Dariusz BURAK

West Pomeranian University of Technology

Parallelization of the Block Encryption Algorithm Based on Logistic Map

Abstract. *In this paper the results of parallelizing the block encryption algorithm based on logistic map are presented. The data dependence analysis of loops was applied in order to parallelize this algorithm. The OpenMP standard is used for presenting the parallelism of the algorithm. The efficiency measurement for a parallel program is shown.*

Streszczenie. *W artykule zaprezentowano wyniki zrównoleglenia blokowego algorytmu szyfrowania opartego na odwzorowaniu logistycznym. W celu zrównoleglenia algorytmu zastosowano analizę zależności danych. Celem przedstawienia równoległości algorytmu użyto standardu OpenMP. Pokazano wyniki pomiarów efektywności programu równoległego. (**Zrównoleglenie blokowego algorytmu szyfrowania opartego na odwzorowaniu logistycznym**).*

Keywords: chaos-based encryption algorithm, logistic map, parallelization, OpenMP
Słowa kluczowe: algorytm szyfrowania oparty na teorii chaosu, odwzorowanie logistyczne, zrównoleglenie, OpenMP

Introduction

One of the very important features of cryptographic algorithms is a cipher speed. This feature is very important in case of block ciphers based on chaos theory because they have to work with large data sets. Therefore, it is important to parallelize the most time-consuming loops in order to achieve faster processing using multiprocessors and multi-core processors. Nowadays, there are many descriptions of various block ciphers based on chaotic maps, for instance [1], [2], [3], [4], [5], [6]. The important issue of chaotic ciphers is program implementation. Unlike parallel implementation of classical block ciphers, for instance AES [7], IDEA [8] there are only few parallel implementations of chaotic block ciphers, for instance [9]. It looks like a research gap because only software or hardware implementation will show real functional advantages and disadvantages of encryption algorithms.
Considering this fact, the main contribution of the study is developing a parallel algorithm in accordance with OpenMP of the cipher designed by Kocarev and Jakimoski [10] (called further KJ encryption algorithm) based on the transformations of a source code written in the C language representing the sequential algorithm.

The KJ Encryption Algorithm

The KJ encryption algorithm is a block encryption algorithm developed by Kocarev and Jakimoski and published in 2001 [10] based on logistic map that operates on 64-bit data blocks with a 128-bit encryption.

An input plaintext block is partitioned into eight sub-blocks, each one consists of 8 bits. The cipher consists of *r* rounds of identical transformations applied in a sequence to the plaintext block. Encryption transformation is given with:

$$\begin{aligned} x_{i,2} &= x_{x-1,1} \square f_0, \\ x_{i,3} &= x_{x-1,2} \square f_1, \\ &\ldots \\ x_{i,0} &= x_{x-1,7} \square f_6, \\ x_{i,1} &= x_{x-1,0} \square f_7, \end{aligned} \tag{1}$$

where: i=1,...,*r*.

The functions $f_1,...,f_7$ have the following form:

$$f_j = f[x_{i-1,1} \square \ldots \square x_{i-1,j} \square z_{i-1,j}], \tag{2}$$

where: j=1,...,7 and f: $M \rightarrow M$, $M = \{0,255\}$, is a map derived from a chaotic map. $f_0 = z_{i,0}$ and $z_{i,0},\ldots, z_{i,7}$ are the eight bytes of the subkey z_i which controls the ith round. The output block is input in the next round, except with the last round. The length of the ciphertext block is 64 bits. Each round i is controlled by one 8-byte subkey z_i. There are *r* subkeys derived from the key in a procedure for generating round subkeys. *f* is obtained via discretization of logistic map.
Decryption process is similar to encryption one, where round subkeys are applied in reverse order in comparison with the encryption process.
More detailed description of KJ encryption algorithm is given in [10] or [11].

Parallelization Process of the KJ Encryption Algorithm

Considering the fact that proposed algorithm can work in block manner it is necessary to prepare a C language source code representing the sequential KJ encryption algorithm working in ECB mode of operation before we start parallelizing process. The source code of the KJ encryption algorithm in the ECB mode contains twenty one "for" loops. Seventeen of them include no I/O functions.

In order to find dependences in program loops we have applied Petit developed at the University of Maryland under the Omega Project and freely available for both DOS and UNIX systems. Petit is a research tool for analyzing array data dependences [12], [13].

In order to present parallelized loops, we have used the OpenMP standard. The OpenMP Application Program Interface (API) [14], [15] supports multiplatform shared memory parallel programming in C/C++ and Fortran on all architectures including Unix and Windows NT platforms. OpenMP is a collection of compiler directives, library routines and environment variables which could be used to specify shared memory parallelism. OpenMP directives extend a sequential programming language with some constructs: Single Program Multiple Data (SPMD) constructs, worksharing constructs, synchronization constructs and help us to operate on both shared and private data. An OpenMP program begins execution as a single task (called a master thread). When a parallel construct is encountered, the master thread creates a team of threads. The statements within the parallel construct are executed in parallel by each thread in the team. At the end of the parallel construct, the threads of the team are synchronized. Then only the master thread continues

execution until the next parallel construct will be encountered. To build a valid parallel code, it is necessary to preserve all dependences, data conflicts and requirements regarding parallelism of a program [14], [15].

The parallelization process of the KJ encryption algorithm consists of the following stages:

- carrying out the data dependence analysis of a sequential source code in order to detect parallelizable loops,
- selecting parallelization methods based on source code transformations,
- constructing parallel forms of `for` loops in accordance with the OpenMP standard.

There are the following basic types of the data dependences that occur in "for" loops [16], [17]:

- a Data Flow Dependence indicates that write-before-read ordering that must be satisfied for parallel computing,
- a Data Anti-dependence indicates that read-before-write ordering should not be violated when performing computations in parallel,
- an Output Dependence indicates a write-before write ordering.

Additionally, control dependence [16], [17] determines the ordering of an instruction *i*, with respect to a branch instruction so that the instruction *i* is executed in correct program order.

At the beginning of the parallelization process, we carried out experiments with sequential KJ algorithm for an about 10 megabytes input file in order to find the most time-consuming loopsin this algorithm.

It appeared that the algorithm has two computational bottlenecks: the first is enclosed in the function `kj_enc()` and the second is enclosed in the function `kj_dec()`. We developed the `kj_enc()` function in order to enable enciphering the whichever number of data blocks and the `kj_dec()` one for deciphering (by analogy with functions included in the C language source code of the classic cryptographic algorithms like DES- the `des_enc()`, the `des_dec()`, LOKI91- the `loki_enc()`, the `loki_dec()` or IDEA- the `idea_enc()`, the `idea_dec()` presented in [18]). Thus the parallelization of these functions has a unique meaning.

The bodies of the most-time consuming loops included in these functions (the first loop is included in the `kj_enc()`function, the second in the `kj_dec()`) are the following:

```
for(l=0;l< PARALLELITY;l++) {
copy(tmplain, src[8*l+8]);
for (i=1;i<=rounds;i++) {
generatekey(newkey,key,i,sbox,rounds);
dst[8*l+2]=tmpplain[1]^newkey[0];
dst[8*l+3]=tmpplain[2]^sbox[newkey[1]
^tmpplain[1]];
dst[8*l+4]=tmpplain[3]^sbox[newkey[2]
^tmpplain[1]^tmpplain[2]];
dst[8*l+5]=tmpplain[4]^sbox[newkey[3]
^tmpplain[1]^tmpplain[2]^tmpplain[3]];
dst[8*l+6]=tmpplain[5]^sbox[newkey[4]
^tmpplain[1]^tmpplain[2]^tmpplain[3]
^tmpplain[4]];
dst[8*l+7]=tmpplain[6]^sbox[newkey[5]
^tmpplain[1]^tmpplain[2]^tmpplain[3]
^tmpplain[4]^tmpplain[5]];
dst[8*l]=tmpplain[7]^sbox[newkey[6]
^tmpplain[1]^tmpplain[2]^tmpplain[3]
^tmpplain[4]^tmpplain[5]^tmpplain[6]];
dst[8*l+1]=tmpplain[0]^sbox[newkey[7]
^tmpplain[1]^tmpplain[2]^tmpplain[3]
^tmpplain[4]^tmpplain[5]^tmpplain[6]
^tmpplain[7]];
copy(tmplain, dst[8*l+8]);
}
}.

for(l=0;l< PARALLELITY;l++) {
copy(tmpcipher,src[8*l+8]);
for (i=rounds;i>=1;i--) {
generatekey(newkey,key,i,sbox,rounds);
dst[8*l+1]=tmpcipher[2]^newkey[0];
dst[8*l+2]=tmpcipher[3]^sbox[newkey[1]
^dst[1]];
dst[8*l+3]=tmpcipher[4]^sbox[newkey[2]
^dst[1] ^dst[2]];
dst[8*l+4]=tmpcipher[5]^sbox[newkey[3]
^dst[1] ^dst[2]^dst[3]];
dst[8*l+5]=tmpcipher[6]^sbox[newkey[4]
^dst[1] ^dst[2]^dst[3]^dst[4]];
dst[8*l+6]=tmpcipher[7]^sbox[newkey[5]
^dst[1] ^dst[2]^dst[3]^dst[4]^dst[5]];
dst[8*l+7]=tmpcipher[0]^sbox[newkey[6]
^dst[1]^dst[2]^dst[3]^dst[4]^dst[5]^dst[6]];
dst[8*l]=tmpcipher[1]^sbox[newkey[7]^dst[1]
^dst[2]^dst[3]^dst[4]^dst[5]^dst[6]^dst[7]];
copy(tmpcipher,dst[8*l+8]);
}
}.
```

Taking into account a high degree of similarity of bodies of these loops, we examine only the first loop. However, this analysis is valid also in the case of the second loop.

The actual parallelization process of the first loop consists of the three following stages:

- filling in the loop by the body of the `generatekey()` function (otherwise, we cannot apply a data dependence analysis)
- suitable variables privatization (l, i, ii, k, newkey, sbox, tmpplain) using OpenMP (based on the results of data dependence analysis);
- adding appropriate OpenMP directive and clauses (#pragma omp parallel for private() shared()).

The steps above result in the following parallel form of loop in accordance with the OpenMP standard:

```
#pragma omp parallel for private(l, i, ii,
k, newkey, sbox, tmpplain)
for(l=0;l< PARALLELITY;l++) {
…
}.
```

The second loop was parallelized in the same way as the first one.

Experimental Results

In order to study the efficiency of the presented KJ parallel code we used a computer with two Quad-Core Intel® Xeon Processors 5300 Series - 1,60 GHz and the Intel® C++ Compiler ver. 12.1 (that supports the OpenMP 3.1). The results received for a 20 megabytes input file using two, four and eight cores versus the only one are shown in Table 1.

Table1. Speed-ups of the parallel KJ encryption algorithm in ECB mode of operation

Number of processors	Number of threads	Speed-up		
		Encryption	Decryption	Total
2	2	1.92	1.99	1.45
4	4	3.50	3.70	1.90
8	8	6.00	6.40	2.30

The total running time of the KJ algorithm consists of the following operations:

- data reading from an input file,
- subkeys generation,
- data encryption,
- data decryption,
- data writing to an output file (both encrypted and decrypted text).

Thus the total speed-up of the KJ parallel algorithm depends heavily on the four factors:

- the degree of parallelization of the loop included in the `kj_enc()` function,
- the degree of parallelization of the loop included in the `kj_dec()` function,
- the method of reading data from an input file,
- the method of writing data to an output file.

The results confirm that the loops included both the `kj_enc()` and the `kj_dec()` functions are parallelizable with relatively high speed-up.

The block method of reading data from an input file and writing data to an output file was used. The following C language functions and block sizes were applied:

- `fread()` function and 8192-bytes block for data reading,
- `fwrite()` function and 128-bytes block for data writing.

Using the fwrite() function is especially important; choosing, for example, the fprintf() function we got much longer time of executing our tasks.

Conclusions

In this paper, the parallelization process of the KJ encryption algorithm which was divided into parallelizable and unparallelizable parts was presented. We have shown that the time-consuming "for" loops included in the functions responsible for the encryption and decryption processes are parallelizable. The experiments have shown that the application of the parallel KJ encryption algorithm for multiprocessor and multi-core computers would considerably boost the time of the data encryption and decryption. We believe that the speed-ups received for these operations are satisfactory. Moreover, the parallel KJ encryption algorithm can be also helpful for hardware implementations or GPU implementations.

Wydanie publikacji zrealizowano przy udziale środków finansowych otrzymanych z budżetu Województwa Zachodniopomorskiego.

REFERENCES

[1] Nishio, Y., Sasase, I., Mori, S., A Secret Key Cryptosystem Using a Chaotic Map, Trans. IEICE Japan, 73 (1990), No.7, 1041--1044

[2] Fridrich, J., Symmetric Ciphers Based On Two-Dimensional Chaotic Maps, Int. J. Bifurcation and Chaos, 8 (1998), No.6, 1259—1284

[3] Scharinger, J., Fast Encryption of Image Data Using Chaotic Kolmogorov Flows, J. Electronic Imaging, 7 (1998), No.2, 318—325

[4] Yi, X., Tan, C.H., Siew, C.K., A New Block Cipher Based on Chaotic Tent Maps, IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, Volume: 49, Issue:12, (2002)

[5] Xua, S., Wang, J., Yang, S., A Novel Block Cipher Based on Chaotic Maps, Congress on Image and Signal Processing, Vol. 3, (2008)

[6] Pareek, N.K., Patidar, V., Sud, K.K., Block cipher using 1D and 2D chaotic maps, International Journal of Information and Communication Technology, Volume: 2, Issue: 3, (2010)

[7] Bielecki, W., Burak, D., Exploiting Loop-Level Parallelism in the AES Algorithm, WSEAS Transactions on Computers, Issue 1, vol. 5, January, (2006), pp. 125—133

[8] Bielecki, W., Burak, D., Parallelization of the IDEA Algorithm, Lecture Notes in Computer Science, 3036, (2004), pp. 635—638

[9] Burak, D., Chudzik, M., Parallelization of the Discrete Chaotic Block Encryption Algorithm, , Lecture Notes in Computer Science, 7204, (2012), pp. 323—332

[10] Kocarev, L., Jakimoski, G., Logistic Map as a Block Encryption Algorithm, Physics Letters A,. 289(4-5), (2001), pp.199--206

[11] Pejaś J., Skrobek A., Chaos-Based Inforamtion Secutity, Handbook of Information and Communication Security, (2010), 91—128

[12] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., Wonnacott, D., New User Interface for Petit and Other Extensions. User Guide, (1996)

[13] The Omega Project: Frameworks and Algorithms for the Analysis and Transformation of Scientific Programs, http://www.cs.umd.edu/projects/omega/

[14] OpenMP Application Program Interface. Version 3.1, July 2011, (2011)

[15] Chapman, B., Jost, G., van der Pas, R., Using OpenMP - Portable Shared Memory Parallel Programming, The MIT Press, (2007)

[16] Aho, A., Lam, M., Sethi, R., Ullman, J., Compilers: Principles, Techniques, and Tools (2nd edition), Prentice Hall, (2006)

[17] Allen R., Kennedy K., Optimizing compilers for modern architectures: A Dependence-based Approach, Morgan Kaufmann Publishers, Inc., (2001)

[18] Schneier B.: Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition, John Wiley & Sons, 2 edition, (1995)

***Author**: dr inż. Dariusz Burak, Zachodniopomorski Uniwersytet Technologiczny w Szczecinie, Wydział Informatyki, ul. Żołnierska 49 71-210 Szczecin, E-mail:* dburak@wi.zut.edu.pl.