Bulgarian Chemical Communications, Volume 52, Issue A (pp. 220-224) 2020 DOI: 10.34049/bcc.52.A.218

Implementation of cryptographic algorithms *via* multithreading

N. Sinyagina[1], V. Todorov[2], G. Kalpachka[1]*

[1]*South-West University "Neofit Rilski", 66 Ivan Mihailov Str., 2700 Blagoevgrad, Bulgaria*
[2]*Sofia University "St. Kliment Ohridski", 15 Tsar Osvoboditel Blvd., 1504 Sofia, Bulgaria*

Received August 11, 2019; Accepted November 08, 2019

Speed of the encryption and decryption processes is one of the main factors when it comes to implementation of cryptographic algorithms. The use of multithreading can significantly decrease the time required for these processes. The article describes a research on multithreaded execution of the RSA algorithm and proposes a conceptual model for implementation, outlining the main points of its software design. The choice of resources, methods and tools for the application deployment has been made as well as an assessment of the used operating system and programming language.

Keywords: Multithreading, Cryptographic Algorithms, Asymmetrical Encryption Algorithm RSA.

INTRODUCTION

In today's information age the cyber-attacks become more and more frequent and dangerous. The Internet (the biggest unsecured global system for data transmission) carries large amounts of data that should not be accessible by anyone. Cryptography is a necessary element for most of the current online applications.

Despite high level of security, the methods for encryption and decryption of data are expected to also have minimum execution delay. Unfortunately, security and speed are inversely proportional – faster algorithms have more vulnerabilities, while the secure ones need more computational resources. The main goal when selecting a cryptographic algorithm is finding the optimal balance between execution time and security.

Fortunately, the technologies are rapidly improving. The computer components are always being upgraded for faster handling of data and computational tasks. The cryptographic algorithms also need to improve along with the technology.

In the recent years the parallel programming has been established as the best approach for getting higher speed in data processing. Most of the current CPUs use architecture with more than one operational thread [1]. The operating systems are also adapted to multicore technology.

Software development on a multicore hardware requires additional effort. Existing serial algorithms need to be adapted to parallel solutions. The breaking down of a process to its main components and running them simultaneously is the main principle of modern programming. Unfortunately, not all algorithms are adapted and optimized for the multithread architecture.

Data encryption is a typical example for an algorithm, usually executed as a serial program. Serial programs are directly dependent on the frequency of the used CPU core. Such algorithms do not benefit from multiple cores or threads.

The aim of this article is to research and analyze the pros and cons of RSA implementation on a multithread architecture. The main focus will lay on evaluating the speed when running the same task on a different number of threads.

MAIN STATES OF THE MULTITHREAD ARCHITECTURE AND THE RSA CRYPTOGRAPHIC ALGORITHM

Multithread architecture – hardware allowing using multiple CPU cores or threads in parallel. Multithread (or parallel) programming is the method of coding, utilizing hardware running with multiple cores and/or threads [2].

Cryptographic algorithm – method for data conversion using standardized encrypting and decrypting operations, aiming to protect the data from unauthorized access. The RSA algorithm is an example for a cryptographic algorithm and more specifically an asymmetrical one.

In the middle of the first decade of the 21st century the multithreaded architecture started to rapidly appear in personal computers.

The main strength of the multithread architecture is the multitasking. It allows the system processes and the user applications to be physically separated. The different tasks can be run actually in parallel. Additionally, different jobs can be assigned with different priorities. A lot of the multithread hardware also provides a separation of the cache memory between the different cores.

* To whom all correspondence should be sent.
E-mail: kalpachka@swu.bg

With the increasing popularity of the mobile technologies the problem with power consumption raises its priority. The multithread architecture has better energy utilization. If a given core is not used, it can be set to idle until it is needed again, which drastically reduces its power consumption.

On top of that, modern technology has almost reached the physical limits of single core clock speed. The only way to get more computational power is to use additional processing units.

The pros of a multithread architecture are many and are undoubtedly more than enough to overwhelm the single thread systems but the cons should also be considered [3].

Using multiple threads definitely expands the possibilities for software development but also raises some complications. New issues appear that were previously not present on serial programs. One of them is the access to shared memory. In serial programming memory conflicts (a process modifying another process' memory during its execution) are rare. In parallel programming it is common for two or more processes to share and modify the same memory. If the threads are not properly synchronized problems can occur. To avoid them, mutual exclusions and semaphores need to be implemented to ensure proper system operation. However, their management requires more complex software solutions, as well as additional resources for their implementation [4].

The order of executing tasks on the different threads is also a big problem in multithread architectures. Often the processes depend on one another, which requires timeouts and checks for the completion and integrity of the results of a given job.

The separation of the processes and the tracking of their execution require additional system resources. The use of two cores instead of one does not mean double performance. The complexity of the job synchronization increases with the number of threads used. There even are cases where dividing a process into multiple ones actually slows down its execution. When developing a multithreaded software there always are resource losses because of the needed bandwidth for the communication and management of the sub tasks.

The multithread architectures fit the modern requirements for development of processor technologies but on the other hand they put more weight on the software developers. Additionally, to optimize any existing serial algorithms, they need to be adapted for the new hardware architectures. The task execution time does not depend as much on the hardware as on the software optimizations [5].

Published in 1977, the RSA algorithm remains one of the most widely spread and secure algorithms for data transfers over unsecured channels.

The basic implementation of the RSA algorithms does not require complex software design but executing it with a high security level requires significant computation power. Its relatively slow run speed is the main reason why it is not more widely spread [6].

The main plus of the RSA algorithm is the fact that it is asymmetric. By definition it does not require exchange of private information between the parties at any point during the data transfer. This is important because any sharing of private information has a risk of directly violating the security. On top of that, if the public key is big enough in length, the cypher becomes virtually impossible to crack.

METHOD OF IMPLEMENTING RSA ON MULTITHREAD ARCHITECTURE

Proposal for speeding up the RSA process via multicore architecture

The RSA algorithm is implemented in 4 steps [7]:

1. Key generation – the recipient of the encrypted messages should generate a pair of keys: public and private. The public key can be freely distributed while the private key should not be shared. The public key is used for encryption while the private key is used for decryption.
2. Sharing of public keys – after generating the key pair, the public key should be delivered to the sender of the encrypted messages. The sharing can occur over unsecured channels because the public key by itself does not present any security risks.
3. Encryption of the message – anyone having the public key can generate encrypted messages using the RSA procedure [8].
4. Decryption of the message – the encrypted message can be decrypted only with the private key, which should only be available for the recipient.

Method for multithreaded implementation of the RSA algorithm

1. Creating a number of threads – a number (n) of threads is created at the begging of the task. The number n is predefined and corresponds to the number of threads to be used. They cannot exceed the number of threads the operating system can

access. All threads should be able to work in parallel with each other.

2. Splitting the data for encryption or decryption into chunks – the input for encryption should be split in relatively even blocks, the number of which should be the same as the number of threads. Verifications need to be implemented to make sure all data are collected and false data are not accidentally added.

3. Passing the split data blocks for processing – at the beginning of the execution of every thread the position of the processed data needs to be marked. This makes it possible to put the data back in order when merging the output of all threads.

4. Running and managing the threads – all threads are independent from each other. There should not be any resource conflicts between them. A process is created to check the state of the separate threads during their execution: not started, running or finished. This process should be highly optimized and need minimum amount of computing resources.

5. Data collection – the parallel executing of the different tasks does not ensure that the processes will finish in the same order they were started. The data from all treads need to be collected and assembled in the same order they were split. After a thread finishes its task, its output is being saved in its corresponding place according to the input order. Once all threads are done and all data are assembled, the process has successfully finished.

CONCEPTUAL MODEL

The main aim in the current article is a presentation of the created application for evaluating the speed during encryption and decryption, using the asymmetrical RSA algorithm on a multicore platform. Such task requires implementation of multiple different sub blocks, working successively.

The sub blocks are divided as follows:

- *Starting block* (input data set up) – this block sets up the input parameters, which is important for fast and easy modification of the initial parameters for the different tests. The comparison analysis requires a large number of various tests with different input variables. The input parameters consist of: size of the encryption data, number of threads used, size of the encryption key, number of consecutive tests to run with these settings. This module lays the base for the execution of the actual encryption and decryption procedures.
- *Test block* (management of the testing functions)–the comparison analysis of the test is based on measuring the executing speed of the encryption and decryption processes using different hardware resources with the same input parameters. It is responsible for running and managing all tests with the requested input parameters.
- *RSA process management block* – connects the actual RSA algorithm implementation with the test functionalities. The main task of the block is splitting the data, creation of encryption and decryption threads, managing, collecting and arranging the output data from the threads.
- *Results block* (time measurement and result calculation) – measures the time needed for all tasks, calculation of average result scores (minimizing errors), output of the final test results. The final result analysis is made based on the output of this module.
- *Blocks for RSA encryption and decryption* – executing of the base RSA encryption and decryption functions.

The connections between the blocks are marked on the algorithm flowchart (Fig. 1).

The algorithm starts with setting up the input parameters. They are used for initial setting of the test environment and do not change during the test execution. The input variables are also used for pre-calculation of other values that will remain constant during all tests. These include: generation of a key pair with a given length and generation of a random character string (used for encryption), again with a predefined length. The key pair, the data for encryption (the message) and the input parameters create the test environment. After creating it, the actual test can begin.

After the *Starting block* has finished its task it calls the *Test block*, passing all test environment variables to it. Based on those parameters the *Test block* prepares the tests for execution. For each test the block calls the *RSA process management block* once to encrypt and once to decrypt the data. Apart from starting the tasks, the *Test block* also aims to minimize the measurement error of the procedure. To accomplish that it replays each test multiple times allowing a possibility to even out any errors caused by outside factors. After all tests are done, the *Results block* is called to give the final test results.

The most complicated part of the application is the *RSA process management block*. It prepares the input data and passes them in parallel to the working threads. The flow of the block can be summarized in four steps: splitting of the input data in relatively equal chunks depending on the number of used threads; creation of the threads; starting and managing of the threads; collection and

arrangement of the output data of each thread. These steps are similar for both encryption and decryption functions.

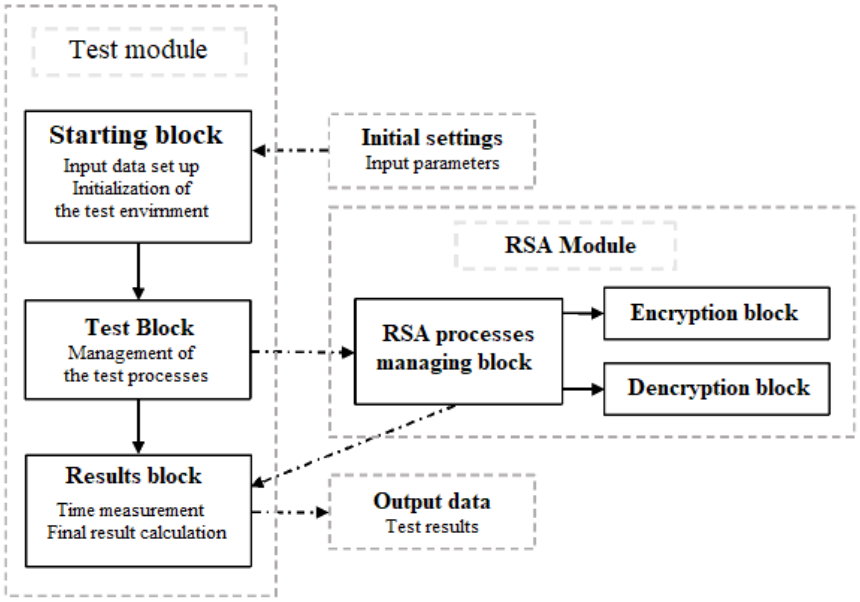

Fig. 1. Algorithm flowchart.

Additionally, the *RSA process management block* also has to communicate with the *Results block*. The time needed for each step has to be measured and saved. Since the steps are created inside the RSA modules, time measuring functions from the test blocks need to be implemented in the RSA modules. The blocks for encryption and decryption contain the main computing functionalities of the RSA processes. The possibility to use multiple processes for encoding and decoding at the same time is a requirement for the current implementation. The parallel processes should not have any conflicts between them. The encryption and decryption modules do not call out other blocks after their completion. Instead, they just notify their management block when their task is completed.

The last part of the test application is the *Results block*. It gathers the time information during the execution of the program and outputs its final version at the end of all tests. All results need to be recorded, averaged and presented in an easy to read form.

The output information consists of: a number of threads for completing the task, time for encryption/decryption and preparation of the test environment. The end results can be used for additional calculations and studies.

CONCLUSIONS

The implementation of the asymmetric cryptographic RSA algorithm on multithread architecture proves that using multiple parallel threads for data encryption and decryption leads to significant speed improvement of the algorithm.

The time for encryption and decryption is reduced almost twice when executing the task on two threads instead of just one. At eight threads, the speed is more than three times faster than at one. Even though the needed time does not decrease linearly with the number of threads used, the results show quite an improvement.

The only downside of the implementation is the lack of improvement when working with small in size messages. In that case the resources lost for thread management actually slow down the actual execution of the task.

Even with the proposed speed improvements, the RSA algorithm still remains on the slow end of cryptographic algorithms, especially compared to the symmetrical ones. The reached level of optimization might not make it a preferred choice over its rivals but it will definitely make it a better option for the cases where a high level of security is required.

REFERENCES

1. J. Stokes, Introduction to Multithreading, Superthreading and Hyperthreading, 2002, https://arstechnica.com/features/2002/10/hyperthreading/.
2. D. Marr, F. Binns, D Hill, G. Hinton, D. Koufaty, J. Miller, M. Upton, *Intel Technology Journal*, **6(1)**, 4 (2002).
3. S. Casey, How to Determine the Effectiveness of Hyper-Threading Technology with an Application, 2011, https://software.intel.com/en-us/articles/how-

to-determine-the-effectiveness-of-hyper-threading-technology-with-an-application.
4. J. Hruska, Maximized Performance: Comparing the Effects of Hyper-Threading, Software Updates, 2012.
5. Hyper-Threading Technology – Operating Systems That Include Optimizations for Hyper-Threading Technology, Intel, 2011.
6. H. Fadhil, M. Younis, *International Journal of Computer Applications*, **87(6)**, 15 (2014).
7. J. Jonsson, B. Kaliski, Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1, 2003, https://www.ietf.org/rfc/rfc3447.txt.
8. B. Aleksandrov, Hybrid Cryptographic Methods And Tools For Information Protection in the Computer Networks and Systems, in: UNITECH'2012 (Proc. Int. Sci. Conf.), Technical University of Gabrovo, part I, p. 382, 2012.