Increasing Speed of Elliptic Curve ElGamal using Parallelization

Hokki Suwanda
School of Electrical Engineering and Informatics
Bandung Institute of Technology
Bandung
hokkyss2@gmail.com

Rinaldi Munir
School of Electrical Engineering and Informatics
Bandung Institute of Technology
Bandung
rinaldi@staff.stei.itb.ac.id

***Abstract*—To attain higher security level, public key cryptography, like RSA uses bigger key size. However, elliptic curve cryptography, like Elliptic Curve ElGamal, uses much smaller key size compared to RSA. However, the key size remains big, which makes encrypting and decrypting messages takes more time. A way to increase the speed of an algorithm is parallelization. The solutions in this paper are implemented in three variations. They are distributed-memory parallel program, shared-memory parallel program, or hybrid parallel program. Each program will output its encryption and decryption duration in seconds. These times will be used to calculate throughput, the number of characters processed each second. Throughput is further used for calculating speedup. The experiment determines whether the solution is successful in increasing speed. From the result, it was found that all solutions are capable of increasing algorithm speed, with shared-memory parallel program being the best, followed by distributed-memory parallel program and hybrid parallel program.**

Keywords—cryptography; distributed-memory; shared-memory; hybrid; speed

I. Introduction

Cryptography is a method to secure a message. An example of cryptography is the encryption and decryption of messages. Based on the key used, cryptography can be classified into symmetric key cryptography and asymmetric key cryptography. Asymmetric key cryptography has more complex computations which makes it more secure than symmetric key cryptography. However, the complexity backfired, making its performance, especially speed, worse than symmetric key cryptography.

To attain higher security level, asymmetric key cryptography needs to be using big numbers as keys. To solve this issue, elliptic curve cryptography is invented. Using elliptic curve as its base, elliptic curve cryptography significantly reduces the size of the key used in the algorithm. According to [1], 256-bit elliptic curve cryptography is equivalent to 1620-bit RSA. Both key sizes are big integers.

From 1986 to 2002, processor performance increased by 50% in average every year. However, as of 2002, the increase in performance decreased to just 20% in average every year. Performance can be increased by using processors with higher speed, in other words, upgrading to faster processors. However, the performance gain is insignificant compared to the price of the processor. This was the case in using only one processor [2].

However, processor trend began to shift in 2005. The manufacturers developed and integrated multiple processors in a single integrated circuit [2], called CPU. This gives more variations of approaches in increasing speed. Instead of upgrading processors, people can now use more processors to increase speed by running multiple instances of a program at one time.

Requirements for program to be able to utilize multiple processors are emerging. This is due to serial programs being unaffected. Serial programs are unaffected because they use only one processor. Running the program multiple times will just run multiple instances of the same program, with the same instructions and data. Parallel program is defined as a group of subprograms that collaborate with each other and running simultaneously, where it can utilize multiprocessors.

The speed of Elliptic Curve ElGamal (ECEG), an asymmetric key cryptography algorithm, can be potentially increased by running it in parallel. There are some researches specializing in elliptic curve, ElGamal, and parallel programs.

[3] focuses on reducing the number of operations in elliptic curve scalar multiplication. It does not parallelize the encryption and decryption phase of the algorithm, where both are essential in ECEG. There are some limitations and conditions for using the algorithm the paper proposed.

[4] has the same focus as [3]. In fact, it adds another alternative to the latter by changing coordinate system from one to another and vice versa. However, it retains properties from [3], with the addition of complexity from changing coordinate systems.

[5] was very similar to this paper. It parallelized ElGamal algorithm using CUDA, a platform for GPU programming. It utilizes the fact that GPU has higher computational capabilities than a CPU. However, CUDA is very limited as it is only available on GPUs developed by Nvidia.

II. Literature Study

A. Elliptic Curve ElGamal

Elliptic Curve ElGamal is a public key cryptography algorithm. It consists of a private key and public key. The private key in Elliptic Curve ElGamal is an integer g. The public key consists of three components E_p, α, and β. E_p is an elliptic curve on $\mathbb{Z}_p$. α is a point on E_p, and β is the result of scalar multiplication between g and α. Integer p is a prime number [6] [7].

Private key is used to decrypt a message, while public key is used to encrypt a message. Encrypting is a process of transforming readable message to an unreadable message, called ciphertext. Decrypting is the opposite, it transforms an unreadable message to readable message, called plaintext [6] [7].

To encrypt a message x, a point in E_p, the ciphertext tuple (y_1, y_2) is calculated using (1) and (2). The ciphertext can be decrypted into plaintext by using (3) [6] [7].

$$y_1 = k\alpha \tag{1}$$

$$y_2 = x + k\beta \tag{2}$$

$$x = y_2 - gy_1 \tag{3}$$

B. Parallel Programming

In a parallel system, a parallel program is defined as a group of subprograms that collaborate closely with each other and running simultaneously to solve a problem. An important aspect in parallel programming is load balancing, synchronization, and communication [2].

Load balancing involves dividing the work among instances in a way that minimizes the amount of communication between instances and every instance gets roughly the same amount of work. Tasks and data must be divided evenly into each program instance. This is done to minimize idle time of processors, as a parallel program is meant to utilize multiple processors at the same time [2].

Synchronization involves making all processors start the next instruction together because each processor works at its own pace. In distributed-memory programs, synchronization is implicitly conducted by communication. In contrast, in shared-memory programs, communication is done by synchronizing threads [2].

There are many models in writing parallel programs. They are the data-parallel model, task graph model, work pool model, master-slave model, pipeline model, producer-comsumer model, and hybrid model. Hybrid model combines multiple models [8].

Some metrics have been defined to measure speed for parallel programs. Some of these metrics are execution time and speedup. Execution time denotes the time that elapses from the start of a parallel computation until the last processed element. Speedup represents the speed gain achieved by parallelizing a program [8].

C. Distributed-Memory Parallel Programming

Distributed-memory programming is one way to parallelize a serial program. Distributed-memory parallel program splits either tasks or data to processes. Each process is isolated from another process, so they cannot access each other's memory stacks. To communicate with each other, message-passing interface (MPI) is used [2] [8].

In MPI, if the number of processes is p, each process is ranked from 0 to $p - 1$. The rank of each process corresponds to its number. A process can communicate by sending and receiving messages. Other than sending and receiving messages, processes can also communicate by broadcasting messages to all processes [2] [8].

An implementation of MPI is OpenMPI, a library for C and Fortran programming language. In OpenMPI, sending and receiving messages are done by calling `MPI_Send` and `MPI_Recv` function respectively. Other than calling MPI_Send and MPI_Recv to individually send and receive messages, MPI_Broadcast and MPI_Gather can also be used. [2] [8].

D. Shared-Memory Parallel Programming

Shared-Memory programming is a way to parallelize a serial program. Shared-memory parallel program splits tasks or data to threads, often called a mini-process. Each threads share memory, thus able to access each other's memory. In shared-memory parallel programs, threads communicate by reading values from and writing values to memory addresses [2].

Each thread, like process, is numbered from 0 to $p - 1$ if there are p threads. Each thread can read and write values to a memory address. It means that a memory address could be written by multiple processes at the same time. This memory address is called critical section. Critical sections must be handled very carefully by programmers to avoid race condition. Race condition can cause an undesired behavior in a parallel program [2]. Proposed Method

The method proposed by [3] and [4] to parallelize an elliptic curve scalar multiplication has limitations on the points that can be used. Moreover, there is also a need to transform one coordinate system to another. The most important reason is that it is not used for cryptographic process.

Solution proposed by [5] parallelize ElGamal Algorithm in GPU using CUDA. Although similar, Elliptic Curve ElGamal has different domain than normal ElGamal Algorithm. Not only that, CUDA is limited to GPUs developed by Nvidia. Which means that it is limited to some devices.

III. Proposed Solution

To parallelize Elliptic Curve ElGamal in CPU, three solutions are proposed, distributed-memory parallel program, shared-memory parallel program, and hybrid parallel program. The solutions will be called *Parallel One*, *Parallel Two*, and *Parallel Three* respectively. Each solution is implemented in C++ programming language.

A. Parallel One

This alternative uses distributed-memory programming, thus uses OpenMPI. In this alternative, processes interact by calling `MPI_Send` and `MPI_Recv`, two functions provided in OpenMPI. Step by step of this alternative is, as follows:

1. Process 0 receives input of message, public key, and private key.
2. Process 0 balances the load of every process by dividing messages to each process equally.
3. Each process, for each character they received, encodes the character, encrypts the character, decrypts the ciphertext, decodes the plaintext, and asserts the equality of the plaintext and the processed character.
4. Each process calculates execution time encryption and decryption, encoding and decoding characters included.
5. Every process except process 0 sends their execution time to process 0.
6. Process 0 will output the maximum execution time of all processes for encryption and decryption.

B. Parallel Two

This alternative uses shared-memory programming and OpenMP, a directive-based shared-memory Application Programming Interface (API). Step by step of this alternative is as follows:

1. The main program receives a message, public key, private key, and the number of threads as input. Message, public key, and private key will be shared to all threads.
2. The main program will fork several threads. The number of forked threads will be taken from the input.
3. The main program divides the message to all forked threads equally.
4. Each thread, for each character they received, encodes the character, encrypts the character, decrypts the ciphertext, decodes the plaintext, and asserts the equality of the plaintext and the processed character.
5. Each thread will change the value of maximum and minimum execution time, which is a variable shared to all threads.
6. The main program will output the maximum and minimum execution time for both encryption and decryption.

C. Hybrid Parallel Program

This alternative combines the use of distributed-memory programming and shared-memory programming. In this alternative, distributed-memory programming is first used to split the main program into instances of process. Then, shared-memory programming is used to fork several threads inside each process. Simply put, this alternative runs Parallel Two for every process. The step by step of this alternative is:

1. Process 0 receives input of message, public key, private key, and number of threads.
2. Process 0 balances the load of every process by dividing messages to each process equally.
3. Each process forks several threads and further divides the message partition they received.
4. Each thread, for each character they received, encodes the character, encrypts the character, decrypts the ciphertext, decodes the plaintext, and asserts the equality of the plaintext and the processed character.
5. Each thread will change the value of maximum and minimum execution time, which is a variable shared to all threads in every process.
6. Every process except process 0 sends their execution time to process 0.
7. Process 0 will output the maximum execution time of all processes for encryption and decryption.

IV. Experiment

A. Environment

Experiment is done on a hardware with hardware specification software versions as shown in Table I and Table II. OpenMPI is an implementation of message-passing interface written for C, C++, and Fortran programming language. OpenMP is an API for writing shared-memory parallel programs in C and C++. Boost is a library for C++ that wraps and adds to C++ standard libraries and data structures, including MPI, multiprecision data types, and data serialization.

B. Metrics

There are three metrics used for experiments in this paper, they are execution time, throughput, and speedup. Execution time is the duration in which a program is running, represented in seconds. Throughput is the number of characters processed every second, represented in characters per second. Speedup is the ratio between throughput of parallel program and throughput of serial program.

The output of both serial and parallel programs is execution time. Execution time will then be used to calculate throughput by using the message length in a test case. For parallel programs, speedup will then be calculated by comparing its throughput to serial program’s throughput.

TABLE I. HARDWARE SPECIFICATION

Processor	Intel® Core™ i3-2370M DualCore CPU @2.40GHz
RAM	16 GB

TABLE II. SOFTWARE VERSION

Operating System	Debian GNU/Linux 11 (bullseye)
C++ Programming Language	14
GNU g++ Compiler	(Debian 10.2.1-6) 10.2.1.20210110
OpenMPI	4.1.0
OpenMP	4.5
Boost	1.80.0
Boost Serialization	1.32.0
Boost MPI	1.35.0
Boost Multiprecision	1.53.0

C. Test Cases

There are several variables for experiments in this paper. The variables are used to determine the effect of the variable to the result. There are three different kinds of variables, they are message length, number of threads, and key size. There is no difference in the number of processes due to the limitation of testing environment.

There are five different values for message length. The values are 1024 characters, 4096 characters, 16384 characters, 32768 characters, and 65536 characters. There are also five possible values for the number of threads. They are 2 threads, 4 threads, 10 threads, 100 threads, and 1000 threads. There are two key sizes, 64 bits and 256 bits.

For each variation, every solution will be run for 10 times. Each solution is designed to output its execution time for both encryption and decryption. So, there will be 10 pairs of encryption and decryption duration. Average of the 10 will be used to calculate throughput. The throughput of each solution will then be compared to the throughput of the serial program to measure the speed increase, speedup.

D. Result and Analysis

Table II to Table XI show that all speedup is greater than 1. This proves that the parallelization is successful in increasing algorithm speed. Increasing the number of threads can increase and decrease speed, it depends on the size of the data. The decrease in speed can be seen from Table II and Table XI, after 1000 threads.

TABLE III. SPEEDUP FOR 64-BIT PARALLEL ONE

Message length	Encryption (2 processes)	Decryption (2 processes)
1024	1.6752	1.2210
4096	1.6629	1.4159
16384	1.7368	1.6341
32768	1.7011	2.0970
65536	1.7059	2.2301

TABLE IV. SPEEDUP FOR 256-BIT PARALLEL ONE

Message length	Encryption (2 processes)	Decryption (2 processes)
1024	1.6900	1.5705
4096	1.6317	1.6301
16384	1.6638	1.4130
32768	1.6847	1.3990
65536	1.6715	1.3990

TABLE V. SPEEDUP FOR ENCRYPTING WITH 64-BIT PARALLEL TWO

Message length	2 threads	4 threads	10 threads	100 threads	1000 threads
1024	1.6833	2.1095	2.0672	2.1762	2.7749
4096	1.6963	1.9706	2.0375	2.2560	2.6232
16384	1.7487	2.0698	2.1872	2.2780	2.4780
32768	1.7284	1.9855	2.1885	2.2320	2.3463
65536	1.7215	1.9239	2.1733	2.1990	2.2918

TABLE VI. SPEEDUP FOR ENCRYPTING WITH 256-BIT PARALLEL TWO

Message length	2 threads	4 threads	10 threads	100 threads	1000 threads
1024	1.6831	1.8006	2.1160	2.1613	2.1320
4096	1.6378	1.8378	2.1780	2.1672	2.0868
16384	1.6990	1.9184	2.1799	2.1934	2.1531
32768	1.7216	1.7465	2.0563	2.1773	2.1695
65536	1.7042	1.8895	2.1532	2.1586	2.0105

TABLE VII. SPEEDUP FOR DECRYPTING WITH 64-BIT PARALLEL TWO

Message length	2 threads	4 threads	10 threads	100 threads	1000 threads
1024	1.9306	2.4428	2.6144	6.3157	1.4491
4096	1.7493	2.0535	2.3924	4.6746	3.8890
16384	1.7410	2.0971	2.2610	3.5554	12.7803
32768	2.1844	2.5397	2.7497	4.0328	14.7390
65536	2.3208	2.6139	3.0090	3.6217	13.0621

TABLE VIII. SPEEDUP FOR DECRYPTING WITH 256-BIT PARALLEL TWO

Message length	2 threads	4 threads	10 threads	100 threads	1000 threads
1024	1.6256	1.7278	2.1425	3.9573	6.7452
4096	1.7179	1.9123	2.3577	2.9613	13.9507
16384	1.6086	1.8246	2.0889	2.3706	5.2421
32768	1.5940	1.6053	1.9084	2.1396	3.7131
65536	1.6875	1.8881	2.1600	2.2881	3.2143

TABLE IX. SPEEDUP FOR ENCRYPTING WITH 64-BIT PARALLEL THREE

Message length	2 processes/ 2 threads	2 processes/ 4 threads	2 processes/ 10 threads	2 processes/ 100 threads	2 threads/ 1000 threads
1024	1.6450	1.6431	1.6245	1.7301	2.1031
4096	1.6024	1.5962	1.6126	1.6832	1.9471
16384	1.6108	1.6234	1.6250	1.6540	1.8801
32768	1.6045	1.6130	1.6026	1.6341	1.8158
65536	1.6077	1.6016	1.6101	1.6093	1.6914

TABLE X. SPEEDUP FOR ENCRYPTING WITH 256-BIT PARALLEL THREE

Message length	2 processes/ 2 threads	2 processes/ 4 threads	2 processes/ 10 threads	2 processes/ 100 threads	2 threads/ 1000 threads
1024	1.5989	1.5923	1.5955	1.5917	1.5048
4096	1.5949	1.5738	1.5902	1.5896	1.5604
16384	1.5992	1.6033	1.6005	1.5837	1.6056
32768	1.5858	1.5804	1.5879	1.5796	1.5700
65536	1.5885	1.6005	1.5955	1.5829	1.5906

TABLE XI. SPEEDUP FOR DECRYPTING WITH 64-BIT PARALLEL THREE

Message length	2 processes/ 2 threads	2 processes/ 4 threads	2 processes/ 10 threads	2 processes/ 100 threads	2 threads/ 1000 threads
1024	1.7228	2.1411	2.5355	3.9521	1.5520
4096	1.5183	1.5709	1.7076	4.0299	4.9646
16384	1.2582	1.3676	1.4332	2.5032	12.081
32768	1.5832	1.6242	1.6371	2.7045	18.254
65536	1.6970	1.6880	1.7835	2.1803	14.043

TABLE XII. SPEEDUP FOR ENCRYPTING WITH 256-BIT PARALLEL THREE

Message length	2 processes/ 2 threads	2 processes/ 4 threads	2 processes/ 10 threads	2 processes/ 100 threads	2 threads/ 1000 threads
1024	1.6091	1.5480	1.6652	3.3497	5.2059
4096	1.6442	1.6750	1.6943	2.3876	11.170
16384	1.4198	1.4667	1.4640	1.7184	4.8313
32768	1.4590	1.4771	1.4885	1.6360	3.5379
65536	1.4662	1.4714	1.4808	1.5554	2.4797

The speed decrease happens mostly on smaller data. For smaller data, processing the data takes less time. However, with more threads or processes, more time will be needed for data transmission, which contributes to overhead time. Smaller data with a lot of threads or processes will cause the overhead time to contribute to execution time.

Increasing the length of message does not affect speedup. This is due to both the serial program and parallel programs having unchanged throughput. This, however, only applies to cases when messages are divided evenly among the threads or processes. Using 1000 threads in 64-bit ECEG Parallel 2 is an example, as it is not divided evenly.

[5] shows higher speedup than this paper. It uses GPU in parallelizing an algorithm. GPU has higher computational capabilities than CPU. It means that GPU is so much faster than CPU.

V. CONCLUSION AND FUTURE WORKS

A. Conclusion

In this paper, three alternatives of parallel Elliptic Curve ElGamal algorithm on CPU are proposed to increase the speed of the algorithm each named Parallel One, Parallel Two, and Parallel Three respectively. Parallel One makes use of distributed memory parallel programming using OpenMPI. Parallel Two makes use of shared memory parallel programming with OpenMP. Parallel Three combines Parallel One and Parallel Two, using both OpenMPI and OpenMP. All alternatives successfully increased the speed of Elliptic Curve ElGamal. However, parallelization does not always increase the speed of an algorithm, as it depends on the data being processed by the algorithm and the degree of parallelization.

B. Future Works

There are some improvements that could be made. Parallelization can be done on GPU instead of CPU as it has higher computational capabilities than the latter. One goal of parallelization is making use of CPU. So, another metric, CPU utilization can be used. It can be used to identify whether the CPU is used as much as possible when running a parallel program. If the CPU utilization is low, then parallelization can be deemed a failure.

ACKNOWLEDGEMENTS

The writer would like to thank the God for His blessings while completing this research. The writer hoped that this research can help increase the speed of ECEG and help future works related to this research. While researching, the writer received help from many people. The writer would like to thank the writer's family for motivating the writer. The writer would like to thank Mr. Dr. Ir. Rinaldi Munir, M.T. for lecturing the writer regarding cryptography and this research. The writer would like to thank all the writer's friends for their help and support while doing this research.

REFERENCES

[1] S. Burnett and S. Paine, RSA Security's Official Guide to CRYPTOGRAPHY, Osborne: McGraw-Hill, 2001.

[2] P. Pacheco, An Introduction to Parallel Programming, Burlington: Elsevier, 2011.

[3] C. Negre and J.-M. Robert, "New Parallel Approaches for Scalar Multiplication in Elliptic Curve over Fields of Small Characteristics," *IEEE Transactions on Computers,* pp. 2875-2890, 2015.

[4] X. Li, W. Yu and B. Li, "Parallel and Regular Algorithm of Elliptic Curve Scalar Multiplications over Binary Fields," *Security and Communication Networks,* p. 10 pages, 2020.

[5] H. O. Purba, Peningkatan Kinerja Algoritma ElGamal dengan Pemrograman Paralel pada Platform CUDA, Bandung: Institut Teknologi Bandung, 2018.

[6] Lembaga Sandi Negara, Jelajah Kriptologi, Jakarta: Lembaga Sandi Negara Republik Indonesia, 2007.

[7] R. Munir, Kriptografi, Edisi Kedua, Bandung: Penerbit Informatika, 2019.

[8] A. Grama, A. Gupta, G. Karypis and V. Kumar, Introduction to Parallel Computing, Second Edition, Edinburgh: Addison Wesley, 2003.