An Implementation of the Efficient Huge Amount of Pseudo-random Unique Numbers Generator and the Acceleration Analysis of Parallelization

Yun-Te Lin[1,2], Yung-Hsiang Huang[1], Yi-Hao Hsiao[1], Yu-Jung Cheng[1], Jih-Sheng Chang[1], Sheng-Wen Wang[1], Fang-Pang Lin[1], and Chung-Ming Wang[1,2]

[1]*National Center for High-Performance Computing, NARLabs, Hsinchu, Taiwan*
[2]*National Chung Hsing University, Taichung, Taiwan*
lsi@narlabs.org.tw

ABSTRACT

Random unique number generator can be used for generating a series of unpredictable and unrepeatable numbers within limited ranges of data and numbers. These numbers are usually distributed equally, random, independent, unpredictable and unrepeatable. A good random number generator has to be effective for a long period and has good statistical distribution and efficient generating performance. This study proposes a computational methodology to generate pseudo-random numbers based on random base polynomial, which uses less memory but generates a great deal of unrepeated pseudo-random numbers. Then this method adopts the multi-thread parallelization to effectively get the benefits of multi-core processors to accelerate the generation of a huge amount of pseudo-random numbers.

Keywords: random unique numbers, pseudo-random number generator, parallelism, multi-thread, multi-core.

I. THE IMPORTANCE OF RANDOM NUMBER GENERATOR AND PARALLELIZATION

This template provides authors with most of the formatting specifications needed for preparing electronic versions of their papers. Margins, column widths, line spacing, and type styles are built-in.

Random unique number generator can generate a series of unpredictable and unrepeatable numbers within limited ranges of data and numbers. These numbers are usually distributed equally, random, independent, unpredictable and unrepeatable, which means that the probability of each generated number is the same. Also, these numbers cannot be deduced from other numbers; in other words, every time the generated numbers have no dependency to each other, and the next generated number cannot be deduced from the previous number [1][2].

A good random number generator has to be effective for a long period and has good statistical quality and efficient generating performance. Repeated random number sequence does not easily occur on a random number with a long period, and it will take a long time to re-circulate the numbers. Good statistical qualities suggest that the random number sequence is independent and with the characteristic of normal distribution. An efficient computing performance means that when using the same random seed, the computer generates the same random number sequence no matter how many times it executes. The algorithms have to be fast, and the less memory it uses, the better [3][4].

Sometimes scientific research and experiments require a great deal of pseudo-random numbers to conduct simulations, for instance, producing a large amount of binary random simulation files (fitting the 0 and 1 normal distribution) or generating a large number of sine/cosine wave to simulate the general norm of electronic signals. In the cloud computing environment which has been very popular, the correlation between physical and virtual machines and resource allocation can also use random number sequence for assigning. In this case, apart from enhancing the security of being tracked, resources can be allocated more evenly. However, when the amount of the random number sequence grows up and cannot be repeated with high randomness, it requires more computing complexity and executing time [5].

In the development of high-performance computing system, because semiconductor manufacturing technology has been affected by physical limitations (including process size, power consumption or cooling problems, etc.), this causes problems for the CPU clock cycle increasing. Hence, CPU producers turn to develop multi-core architecture by putting multiple computing cores into a CPU die to enhance the computing performance of the whole CPU. The development of multi-core architecture helps to improve the efficiency and advantages of the parallel processing, so the parallel processing has been widely researched and used in recent years.

Usually, on single computer, multi-threading can be used to perform parallel processing for the complicated computing. If there are several computers, Message Passing Interface can be used to conduct the parallel processing. Both need to rewrite the original program and usually we will divide the computing and data into smaller subsets, and distribute them to different cores or computing nodes to conduct calculation. Then collect the results from different cores and computing nodes to process, then export the final result. This study uses multi-threading methodology to develop parallel pseudo-random number generator, and analyzes the accelerating status through the designed random number generation when a huge amount of pseudo-random numbers are generated.

The method proposed in this study has faster execution and less memory usage. Yet, when we generate a large amount of random numbers, the computing time will increase with the amount of random numbers. So it WILL TAKE A LONG TIME TO EXECUTE THE COMPUTING. IN THE Last few years, the multi-core technology has been more mature and parallel processing can accelerate the execution speed of generating huge amounts of pseudo-random numbers. In this study, we efficiently use multi-core processors of the computing resources in order to reach the goal of the parallel processing acceleration.

II. THE INTRODUCTION TO THE RANDOM NUMBER SEQUENCES AND BASIC THEORY

The traditional method to generate pseudo-random numbers usually uses the system clock on the computer as the random seed and algorithm to generate random numbers. The random numbers generated through this method will always generate different random numbers with different system time. The advantages of this method are that seeds are easily retrieved but not easily repeated, which is sufficient to provide general randomness of random numbers. However, this method has some shortcomings. For instance, in need of high quality of random numbers, like applications of encryption, statistical analysis or numerical analysis, the randomness of random numbers will not be accurate exactly. Also, the random numbers may repeat when tens of millions of random numbers are being generated. When unrepeated random numbers are generated through the method using time as seeds, the pseudo-random number generator needs to offer an inspecting mechanism to check the repeated random numbers; if a repeated random number appears, the generator has to discard the random number generated this time and re-generate another new random number until it is unique, and then acquires sufficient amount of unrepeated random numbers sequentially. When huge and unrepeated random numbers are generated through this method, with the rising amount of random numbers, it is more possible that the new generated number will repeats, leading to the need of more extra random number generating cycles and more comparing time of random number uniqueness. This wastes a great deal of computing resources [6].

According to different needs and goals, the pseudo-random number generator can be designed to generate different types of derivative random numbers. For example, the random number sequence that generates 0 and 1 can be used to simulate the bit stream of general files; the 1 and -1 random number can be used to simulate vibration wavelet of transmitting signals; the random number sequence that include the natural numbers can be used as the encryption index for the purpose of data hiding; or to generate similar natural random number to simulate Monte Carlo sampling methods [7].

In order to provide pseudo-random number sequence that have huge amount and random enough and unrepeated random numbers, this study proposes a highly efficient pseudo-random number generating method to generate a series of natural random numbers. This method uses the minimum of memory to reach the goal of quickly generating random number sequence and assures that the random number sequence will not repeat. Besides, we use the password that user inputted as the seed of random numbers to assure that typing the same password always can generate the same random number sequence. Finally we conduct the parallel mechanism to enhance the executing speed of the whole huge amount of pseudo-random numbers computing.

III. THE PRNG RANDOM NUMBER GENERATION ALGORITHM

Unlike the traditional pseudo-random number generator to generate unrepeated random number sequence, as Fig. 1 shows, the pseudo-random number generator method offered in this study is to use several combination of unrepeated random number sequence that range from 0 to 31 as polynomial calculation coefficients (the 32 decimal system as the base, which is similar to a two-dimensional array; the size of the internal dimension is 32 and the size of the outer dimension is the amount of layers; and the layer numbers of the outer dimension happen to be exponents of corresponding 32 to the power of numbers). Then we use the random value of every layer as the index, and have it multiplied by 32 exponential rate corresponded to the coefficient (the power number). We continue to recursively combine the numbers of every layer and finally have random numbers of 32 to the power of K. The number demand of the random number sequence depends on the to-be-produced maximum pseudo-random number N, and then use 32 as the base to proceed factoring polynomials to find out the layer number K that can include this maximum pseudo-random number. The series numbers in the layer index also suggest that every random number series need to be multiplied by the power number of 32. For example, the exponential rate in layer 1 is 0, meaning zero power of 32; the exponential rate in layer 2 is 1, meaning first power of 32; and so forth. This method allows to quickly get the unrepeated random number of the K-th power of 32 that is based on 32, and exports the random number sequence that meets the user-specified range. The Fig. 2 explains the principals of how the pseudo-random numbers are generated and the algorithmic method.

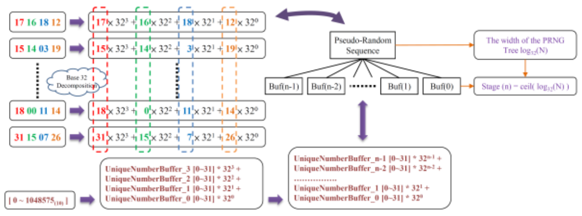
17 16 18 12
15 14 03 19
Base 32 Decomposition
18 00 11 14
31 15 07 26
17 × 32³ + 16 × 32² + 18 × 32¹ + 12 × 32⁰
15 × 32³ + 14 × 32² + 3 × 32¹ + 19 × 32⁰
18 × 32³ + 0 × 32² + 11 × 32¹ + 14 × 32⁰
31 × 32³ + 15 × 32² + 7 × 32¹ + 26 × 32⁰
Pseudo-Random Sequence
The width of the PRNG Tree log32(N)
Buf(n-1)
Buf(n-2)
Buf(1)
Buf(0)
Stage (n) = ceil(log32(N))
[0 ~ 1048575(10)]
UniqueNumberBuffer_3 [0~31] * 32³ +
UniqueNumberBuffer_2 [0~31] * 32² +
UniqueNumberBuffer_1 [0~31] * 32¹ +
UniqueNumberBuffer_0 [0~31] * 32⁰
UniqueNumberBuffer_n-1 [0~31] * 32ⁿ⁻¹ +
UniqueNumberBuffer_n-2 [0~31] * 32ⁿ⁻² +
UniqueNumberBuffer_1 [0~31] * 32¹ +
UniqueNumberBuffer_0 [0~31] * 32⁰

Figure 1. Generating random number via the random number base polynomial method.

As Fig. 2 shows, the first step is to generate the random number base series. Use users' password as the seed of random number to generate the base of random number sequence. In order to make sure the randomness is sufficient and the numbers of seed in every layer can be completely different. First, use users' password and the default 128-characters password to do the dual composition of "replace" and "combine." Then have every character rotational shift forward by one character so that the original first character moves left and becomes the last character and the second character becomes the first. And bring in MD5 hashing function to calculate the character array of 32 hexadecimal values in layer 1. The second layer is the first layer composition in accordance with another rotational shifting character to the left. Also bring in the MD5 hashing function to calculate the 32 character array of hexadecimal numeral system. When all the needed character arrays of hexadecimal numeral system (i.e. the number series of layers) are generated, we can start to generate the random number series of the base.

We apply the "Unique Number" generating functions designed in this study for the character arrays of every layer to change into unrepeated 32 random number series between 0 and 31. Since the range is not too large and there are only 32 numbers, we can compare them to see if there are any repeated numbers and thus the executing speed of generating the unique number is rather fast. To generate the MD5 base random numbers is changing the characters into Long Integer through hexadecimal numeral system as the start position. Then add the number of "Start Position" with "Count" and have it divided by "Key Length", and make the remainder as the "Current Position". Induce the "Current Position" to the index of base series to get the corresponding bit and do another hexadecimal numeral system transformation, and finally get a long "Value". When the "Value" is even, enlarge the "Value", which means to have the length of characters minus the "Value" and then divided by the length of characters, and get the remainder. Through the aforementioned method, we can calculate the "Values" between 0 and 31, and finally check whether there are any repeated numbers. When the random numbers between 0 and 31 are repeated, then we conduct the Bi-Direction Neighboring Diffusion to search for the closest "Value" that is not repeated.

The Bi-Direction Neighboring Diffusion expands to the left or the right in the way of Offset, which means retrieve unrepeated numbers from the neighboring numbers. When the Value minus the displacement value and equals or is bigger than 0, or plus the displacement value is bigger than the length of the characters, then Offset to the left diffusion calculation; if it's the contrary, then Offset to the right diffusion calculation. Through this continuous Bi-Direction Neighboring Diffusion, we can check if the value is repeated and quickly find the unrepeated value. The advantage of this Bi-Direction Neighboring Diffusion is that you can use the minimum times of Offset (or search) to find the usable and unrepeated value and there is no need to examine all the available values from the beginning (the worst condition is O(n)=16, far smaller than O(n)=31).

Last but not least, when the password characters are changed into the base series for calculating random numbers, we can start to generate 32 to the K power pseudo-random numbers. When the random number is bigger, there are more base layers needed, and have exponents of 32 as the key to decide the required maximum number of layers. For instance, when generating one million unrepeated pseudo-random numbers of which the minimum value is 1 and the maximum value is one million, we need at least four bases, i.e. at least generating 32 to the fourth power random numbers, and retrieve all the random numbers with the range.

We can also change the random natural number generated through the pseudo-random number generator into the 0/1 random bit number of Normal Distribution between 0 and 1. 0/1 random bit number means a random number combined from 0 and 1, and equals the 0 and 1 Bit in computer. So it can be used to simulate the archives or the payload of the network packets. There are many ways to changing into 0/1 random bit numbers, the easiest way of which is to do the modulus of the random number and have it divided by 2, and the remainder is 0 or 1, which accords with the characteristics of Normal Distribution.

IV. Parallel Acceleration of the Random Number Generation Algorithm

When the random number generator generates random numbers, all the random numbers have to be saved immediately, and usually arrays are used to store all the random numbers temporarily. To use the method of array requires the need to define an array of fixed size, and then start to save the data in the array. The data cannot be stored outside the array definition, or else there will be a buffer overflow problem. This study adopts the multi-threaded method to do the parallelization of the pseudo-random number generator, and user can decide how many threads to perform the parallel random number generation. Since using the multi-thread, the amount of the random

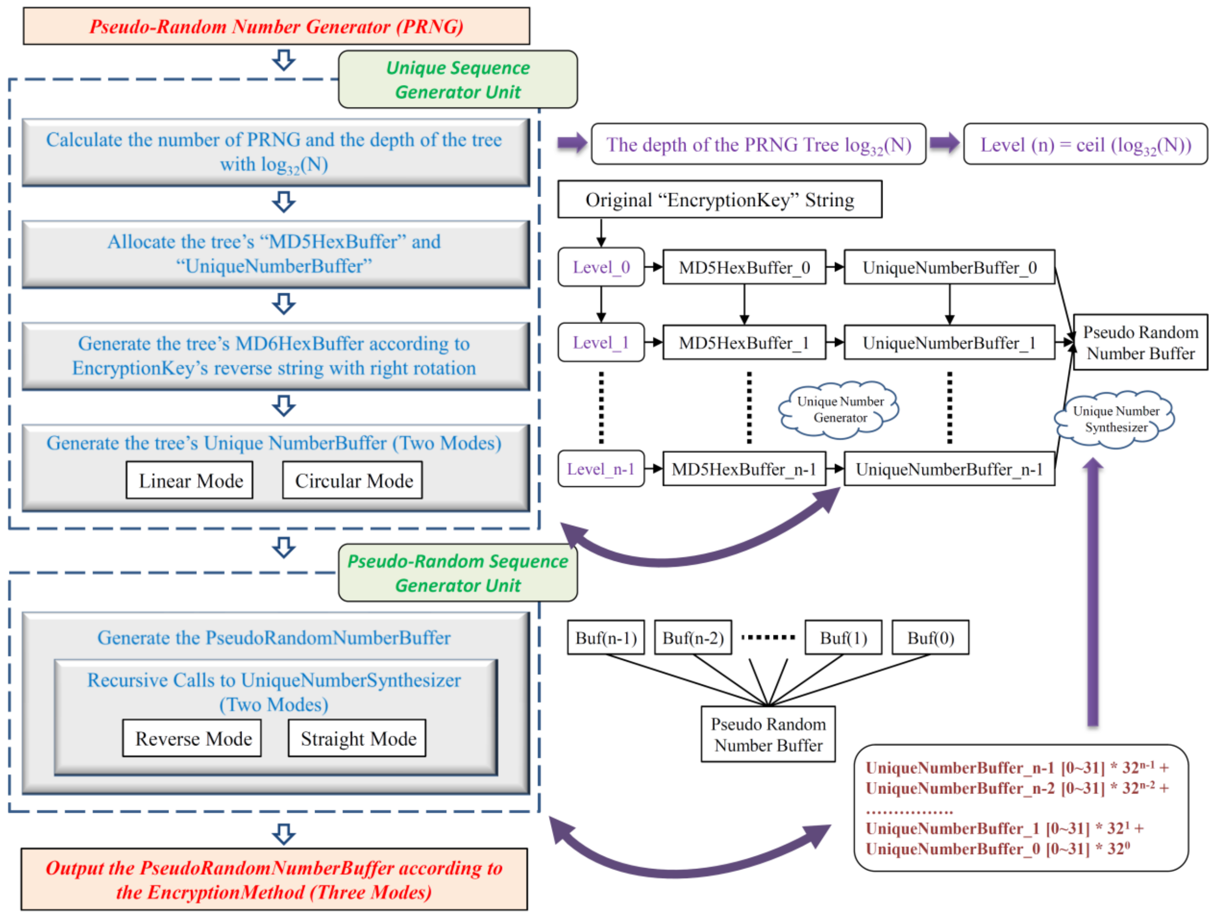
Pseudo-Random Number Generator (PRNG)
Unique Sequence Generator Unit
Calculate the number of PRNG and the depth of the tree with log$_{32}$(N)
Allocate the tree's "MD5HexBuffer" and "UniqueNumberBuffer"
Generate the tree's MD6HexBuffer according to EncryptionKey's reverse string with right rotation
Generate the tree's Unique NumberBuffer (Two Modes)
Linear Mode
Circular Mode
The depth of the PRNG Tree log$_{32}$(N)
Level (n) = ceil (log$_{32}$(N))
Original "EncryptionKey" String
Level_0
MD5HexBuffer_0
UniqueNumberBuffer_0
Level_1
MD5HexBuffer_1
UniqueNumberBuffer_1
Pseudo Random Number Buffer
Unique Number Generator
Unique Number Synthesizer
Level_n-1
MD5HexBuffer_n-1
UniqueNumberBuffer_n-1
Pseudo-Random Sequence Generator Unit
Generate the PseudoRandomNumberBuffer
Recursive Calls to UniqueNumberSynthesizer (Two Modes)
Reverse Mode
Straight Mode
Buf(n-1)
Buf(n-2)
Buf(1)
Buf(0)
Pseudo Random Number Buffer
UniqueNumberBuffer_n-1 [0~31] * 32^{n-1} +
UniqueNumberBuffer_n-2 [0~31] * 32^{n-2} +
....
UniqueNumberBuffer_1 [0~31] * 32^{1} +
UniqueNumberBuffer_0 [0~31] * 32^{0}
Output the PseudoRandomNumberBuffer according to the EncryptionMethod (Three Modes)

Figure 2. The process of generating random numbers designed in the study.

numbers generated within the range of random number demand by each thread will be uncertain, it is hard to define an array effectively that fits the size to store the random numbers of this thread. This study proposes four different methods to solve the random number storing problem of every thread, and compare these four methods to see the advantages and shortcomings.

The first method is to use link-list to increase or decrease the temporary space of storing random numbers according to the need. In order to use the minimum memory, this study has designed the simplest node data structure, which aims to use the least memory. In the node structure, define a Long integer variable for storing random number and define a Pointer for pointing to the next Node address. This Pointer variable can realize the Single link-list way to store the next generated random number; when a new random number is generated, then a Node is added to store this random number. The advantage of Single link-list is that it can dynamically increase the length of link according to the amount of random numbers, but the shortcoming is that every node requires extra space recording the address of next random number node. Generally speaking, in the 32-bit operating system, every node requires four bytes to store the memory address of next node, and 64-bit operating system needs eight bytes (it also differs based on different computer architecture and compiler). Therefore, the link-list method to store random numbers costs more memory space than the array method.

The method regarding the link-list requires more extra memory, and hence use array to temporarily store the random number sequence, and dynamically adjust the size of memory of this array through realloc() function in C/C++ to increase the size of the array to prevent from the buffer overflow situations. The other three methods respectively use different call timing design of realloc() function. The first one is that whenever there is a new random number, it will call the realloc() function to add the array by 1 in every thread. The second is that configure an array in every thread according to the amount of all random numbers; after the random numbers are generated, realloc() function is called to reduce the size of the array to accord with the amount of random numbers that are actually generated. The third design is that after all the random numbers are divided by the amount of threads, and there will be an average number p (p=N/m). Each thread will be assigned to the array of the p, which means all the arrays of the random numbers are distributed to every thread. When the amount of random numbers exceeds the preconfigured size of array, the first method will be adopted to increasing the array through realloc() function. On the contrary, if after the random numbers are generated and the amount of random numbers is less than the size of the array, then the second method will be adopted to decreasing the array to meet with the actual size of the array of the random numbers.

The aforementioned three methods to dynamically adjusting the array to temporarily store random numbers have their own advantages and shortcomings. The advantage of the first method is that it can completely and dynamically adjust the size of the array according to the amount of random numbers, and always maintain the optimal memory usage. However, since realloc() function is required to adjust the size of the array whenever random numbers are generated, to re-allocate the virtual memory frequently is necessary, which leads to more extra time on the reallocation of memory. And the executing number of re-allocating the size of memory approximately equals the amount of random numbers (N). The advantage of the second method is that the realloc() function is used the least, the number of which equals the amount of multi-thread m, but the shortcoming is that in the beginning it will occupy the virtual memory with m arrays of all the random numbers (m*N). The third method combines the advantage of the first with that of the second; the occupied total virtual memory space in the beginning is the size of the array of the total amount of random numbers N (m*p=N). The realloc() function only appears when the amount of random numbers exceeds the size of the array within the threads and when reducing the size of the array, so it is used far less than the first method.

The random number generator in this study uses polynomial algorithm, and the main structure rests on the multi-layer and fixed-length random number base array (Length =32), and therefore there are several method to taken the computing apart when processing the parallelization. The most basic method is parallel cutting, which aims to do the partition of the random number computing base in Layer 1. As Fig. 3 shows, when using the 4 threads to do the parallel computing, have the random number base array in Layer 1 cut into 4 sets of random number array (each set has 8 random numbers), and respectively bring in the sets of random numbers and the random number base array of the lower layers (from Layer 2 to Layer K) to every thread to do the computing. After finishing the computing, have the random numbers from each thread combined according to the series to generate all the pseudo-random number sequence. Yet, the shortcoming of this method is that when the amount of threads cannot be divided by 32, then all the computing cannot be equally distributed to each thread. In addition, since each layer has only 32 random numbers, it can only use the 32 threads at most.

Given that there are several restrictions regarding the first method of parallel computing partition, this study adopts the method of equally distributing the total amount to do the computing partition. When doing the partitioning of equal distribution, have the combinations of all the random number computing

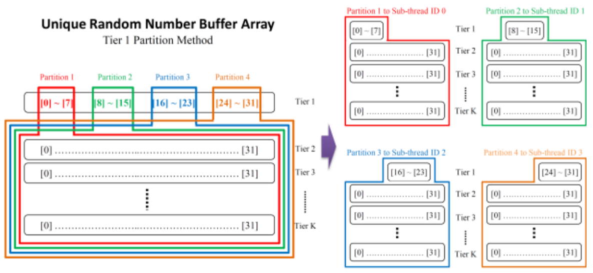
Unique Random Number Buffer Array
Tier 1 Partition Method
Partition 1
Partition 2
Partition 3
Partition 4
[0] ~ [7]
[8] ~ [15]
[16] ~ [23]
[24] ~ [31]
Tier 1
[0] .... [31]
Tier 2
Tier 3
Tier K
Partition 1 to Sub-thread ID 0
Partition 2 to Sub-thread ID 1
Partition 3 to Sub-thread ID 2
Partition 4 to Sub-thread ID 3

Figure 3. The partition of random number sequence base in layer 1.

base array cut and distribute the cut range of the array to each thread to do the computing. The combinations of all the random number base arrays equal the computing number of times of all the random numbers; if the random number base array of 3 layers is used, then the total number of times of computing the random numbers is 32 to the third power. After cutting and equally distributing, the computing range of each thread appears to be a vertical partition. Take 4 threads for example, the array computing range of the first thread starts computing from [the first in Layer 1/ the first in Layer 2/ the first in Layer 3] and circulate the computing to [the eighth in Layer 1/ the thirty-second in Layer 2/ the thirty-second in Layer 3] (equivalent to multidimensional arrays [0][0][0] ~ [7][31][31]). The computing range of the second thread starts from [the ninth in Layer 1/ the first in Layer 2/ the first in Layer 3] to [the sixteenth in Layer 1/ the thirty-second in Layer 2/ the thirty-second in Layer 3] (equivalent to multidimensional arrays [8][0][0] ~ [15][31][31]). Nearly every thread can get to 8 * 32 * 32 random number computing combinations, i.e. approximate amount of random numbers generated. Fig. 4 shows the diagram of the aforementioned partition method of equal distribution.

Fig. 5 shows the detailed executing progress of the method of equally distributing the total amounts. According to the demanding base layer numbers, have the total amount of random number calculated and equally distribute the total amount based on the amount of threads. After that, every thread gets the computing range among the total amount, and the computing ranges turns into the starting position and ending position of the array, and then it is the computing range of an array. Every thread with different array ranges is calculated to get the random numbers that meets the demand, and send the result to the main thread, and finally main thread collects all of the random number series of every thread to get a complete computing result of random numbers.

The method of equally distributing and partitioning the total amount to cut the computing base arrays can not only effectively solve the problem that the first partition method can only use 32 threads at most, but can also more equally distribute random number computing times to each thread, making the parallelized pseudo-random number generator generate pseudo-random number sequence more efficiently.

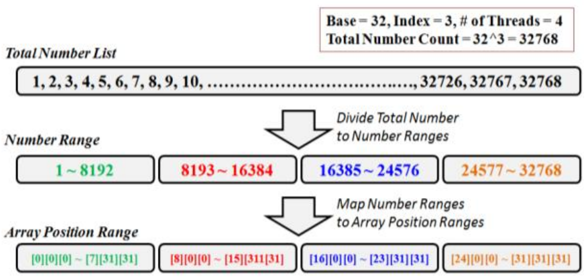

Figure 4. Diagram of partition method of equal distribution of the total amount.

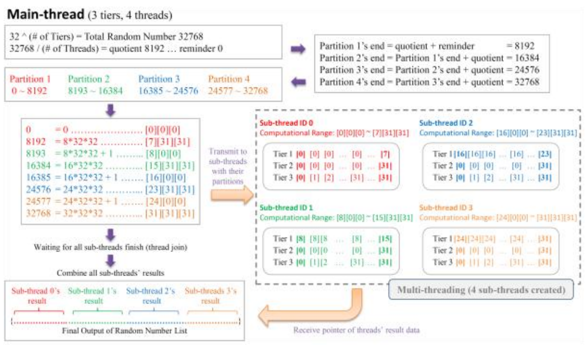
Main-thread (3 tiers, 4 threads)
Transmit to sub-threads with their partitions
Waiting for all sub-threads finish (thread join)
Combine all sub-threads' results
Multi-threading (4 sub-threads created)
Receive pointer of threads' result data
Final Output of Random Number List

Figure 5. The process of partition method of equal distribution of the total amount (4 threads).

V. The Design of Parallelized Experimental Model and Analysis of Experimental Data

To focus on the pseudo-random number generation and the parallelization designed in the study, we start from one thread to gradually increase the amount of threads (increasing four threads every time) to test the speeding effect of its generation of random numbers and test the using rate of memory of the four methods of temporarily storing random numbers. The experimentation is divided into two parts, multi-threading acceleration test and memory usage amount test. (1). During the experiment of the multi-threading acceleration test, we respectively test the four methods of temporarily storing random numbers and test the amount of random numbers, of which the number of layer of the random number base is 5, 6 and 7, and then take down the average time of each number of layers of generating random numbers. (2). During the memory usage amount test, we use 18 threads to respectively test memory usage amount when the four methods of temporarily storing random numbers generate one billion (1G) pseudo-random numbers, and the sampling interval is 1 second. Table 1 shows the hardware specifications of the test host computer and the experimental design model.

As for the part of multi-threading acceleration test, first of all, we conduct the test on the original pseudo-random number generator (without the multi-threaded parallelization). This pseudo-random number generator runs the usual single-thread program and uses arrays to temporarily store all the generated random numbers. Table 2 is the result data of the time of generating pseudo-random numbers. If the index is the same, then the time of generating random numbers is practically the same. The rightmost column is the average time of the random number generation of each layer.

Table 3 demonstrates the result data of speeding test of the parallelized pseudo-random number generator designed in this study. The left column is the amount of the threads, and we start from one thread and gradually increase four threads to test until reaching 60 threads. In the test of every thread amount, we also test the links and the three method of temporarily storing random numbers which use arrays. To focus on all the random numbers in every random number base layers (index=5,6,7) and test the time they are generated, and then calculate to get the average time of random number generation of this layer. The experiment data do not include the

TABLE 1. HARDWARE SPECIFICATIONS OF THEST HOST COMPUTER AND THE MODEL OF EXPERIMENT.

Specification of the computer:	
CPU	Intel(R) Xeon(R) E7530 CPU @ 1.87GHz (true 24 cores CPU, HT enabled)
Memory	529,322,584 KB
OS	Linux 2.6.32-5-amd64 x86 64 GNU/Linux
Preparation operations:	
1.	Clear the Linux cache - "echo 3 > /proc/sys/vm/drop_caches"
2.	Flush the disk cache - "sync"
The number of base layers and the amount of demanding random numbers:	
1.	Multi-thread acceleration test: range of random numbers between 1 ~ following values respectively - 5 layers {1048577, 2097152, 4194304, 8388608, 16777216, 33554432} - 6 layers {33554433, 67108864, 134217728, 268435456, 536870912, 1073741824} - 7 layers {1073741825, 2147483648, 4294967296}
2.	Memory usage test: range of random numbers between 1 ~ 1,073,741,824
Multi-thread stepping interval:	
	1, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60
Methods of temporarily storing random numbers:	
1.	Link List: Single Direction
2.	Array Method 1: Reallocate Array Size Random Number N Times
3.	Array Method 2: Reallocate Array Size T Threads Times
4.	Array Method 3: Reallocate Array Size 1 Time or size exceed N Times
Memory usage amount:	
1.	Run the "ps -au" command to get the values of "VSZ" and "RSS"
2.	Use 18 threads to respectively test memory usage amount
3.	Sampling interval is 1 second
4.	Take down the memory usage since the threads generated and the computation finished

random number layers that are 2,3,4; when the random number base layers are four and the lower layers, the generation time is less than 1 second and thus cannot effectively have the speeding effect. About the characteristics of multi-threading acceleration curve, please refer to Fig. 6, Fig. 7 and Fig. 8.

Fig. 6 shows the graph of the average time of generating random number of each thread amount when using 5-layers random number base. When using the link-list as the method to temporarily store random numbers, if the amount of threads is 24, it will reach the best speeding effect; but when the amount of threads is 28, the speed will start to decrease, and then the random number generation time of each thread will be half of time one thread (the average time of one thread is 18 seconds, and the average time of 36 to 60 threads is 9 seconds). When using the first array storage method, the acceleration effect of multi-threads will stop accelerating after using approximately 12 threads. Compared to the previous two results, the second array method and the third array storage method have better multi-thread acceleration effect but stop accelerating after the thread usage amount reaches 24.

Fig. 7 is the graph of the average time of generating random number of each thread amount when using 6-layers random number base. Regarding the link-list, when the amount of threads is 16, it reaches the highest acceleration effect; when the amount of threads is more than 16, the acceleration will decrease. In the first method of array storage, when the amounts of threads are 4 and 8, the speed significantly decreases to be slower than 1 thread by one time. When the amount of threads is 16, the acceleration effect is the most obvious, but when the

TABLE 2. THE RESULT DATA OF ORIGINAL PSEUDO-RANDOM NUMBER GENERATOR

# of index	**Number of values**		**Generation times**	
			Times (sec)	***Avg.***
5	32^4+1	1,048,577	17.442	17.471
	32^4*2	2,097,152	16.973	
	32^4*4	4,194,304	17.375	
	32^4*8	8,388,608	17.670	
	32^4*16	16,777,216	17.809	
	32^5	33,554,432	17.559	
6	32^5+1	33,554,433	698.639	703.980
	32^5*2	67,108,864	694.572	
	32^5*4	134,217,728	696.448	
	32^5*8	268,435,456	704.053	
	32^5*16	536,870,912	711.341	
	32^6	1,073,741,824	718.831	
7	32^6+1	1,073,741,825	25948.942	26011.56 2
	32^6*2	2,147,483,648	26010.398	
	32^6*4	4,294,967,296	26075.346	

TABLE 3. THE RESULT DATA OF MULTI-THREADING PARALLEL ACCELERATION.

lex = 5 **48577, 2097152, 4194304, 8388608, 16777216, 33554432}**				**Index = 6** **{33554433, 67108864, 134217728, 268435456, 536870912, 1073741824}**				**Index = 7** **{1073741825,2147483648,4294967296}**			
ink List	**Array Method**			**Link List**	**Array Method**			**Link List**	**Array Method**		
	1	**2**	**3**		**1**	**2**	**3**		**1**	**2**	
18.39	18.19	17.95	17.76	730.77	721.86	710.72	710.10	27411.18	27231.94	27206.41	2
4.91	7.57	4.52	4.53	189.25	1525.99	177.73	177.40	6903.46	57735.83	6831.66	
3.02	4.98	2.30	2.30	106.20	1265.22	88.82	88.83	3482.71	10093.27	3419.55	
2.53	3.89	1.76	1.71	84.35	600.17	59.55	59.58	2356.13	16123.72	2279.62	
2.38	3.67	1.13	1.11	83.45	256.46	44.69	44.71	2027.33	12746.84	1786.12	
2.37	3.62	0.92	0.92	157.06	306.62	35.81	36.21	1653.19	7721.28	1431.27	
2.34	3.65	0.77	0.80	249.21	256.58	29.82	30.40	1414.60	11899.90	1199.95	
2.36	3.70	0.85	0.85	270.89	266.41	33.91	33.93	1420.65	6903.53	1366.39	
7.24	3.61	0.76	0.77	293.45	239.03	29.69	29.75	1417.06	6762.04	1202.81	
8.71	3.76	0.68	0.68	302.95	207.93	26.80	26.52	1475.52	6594.05	1073.29	
8.66	3.99	0.75	0.77	300.35	178.74	26.45	26.51	1739.71	5862.46	1108.14	
8.75	4.10	0.70	0.70	323.99	168.95	24.68	24.68	2150.40	5090.79	1035.14	
8.62	4.67	0.73	0.76	331.93	177.47	23.59	23.48	2267.76	5361.10	971.01	
9.02	4.71	0.80	0.79	334.09	157.68	24.52	24.59	2657.33	4861.39	974.48	
9.03	5.06	0.75	0.77	336.50	189.63	24.30	24.63	2756.16	3858.91	976.37	
8.89	5.18	0.77	0.75	338.10	169.64	24.21	24.30	2832.96	4552.38	986.64	

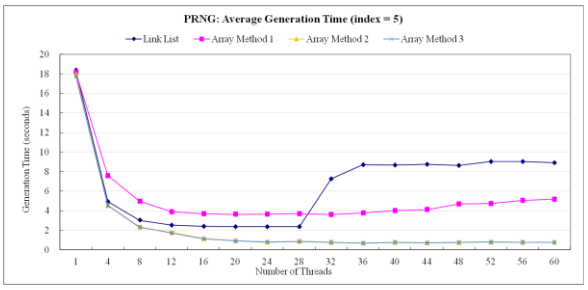

Figure 6. The average time of generating random numbers of 5-layers random number base.

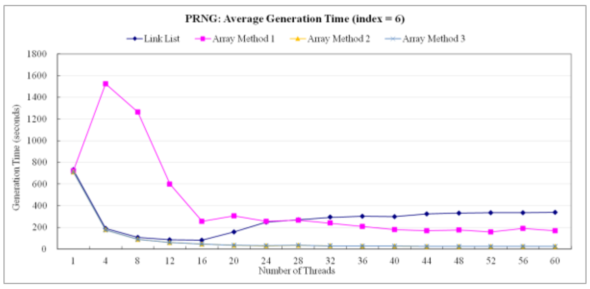

Figure 7. The average time of generating random numbers of 6-layers random number base.

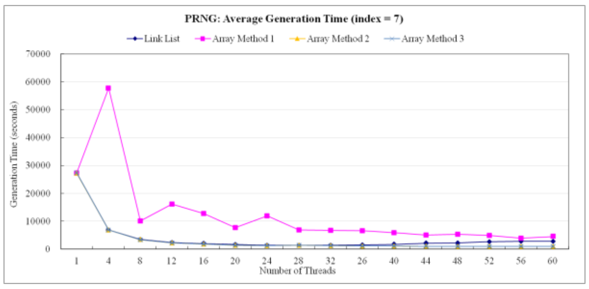

Figure 8. The average time of generating random numbers of 7-layers random number base.

amount of threads is more than 20, the acceleration effect is not clear. In the second and third array storage methods, it is obvious to tell there are two phases of acceleration effect. When the amount of threads is 24, it reaches the best effect in the first phase; when the amount of threads is 48, it gets a faster acceleration effect than the first phase, and the acceleration effect is close to zero after that.

Fig. 8 is the graph of the average time of generating random number of each thread amount when using 7-layers random number base. When using 40 threads, the link-list reaches the best acceleration effect and then begins to slow down. When using the first array storage method, the acceleration effect is the worst and cannot compete with the other three. When using four threads, the acceleration effect is negative; when using 8 threads, the acceleration effect begins to be clear but is not stable. Using the second and the third array storage methods can reach the best acceleration effect of two phases with the 24 threads and 48 threads.

Table 4 shows the four parallelized random number storage methods that have the threads amount of the best acceleration effect and acceleration rate. Using the second and third method of array storage can reach the best acceleration rate, 29.83 and 29.97 in the 6-layers random number base.

The four methods to temporarily storing random numbers proposed in this study helps solve the problem of the inability to accurately allocate random number storage space and allows each thread to temporarily store all the generated random numbers of this thread. However, according to the test result, when generating different amount of pseudo-random numbers, the four methods have different acceleration and even deceleration. When using only 1 thread, there is no significant difference between the four methods' speed of generating random numbers, but when the amount of threads starts to rise, the difference comes to be obvious. The second and third array methods of storing random numbers have nearly the same and best acceleration effect. According to Table 4, when the random number base is 5, the second method and the third method of array storage use 36 threads and can reach the acceleration by 25 times at most. When the random number base is 6, the second method and the third method of array storage use 48 threads and can reach the acceleration by 29 times (nearly 30 times) at most. When the random number base is 7, the second method and the third method of array storage use 48 threads and can reach the acceleration by 26 times at most. Also, it is ostensible that when there are 24 and 48 threads, the acceleration reaches the best effect, and this has to do with our test host computer and meets with our initial expectation that the idealist number of acceleration should be that the number of actual core number is between 24 and 48 simulated by Hyper-Threading.

Combining the acceleration effect of the multi-threaded parallel experiments with the test result of the two parts of memory usage rate, we find that the acceleration effect is the best when using the second method and the third method of array storage. And between these two, the third method has better memory usage amount because the usage amount of VSZ is far smaller than the second method. In other words, if we apply the second array method to some

TABLE 4. THE BEST ACCELERATION RATE OF MULTI-THREADS.

Index		**Link List**	**Array Method**		
			1	***2***	***3***
5	# of threads	24	32	36	36
	Best speed up rate	7.46	4.83	25.35	25.50
6	# of threads	16	52	48	48
	Best speed up rate	8.43	4.46	29.83	29.97
7	# of threads	24	56	48	48
	Best speed up rate	18.38	6.74	26.78	26.80

host computers that do not have big memory space enough, the system might out of memory.

VI. Related Application and Experiment with our PRNG

Fig. 9 and Fig. 10 show an application of the 3D steganography with mesh permutation algorithm in 3D model, the order sequence of the vertices and polygons was generated with our PRNG method.

Fig. 11~14 show four applications of the 2D HDRI (high dynamic range image) steganography with multiple-base algorithm, the embedding sequence of the secure message was also generated with our PRNG method.

VII. Conclusion and Future Work

The pseudo-random number generator in this study can effectively use small amount of memory to generate a great deal of pseudo-random numbers, and make sure there will not be any repeated random numbers. Also, the parallel architecture of multi-threads helps increase the speed of generating pseudo-random numbers, and the acceleration effect can reach 30 times at most. Multi-threads apply the method of equal distribution to assign random number computing to every thread and further increase the efficiency of multi-threads. At the same time, through the equal distribution of array sizes, we adjust the method of temporarily storing random numbers of the array sizes according to the actual usage amount to offer a better memory usage rate. Thusly, the pseudo-random number generator we propose has the three major characteristics of memory saving, fast execution efficiency, and the usage of multi-thread parallelization to generate a huge amount of pseudo-random numbers.

This study has finished implementing the method of combining the multi-layer random number base to generate a great deal of pseudo-random numbers and the multi-thread version of pseudo-random number generator. In the future, we can further the study to see how to distribute the computing more equally through parallelized partition and how to reach the allocation optimization of multi-threading dynamic memory.

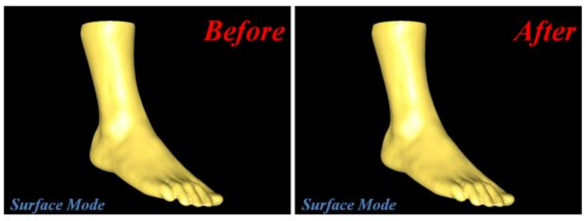

Figure 9. Our PRNG used in the 3D Permutation Steganography (shown in Surface mode).

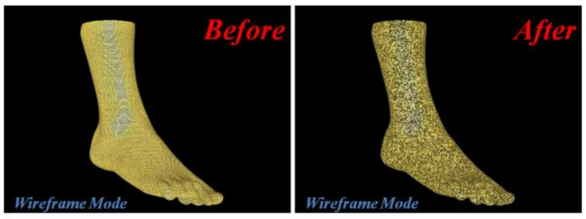

Figure 10. Our PRNG used in the 3D Permutation Steganography (shown in Wireframe mode).

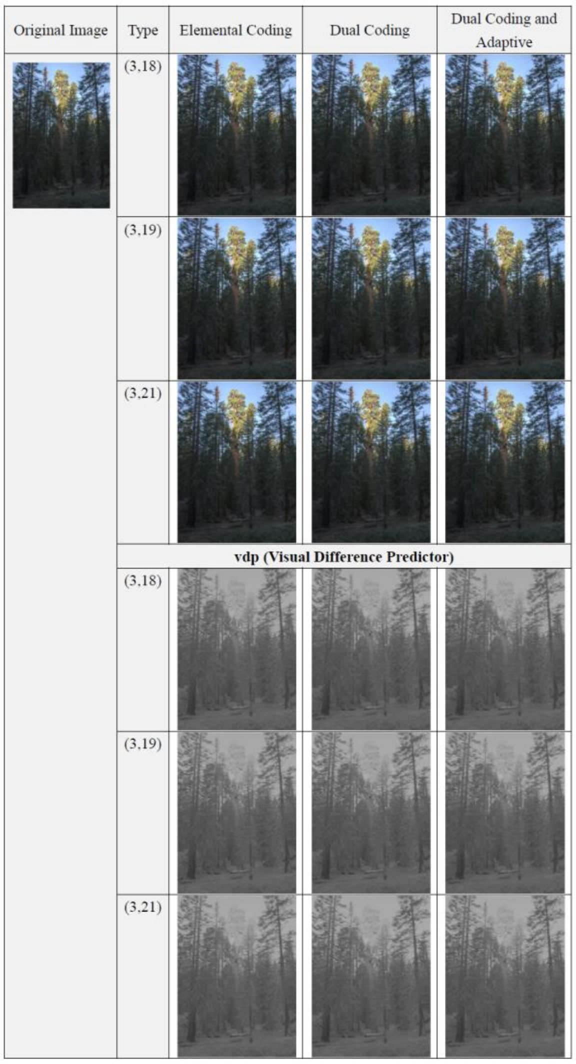

Figure 11. Our PRNG used in the 2D HDRI Steganography.

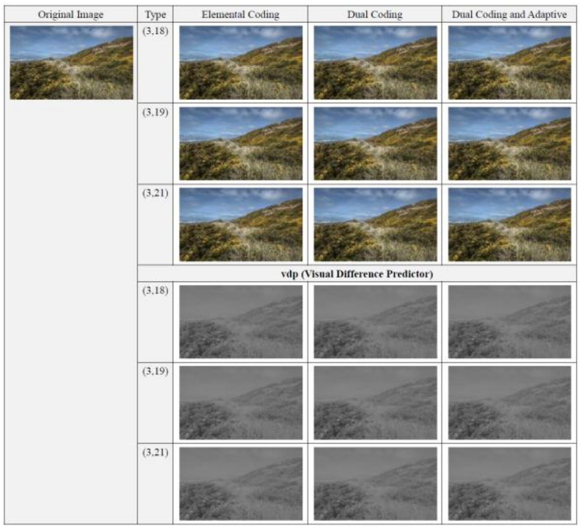

Figure 12. Our PRNG used in the 2D HDRI Steganography.

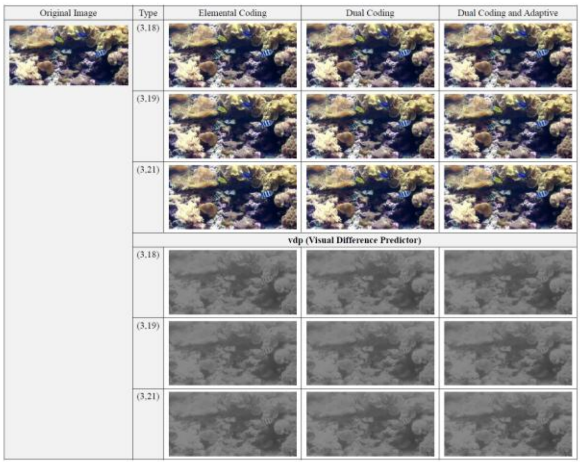

Figure 13. Our PRNG used in the 2D HDRI Steganography.

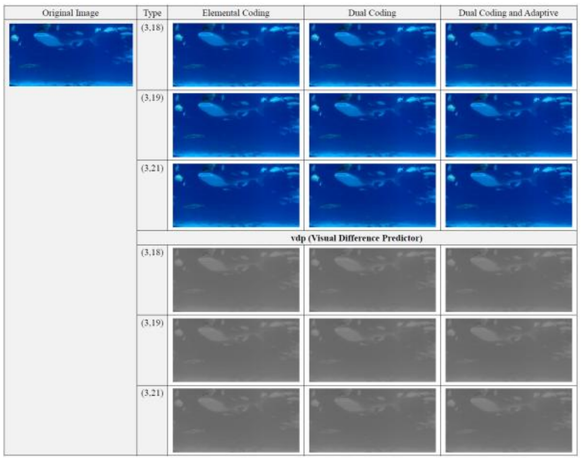

Figure 14. Our PRNG used in the 2D HDRI Steganography.

REFERENCES

[1] A. De Matteis, S. Pagnutti, "Long-range correlations in linear and non-linear random number generators", Parallel Computing 14, 1990, pp. 207-210.

[2] David K. Gifford, "Natural Random Numbers", Laboratory For Computer Science, Massachusetts Institute of Technology, 1998.

[3] Charles W. O'Donnell, G. Edward Suh, and Srinivas Devadas, "PUF-Based Random Number Generation", Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004.

[4] P. Hellekalek, "Good random number generators are (not so) easy to find", Mathematics and Computers in Simulation 46, 1998, pp. 485-505.

[5] P. L'Ecuyer, “Random number generation”, In Jerry Banks (Ed.), Handbook on Simulation, Wiley, New York, 1997.

[6] D.E. Knuth, “The Art of Computer Programming”, Vol. 2, Addison-Wesley, Reading, Mass., 2nd ed., 1981.

[7] H. Niederreiter, “Random Number Generation and Quasi-Monte Carlo Methods”, SIAM, Philadelphia, 1992.