Improving throughput of RC4 algorithm using multithreading techniques in multicore processors

T.D.B Weerasinghe
MSc.Eng, BSc.Eng (Hons),
MIEEE, AMIE (SL), AMCS (SL)
Software Engineer
IFS R&D International, 363,
Udugama, Kandy, Sri Lanka
tharindu.weerasinghe@gmail.com

ABSTRACT
RC4 is the most widely used stream cipher around. So, it is important that it runs cost effectively, with minimum encryption time. In other words, it should give higher throughput. In this paper, a mechanism is proposed to improve the throughput of RC4 algorithm in multicore processors using multithreading. The proposed mechanism does not parallelize RC4, instead it introduces a way that multithreading can be used in encryption when the plaintext is in the form of a text file. In this particular research, the source codes were written in Java (JDK version: 1.6.0_21) in Windows environments. Experiments to analyze the throughput were done separately in an Intel® P4 machine (O/S: Windows XP), Core 2 Duo machine (O/S: Windows XP) and Core i3 machine (O/S: Windows 7).

Outcome of the research: Higher throughput of RC4 algorithm can be achieved in multicores when using the proposed mechanism in this research. Effective use of multithreading in encryption can be achieved in multicores using this technique.

General Terms
Accelerating RC4, Parallel RC4 Encryption, Multithreading for Encryption.

Keywords
Throughput of RC4, Multithreading in Encryption

1. INTRODUCTION
The main objective of the research was to study the robustness of RC4 (implemented purely in software) when encryption is done in multiple threads. In other words the intension was to improve the throughput of RC4 by using parallelism. For that, the following mechanism was used:

Dividing the input file into similar sized portions, then encrypting each portion by RC4 (key size: 128 bits) and merging all encrypted portions together then finally saving the final encrypted file in a folder. This process was done multiple threads to get the advantages of eligible parallelism in multicores. (Using Executors in Java).

Measurement criteria for parallel execution: Average encryption time (i.e. time taken to divide the file into similar parts, encrypt and form the final encrypted files by merging all the portions) of different plaintext data sizes in different types of multicores (after similar number of experiments for each data size).

Measurement criteria for sequential execution (to compare the results of the parallel execution and come to a conclusion): Average encryption time of different plaintext data sizes in different types of multicores (after similar number of experiments for each data size).

RC4 is the most popular stream cipher in the world of cryptography [3]. When it comes to accelerating the algorithm, the focus of many researches has been pivoted around hardware implementations. [2, 3, 4, 5]. Since RC4 is commonly used in WEP, variety of studies has been done in the areas of WEP to improve the latency of RC4 or in other words improve the throughput. [4], [6]. So, in this research the idea was to identify parallel programming model or mechanism to improve the throughput of the algorithms which is cost effective.

Improving throughput of encryption algorithms is essential as the connectivity of computers have increased rapidly. Irrespective of the cipher used, the major reason for the lower throughput is lack of parallelism. [7] Thus focus of this work was to introduce a parallel execution mechanism to RC4 in-order to enhance the throughput. In open literature, there is no evidence of using multithreading (in Java) to execute the mechanism used in this research in multicores to accelerate RC4.

Initially the input text file was divided into similar sized parts (chunks). This mechanism was adopted from a research done by Barnes A. et al. [1] but in that research the file chunking mechanism was not executed using parallelism. But here, in this research, file chunking and encryption operations were executed in multiple threads using Java's executors.

In the parallel approach, for each plaintext data size, the encryption time was measured in nanoseconds once all the threads have been executed. Then the average time was calculated after similar number of experiments for each data size respectively. To compare the results with the sequential implementation, original input text file was encrypted (without dividing into similar parts) and the average times were calculated accordingly.

All these experiments were done in 3 computers, Intel® P4, Inter® Core™ 2 Duo and Core™ i5 respectively.

2. RC4 ALGORITHM

RC4 is the most commonly used, stream cipher. It is used in applications like WEP, WPA, SSL, PDF, Bit-Torrent protocol system, MS Point-to-Point Encryption, Remote Desktop Protocol and Skype.

In this section, the algorithm RC4 is described with respect to its major parts: The key scheduling algorithm (KSA) and the pseudo-random generation algorithm (PRGA). In most applications RC4 is used with a word size $n = 8$ and array size $N = 2^8$. [7]

The RC4 algorithm can be illustrated as follows because n = 8 and N = 2^8 (256)

KSA:

```
for i = 0 to 255
  S[i] = i;
j=0
for i = 0 to 255
 j = (j+S[i]+K[i mod l]) mod 256;
 swap (S[i] and S[j]);
end loop;
```

PRGA:

```
i = 0, j=0;
for x = 0 to L-1
  i = (i+1) mod 256;
  j = (j+S[i]) mod 256;
  swap S[i] and S[j];
  GeneratedKey = S[ (S[i] + S[j]) mod 256] ;
  Output = M[x] XOR GeneratedKey;
```

Where 'M' is the plain text message and 'L' is its length. [4]
Output is the Ciphertext

3. MULTITHREADING IN JAVA

A thread is the smallest unit of processing that is scheduled by the operating system. A process is a unit of execution in operating system level. Normally a computer executes more than one process at a time. A single program can be multi-threaded. Time slicing is done like in multiprocessing. The threads share the same memory. [9]

Tasks can be regarded as logical units of work, and threads are a mechanism that enables tasks (units of work) run asynchronously. There are two policies for executing tasks using threads: execute tasks sequentially in a single thread, and execute each task in its own thread. Both carry limitations: the sequential approach suffers from poor throughput, and the thread-per-task approach suffers from poor resource management. [10]

A thread pool, manages a homogeneous pool of worker threads. A thread pool is strictly dedicated to a work queue that is holding tasks waiting to be executed. Worker threads own a simple routine: request the next task from the work queue, execute it, and go back to waiting for another task. [10]

Executing tasks in pool threads is advantageous over the thread-per-task approach. Reusing an existing thread instead of creating a new one omits thread creation and teardown costs over multiple requests. And also, since the worker thread already exists at the time the request arrives, the latency associated with creation of threads does not delay the execution of tasks, thus responsiveness is improved. By properly tuning the size of the thread pool, enough threads can be obtained to keep the processors busy while not having so many or thrashes due to competition among threads for resources. [10]

Thus, to achieve a rich throughput, the sequential approach should be avoided; in other words, the multithread mechanism should be used (executing tasks in pool threads using Executors). ***Resource management*** will be fixed because a resource manager is used. Such a resource manager is supplied by Java in the form of "Executors"

The particular class library provides a flexible thread pool implementation along with some useful predefined configurations. [10]

newFixedThreadPool - A fixed-size thread pool creates threads as tasks are submitted, up to the maximum pool size, defined by the user, and then attempts to keep the pool size constant (will add new threads if a thread dies due to an unexpected Exception in the middle of execution). [10]

newCachedThreadPool - A cached thread pool has more flexibility to reap idle threads when the current size of the pool exceeds the demand for processing, and to add new threads when demand increases, but places no bounds on the size of the pool.[10]

In this research *newCachedThreadPool is used* as it has advantages.

An Executor is an object that manages tasks which are running. A Runnable can be submitted to the Executor's '***execute ()***' method in-order to be run with it. Instead of creating a thread for a Runnable that you have defined, and calling, '***start ()***', the following can be done[9] :

- Select the Executor object, say called exec
- Create your Runnable, say called myTask
- Submit for running: exec.execute(myTask)

Executors are designed relying on the producer-consumer pattern, where activities that submit tasks are the producers (producing units of work to be done) and the threads that execute tasks are the consumers (consuming those units of work). [10]

Summary of the usage of Executors: obtained from [9]

- Create a class that implements a Runnable to be the "*task object*".
- Create the task objects.
- Create the Executor.
- Submit each task-object to the Executor which starts it up in a separate thread.

4. RESEARCH METHOD

4.1 Mechanism which adhered Parallelism

Each input text file is divided into similar sized chunk. Chunk size is taken as 10000 bytes. After that, each portion is encrypted by RC4 cipher and saved in a folder. These two operations are done using multiple threads. Encryption time is calculated for the whole process (chunking, encrypting and saving) for all threads. Number of threads is decided by the JVM [as the static method newCachedThreadPool() is used].

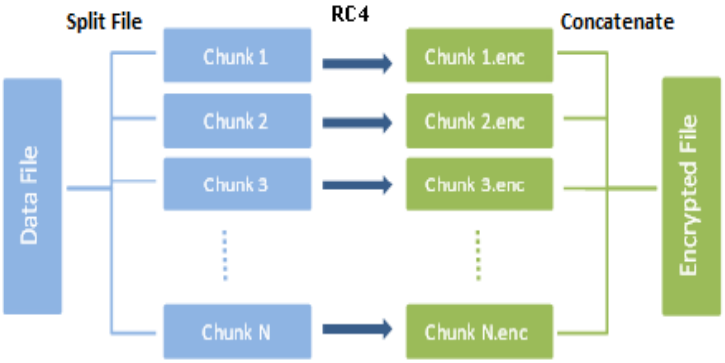

Fig.1 Overview of the mechanism (referred [1])

4.2 Sequential approach for comparison

A sequential encryption mechanism is needed to compare the results of the parallel implementation and to analyze the results. Hence the sequential encryption mechanism is used as follows:

Input file is read (without chunking) line by line and then the whole string buffer is encrypted by RC4. This is the simplest sequential approach that could have been adopted!

4.3 Use of Multithreading and Execution Time:

Below code segment shows how the executors used and how the time calculation is done:

```
public static void main(String[] args) throws IOException {


System.out.println("Multithreading started...");

ExecutorService executor = Executors.newCachedThreadPool();

        startTime1 = System.nanoTime();

        for (int i= 0; i<=2532271 ; i=i+10000)

        {

                executor.execute(new FileChunkAndEncrypt(i));

        }

endTime1 = System.nanoTime();



        System.out.println("Starting shutdown...");

        executor.shutdown();



        //Calculate the elapsed time in milliseconds

        long timeElapsed1 = CalculateTimeDifference.GetTimeElapsed(startTime1, endTime1);



        try {


executor.awaitTermination(100, TimeUnit.SECONDS);

        } catch (InterruptedException ex) {


System.out.println("Interrupted...");

        }

        System.out.println("All executed!");

        System.out.println("Time Elapsed: " + timeElapsed1 + "ns");

}
```

Value of 'i' is the chunk size and the maximum value is hard coded and it is the total file size in bytes.

The following methods are written in the class "FileChunkAndEncrypt" which implements Runnable. This is where the task is defined for executions.

```
public FileChunkAndEncrypt(int chunkSize)

{

       this.chunkSize = chunkSize;

}
```

```
@Override

public void run() {

try {

// File chunking in parallel goes
here....
readFragmentEncrypt("D:/testChunk.txt",
chunkSize);

} catch (IOException ex) {
Logger.getLogger(FileChunkAndEncrypt.clas
s.getName()).log(Level.SEVERE, null, ex);

}

}
```

4.4 Initial Key Generation:

The initial key (in both occasions – parallel and sequential) is generated using `java.security.SecureRandom` and `java.math.BigInteger`. Stream 128bit random bits were generated every time, the experiment was done. The size of the key was not changed throughout the research because the variable was the input data size.

5. RESULTS AND ANALYSIS

Experiments were done for both sequential and parallel implementations in a single core (P4), Core™ 2 Duo and Core™ i3 processors. Encryption times were measure 100 times and the average is taken. Then the throughput is calculated accordingly. Encryption time includes the time taken to chunk the input file into similar sized parts, encrypt each of those chunks and finally merge all of them to form one file. So, Encryption Time is referred as Execution Time.

These are the results:

5.1 Tables which contains the obtained results

5.1.1 Average Execution Time (ns) Vs Data Size (KB)

Table 1: Average Encryption Time Vs Data Size *(Sequential in Core i3)*

Data Size/KB	***Average Execution Time/ns***
20	4791
40	7417
60	9959
80	10507
100	12153
120	15397
140	17087
160	19015
180	20874
200	22504

Table 2: Average Encryption Time Vs Data Size *(Parallel in Core i3)*

Data Size/KB	***Average Execution Time/ns***
20	2275
40	2631
60	2794
80	2918
100	3365
120	3748
140	3908
160	3991
180	4488
200	4715

Table 3: Average Encryption Time Vs Data Size *(Sequential in Core 2 Duo)*

Data Size/KB	***Average Execution Time/ns***
20	6938
40	10267
60	14802
80	17892
100	18761
120	21252
140	24341
160	27522
180	29419
200	33284

Table 4: Average Encryption Time Vs Data Size *(Parallel in Core 2 Duo)*

Data Size/KB	***Average Execution Time/ns***
20	1912
40	2331
60	2446
80	5992
100	5826
120	7163
140	6625
160	7083
180	8037
200	8265

Table 5: Average Encryption Time Vs Data Size *(Sequential in Single Core P4)*

Data Size/KB	***Average Execution Time/ns***
20	12086
40	16677
60	20625
80	24015
100	27470
120	31737
140	35346
160	38841
180	42252
200	45609

Table 6: Average Encryption Time Vs Data Size *(Parallel in Single Core P4)*

Data Size/KB	***Average Execution Time/ns***
20	5890
40	7305
60	8617
80	8853
100	10337
120	10753
140	12481
160	17318
180	19060
200	22113

5.1.2 Throughput (MBps) Vs Data Size (KB)

Table 7: Throughput Vs Data Size *(All Sequential Implementations)*

Data Size/KB	***Throughput/MBps***		
	Corei3	**Core 2 Duo**	**P4**
20	4076.654143	2815.112424	1616.022671
40	5266.617231	3804.665433	2342.297775
60	5883.497339	3958.502229	2840.909091
80	7435.519178	4366.476638	3253.175099
100	8035.56735	5205.279569	3555.014561
120	7611.060596	5514.1869	3692.456754
140	8001.331422	5616.809088	3868.011939
160	8217.19695	5677.276361	4022.810947
180	8421.062087	5975.092627	4160.306021
200	8679.01262	5868.059728	4282.323664

Table 8: Throughput Vs Data Size *(All Parallel Implementations)*

Data Size/KB	***Throughput/MBps***		
	Core i3	**Core 2 Duo**	**P4**
20	8585.1648	10215.089	3316.002
40	14847.016	16757.829	5347.365
60	20971.278	23954.926	6799.785
80	26773.475	13038.218	8824.692
100	29021.174	16762.144	9447.253
120	31266.676	16360.114	10898.12
140	34984.327	20636.792	10954.15
160	39150.589	22059.862	9022.404
180	39166.945	21871.501	9222.521
200	41423.648	23631.276	8832.474

5.2 Graphs which illustrates the results

5.2.1 Average Execution Time Vs Data Size

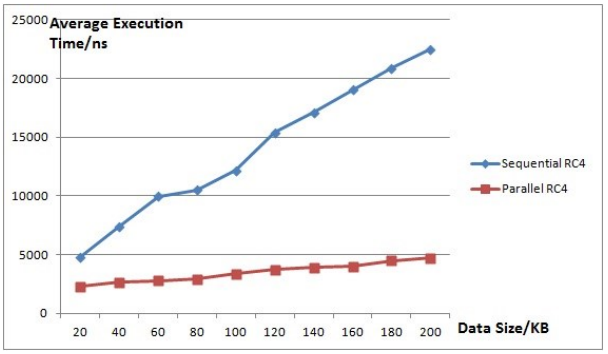

Fig.2 Average Execution Time Vs Data Size (Core i3)

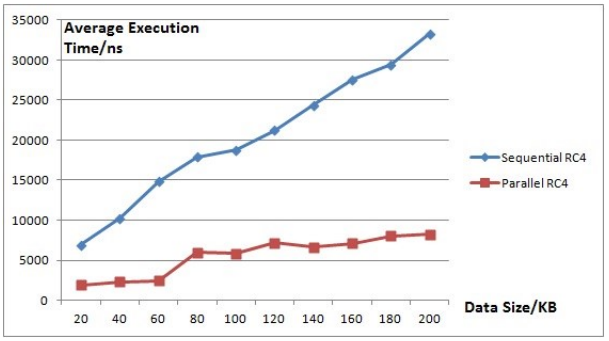

Fig.3 Average Execution Time Vs Data Size (Core 2 Duo)

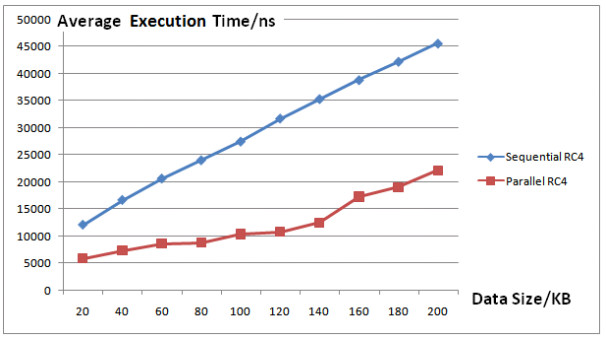

Fig.3 Average Execution Time Vs Data Size (P4)

5.2.2 Throughput Vs Data Size

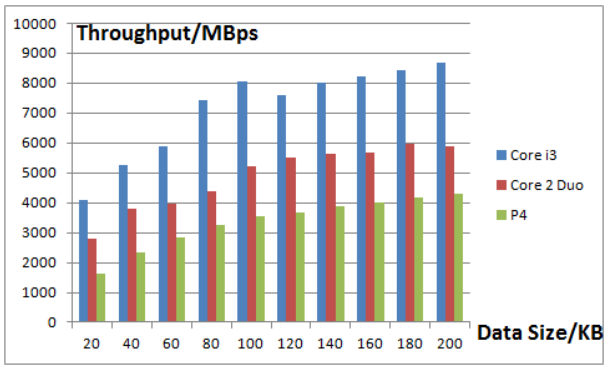

Fig.4 Throughput Vs Data Size (Sequential Implementations)

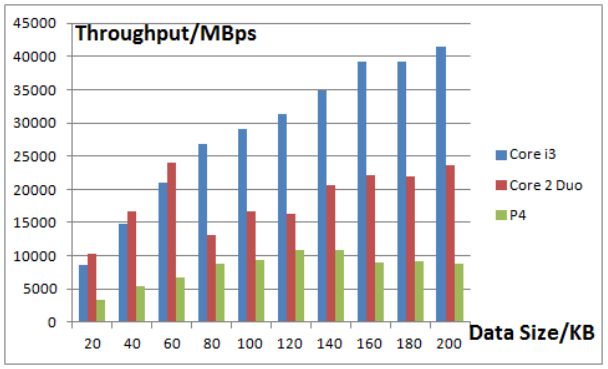

Fig.5 Throughput Vs Data Size (Parallel Implementations)

6. CONCLUSIONS AND FUTURE WORK

By looking at the results the following conclusions can be mentioned:

Parallelism mechanism made RC4 robust on any architecture. Performance or in other words, throughput increment is noteworthy. Anyways, effective improvements can be seen in multicores; especially in Core i3 as it is a quad-core processor. Since Core 2 Duo is a dual-core processor it doesn't have a better performance than Core i3 which is an obvious fact.

In this research Java Executors were used and the JVM decided the no. of threads were used in the parallelism process. According to the available theory and literature it is one of the most cost effective ways of using of Executors in multithreading.

Performance/throughput of RC4 has been boosted due to the mechanism of parallel encryption introduced by this research. Obviously better results can be achieved in Core i5, Core i7 and other high end processors and GPUs. The most important factor is the use of multithreading techniques in multicores! Thus it is obvious that the throughput of RC4 can be immensely increased by the proposed mechanism.

Suggested future work:

1. Improving throughput of block cipher algorithms using multithreading techniques.

2. Evaluation of the robustness of the symmetric key algorithms in multicores and GPUs.

3. Analysis of symmetric key algorithms in multicores and GPUs based on Linux and C language.

REFERENCES

[1] Barnes, A.; Fernando, R.; Mettananda, K.; Ragel, R.; , "Improving the throughput of the AES algorithm with multicore processors," *Industrial and Information Systems (ICIIS), 2012 7th IEEE International Conference on* , vol., no., pp.1-6, 6-9 Aug. 2012 doi: 10.1109/ICIInfS.2012.6304791

[2] Sen Gupta, S.; Chattopadhyay, A.; Sinha, K.; Maitra, S.; Sinha, B.; , "High Performance Hardware Implementation for RC4 Stream Cipher," *Computers, IEEE Transactions on* , vol.PP, no.99, pp.1, 0 doi: 10.1109/TC.2012.19

[3] Zong Wang; Arslan, T.; Erdogan, A.; , "Implementation of Hardware Encryption Engine for Wireless Communication on a Reconfigurable Instruction Cell Architecture," *Electronic Design, Test and Applications, 2008. DELTA 2008. 4th IEEE International Symposium on* , vol., no., pp.148-152, 23-25Jan.2008 doi: 10.1109/DELTA.2008.100

[4] Kitsos, P.; Kostopoulos, G.; Sklavos, N.; Koufopavlou, O.; , "Hardware implementation of the RC4 stream cipher," *Circuits and Systems, 2003 IEEE 46th Midwest Symposium on* , vol.3, no., pp. 1363- 1366 Vol. 3, 27-30 Dec.2003 doi: 10.1109/MWSCAS.2003.1562548

[5] Jun-Dian Lee; Chih-Peng Fan; , "Efficient low-latency RC4 architecture designs for IEEE 802.11i WEP/TKIP," *Intelligent Signal Processing and Communication Systems, 2007. ISPACS 2007. International Symposium on* , vol., no., pp.56-59, Nov. 28 2007-Dec. 1 2007 doi: 10.1109/ISPACS.2007.4445822

[6] Dongara, P.; Vijaykumar, T.N.;, "Accelerating private-key cryptography via multithreading on symmetric multiprocessors," *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on* , vol., no., pp. 58- 69, 6-8 March 2003 doi: 10.1109/ISPASS.2003.1190233

[7] Nawaz, Y.; Gupta, K.C.; Gong G.; "A 32-bit RC4-like Keystream Generator," *Cryptology ePrint Archive: Report 2005/175*.

[8] Mousa, A.; Hamad, A.; "Evaluation of the RC4 Algorithm for Data Encryption,"*International Journal of Computer Science and Applications*, vol.3, no.2, pp. 44-56

[9] Lecture notes: www.cs.virginia.edu/~cs201/slides/cs2110-16-parallelprog.ppt

[10] Bloch J, Bowbeer J, Goetz B, Holmes D, Lea D, Peierls T**.** Java Concurrency in Practice. (May 2006). Chapter 6.2