that value will be encrypted and cannot be decrypted by the FHE program (server). Since, DFS or BFS cannot be employed for solving the problem with FHE, a new approach, FHE friendly approach is required. FHE friendly approach, that is developed in this term project as follows:

[2]Bootstrapping is a technique for resetting the noise level of a ciphertext to a fixed lower level.

1. (Optional) From the 2D input matrix, calculate the reduced matrix.
2. From the 2D input matrix (or reduced matrix), obtain the adjacency matrix.
3. From the adjacency matrix, calculate the distances matrix.
4. From the input matrix (or reduced matrix) and distances matrix, count the number of islands.

In the following subsections, the steps above are explained in more details. For simplicity, small sized matrices (such as 2x2, 3x3 etc.) are used as input for visuals.

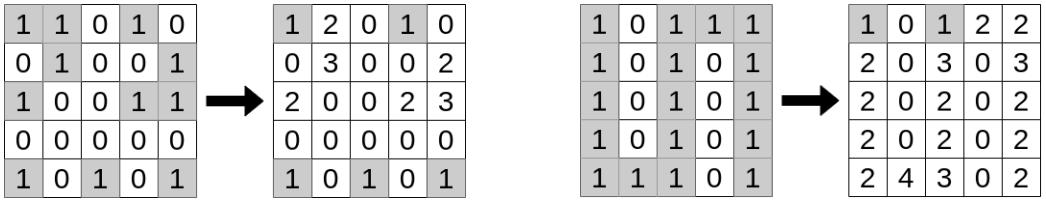
3.1 Reduce Ones

This step is optional as the number of islands can be calculated with either the original input matrix or the reduced matrix. The main reason of this step is to reduce the ones in the input matrix so that the client (with the secret key) can solve the problem with much more less computations after it receives the distances matrix from the FHE program (server). The input matrix is converted into reduced matrix with the following calculation:

Let the G be boolean 2D input matrix of size nxn, the R be the reduced matrix of size nxn, notation $G[i, j]$ represent the cell value of i^{th} row and j^{th} column in matrix G and out of bound elements of matrices are assumed to be zero (e.g. $G[x, y]$ equals to zero if $x \notin [1, n]$ and $y \notin [1, n]$). So, R can be calculated with the following equation:

$$R[i,j] = G[i,j].\Big(G[i-1,j-1] + G[i-1,j] + G[i-1,j+1] + G[i,j-1] + G[i,j]\Big)$$

With this calculation, the number of 1 valued cells are decrease as the $R[i, j]$ value is the multiplication of the corresponding cell in G ($G[i, j]$) with the summation of some neighboring cells (such as left ($G[i, j-1]$), upper left ($G[i-1, j-1]$), upper ($G[i-1, j]$) and upper right ($G[i-1, j+1]$) neighbors) together with the cells own value ($G[i, j]$). In Figure 2 , there are examples of this step.



(a) Input matrix with 11 ones and 5 islands is converted into reduced matrix with 5 ones.

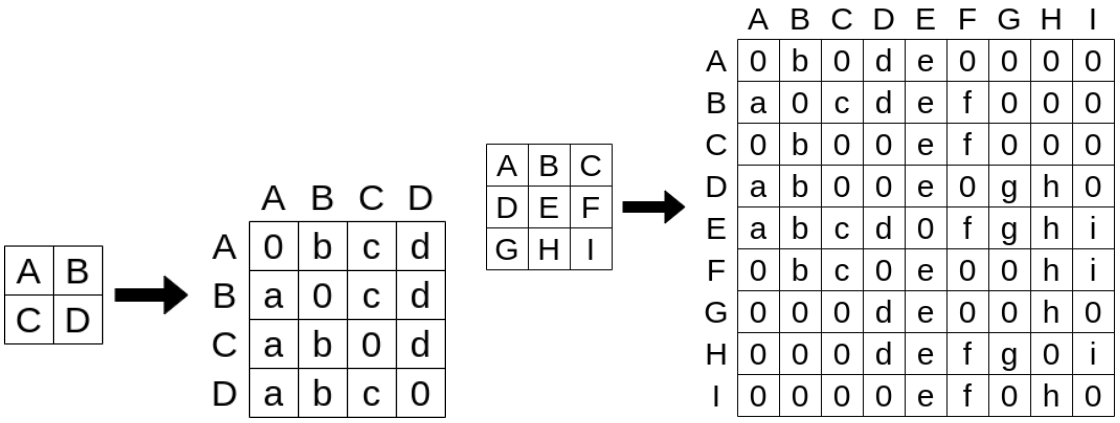
(b) Input matrix with 17 ones and 1 island is converted into reduced matrix with 2 ones.

Figure 2 . 2D matrices and their conversion into reduced matrices

Also, reduced matrix can be used to find approximate solution or to set an upper bound for island counts. For approximation, multiple reduce ones step with different neighboring settings and matrix traversal methods can be used and the output with minimum number of ones can be used as an approximate solution to the problem. Approximate solution might be useful for some of the cases but upper bound for approximation ratio might need to be found in that case.

3.2 Adjacency Matrix

Input matrix or reduced input matrix needs to be converted into adjacency matrix in order to find the distances between each cells/nodes. Each cell in 2D matrix is treated as a node so that n^2xn^2 adjacency matrix is created from nxn input matrix. Cell (i, j) of the input matrix has the index value of $i.n + j$ in rows and columns of the adjacency matrix. In Figure 3 , there are examples of adjacency matrix conversions.



(a) 2x2 matrix and its 4x4 adjacency matrix

(b) 3x3 matrix and its 9x9 adjacency matrix

Figure 3 . 2D matrices and their conversions into adjacency matrices

3.3 Distances Matrix

Since rows of the adjacency matrix gives 1-hop distances information, it is possible to calculate distances from each node to every other reachable nodes. After distances are obtained, distances matrix is used to check whether there is a path from a node to other nodes. Distances matrix is initialized with adjacency matrix, then the rows of distances matrix are updated as the follows:

Let $N = n^2$, ADJ be the adjacency matrix of size NxN, DIST be the distances matrix of size NxN, $G[i]$ represent the i^{th} row of the matrix G and $G[i, j]$ represent the value at i^{th} row and j^{th} column. Then, do the following for each row r:

$$\text{DIST}[r] = \sum_{\text{R=1}}^{\lceil N/2 \rceil} \sum_{\text{c=1}}^{N} \text{DIST}[r, c].\text{ADJ}[c]$$

R variable above is for repeating the summation, and $\sum_{\text{c=1}}^{N} \text{DIST}[r, c].\text{ADJ}[c]$ will be done $\lceil N/2 \rceil$ times. $\lceil N/2 \rceil$ is equal to or greater than length of the longest possible path in 2D input matrix. If $\text{DIST}[n.i + j, n.x + y]$ is greater than 0, that means the node/cell with (i, j) index can reach to the node/cell with (x, y) index.

In Figure 4 , there is an example of calculating distances matrix for the first row without doing any repetition. The graph and its adjacency matrix is different than in our problem. Yet, it might be a good example to demonstrate the calculation of distances matrix. The calculation done in the figure is just $\text{DIST}[1] = \sum_{\text{c=1}}^{4} \text{DIST}[1, c].\text{ADJ}[c]$. At the beginning, node A can only reach to node B, but after calculations it can be seen that node A can reach

to every other nodes (including itself).

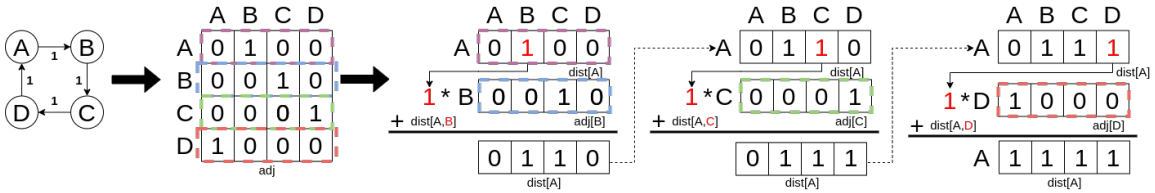


Figure 4 . Directed graph with 4 nodes together with its adjacency matrix and partial calculation of its distances matrix for the first row.

3.4 Islands

In this step, the client has input matrix, reduced matrix (optionally) and distances matrix. From these, client can find the number of islands. Distances matrix has the connectivity information of every nodes such that the client can check whether a node is connected to another node. The client starts from checking 1 valued node/cell and it populates every reachable 1 valued nodes/cells from that cell. Every populated/grouped 1's represent the islands and their counts is the result of this problem. Using reduced matrix instead of input matrix helps to save additional computations on client side since it reduces the number of 1s in the input matrix.

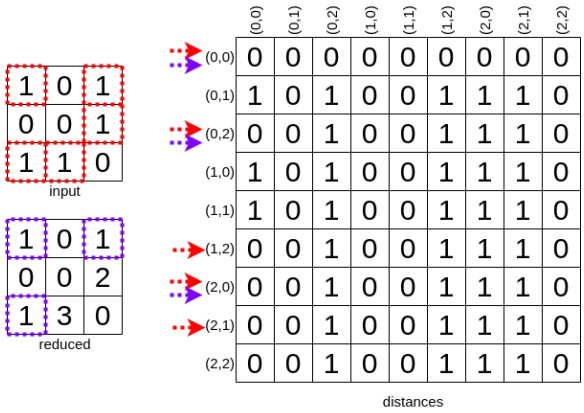


Figure 5 . Input matrix and its reduced version with distances matrix. Note that 1 values in distances matrix means that there is a path/paths from the cell in the row to the cell in the column.

Figure 5 includes example input matrix and its reduced version together with the distances matrix. Without using reduced input matrix, the client needs to create islands from 5 cells $((0,0),\ (0,2),\ (1,2),\ (2,0)$ and $(2,1))$: $(\mathrm{DIST}[(i,j),(x,y)] = \mathrm{DIST}[i.n+j, x.n+y])$

1. The client checks whether $(0,0)$ has path to any other cells and decides there is no path to other cells since every value in $\mathrm{DIST}[(0,0)]$ equal to 0. Then, $(0,0)$ is an island itself.
2. There are 4 cells $((0,2),\ (1,2),\ (2,0)$ and $(2,1))$ left, and then then client checks whether $(0,2)$ has path to any other cells. Since $\mathrm{DIST}[(0,2),(1,2)]$, $\mathrm{DIST}[(0,2),(2,0)]$ and $\mathrm{DIST}[(0,2),(2,1)]$ are not equal to zero, those 4 cells form an island.

3. The number of island is **2**.

With the reduced input matrix, the client needs to create islands from only checking distance information of 3 cells ((0, 0), (0, 2), and (2, 0)): (DIST$[(i, j), (x, y)]$ = DIST$[i.n + j, x.n + y]$)

1. The client checks whether $(0, 0)$ has path to any other cells and decides there is no path to other cells since every value in DIST$[(0, 0)]$ equal to 0. Then, $(0, 0)$ is an island itself.
2. There are 2 cells ($(0, 2)$ and $(2, 0)$) left, and then the client checks whether $(0, 2)$ has path to any other cells. Since DIST$[(0, 2), (2, 0)]$ is not equal to zero, those 2 cells form an island.
3. The number of island is **2**.

Using the reduced input matrix reduces the number of 1 valued cells which are also used in forming islands. As a result, there are less computation on the client side.

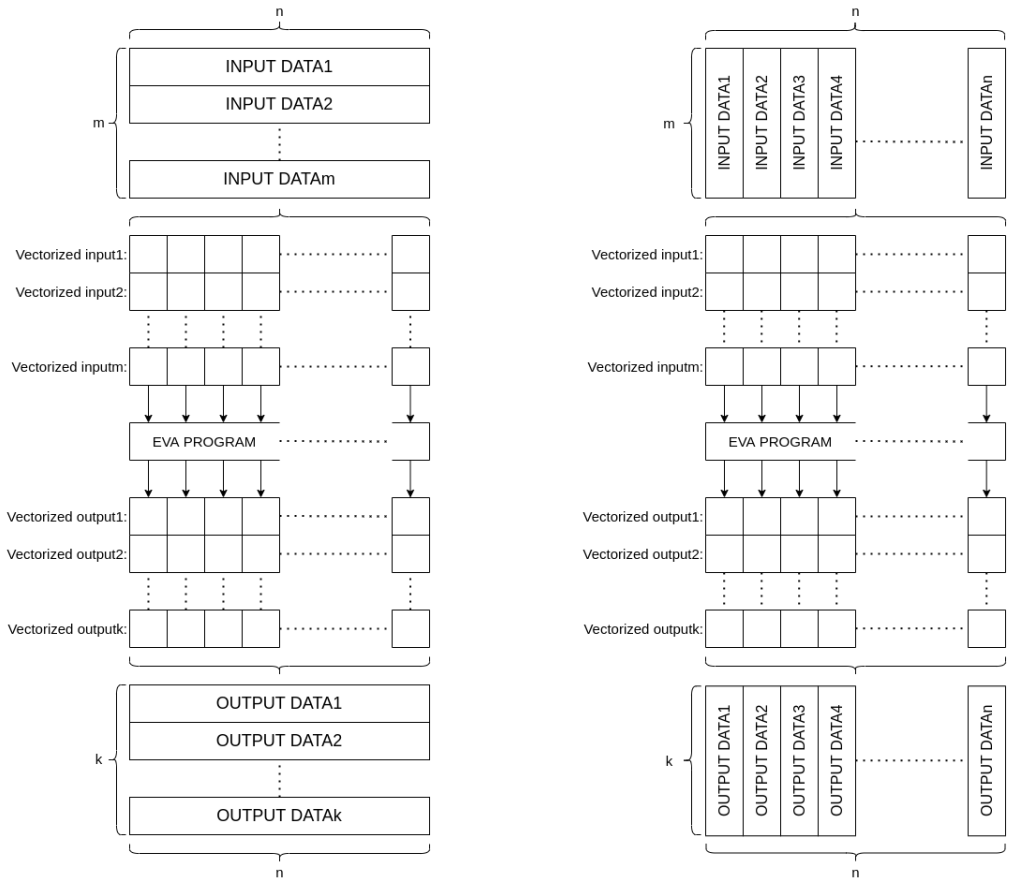
4 Results and Discussion

4.1 Methodology

Details of FHE friendly solution for the problem is explained in the previous section. In this section, FHE programs written with EVA compiler backed with CKKS scheme of SEAL library is explained in details. There are 2 types of EVA Programs (Reduce Ones and Compute Distances) and 3 different approaches (Vectorized, Parallelized and Parallelized without Loops). For the approaches, the EVA Programs have slight differences in terms of implementation but the underlying algorithm is same as explained in the previous section. Before describing the EVA programs, these are the common concepts in the programs.

- **Client:** The client first generates public and private keys with defined parameters (such as security level). And then, encrypts the input data that will be used by the EVA program (in the server side). After the data is encrypted, it is sent to the server which executes the compiled version of the EVA program with the encrypted data. Finally, the received results are decrypted with the private key. For the sanity checking and the evaluation of the FHE programs, there is an evaluation process which simply runs the same EVA program on the data itself without any encryption. This evaluation process is needed as CKKS scheme does approximate the values, and if inadequate encryption parameters (such as small modulus coefficients with wrong security level) used the resulting values can differ a lot than expected values. Therefore, output and reference output of the EVA programs are compared in terms of their Mean Squared Error (MSE) scores. Lesser the MSE means EVA program works as intended.
- **Server:** The server runs compiled version of the EVA program on encrypted inputs and sends back encrypted outputs of the program to the client. Server never decrypts the encrypted inputs regardless of required computations.

There are 3 implementations for solving the problem: **Vectorized** Implementation, **Parallelized** Implementation and **Parallelized** Implementation **without Loops**. Their implementation details and differences are explained in the following subsections. EVA Program can be implemented with the two options: **vectorized** and **parallel**. Illustrations of the two options are in Figure 6 :



(a) Vectorized (b) Parallel

Figure 6 . Vectorized vs Parallel Input/Output Options

- **Vectorized:** Multiple input data are vectorized and each vector elements are passed to the EVA Program. Therefore, the EVA Program is needed to be written accordingly as the program will process only vector elements. Shift operations are much more suitable for this approach as it enables accessing adjacent vector elements of same input data. One drawback of this approach is that EVA Program receives elements from different inputs rather than the same input data.
  - For example, let the EVA Program be multiplication of the input elements with their indices. So, m different input data with data size n (vector size) must be $\{$**input_1:**$[x_{11}, x_{21}, \ldots, x_{m1}]$, **input_2:**$[x_{12}, x_{22}, \ldots, x_{m2}], \ldots,$ **input_n:** $[x_{1n}, x_{2n}, \ldots, x_{mn}]\}$, the program is simply returning '1.**input_1**, 2.**input_2**, $\ldots$, n.**input_n**', and the output data is $\{$**output_1:**$[\hat{x}_{11}, \hat{x}_{21}, \ldots, \hat{x}_{m1}]$, **output_2:** $[\hat{x}_{12}, \hat{x}_{22}, \ldots, \hat{x}_{m2}], \ldots,$ **output_n:**$[\hat{x}_{1n}, \hat{x}_{2n}, \ldots, \hat{x}_{mn}]\}$.
  - Each output is equal to multiplication of its input index with its input.
- **Parallel:** Multiple input data are parallelized instead of vectorized and each input data is passed to the EVA Program. Therefore, the EVA Program is needed to be written accordingly since the whole input will be processed in the program. In the EVA Program, whole input data can be converted into *numpy* array with data type of *eva.Expr* and computations in the program can be done over that array. In the

end, the *numpy* array must be converted back to the dictionary compatible with EVA Program's Output. Even though computations of the program are done with *numpy*, there are still computation limitations such as broadcast operations between *numpy* arrays with *eva.Expr* data types which results in exception/error.

- For example, let the EVA Program be multiplication of the input elements with their indices. So, n (vector size) different input data with data size m must be $\{$**input_1:**$[x_{11}, x_{12}, \ldots, x_{1m}]$, **input_2:**$[x_{21}, x_{22}, \ldots, x_{2m}], \ldots,$ **input_n:** $[x_{n1}, x_{n2}, \ldots, x_{nm}]\}$, the program is simply returning '1.**input_1**, 2.**input_2**, $\ldots$, n.**input_n**', and the output data is $\{$**output_1:**$[\hat{x}_{11}, \hat{x}_{12}, \ldots, \hat{x}_{1m}]$, **output_2:** $[\hat{x}_{21}, \hat{x}_{22}, \ldots, \hat{x}_{2m}], \ldots,$ **output_n:**$[\hat{x}_{n1}, \hat{x}_{n2}, \ldots, \hat{x}_{nm}]\}$.
- Each output is equal to multiplication of $[1, 2, \ldots, n-1, n]$ with its input.

4.1.1 Vectorized Implementation

Input matrix of size nxn is converted into vector whose size is power of 2 and bigger than or equal to $(n+2)^2$ (+2 comes from the zero paddings.). Then, **Reduce Ones EVA Program** is deployed to convert the input matrix into reduced input matrix. Then, from the input matrix, the client itself creates the adjacency matrix and the client itself creates two more vectors from the adjacency matrix: distances matrix and extended matrix. Those 3 vectors' size is power of 2 and bigger than or equal to n^4 (There are n^4 elements in the adjacency matrix since there are n^2 nodes in the 2D input matrix.). Distances matrix and extended matrix is initialized from the adjacency matrix at the beginning.

With adjacency, distances and extended matrices as input vectors, **Compute Distances EVA Program** is run exactly $\lceil n^2/2 \rceil . n^2$ times. $\lceil n^2/2 \rceil$ represents length of the longest possible path in the input matrix and n^2 is for calculating distances for each element/node. With every iteration distances and extended matrices are updated. Note that distance matrix is updated within the EVA Program at each iteration, and then, the client adjust the extended matrix for the next iteration from the distances matrix and the next element/node.

After the client has obtained reduced input matrix (optional) and distances matrix, the number of islands can be found at the client side. nxn is the size of input matrix. In the vectorized implementation:

- **Reduce Ones EVA Program** is called just once with 1 vector with size of $2^{\lceil 2\log(n+2) \rceil}$.
- **Compute Distances EVA Program** is called $\lceil n^2/2 \rceil . n^2$ times with 3 vectors with size of $2^{\lceil 4\log(n) \rceil}$.
- The problem is solved for only one instance.

4.1.2 Parallelized Implementation

2^m input matrices whose sizes are nxn is converted into n^2 vectors with size 2^m. Then, **Reduce Ones EVA Program** is deployed to convert all input matrices into reduced input matrices. In the program, *numpy* matrices are constructed from the inputs where each value is type of *eva.Expr* and calculations are pretty straightforward as they are done with *numpy*[3]. After the calculations are done, the *numpy* arrays are converted back to appropriate output dictionary.

[3]Data type of *eva.Expr* has built-in operations such as multiplication with scalar or another *eva.Expr*, addition with a scalar or another *eva.Expr*, negation, shift etc. Those operations enable the use of *eva.Expr* data type with *numpy* and makes the implementations very easier. Even though there are built-in defined operations, some of the operations with the *numpy* such as broadcast operation can cause exceptions/errors.

Then, **Compute Distances EVA Program** is deployed to calculate distances matrix. This program converts the input vectors into *numpy* input matrices. From that matrices, adjacency matrices[4] are obtained and used further to get distances matrices. After the distances matrices are calculated, they are converted back to appropriate output dictionary.

After the client has obtained reduced input matrices (optional) and distances matrices, the number of islands for multiple inputs can be found at the client side. nxn is the size of input matrices and 2^m is the vector sizes and the number of different input graphs/matrices. In the parallelized implementation:

- **Reduce Ones EVA Program** is called just once with n^2 vectors with size of 2^m.
- **Compute Distances EVA Program** is called just once with n^2 vectors with size of 2^m.
- The problem is solved for 2^m instances.

4.1.3 Parallelized Implementation without Loops

Reduce Ones step is same as in the Parallelized Implementation. But, Compute Distances part is different. Even though **Compute Distances EVA Program** of Parallelized Implementation is straightforward and easy, it has a crucial problem: Multiplicative depth grows larger with the increase of n, and the implementation cannot compute matrices with size bigger than 2x2 (e.g. 3x3, 4x4 etc.) due to insufficient **bit modulus**. In order to tackle this problem, loops in the **Compute Distances EVA Program** is moved to the client side. So, the client will call this program $\lceil n^2/2 \rceil . n^2$ times. This approach is pretty similar to the approach in the Vectorized Implementation where $\lceil n^2/2 \rceil$ represents length of the longest possible path in the input matrices and n^2 is for calculating distance for each element. Also, input vectors differ from the previous implementation as the adjacency and distances matrices are created in the client side and passed as input to the **Compute Distances EVA Program**. Distances matrices are updated at each iteration/program call depending on the each row values.

After the client has obtained reduced input matrices (optional) and distances matrices, the number of islands for multiple inputs can be found at the client side. nxn is the size of input matrices and 2^m is the vector sizes and the number of different input graphs/matrices. In the parallelized implementation:

- **Reduce Ones EVA Program** is called just once with n^2 vectors with size of 2^m.
- **Compute Distances EVA Program** is called $\lceil n^2/2 \rceil . n^2$ times with $2.n^4$ vectors with size of 2^m.
- The problem is solved for 2^m instances.

4.1.4 Implementation Comparisons

In this subsection, similarities or differences of the three implementations that are described previously are discussed. In the Table 1, solution steps and where they are executed in the implementations are shown. Reduce Ones and calculation of Distances Matrix steps are

[4]Direct conversion of input matrix into adjacency matrix causes an exception regarding the use of scalar 0 values in it as SEAL library prohibits multiplication of encrypted value with scalar 0 which will result in **transparency of ciphertext**. In order to solve this, zero input vector is used where each element is the encrypted version of zero values. This might cause security problems as it resembles the **known-plaintext attacks**. If that is the case, conversion into adjacency matrix can be done in the client side and sent it to the program as done in the Vectorized Implementation in order to solve the security problem.

done by the server in all of the implementations. However, conversion of input matrix into adjacency matrix is done only by the server in Parallelized Implementation. This step is done together with the calculation of Distances Matrix. With the additional EVA Program (such as *Obtain Adjacency Matrix EVA Program*), adjacency step can be passed to the server in the Parallelized Implementation without Loops. However, such thing may not be possible for the Vectorized Implementation. Finding the number of islands step cannot be done in the EVA Program as it requires checking connectivity of 1 valued cells.

Step\Implementation	**Vectorized**	**Parallelized**	**P. w/out Loops**
Reduce Ones	Server	Server	Server
Adjacency Matrix	Client	Server	Client
Distances Matrix	Server	Server	Server
Islands	Client	Client	Client

Table 1. Solution steps and where they are executed in the implementations

In Table 2, the number of times that each EVA Program is called by the implementations are shown. Reduce Ones EVA Program is called only once in every implementations. However, Compute Distances EVA Program is called $\lceil n^2/2 \rceil . n^2$ times in both Vectorized Implementation and Parallelized Implementation without Loops where n^2 is the size of the input matrix/matrices.

EVA Program\Impl.	**Vectorized**	**Parallelized**	**P. w/out Loops**
Reduce Ones	1	1	1
Compute Distances	$\lceil n^2/2 \rceil . n^2$	1	$\lceil n^2/2 \rceil . n^2$

Table 2. The number of times that EVA Programs are called in the implementations where n^2 is the size of input matrix/matrices

Input sizes, vector sizes and number of inputs of implementations' EVA Programs are shown in Table 3. n^2 is the size of input matrix/matrices and 2^m is the number of different input matrices. Vectorized Implementation works on only one input matrix whereas Parallelized Implementations can work on multiple input matrices. In Vectorized Implementation, input sizes to the Reduce Ones and Compute Distances EVA Programs are 1 and 3 respectively while this is not the case for Parallelized Implementations since whole input matrix or distances matrix are used as input to EVA Programs.

Implementation	**Vectorized**		**Parallelized**		**P. w/out Loops**	
EVA Program	**Reduce**	**Compute**	**Reduce**	**Compute**	**Reduce**	**Compute**
Input Size	1	3	n^2	n^2	n^2	$2.n^4$
Vector Size	$2^{\lceil 2\log(n+2) \rceil}$	$2^{\lceil 4\log(n) \rceil}$	2^m	2^m	2^m	2^m
Number of Inputs	1		2^m		2^m	

Table 3. Input sizes, vector sizes and number of inputs that are used in EVA Programs of the implementations where n^2 is the size of input matrix/matrices and 2^m is the number of different input matrices in the Parallelized Implementations

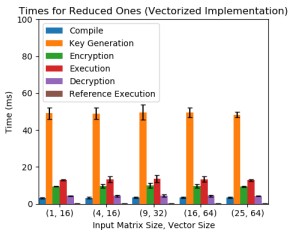
4.2 Results

FHE comes with computational overhead which means computations that are done over encrypted data will be slower than the computations on unencrypted data. In this section, implementations are run through 100 simulations with different parameters (such as matrix size, number of inputs) and their averaged results are compared with each other. **Compile**, **Key Generation**, **Encryption**, **Execution**, **Decryption**, **Reference Execution** times and **MSE** scores are saved for each EVA Program of the implementations in each simulation run, and then, those timings and scores are averaged for each EVA Program and the implementation. Since, Parallelized Implementations can also process multiple input data, their amortized running times are also considered. This section contains plots of those timings and MSE scores.

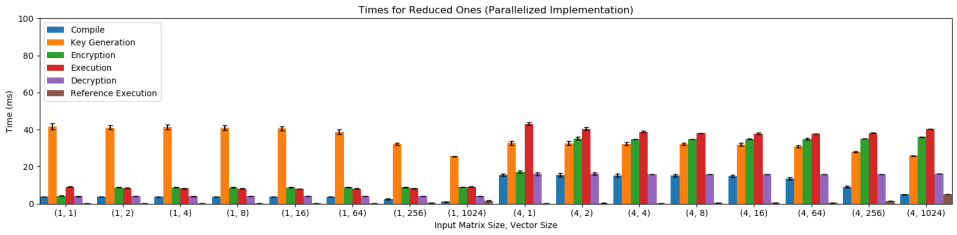
4.2.1 Plots of Reduce Ones EVA Programs

Means and standard deviations of timings and scores of Reduce Ones EVA Programs by the input matrix and vector sizes are shown in Figures 7 and 8 . From the figures, it can be said that increasing the vector size helps to decrease compile, key generation times and also MSE scores. As expected, increasing the input matrix size increases all the measured timings. However, it affects greatly the encryption, execution, decryption timings for the Parallelized Implementations due to increasing number and size of input vectors to the programs. In each Compute Distances EVA Programs, 30-bits are used for output ranges and input scales. In case of obtaining high MSE scores, increasing this values helps to reduce scale of errors on ciphertexts (and so, MSE scores).

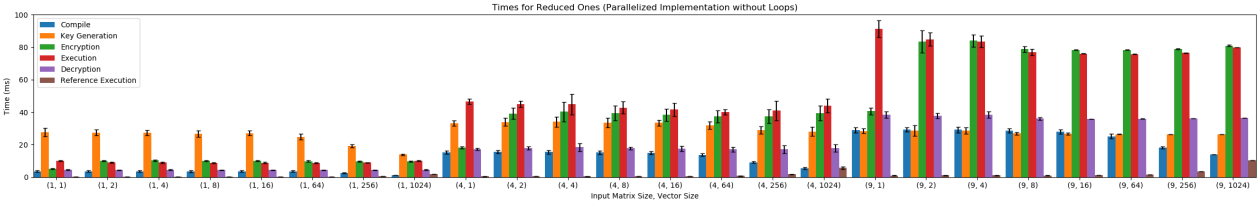
Note that, y-axis of timing plots are same, that means each implementations' Reduce Ones EVA Program runs in the same time-scale even though there are up to 10 times timing differences between the implementations' programs.



(a) Vectorized Imp.

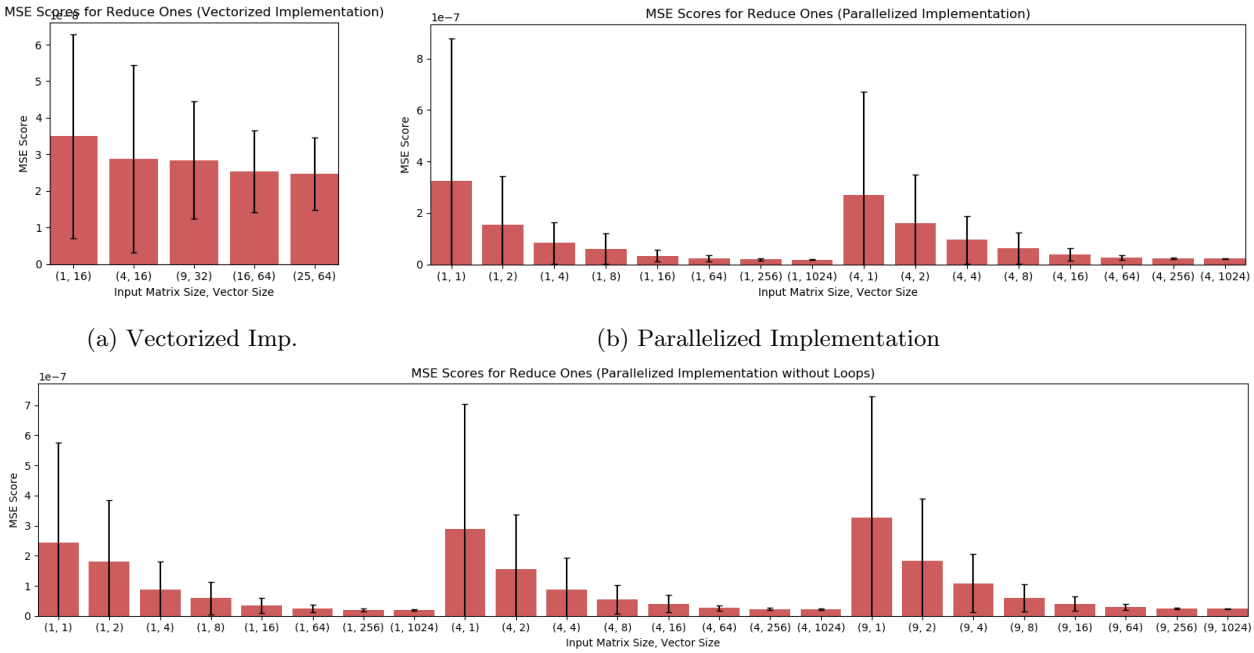


(b) Parallelized Implementation



(c) Parallelized Implementation without Loops

Figure 7 . Timings of all implementations' Reduce Ones EVA Programs by Input Matrix and Vector Sizes



(a) Vectorized Imp.

(b) Parallelized Implementation

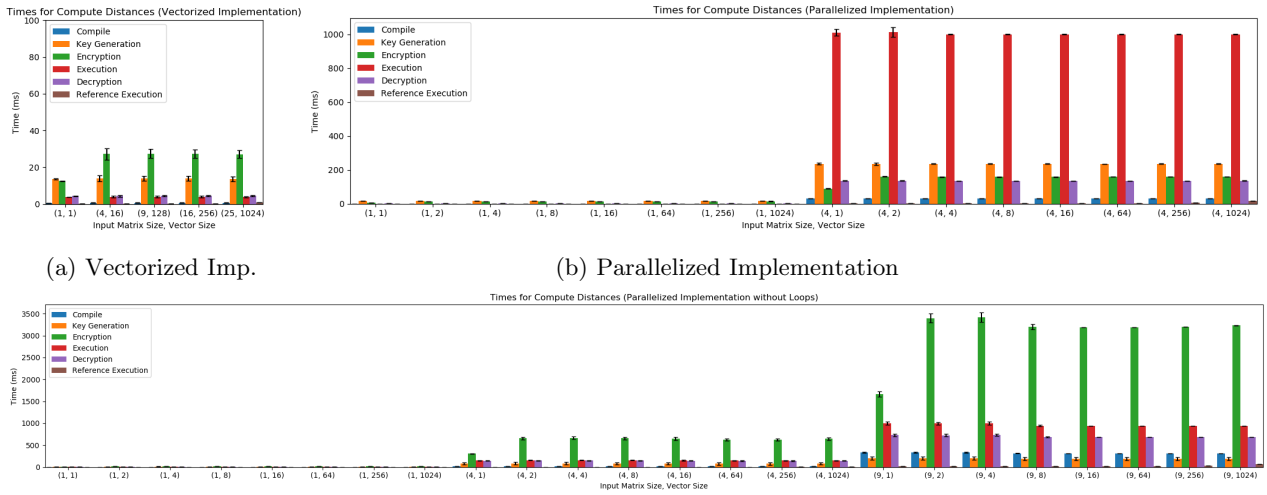
(c) Parallelized Implementation without Loops

Figure 8 . MSE scores of all implementations' Reduce Ones EVA Programs by Input Matrix and Vector Sizes

4.2.2 Plots of Compute Distances EVA Programs

Means and standard deviations of timings and scores of Compute Distances EVA Programs by the input matrix and vector sizes are shown in Figures 9 and 10 . From the figures, it can be said that increasing the vector size slightly helps to decrease MSE scores. As expected, increasing the input matrix size increases all the measured timings. However, it affects greatly the encryption, execution, decryption timings for the Parallelized Implementations due to increasing number and size of input vectors to the programs. From the plots, it can be seen that Parallelized Implementation without Loops has an overhead related with the input vectors to the programs as encryption timings are the most time consuming part of the program. Other than that, Parallelized Implementation has an overhead related with the Execution which is expected because the program is used to calculate distances matrix at one call. The other programs are called multiple times to calculate the distances matrices. Also, MSE scores of Parallelized Implementations are much more higher than the Vectorized Implementation due to high number of multiplications in the parallelized programs. In each Compute Distances EVA Programs, 30-bits are used for output ranges and input scales. In case of high MSE scores, increasing those values helps to reduce scale of errors on ciphertexts (and so, MSE scores).

Note that, y-axis of timing plots are not same, that means Vectorized program runs faster than all others, and Parallelized program is running faster than the program without loops. Also, it is important to keep in mind that Compute Distances EVA Program of Parallel Implementation is called just once to calculate distances whereas other programs are called multiple times. Comparison of implementations in terms of amortized runtime is also plotted in the next subsections. Another reminder is that Parallelized Implementations works on multiple inputs while Vectorized Implementation runs for only one input.

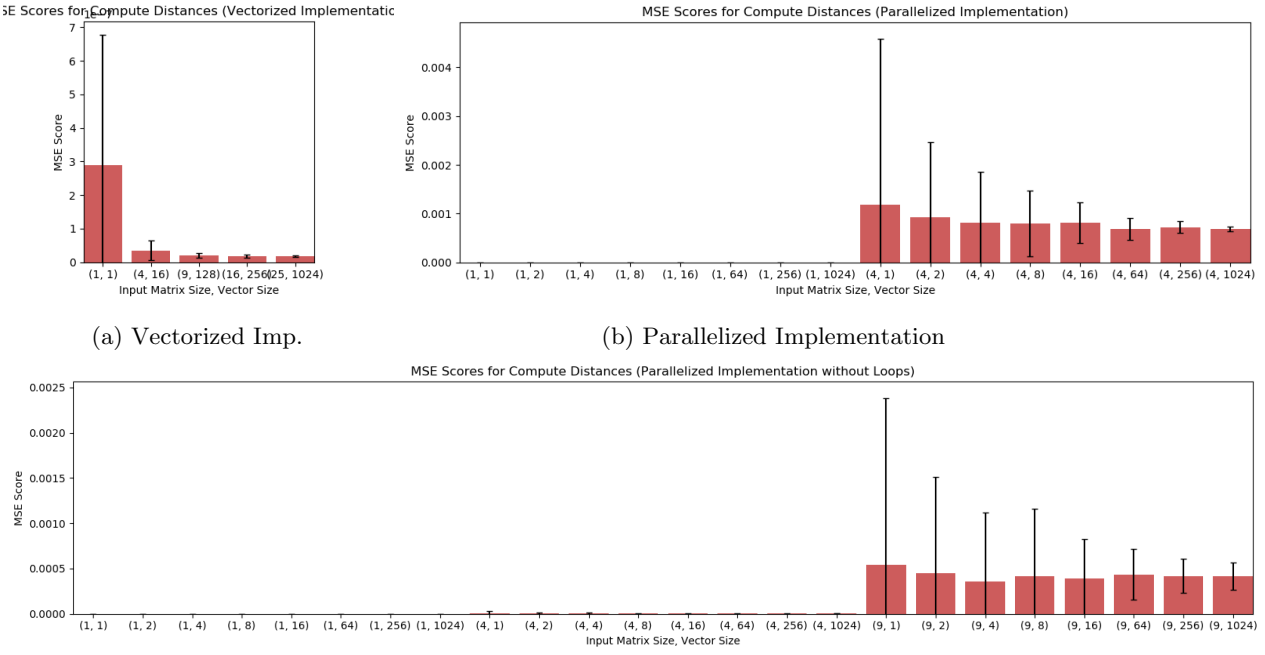


(a) Vectorized Imp.

(b) Parallelized Implementation

(c) Parallelized Implementation without Loops

Figure 9 . Timings of all implementations' Compute Distances EVA Programs by Input Matrix and Vector Sizes



(a) Vectorized Imp.

(b) Parallelized Implementation

(c) Parallelized Implementation without Loops

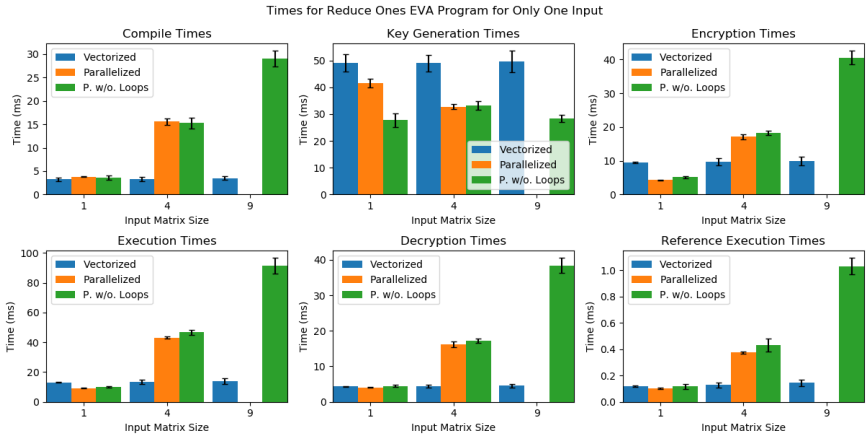
Figure 10 . MSE scores of all implementations' Compute Distances EVA Programs by Input Matrix and Vector Sizes

4.2.3 Comparison of EVA Programs with Only One Input

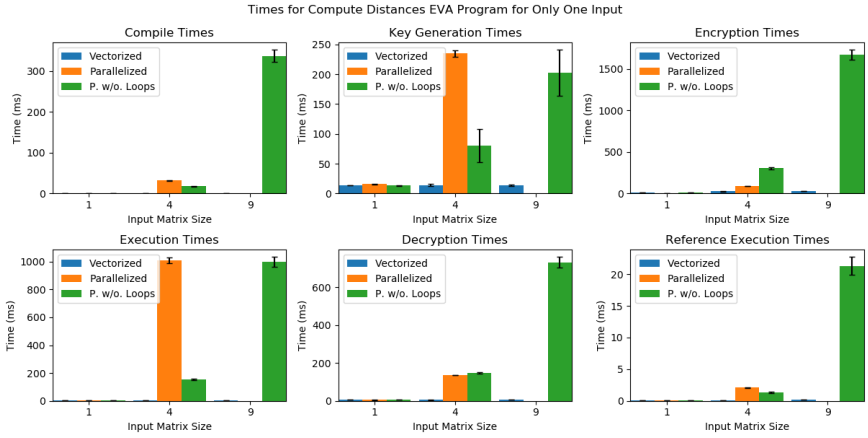
In the Figure 11 , timings and MSE scores of EVA Programs are plotted with respect to the implementations. Input matrix size and number of inputs are same for all of the three implementations. So, only one input matrix is used and input matrix sizes are 1x1, 2x2 and

3x3. Unfortunately, Parallelized Implementation cannot work with matrix sizes above 2x2. The vector size is 1 for Parallelized Implementations due to using only one input.

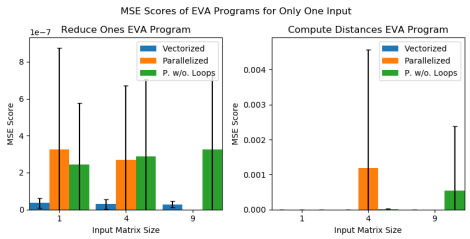
Vectorized Implementation is faster than Parallelized Implementation in almost every timings and MSE scores except for key generation times in Reduce Ones EVA Programs. In terms of Compute Distances EVA Programs, Vectorized Implementation is much more faster than the other implementations. Parallelized Implementations runs with similar timings in terms of Reduce Ones EVA Programs. Yet, Parallelized Implementation is only faster in terms of encryption and decryption time which can be explained with reduced number of input vectors to Compute Distance EVA Program. Parallelized Implementation's MSE scores are much more greater than the rest due to having greater multiplicative depth, although this cannot be seen clearly from the plots.



(a) Timings comparisons of Reduce Ones EVA Programs by implementations



(b) Timings comparisons of Compute Distances EVA Programs by implementations



(c) MSE scores comparisons of EVA Programs by implementations

Figure 11 . Timings and MSE scores comparison of EVA Programs by implementations

As mentioned in Subsection 4.1.4, EVA program can be called more than once depending on the implementation. Therefore, total running times of the EVA programs should also be analyzed. Reduce Ones EVA Programs are called only once for every implementation but only Compute Distances EVA Program of Parallelized Implementation is run once whereas the rest of the implementations are called $\lceil n^2/2 \rceil . n^2$ times.

In the Figure 12 , total running times of each implementations with only one input are shown. By looking the figure, Vectorized Implementation seems faster than Parallelized Implementations if each implementation is run with only one input (i.e. vector size is 1 for Parallelized Implementations). Even though Reduce Ones EVA Programs are called only once, Parallelized Implementations seem to run in $O(n)$ instead of $O(1)$ which is similar to Vectorized Implementation runtime behaviour. Compute Distances EVA Programs (except Parallelized Implementation) seem to have exponential runtime with respect to input size. This or the opposite cannot be said about Parallelized Implementation's runtime due to lack of data. Also note that Compute Distances EVA Programs can be finished in several hours as the input matrix size is increased and Parallelized Implementation without Loops seems to be much more slower than Vectorized Implementation.

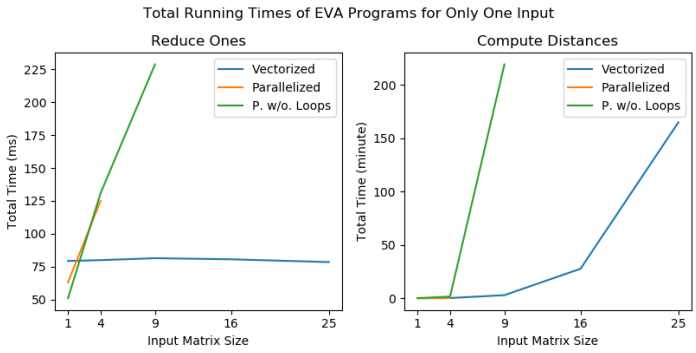
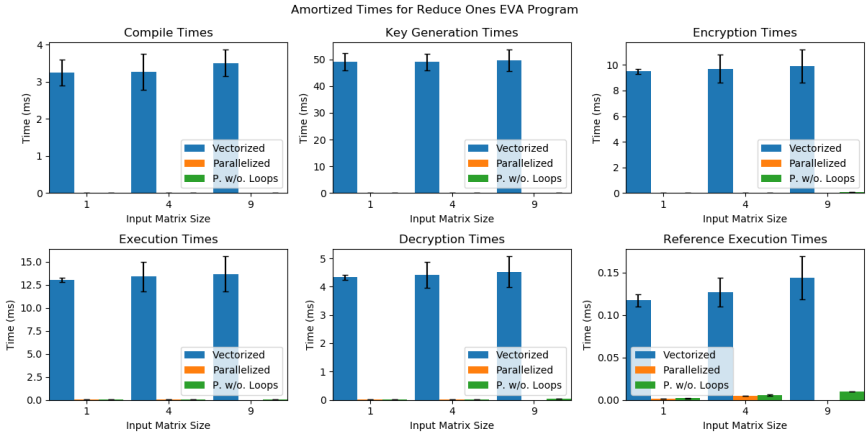


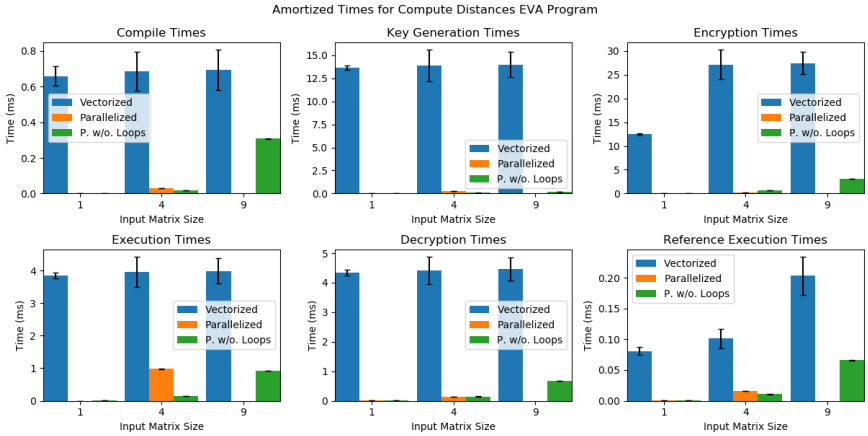
Figure 12 . Total runtimes of EVA Programs for only one input

4.2.4 Amortized Comparison of EVA Programs

In the previous subsections, every implementations' EVA Programs are compared with each other when they work on only **one input**. However, Parallelized Implementations can handle more than one inputs unlike Vectorized Implementation. Amortized comparison of each EVA Programs are displayed in Figure 13 . For the amortized analysis, total running times are divided with 1024 (the number of inputs which is maximum vector size for Parallelized Implementations in the simulation) and 1 for Vectorized Implementation. With amortized runtimes of EVA Programs in Figures 13 a and 13 b, it can be said that increasing the vector size decreases the amortized runtimes for Parallelized Implementations.



(a) Amortized timing comparisons of Reduce Ones EVA Programs by implementations



(b) Amortized timings comparisons of Compute Distances EVA Programs by implementations

Figure 13 . Amortized timings of EVA Programs by implementations

If total runtimes of each implementations are considered as in Figure 14 , Vectorized Implementation is slower than the Parallelized Implementations. However, further experiments/simulations with larger input matrix sizes are required to decide this for larger input matrix sizes. Also, further experiments/simulations should be conducted with Vectorized Implementation where it can work on multiple input data.

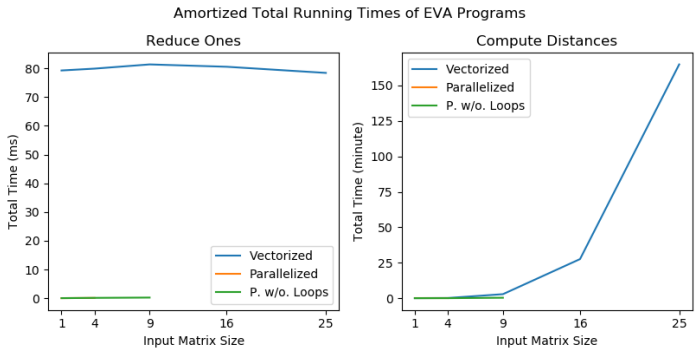


Figure 14 . Amortized total runtimes of EVA Programs by implementations

4.3 Discussion

Even though all of the implementation can solve the problem, Parallelized Implementation couldn't work on input matrices with sizes greater than 2x2 due to insufficient bit modulus. The reason that Parallelized Implementation cannot work on the bigger matrices is that the implementation has more multiplications than the other implementations' and this causes the insufficient bit modulus. For bigger matrices, Parallelized Implementation without Loops will also suffer from this problem. However, Vectorized Implementation can work on any sizes as long as the inputs are fitted into the input vectors (There may be vector size limitations from EVA compiler or SEAL library). The insufficient bit modulus problem can be solved by decreasing multiplication in the EVA Program as in the Parallelized Implementation without Loops where Compute Distances EVA Program is called more than once to obtain the same result of the Parallelized Implementation.

In all of the implementations, the results of the EVA Programs are first rounded to the closest integers before passing them to the next step. This is done in order to reduce accumulated error of CKKS approximation. For example, a cell value in distances matrix could be approximated to 0.99 (which is possible as there exist MSE scores above $(1-0.99)^2 = 0.0001$ in Figures 8 and 10) instead of 1 and if it calculated with similar error $\lceil n^2/2 \rceil \, n^2$ times, then it will be equal to 0.63 for $n = 3$ and 0.27 for $n = 4$.

There could be few optimizations in the implementations:

1. Compute Distances EVA Programs of Vectorized Implementation and Parallelized Implementation without Loops are employed $\lceil n^2/2 \rceil \, n^2$ times. This increases the total running time greatly. So, the client could reduce the number of program calls by checking the input matrix, and calling Compute Distances EVA Program for only 1-valued cells. Using the reduced input matrix will also decrease the number of EVA Program calls as it reduces the number of 1s in the input matrix.
2. The client can check consecutive distances matrices and decides to stop earlier. For example, if every 1-valued cells can reach to every other 1-valued cells, the client can stop the iteration earlier as the client has the information enough to continue solving the problem correctly.
3. Another optimization regarding the reducing the input vector sizes that the client could do by reducing the size of the adjacency matrix by removing 0-valued cells' rows and columns in the adjacency and distances matrix. This will also result in less computations (especially multiplication).

Although those optimizations can reduce runtimes of the implementations, it would be pretty hard to implement on cases where working on multiple input matrices due to constrained computation capabilities of EVA Programs. EVA Programs are running in parallel for each vector element/s. So, it is impossible to do different calculation depending on the vector element index in EVA Program.

If the solution requires working on only one input, Vectorized Implementation is the fastest. Otherwise, Parallelized Implementations seem to run faster in terms of amortized analysis (where 1024 inputs are used by Parallelized Implementations whereas Vectorized Implementation uses 1 input). However, Vectorized Implementation can be improved to work on multiple input data and it may change the situation. My personal opinion is that improved Vectorized Implementation would be much more faster than the Parallelized Implementations. Unfortunately, I didn't think of improving Vectorized Implementation in this way while I was doing the project. Also, Vectorized Implementation has not multiplication limitations

as in Parallelized Implementation because it has simpler EVA Programs compared to other implementations' programs.

Finding the connected islands problem can be solved using FHE. However, it is important to keep in mind that running times of implementations can be more than hours depending on the input matrix size due to computation limitations and overhead come with FHE.

5 Conclusion

This term project report presents FHE solution to the problem, finding the number of islands. FHE friendly approach to solve the algorithm is explained in details, and then, details of FHE solution implemented with EVA and SEAL are explained. Different implementations are compared with each other in terms of compile, key generation, encryption, execution, decryption and reference execution times, total running times and approximation rates (by MSE scores). There are few optimizations to the implementations which could result in the running time reduction. With optimizations and improvement of the Vectorized Implementation where it can process more than one input data could be the extension of this project. Also, further experiments should be done with larger input matrix and vector sizes.

References

[1] K. Laine, ''Simple encrypted arithmetic library 2.3. 1,'' *Microsoft Research https://www. microsoft. com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1. pdf*, 2017.

[2] F. Armknecht, C. Boyd, C. Carr, K. Gjøsteen, A. Jäschke, C. A. Reuter, and M. Strand, ''A guide to fully homomorphic encryption,'' Cryptology ePrint Archive, Paper 2015/1192, 2015. [Online]. Available: https://eprint.iacr.org/2015/1192

[3] R. L. Rivest, L. Adleman, M. L. Dertouzos *et al.*, ''On data banks and privacy homomorphisms,'' *Foundations of secure computation*, vol. 4, no. 11, pp. 169--180, 1978.

[4] C. Gentry, ''Fully homomorphic encryption using ideal lattices,'' in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169--178.

[5] D. Dolev, C. Dwork, and M. Naor, ''Nonmalleable cryptography,'' *SIAM Journal on Computing*, vol. 30, no. 2, pp. 391--437, 2000. [Online]. Available: https://doi.org/10.1137/S0097539795291562

[6] ''Microsoft SEAL (release 4.0),'' https://github.com/Microsoft/SEAL, Mar. 2022, microsoft Research, Redmond, WA.

[7] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, ''(leveled) fully homomorphic encryption without bootstrapping,'' *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1--36, 2014.

[8] J. Fan and F. Vercauteren, ''Somewhat practical fully homomorphic encryption,'' Cryptology ePrint Archive, Paper 2012/144, 2012, https://eprint.iacr.org/2012/144. [Online]. Available: https://eprint.iacr.org/2012/144

[9] J. H. Cheon, A. Kim, M. Kim, and Y. Song, ''Homomorphic encryption for arithmetic of approximate numbers,'' in *Advances in Cryptology -- ASIACRYPT 2017*, T. Takagi and T. Peyrin, Eds. Cham: Springer International Publishing, 2017, pp. 409--437.

[10] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, ''Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation,'' in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 546–561. [Online]. Available: https://doi.org/10.1145/3385412.3386023