Parallelization of Cryptographic Algorithm Based on Different Parallel Computing Technologies

Lesia Mochurad[a], Glib Shchur[a]

[a] *Artificial intelligence Department, Lviv Polytechnic National University, Lviv, 79013, Ukraine*

Abstract

The analysis of efficiency of application of four technologies of parallel programming for parallelization of algorithm of block encryption Advanced Encryption System is carried out in the work. The obtained results showed that the average execution time of this algorithm can be increased three times with a processor and thousands of times with a graphics processor. The advantages and disadvantages of each of the technologies are analyzed. But it is shown that each of them can be suitable for different scenarios. That is why it is important for a programmer to know OpenMP, Java Threads, Java ForkJoin, CUDA and other technologies that exist for parallelization. The software is developed and a number of numerical experiments are carried out. The reliability of the obtained encryption results is confirmed. To eliminate the influence of external factors on the reporting time, the algorithm was performed 10 times in a row and the average value was calculated. The largest increase in speed is obtained using CUDA to parallelize the Advanced Encryption System algorithm and is more than 300000, which is a very significant improvement. After that, we get an acceleration of three times for OpenMP and 2.8 on average for encryption and decryption using both Java technologies.

Keywords 1

Advanced Encryption System algorithm, OpenMP technology, Java Threads, CUDA, acceleration.

1. Introduction

The demand and need to increase the speed of software applications continues to grow as programs are developed that require more computing power. Until 2004, Moore's Law allowed the number of transistors in a processor to automatically increase software performance [1]. That is, if the processor executes more instructions per second, the software will also run faster. However, due to the physical limitations of processors, as well as the heating of components, it is impossible to continue to rely on Moore's Law to achieve faster and faster execution of algorithms.

One alternative is parallel computing, which takes advantage of a multi-core computer architecture [2-4]. In this model, multiprocessors communicate with each other through a shared cache contained in апарaтi. However, the software needs to be adapted so that it can use multiple processors to work on a single task, that is, use concurrency techniques to change the way code is written and executed.

One of the most interesting and popular areas of modern development is the transition from the implementation of calculations on the central processing unit (CPU) to the calculations on the GPU graphics processing unit (GPU). In particular, Nvidia proposed its solution by developing the CUDA (Common Unified Device Architecture) parallel computing architecture. Nvidia offers examples of practical application of CUDA to solve general-purpose problems [5-8]. The observed increase in performance compared to the CPU in these examples is from 10 to 100 times.

Parallel execution of a sequential flow of instructions on the CPU has certain basic limitations and simply adding executable blocks can not achieve a significant increase in speed. At the same time,

IT&AS'2021: Symposium on Information Technologies & Applied Sciences, March 5, 2021, Bratislava, Slovakia
EMAIL: lesia.i.mochurad@lpnu.ua (L. Mochurad); hlib.shchur.kn.2017@lpnu.ua (G. Shchur).
ORCID: 0000-0002-4957-1512 (L. Mochurad); 0000-0002-3796-2866 (G. Shchur).

GPUs were originally created to execute parallel instructions. Most of the GPU, in contrast to the CPU, is occupied by executable units, which gives an advantage in the speed of tasks related to parallel data processing, namely when the same sequence of operations is applied to a large amount of data and the number of instructions exceeds the number memory accesses. Thus, the GPU architecture allows to achieve greater efficiency in parallel computing.

Another important advantage of using a GPU for general-purpose computing is that when computing on a GPU, the CPU remains less loaded and can be used to perform other tasks.

The paper analyzes different approaches to parallelization of the Advanced Encryption System (AES) block encryption algorithm. This algorithm is a block encryption algorithm that is used today in a wide range of security applications [9].

The object of research is to study the properties that allow to parallelize AES algorithms.

The subject of the research is parallel programming technologies OpenMP, Java Threads, Java ForkJoin and CUDA.

The purpose of this work is to analyze the effectiveness of various parallel computing technologies to the standard algorithm of extended AES encryption.

2. Theoretical Basis

AES algorithm.

This algorithm is known as **Rijndael** - asymmetric block encryption algorithm (block size 128 bits, key 128/192/256 bits). As Selent found, Reindale's algorithm "uses a combination of exclusive OR (XOR) operations, replacing octets to rotate the rows of an S-box array" and a column and a mixture of columns. It was successful because it is easy to implement and can be run without much expense on a regular computer [10].

The AES algorithm works in both directions, used to encrypt and decrypt any type of binary file. As an input file, it receives a file divided into blocks of 128 bits (16 bytes) and a key of 128, 196 or 256 bits. This paper uses a 128-bit key [11]. For each block, there is another 128-bit block with encrypted data. Operations are performed on a 4x4 matrix that contains a corresponding block called a state.

Figure 1 shows a graphical representation of the algorithm, highlighting the main transformations that occur over the state. It is important to note that these transformations are applied several times in CBC (Cipher Block Chaining) mode, which means that after the first round of encryption on plain text, additional operations will be applied on the encrypted text. This increases security. These transformations are described below:

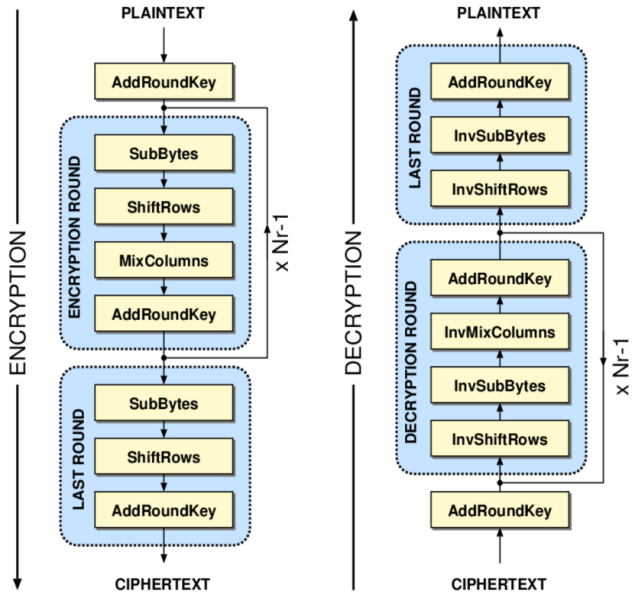

Figure 1: Block diagram of encryption and decryption using the AES algorithm

- ***Key expansion:*** This is a common program used to expand the input key and get 16∗(Nr + 1) new bytes. Therefore, a different key is used in each round of the algorithm. The number of rounds depends on the length of the key, because a 16-byte key will be used, there will be 10 rounds, and the expanded key will be 176 bytes. Methods used to extend the key include cyclic permutations and arithmetic operations on Galois fields. See more on Figure 1 [11].
- ***Add Round Key:*** In this part of the process, the key is added to the state using an XOR operation. Part of the expanded key used depends on the integer number [11].
- ***Sub Bytes/Inverse Sub Bytes:*** This conversion uses a lookup table to perform byte substitution, ie the value of each byte is changed by another in the S-box matrix, or the inverse S-box for decryption. The matrix used was carefully selected for safety. Again, the values of the matrices are the result of modular arithmetic in Galois fields.
- ***Shift Rows/ Inverse Shift Rows:*** In this step, the last three status rows are shifted to the left for encryption or to the right for decryption. Line 1 is scrolled with an offset of 1, line 2 of 2, and line 3 of 3.
- ***Mix Columns/ Inverse Mix Columns:*** This process is really difficult to explain because it uses Galois field operations in each column of the state matrix. The arithmetic operations can be calculated in advance and saved in different search tables. Therefore, the process is reduced to performing multiplications and XOR operations with each status column and the corresponding search table [10].

All of the above transformations apply to each block into which the input file is divided 10 times in a row. Each block that is encrypted is independent of the others, which makes it possible to make a parallel implementation of the algorithm.

3. Setting the Task

To achieve the goal set in the work it is necessary:

- • Explore the application of four different technologies: OpenMP, Java Threads, Java ForkJoin and CUDA, which implement different strategies to achieve the implementation of the AES encryption algorithm.
- Develop software that works in parallel.
- Analyze the advantages and disadvantages of each of these technologies, highlighting applications in which it is convenient to use each of the tools. Similarly, provide a performance analysis when performing the AES algorithm.
- Compare the speed and efficiency of parallel algorithms implemented on the CPU and GPU.

4. Materials and Methods

Parallelization of the AES algorithm

The AES algorithm cannot be parallel by definition, as each round depends on the results of the previous encryption round. Although the conversions can be parallel, it makes no sense to use different threads to calculate operations, in a state of only 16 bytes.

The approach used in this work focuses on the parallel encryption of the blocks into which the file is divided. This is possible because the encryption process for each block is independent of the others. For example, suppose you need to encrypt a file of 1048576 bytes. This file will be divided into 16-byte blocks. Thus, the AES algorithm will be applied to 65536 blocks. Here you can use parallelization to distribute the work between the available threads depending on the technology used.

AES parallelization using OpenMP

The main feature of OpenMP [12, 13] is an easy transition from serial to parallel code only with the help of pre-processing directives that tell the compiler which sections of code will be executed in parallel. This makes it easy for the programmer to parallelize the code. It is important to note that shared and private data are key to OpenMP and should be defined in the directives, thus establishing communication and synchronization between threads.

The following is the main function of encryption in AES using OpenMP:

```
void cipher_control(byte *file_in , byte *file_out , long long file_size , unsigned long
blocks , byte *expanded_key)
{
   unsigned long block ;
   int padding , res ;
   // Check if the size of the input file is multiple of 16
   res = file_size % 16;
   #pragma omp parallel for shared(file_in , file_out , expanded_key)
   for ( block = 0; block < blocks ; block++) {
       // Check if it is necessary to add padding to the last block
       if(block == blocks - 1 && res != 0)
       {
           padding = 16 - res;
           for(int i = res;i < res + padding;i++)
           {
               file_in[block * 16 + i] = 0x00;
           }
       }
       //Invoke the cipher process for the corresponding block
       cipher(file_in + block * 16, expanded_key);
       //Copy the encrypted block to the output file
       memcpy(file_out + block * 16, file_in + block * 16, 16 * sizeof(byte));
   }
}
```

As you can see, this piece of code corresponds to the division of work into blocks. The most important part is the line where the encryption method is called, which performs the AES algorithm to encrypt the current block. There are no significant changes in the consistent implementation. Except for the line that specifies that the loop parallel will be used (`#pragma omp parallel for shared(file_in, file_out, expanded_key)`). Which can share in memory the input file, the output file and the encryption key. Here, in a simple way the algorithm is parallelized by means of OpenMP.

AES parallelization using CUDA

A GPU graphics processing unit is required to run CUDA. Thanks to the GPU architecture, it is possible to control hundreds of threads and obtain large accelerations [5, 14] The main advantage of GPUs is that they can run a huge number of parallel threads. On the other hand, the disadvantages are the high cost of equipment and high energy consumption during execution.

The following is a snippet of the program (AES control function) written in CUDA C, which is used to control the encryption process. This method works on the GPU.

```
__global__ void cipher_control(byte *file_in , byte *file_out , long long * file_size ,
unsigned long *blocks , byte *expanded_key , byte *d_sbox , byte *d_m2, byte *d_m3)
{
   byte state [16];
   int block ;
   int padding, res;
   // Get the number of the block that the current thread is managing
   block = blockIdx . x * blockDim . x + threadIdx . x;
   // Check if the size of the input file is multiple of 16
   res = *file_size % 16;
   // Verify the current block is not out of boundaries
   while(block < *blocks) {
       //Copy the corresponding input data to the state matrix
       memcpy(state, file_in + block * 16, 16 * sizeof(byte));
       // Check if it is necessary to add padding to the last block
       if (block == ((*blocks) - 1) && res != 0)
       {
           padding = 16 - res;
           // Add padding only to the required spaces
           for (int i = res;i < res + padding;i++)
           {
               state[i] = 0x00;
           }
       }
       // Invoke the cipher process for the corresponding block
```

```
        cipher (state , expanded_key , d_sbox , d_m2, d_m3);

        // Copy the encrypted block
        memcpy(file_out + block * 16, state , 16 * sizeof(byte));
        // Update the current block moving to the next section of memory allowed for this
thread
        block += gridDim . x * blockDim . x ;
    }
}
```

This function includes the usual C syntax and some special CUDA functions. `__global__` indicates that the method will be executed on the GPU, which means that the same segment of code will be copied and executed on multiple threads simultaneously. Instruction block `block=blockIdx.x*blockDim.x+threadIdx.x;` gets the block number that the current stream needs to encrypt. Because the program works with multiple blocks of threads, it is important to get the correct index to avoid racing.

Function `cipher (state , expanded_key , d_sbox , d_m2, d_m3)` calls the encryption method, this is where the AES algorithm is actually located and performs operations on the current state. Finally, the current block to be encrypted is updated by adding `block += gridDim.x * blockDim.x`. This operation ensures that the next block to be encrypted by the current stream is not encrypted by another.

Because the size of the input files may not be a multiple of 16, a gasket is added to add the last block during the encryption process. The number of bytes added is stored in the first byte of the source file. Then in the process of decryption the information about the attachment is read, and the last bytes are ignored when saving the decrypted file.

Method `cipher_control` is called by the following statement inside the main program:

```
cipher_control <<< 128, 128>>> (/*params*/);
```

This instruction suggests the GPU to use 128*128=16384 threads to perform the function. In this paper, the optimal number of threads should be equal to the size of the input file, but this size is variable. When the number of threads is greater than required, there are threads that consume only resources, on the other hand, if the number is less, then the required threads will have to do more work. Net 128*128 – this is the optimal number that can work with large and small files.

All methods of reading / writing input / output files are controlled by the processor. All other methods related to the AES algorithm, such as AddRoundKey, MixColumns, ShiftRows, etc. they are controlled by the GPU.

AES parallelization using Java threads (Java Threads)

Java integrates support for concurrent operations into its API using various strategies, one of which is to use Thread objects. These objects are designed for parallel execution. Creating and administering threads is the responsibility of the developer.

In the case of Java threads, the decryption method was implemented by creating an AES_decipher object, which is instantiated within each of the created threads. In the case of Java code, the number of threads used depends on the available hardware. It is recommended to use only 2 threads per core of one processor.

Once the threads are created, the work is assigned to each of them, dividing by an equal number of blocks to decrypt. For example, if you had 800 blocks and 8 threads, each thread must perform the task of decrypting 100 blocks. It is important to take care of the correct distribution of memory, because if one block is assigned more than one stream, race conditions may occur [15].

The following is a piece of code that is used to process the creation, initialization, and completion of each thread. Operator `.start()` indicates that the thread should start running in parallel. On the other hand, the operator `.join()` used to indicate that you need to wait for all threads to complete before continuing the program flow.

```
//Instantiate the object to cipher
ac = new AES_cipher( file_in , key) ;
   the runtime
   // Assign the work to the threads
   for(int thr = 0; thr < MAXTHREADS; thr++)
   {
       if ( thr < MAXTHREADS – 1 )
       {
```

```
            threads [ thr ] = new Thread(new AES_cipher( thr * thread_blocks , ( + 1) *
thread_blocks ) ) ;
        }
        else
        {
            threads [ thr ] = new Thread(new AES_cipher( thr * thread_blocks, aes_blocks ))
; }
    }
    // Start the execution of the threads
    for(int thr = 0; thr < MAXTHREADS; thr++)
    {
        threads[thr].start();
    }
    // Wait for every thread to finish its work
    for(int thr = 0; thr < MAXTHREADS; thr++)
    {
        try
        {
            threads[thr].join();
        } catch (InterruptedException e)
        {
            e.printStackTrace () ;
        }
    }
```

AES using Java ForkJoin

Java also provides a framework for managing threads, so the programmer does not have to deal with the administration and synchronization of threads, because they are created in the pool. This framework is called Java ForkJoin. This works by recursively dividing the initial working block into smaller ones, until the optimal size is assigned to any of the free threads. It works using a labor theft algorithm, which, if one of the threads finishes its work and is idle, it will take part of the work from another busy thread. This is to make execution more efficient and to always support all threads [15].

Two classes were implemented in the work, one for encryption and the other for decryption. The following is a snippet that separates tasks and causes new tasks to be created for threads. As you can see, if the task is small enough, it is processed by calling the decryption control method, which is similar to the one presented in the OpenMP section. Otherwise, the block is split in half and 2 new AES_decipher objects are created, each with half of the original work. New objects are assigned to the pool using the invokeAll method. Basically, all you have to do is create a pool and assign it an initial object that contains all the work.

```
    public void compute() {
    if(this.end - this.start <= THRESHOLD)
    {
        this.decipher_control();
    }
    else
    {
        long mid = (this.start + this.end) / 2;
        invokeAll (new AES_decipher(this.start, mid),
        new AES_decipher(mid, this.end));
    }
}
```

5. Results

The same file was used to evaluate the effectiveness of the parallel algorithms used to encrypt (EC) the file and decrypt it (DC). In this way, the correctness of execution was checked, because the decrypted file corresponds to the same original file. Similarly, the execution time was measured to make appropriate comparisons. The algorithm was performed 10 times in a row, to subtract the average value, thereby eliminating the influence of external factors on the reporting time. The test file for encryption and decryption corresponds to a book in PDF format of 17.9 MB. The key used is read from the TXT file and was the same for all cases.

In all cases, the computer used for testing has the technical characteristics given in Table 1.

Table 1: Technical characteristics of the computer used in numerical experiments

CPU	Intel Core i5-8300H 2.80 GHz
Number of Kernels	8
RAM	8 GB DDR4 SDRAM
Storage capacity	128 GB SSD + 1 TB HDD
GPU	GeForce GTX 1050
OS	Windows 10

The results of the obtained acceleration were calculated by implementing the AES algorithm sequentially using C and Java. In the case of OpenMP and CUDA, the acceleration is compared to the implementation in C; while Java Threads and ForkJoin use a serial version created in Java. The results are summarized in Table 2 and in Figure 2.

Table 2: Summary of execution time for parallel implementation of AES

Execution time in ms											
C		OpenMP		CUDA		Java		Java Threads		Java ForkJoin	
EC	DC	EC	DC	EC	DC	EC	DC	EC	DC	EC	DC
2500,9	3089,4721	826, 2884	1004,1843	0,0089	0,0099	504,6	704	210,6	212,6	203,8	219,3

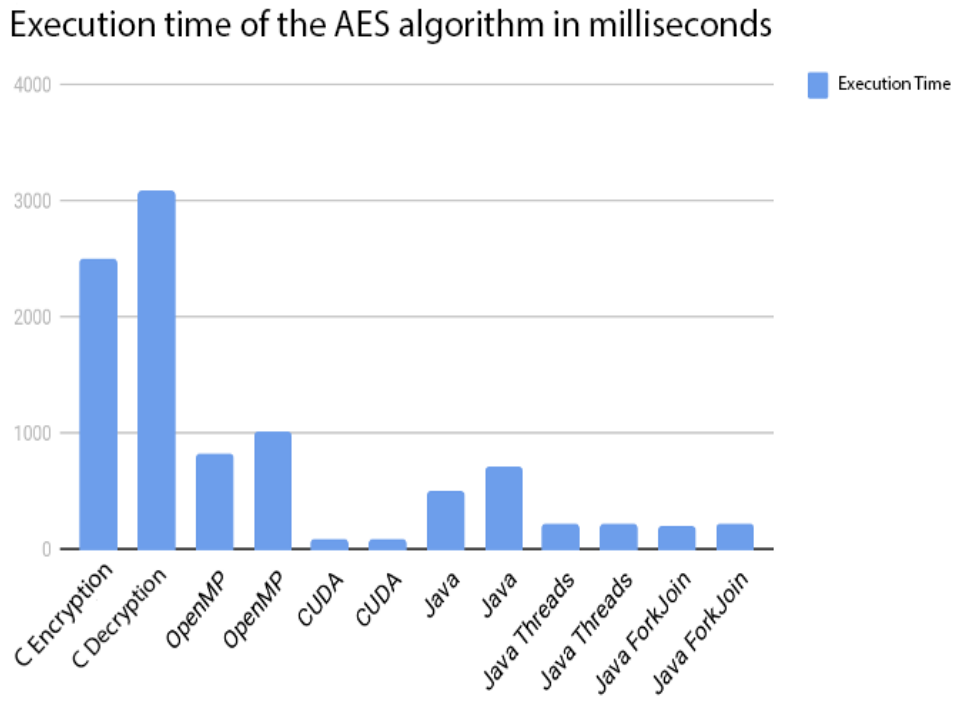

Figure 2: Comparative graph of the execution time of the parallel AES algorithm

Table 3 shows the acceleration obtained for each of the technologies.

Table 3: Acceleration obtained using four technologies for parallel AES

Technology	***Acceleration***	
	ES	**DS**
OpenMP	3.027	3.077
Java Threads	2.396	3.311
Java ForkJoin	2.476	3.210
CUDA	281000.000	312067.889

As you can see, the largest increase in speed is obtained using CUDA to parallelize the AES algorithm and is more than 300,000, which is a very significant improvement. After that, we get an acceleration of three times for OpenMP and 2.8 on average for encryption and decryption using both Java technologies. In this sense, we can conclude that when using calculations in an 8-core processor, the acceleration obtained by AES optimization is close to 3. However, the use of GPUs for the AES algorithm leads to significantly higher performance. We can conclude that the best implementation of the parallel AES algorithm in the processor is obtained using OpenMP. However, Java execution time is less than C. This is due to how Java works and manages memory. Finally, in the case of a GPU, the best optimization is definitely when using CUDA.

6. Conclusions

The choice of technology for parallel implementation of the algorithm depends on the use required for the final application. For example, in the case of AES, it can be used in different scenarios. In a high-level application that has contact with the user, a viable option would be Java Threads or ForkJoin. However, in low-level applications, such as encrypting messages sent to hardware registers, it is convenient to use C and OpenMP. Finally, if you have dedicated hardware with a GPU, a definite option would be CUDA.

The main advantage of OpenMP is related to the simplicity with which the algorithm changes sequentially to become parallel, as it only requires specifying pre-processing directives. The disadvantage is that you do not have the versatility and flexibility to change the desired behavior in parallel. In the case of Java, the disadvantage is that it is necessary to administer the threads manually and solve synchronization problems. One of the advantages would be the ability to easily work with high-level applications. For example, processing images and files does not require a high level of complexity. Finally, CUDA has the advantage of very high acceleration due to the number of flows it provides, but the disadvantage is that it requires special equipment (GPU), which is expensive and requires more resources, such as more power and also heats up more. than a regular processor.

As a result of a series of numerical experiments, the efficiency of parallelization of the AES algorithm by various parallel programming technologies was proved. Obviously, sequential computing is no longer enough to meet the needs that users currently require. Concurrency needs to be integrated into software development to achieve the best results in terms of efficiency. In addition, the architecture of multi-core and multiprocessor systems is fully used in this way. Finally, it should be emphasized that before using technologies available for multiprocessor processing, it is necessary to conduct a thorough analysis of the algorithm to be implemented to avoid runtime problems such as race conditions or even try to parallelize an algorithm that is inherently impossible. Parallelism in software is a very powerful tool that needs to be used properly for optimal results and helps to create more efficient software.

7. References

[1] Sutter, H. The free lunch is over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 30(3), 7 p. (2005).

[2] Mochurad, L., Shakhovska, K., Montenegro, S. Parallel Solving of Fredholm Integral Equations of the First Kind by Tikhonov Regularization Method Using OpenMP Technology. In: Shakhovska N., Medykovskyy M. (eds) Advances in Intelligent Systems and Computing IV. CCSIT 2019. Advances in Intelligent Systems and Computing, vol 1080. Springer, Cham, pp. 25-35. DOI: 10.1007/978-3-030-33695-0_3, 11 p. (2020) (https://doi.org/10.1007/978-3-030-33695-0_3).

[3] Mochurad, L., Albota, S. Optimizing the Computational Modeling of Modern Electronic Optical Systems. In: Lytvynenko V., Babichev S., Wójcik W., Vynokurova O., Vyshemyrskaya S., Radetskaya S. (eds) Lecture Notes in Computational Intelligence and Decision Making. ISDMCI 2019. Advances in Intelligent Systems and Computing, vol 1020. Springer, Cham. pp 597-608. (2020) doi: 10.1007/978-3-030-26474-1_41 (https://doi.org/10.1007/978-3-030-26474-1_41).

[4] Mochurad, L., Boyko, N. Solving Systems of Nonlinear Equations on Multi-core Processors. In: Shakhovska N., Medykovskyy M. (eds) Advances in Intelligent Systems and Computing IV. CCSIT 2019. Advances in Intelligent Systems and Computing, vol 1080. Springer, Cham. pp. 90-106. (2020) DOI: 10.1007/978-3-030-33695-0_8. (https://doi.org/10.1007/978-3-030-33695-0_8).

[5] Mochurad, L.I., Boyko, N.I. Technologies of distributed systems and parallel computation: monograph. Lviv: Publishing House “Bona”, 261 p. (2020). ISBN 978-617-7815-25-8.

[6] Farber, R. CUDA Application Design and Development. Waltham, MA: Morgan Kaufmann, 336 p. (2011).

[7] Sanders, J., Kandrot, E. CUDA by Example: An Introduction to GeneralPurpose GPU Programming. Michigan : Addison-Wesley Professional, 312 p. (2010).

[8] Fedushko, S., Mastykash, O., Syerov, Y., Peracek, T. Model of user data analysis complex for the management of diverse web projects during crises. Applied Sciences (Switzerland), 10(24), pp. 1–12 (2020).
[9] Pomogaeva, P.N. Cryptography: from the beginnings to the present day. Bulletin of young scientists of St. Petersburg State University of Technology and Design 4, 519-525 (2019).
[10] Selent, Douglas. Advanced encryption standard. Rivier Academic Journal, 6(2), pp. 1-14 (2010).
[11] Heron, S. Advanced Encryption Standard (AES). Network Security 2009, 12, pp. 8-12 (2009).
[12] Mochurad, L., Boyko, N., Petryshyn, N., Potokij, M., Yatskiv, M. Parallelization of the Simplex Method Based on the OpenMP Technology. Proceedings of the 4th International Conference on Computational Linguistics and Intelligent Systems (COLINS 2020). Volume I: Main Conference. Lviv, Ukraine, April 23-24, 936-951 p. (2020).
[13] Voss, M.J. OpenMP share memory parallel programming. Toronto, Kanada, 270 p. (2003).
[14] Agarwal, N., Goyal, A., Maheshwari, G., Dugtal, A., Parallel Implementation of Scheduling Algorithms on GPU using CUDA. International Journal of Computer Applications, Vol. 127, No 2, pp. 44–49 (2015).
[15] Z. Hu, S. Gnatyuk, T. Okhrimenko, S. Tynymbayev, and M. Iavich, "High-Speed and Secure PRNG for Cryptographic Applications," IJCNIS, vol. 12, no. 3, pp. 1–10, Jun. 2020, doi: 10.5815/ijcnis.2020.03.01.
[16] M. M. Hassan and G. M. Rather, "Centralized Relay Selection and Optical Filtering Based System Design for Reliable Free Space Optical Communication over Atmospheric Turbulence," IJCNIS, vol. 12, no. 1, pp. 27–42, Feb. 2020, doi: 10.5815/ijcnis.2020.01.04.
[17] Java. The Java Tutorials. url: https://docs.oracle.com/javase/tutorial/ essential/concurrency /index.html (visitado 24-11-2018).