Parallel Algorithms for Accelerating Homomorphic Evaluation

Abstract—Homomorphic encryption is an encryption scheme that allows computations (addition, multiplication) on ciphertext, without decrypting it first. Many algorithms and applications make use of multiple homomorphic evaluations to achieve privacy-preserving computations. However, these evaluations are computationally intensive due to the underlying operations on ciphertext. Carrying out these evaluations sequentially dramatically increases the overall time. For example, in an ongoing project, the evaluation of a random forest is done sequentially. But, each decision tree can be evaluated separately using parallelization techniques, especially in the cloud computing setting. Also, each decision tree contains comparison between two encrypted numbers that can be evaluated concurrently. The comparison on two encrypted numbers has been done using a protocol, namely, secure comparison protocol which is also done sequentially. In this capstone project, we have been looking into techniques, mechanisms, and algorithms that can speed up the evaluation of this protocol.

I. INTRODUCTION AND MOTIVATION

Comparing two numbers is one of the building blocks of many computations, such as, in a decision tree, each node performs this comparison. To perform any operation on a decision tree securely using homomorphic encryption and especially in a practical time, it is important to optimize and improve various secure comparison protocols. In this capstone project, we look into one of the newly proposed secure comparison protocol in which comparing two numbers results in a single bit with a low round complexity. Then, we explain the reasons why computing this algorithm sequentially is not optimal. We also look into the characteristics of the secure comparison circuit. Leveraging these characteristics, two new approaches have been proposed to evaluate this protocol using parallel computing. In the first approach, we try to maximize the reusable computations while eradicating any duplicated computations by using various caching mechanisms. Although, we see significant speed up using Approach 1 from the sequential approach, it doesn’t seem to be good enough. So, we then propose Approach 2 in which we try to minimize the multiplicative depth of the circuit with the help of pairing of computations. We see how the multiplicative depth has a significant impact on the running time of the algorithm. Once we have understood the working of the both the approaches, we analyze the running time of both the approaches for different bit-length input. We also analyze the speed up and efficiency of both the approaches. Finally, we conclude the paper by stating various insights that we have learned as well as various directions this work can be taken forward.

\[
\begin{array}{c|c|c|c}
 x & y & x < y \text{ (} \neg \text{ AND} \ y \text{)} & x = y \text{ (} \times \text{ XNOR} \ y \text{)} \\
0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 \\
\end{array}
\]

Fig. 1. Truth table for LESS THAN and EQUAL TO operations

II. SECURE COMPARISON PROTOCOL

The ability to compare two encrypted numbers without the evaluator knowing the actual numbers, has a lot of applications, for example, private evaluation of a decision tree. Each decision node in a decision tree evaluation needs to compute a boolean operation. There are many secure comparison protocols that can compare two \( \ell \)-bit inputs. Some of them are Veugen [1], [2], Damgård-Geisler-Krøigaard (DGK) [3]. But some of these protocols generate a resultant vector as the result and then need to check if one of the bits in the resultant vector is zero, whereas some others do give a single bit output, but with high communication overheads. In this project, we have been looking into a secure comparison protocol that not only produces a single bit output but also uses low round complexity which is very important while outsourcing a work [4].

A. Theorem and arithmetic circuit

Theorem 1: Given two integers in their binary representation \( x = \{x_{\ell-1}, x_{\ell-2}, \ldots, x_0\} \) and \( y = \{y_{\ell-1}, y_{\ell-2}, \ldots, y_0\} \), \( b = 1(x < y) \) if and only if there exists an index \( j \), where \( 0 \leq j < \ell \), such that \( x_j < y_j \) and all leading bits are equal \( x_i = y_i \), \( i > j \).

This protocol can be represented as an arithmetic circuit as follows:

\[
b = \begin{cases} 
(x_{\ell-1} < y_{\ell-1}) \lor \\
(x_{\ell-1} = y_{\ell-1}) \land (x_{\ell-2} < y_{\ell-2}) \lor \\
\vdots \\
(x_{\ell-1} = y_{\ell-1}) \land \cdots \land (x_1 = y_1) \land (x_0 < y_0)
\end{cases}
\]

where \( x_j < y_j \equiv (\neg x_j \land y_j) \equiv (1 - x_j)y_j \) and \( x_j = y_j \equiv (x_j \oplus y_j + 1) \equiv x_j + y_j + 1 \). The terms compare the two bits \( x_u \) and \( y_u \). In each of the terms, the AND gates ensure that the term is true if and only if all operands are true, i.e. the current x-bit is less than the y-bit and all the prior bits are equal. On the other hand, the OR gates between all the terms form a way to know if any of the term is true [4]. The result of this
Fig. 2. Secure comparison circuit for 4-bit input

Fig. 3. Running time of basic homomorphic operations

protocol is a single encrypted boolean value. The result is 1 if there exists a term where all the operands are true and the term result is 1. Otherwise, the result is 0 [4]. For example, let’s say we have two numbers 12 and 13 which can be represented in their binary form as 1100 and 1101 respectively. By using the above theorem, in the fourth term we would have all the previous bits equal between the two numbers and the current bit of 12 ($x$) less than the current bit of 13 ($y$). Hence, we can confirm that the first number is less than the second number.

B. Sequential Evaluation approach

\begin{algorithm}
\caption{Sequential Evaluation approach}
\begin{algorithmic}
\Procedure{SEQUENTIAL\_APPROACH}{\textbf{for} \(i = \text{bit-length-1} \) \textbf{to} \(0 \) \textbf{do}:
\For{\textbf{for} \(j = \text{bit-length-1} \) \textbf{to} \(i \) \textbf{do}:
\If{\(i==j\)}
\If{\(\text{work not in cache}\)}
\text{lessThan}(x_{i},y_{j});
\text{store result};
\Else
\If{\(\text{work not in cache}\)}
\text{equalTo}(x_{i},y_{j});
\text{store result};
\EndIf
\EndIf
\EndIf
\EndFor
\EndFor
\EndProcedure
\end{algorithmic}
\end{algorithm}

The above protocol does have a naive solution, that is evaluating it sequentially. Fig. 2 is an example of the secure comparison protocol for a 4-bit input. This circuit is responsible for comparing if a given encrypted number is less than the other encrypted number or not. As we can see in Fig. 3 that in homomorphic evaluations, the AND operation takes significant time as it involves multiplication. In homomorphic encryption, to hide the data, noise is added to it. If we perform operations like multiplication, it multiplies the noise as well. At the end, the noise is canceled out using mod operations, so it is important to keep the noise in check. When the noise grows to a great extent, to the point that it would be impossible to recover the plain text back, we perform operations like bootstrapping to reduce the noise. So, our major focus was to reduce the number of times AND operations are applied. Since, this is a sequential algorithm, the best we could do would be not to repeat the computations that we have already performed. As we can see in the circuit at level 2, we have $x_3 = y_3$ operation which is being used again in level 1. Also, we can see that at level 1 we have $x_3 = y_3 \text{ AND } x_2 = y_2$ which is again used at level 0. So, we see a pattern here that all the AND operations except the last one in any level are again reused in the level below. Doing these computations again, would just increase the running time unnecessarily. So we decided to use a caching mechanism to store the previous values that would of be used in the level below. In this way, we did experience a significant speed up. Although we did use caching, this did not reduce the multiplicative depth in any way. The number of consecutive multiplications did remain the same. Due to the high number of consecutive multiplications, we need to keep higher parameters for the ciphertext, in turn increasing the ciphertext size to a great extent. The increased size of the ciphertext negatively affects the running time of the algorithm.

C. Characteristics of the circuit

To develop a good algorithm for the circuit, it is important to understand the characteristics of the circuit. This will give us deeper insights about the circuit, and help us in understanding the different points which we should be avoiding and the points which we should definitely take into consideration while developing the algorithm.

1) Predictable structure of the circuit: The first and the most important characteristic of the circuit is it’s predictable and definite nature. For a given bit-length only the bit values change but the circuit remains the same as seen in Fig.5. Due to this property of the circuit, while developing the algorithm we can take two things into consideration. Firstly, we could break down the circuit into different parts, as the circuit structure doesn’t build dynamically while evaluating the circuit. Secondly, we can store the circuits of different bit-length in data structures and then reuse them for different inputs. This would help us eradicate the time for building the data structures to evaluate the circuit. Also, during run time of some other functionality, if memory allotment becomes an issue, we could always delete the data structures to free up memory and then build them again whenever required.

2) Complexity increases with the bit-length: As the bit-length of the input increases, the complexity of the evaluation also increases. We could notice this behavior when we analyze the multiplicative depth for different bit-length. If the bit-
length increases by a factor of 2, the multiplicative depth also increases approximately by factor of 2. Such an increase in the multiplicative depth results in an increase in the ciphertext parameters as well, results in larger ciphertext which increases the overall complexity of the circuit.

3) Repeated computations: As we discussed in our sequential approach, that a lot of the computations from one level are again repeated in the level below. This leads to two things. Firstly, it adds a dependency in the circuit, that is, we either perform the computations multiple times, or we wait for the computations being performed to get completed, so that we can again reuse it in the level below. In all our approaches, we have avoided any repeated computations and focused on computations that can be carried out based on the results we have so far.

4) Independent operations: Having said that there are a lot of repeated computations, all the operations in the circuit are independent from the other operations. This means that we could have different threads performing different operations at the same time without interacting with each other in any way.

III. PROPOSED SOLUTIONS

Now that we have analyzed the characteristics of the circuit, we will look into the two algorithms that we developed to evaluate the comparison between two encrypted numbers using the secure comparison protocol.

A. Approach 1: Maximize reusable components

In our first approach to parallelizing the protocol evaluation, our major focus has been on maximizing the utilization of the number of cores at all time, as well as eradicating any repeated computations. We used four priority stacks to give preference to computationally intensive tasks that are constrained by data dependency. So, the tasks that are computationally intensive and are required for other tasks to finish, are given priority over others. From Fig.4, we can see how the stacks are placed. Lets look at each of the stacks in detail:

**Priority stack 1:** As the AND operations are the most time consuming operations in the circuit, we wanted to give these operations the highest priority. So if there is an AND operation that can be performed based on the values present in the cache, we go ahead and do that work, store the result in the cache and start over again from priority stack 1. For example, if the number of bits is 4 and this priority stack has an entry 2, then it actually represents 3E AND 2E. So, in short, it represents the AND operations from the highest bit till the bit that is in the stack entry.

**Priority stack 2:** Once we have the AND operations between the EQUAL TO operations, we extend those results by performing AND operation with the only LESS THAN operation in that level. For example, if the number of bits is 4 and this priority stack has an entry 2, then it actually represents 3E AND 2L.

**Priority stack 3:** Once we have the second priority stack ready, we turn our attention to the OR operations between the different levels, these operations are computationally as intensive as the AND operations, but these operations can only be performed once we have the results for each level and hence we put these operations into the third priority stack. For example, if the number of bits is 4 and this priority stack has an entry 1, then it actually represents 3L OR operation 3L or operation 3/L. This means that it should have the answer to LEVEL 3 OR operation LEVEL 2 OR operation LEVEL 1.

**Priority stack 4:** The fourth priority stack includes work for performing the LESS THAN and EQUAL TO operations. For example, if the priority stack has an entry 3L, it means that we need to perform \( x_3 < y_3 \) operation and if the priority stack has an entry 3E, it means that we need to perform \( x_3 = \) operation.

Once we have all the priority stacks filled with tasks, we start as many threads as there are cores in the system. Each thread follows a similar path. First, any thread locks priority stack 1, checks if it can perform the work at the top of the stack based on the results in the cache. If there are dependencies for which there hasn’t been a result yet in the cache, the thread moves to the next stack. The thread then checks the second priority stack, if it can perform the first work in the stack. If the thread is able to perform that task it will perform it, find out the result, store it in the cache and then again start searching from Priority stack 1. If it cannot perform the task, it moves to Priority stack 3 and then finally if it is not able
to do that work also, it moves to Priority stack 4. The threads keep following this order until all the stacks are empty. At this point, the threads come out of the loop and wait for all the other threads to stop working. At the end, we get the result that is stored in the cache with the key $3 \oplus t_0 \oplus t_1$ for a 4-bit input. The extension of this key is 3rd level till 0th level $\lor$. That means that it has the answer to $3 \lor hide operation 2 \lor hide operation 1 \lor hide operation 0$, which represents the entire circuit. In this way, we try to utilize as many resources as we have by performing the computations that can be performed at the current point of time while not performing duplicated tasks. Although we did remove all the duplicated tasks, the multiplicative depth of the approach remains the same as the sequential approach.

B. Approach 2: Minimize multiplicative depth

In our second approach, our major focus has been on reducing the multiplicative depth of the entire circuit. In Approach 2, we convert the secure comparison circuit to an evaluation tree which helps us to reduce the multiplicative depth to a great extent. The reduction in the multiplicative depth is due the pairing of the multiplications. Let’s see this by an example, let’s say we want to compute $1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8$

Now, if we do the computation as $1 \times 2$ then $result \times 3$, and then $result \times 4$ and so on, then the multiplicative depth would be 7. But let’s say we group together $1 \times 2$, $3 \times 4$, $5 \times 6$ and $7 \times 8$ in the first round and the results of the first and last two in the second round and finally, at the end we perform the final result in the third round, then we reduce the multiplicative to just 3 instead of 7. In this way, we reduce the size of the ciphertext to a great extent in turn reducing the running time of the algorithm.

Firstly, the 3,0,2,1 in Fig.5 at the root represents the levels from our previous circuit and the OR represents that this node would have the results to $3 \lor hide operation 0 \lor hide operation 2 \lor hide operation 1$, which represents the entire circuit for a 4 bit input. The reason the levels have been represented in this order is since LEVEL one has just 1 operation where as level 0 has the maximum number of operations, distributing the tree in this structure gives us a relatively balanced tree. If we would have distributed the tree as 0,1,2,3, then the left subtree would have been of relatively larger depth than the right subtree in each case. Also, the AND operation highlighted in red was on purpose kept together. The AND operation could have been between one $\text{EQUAL TO}$ and one $\text{LESS THAN}$ operation as well. But since we have already computed the $\text{AND}$ operation between the $\text{EQUAL TO}$ operation in another subtree, we could use that again, rather than breaking it and performing the computation in a different way. The first part of this algorithm is developing the tree. Once the tree has been developed, we go in the bottom up fashion. We first add in all the work we need to do into a vector of sets, so that we do not have to perform any computations twice in the same level, all the threads keep performing the task at the same level, and the results are stored in a cache. Once all the work has been completed in one level, we move the level above that. In this way, we are sure that any result that we need to perform a computation, would have already been computed by one of the threads in the levels below it. In this way, we can be sure that we are not performing any duplicated computation.

IV. RESULTS

To test the performance of the two approaches, we did perform tests on a 8-core machine. We ran all the test cases
until we got a standard deviation of less than 1%.

A. Analysis of multiplicative depth

As we have seen, the multiplicative depth affects the size of the ciphertext to a great extent, in turn affecting the running time of the evaluation. Hence, it is important to analyze the multiplicative depth for the different approaches and how they increase as the bit-length of the input increases. As we can see from the table in Fig.7, the multiplicative depth grows as the bit-length increases. But the factor at which it grows is different for the approaches we have been looking into. As we can see, in our Approach 1 the multiplicative depth remains the same as the sequential algorithm and it can be derived by the formula $3l - 2$. As we have seen while discussing Approach 2, the multiplicative depth is reduced drastically and it can be derived by the formula $\log(l) + \frac{1}{2} + l$.

B. Analysis of running time

Fig. 9 represents the running times for different bit-length of input in which time is represented in log format. From Fig. 8 and Fig. 9, it is clear that Approach 2 outperforms the sequential algorithm as well as Approach 1. The results are as we expected, as in Approach 1 we have seen that it did not improve the multiplicative depth from the sequential algorithm and also it included a strict multi-threaded management.

C. Analysis of speed up

The speed up for any parallel algorithm can be derived by the formula:

$$Speedup(N, K) = \frac{T_{seq}(N, 1)}{T_{par}(N, K)}$$

where $T_{seq}$ and $T_{par}$ are the evaluation times of the sequential and the first parallel approach, $N$ is the input size and $K$ is the number of cores. We can see the efficiency of our 2 approaches from Fig.10 and Fig.11. From the two figures, we can see that although Approach 1 does show some speed up initially, but it drops when the input bit-length is more than 8 bits. This happens since Approach 1 has a lot of critical sections for accessing the stacks which increases the sequential time of the algorithm to a great extent. On the other hand, we can see that the speed up of Approach 2 keeps increasing as the input bit-length is increased, due to the fact that the effect of the sequential part of any algorithm keeps becoming insignificant as the overall time of the algorithm increases.
D. Analysis of efficiency

The efficiency of any parallel algorithm can be derived by the formula:

\[
Efficiency(N, K) = \frac{Speedup(N, K)}{K}
\]  

(3)

where \( N \) is the input size and \( K \) is the number of cores. After analyzing the speed up, we can justify the efficiency of the two approaches based on the same reasoning. Although the ideal efficiency for any parallel algorithm is 1, we do get 0.8 efficiency for the second approach for an input of 32-bit numbers.

V. CONCLUSION AND FUTURE WORK

We have developed and reviewed two different approaches to improve the secure comparison protocol. We did analyze the reason one performed better than the other. We conclude that we can significantly improve the running time of any homomorphic evaluation by reducing the multiplicative depth which leads to smaller HE parameters, which in turn generates smaller ciphertext reducing the running time of the evaluation. In general, we can say that by understanding the characteristics of any circuit, and by careful design of various parallel algorithms, we can improve the running time of any algorithm to a great extent. Also, these two approaches can further be extended and tested in a multi-cloud setting, as well as various different ways to further reduce the multiplicative depth can be looked into.

REFERENCES


