A PARALLEL RANDOM NUMBER GENERATOR FOR SHARED MEMORY ARCHITECTURE MACHINE USING OPENMP

Sayed Ahmed
Department of Computer Science
University of Manitoba, Canada
email:sayed@cs.umanitoba.ca

Rasit Eskicioglu
Department of Computer Science
University of Manitoba, Canada
email:rasit@cs.umanitoba.ca

Lutful Karim
Department of Computer Science
University of Manitoba, Canada
email:lkarim@cs.umanitoba.ca

ABSTRACT

The fields of probability and statistics are built over the abstract concepts of probability space and random variables. This has given rise to elegant and powerful mathematical theory but exact implementation of these concepts on conventional computers seems impossible. In practice, random variables (called pseudo-random numbers) are simulated using deterministic algorithms whose behavior is very hard to distinguish from that of their truly random counterparts. Finding high-quality and efficient algorithms for random number generation on parallel computers is even more difficult. One of the reasons good parallel random number generators are so hard to create is that any small correlations that exist in the sequential generator may be amplified by the method used to distribute the sequence among the processors, producing stronger correlations in the subsequences on each processor. Inter-processor correlations may also be introduced. Also, the method to specify the seeds for each processor is greatly important, since any correlations between the seeds on the different processors could produce strong inter-processor correlations. However, we found the random number generator by L'ecuyer based on the combination of four linear congruential generators**(LCGs)** is a good example for both sequential and parallel implementations having a large domain and inherently parallel property where each processor is assigned to generate one or more sequence independently and our parallel implementation generates the same random number sequence as the sequential implementation. We mainly present the issues related to the parallel implementation of this algorithm using a shared memory workstation in OpenMP. In the implementation, multiple virtual generators work in parallel. Our solution is scalable with the number of processors, and provides locality also maintains all properties of its sequential implementation with significant absolute and relative speedup, and significant efficiency and iso-efficiency.

KEY WORDS

Random Number, Random Number Generator, Shared Memory Machine, OpenMP

1 Introduction

Random number generators are extensively used for computer simulations in computational science and **engineering [3]**. Computer simulations provide a sound understanding of many real systems and hence play significant role to design, implement, even to modify an existing system. In urban planning, in transport scheduling, in production engineering, in job scheduling, in designing slot machines, in study of nuclear reactors and in many other fields computer simulations are of significantly important. In computer simulations random numbers have a significant importance to simulate various stochastic input parameters. In large simulation systems, the number of input parameters are huge. Hence, for larger simulation systems the random number generator should have a large domain to support huge number of input parameters. Besides, to understand the behavior of any system huge number of simulation runs are needed with parameter values from a large and variable domain. Specially for Monte Carlo simulations huge number of simulation runs are needed when the system is large. Hence, random number generator must have larger domain to support huge number of simulation runs with different parameter values for each run. Moreover, to handle huge systems and to minimize total simulation time today simulation programs are executed in super fast multi-processor parallel computers. Hence, an efficient parallel random number generator having a large domain is of great importance in computer simulations.

Simulation programming languages as well as general purpose programming languages usually provide a random number generator. The package usually provides a single generator where multiple sequence (to support multiple simulation run) of random numbers are generated where each sequence provides data for multiple input parameters in a single simulation. These generators use the same generator but different starting seeds in the main sequence. L'Ecuyer and cote [7] proposed a random number generator that uses 32 generators (virtual) where the seeds for each generator is 2^{50} values apart. Hence, each generator has 2^{50} values. The algorithm is based on the combined linear generators having period length 2^{61}. Each generator again is split into 2^{20} segments (V) and 2^{30} offsets (W). When a generator generates random numbers it uses a pointer to move from one segment to another segment within the

same virtual generator and takes next offset value as the next random number. However, only 32 generators and the small number of segment and offset values are not adequate for many huge current and future applications.

To cope up with huge simulations based on the previous algorithm L'Ecyer and Andres [6] later proposed another package having more robust backbone generators which have period length 2^{121} and improved structural properties. As the period length is much larger, the number of segments and offsets can also be much more than previous. Besides, the choice of the number of segments and offsets are now on the user so they are flexible to choice the segment and offset count hence the total number of generators is user dependent. Users can select generator number based on the number of simulation runs, simulation volume, and simulation nature. Each generator is VW values apart. The seed of the first generator is specified using a four value vectors. The other seeds of the other generators are automatically generated using this seed.

In this paper, we presented our parallel implementation of the algorithm by L'Ecuyer and Andres [6] using OpenMP. OpenMP is a multi-threaded parallel programming interface comes as an extension to C/C++. OpenMP uses light weight thread where each thread is assigned a particular responsibility and the threads work in parallel with each other. The threads share the common data structures among them if needed. OpenMP provides exclusive excess to a data item by the threads to provide data consistency [11]. In our parallel random number generator each thread executes a different generator (virtual) where each generator has a different domain and seed values. This is the same as sequential implementation and hence parallel algorithm can not introduce any new correlation. We tested our algorithm in a 8 processor shared memory parallel computer. We calculated the absolute and relative speedup, and efficiency and iso-efficiency of our implementation.

In the following sections, first, we provided related works in section 2. In section 3, we provided the theory behind combined linear congruential generators. Based on the theory, a sequential algorithm is provided in section 4. In section 5, parallelization issues are presented also a parallel algorithm is proposed. In section 6, the parallel aspects of our implementation are explained. Finally, the paper concludes with conclusions and future works in section 7.

2 Related Works

Extensive research has been done on the generation of random numbers. The principal algorithms used for sequential random number generators are Linear Congruential Generator, Lagged Fibonacci Generator, Shift Register Generator, Combined Linear or Multiplicative Generators [2]. The principal techniques used for parallelizing random number generators usually distribute the random number sequences produced by a sequential generator among the processors in different ways such as Leapfrog, Sequence Splitting, Independent Sequences [2]. Random number generators specially for parallel computers are very hard to be trusted. All simulations should be tested with two or more different potential generators and the results should be compared for better accuracy of the simulation results. On a sequential computer good generators to use are Multiplicative Lagged Fibonacci Generators with a lag of at least 127 and preferably 1279 or more, 48 bit or preferably 64 bit Linear Congruential Generators that performs well in Spectral test and has prime modulus, 32 bit or more Combined Linear Congruential Generators with empirically well chosen parameters (such as in [6]) [2]. Extensive study shows that the best suited random number generators for parallel computers are Combined Linear Congruential Generators using sequence splitting, and Lagged Fibonacci Generators using independent sequences with careful initialization to ensure the seed tables on each processor are random and uncorrelated. In this paper, we presented our implementation of a parallel random number generator based on the Combined Linear Congruential Generators provided in [6]. For parallelization, we used sequence splitting among different processors . Our study shows it's among the mostly recommended random number generator for parallel computers. Moreover, to our knowledge there is no such work in OpenMP for shared memory.

3 Random Number Generators based on Four LCGs

The most commonly used generator for pseudo-random numbers is the Linear Congruential Generator. We denote this pseudo-random number generator with the underlying recursion.

$$x_n = ax_{n-1} + b \,(mod\ m) \tag{1}$$

The values a, b and m are pre-selected constants. a is known as the multiplier, b is the increment or additive constant, and m is the modulus. The size of the modulus constrains the period, and it is usually chosen to be either prime or a power of 2.

The quality of the generator is strongly dependent upon the choice of these constants. The method is appealing however, because once a good set of the parameters is found, it is very easy to program. One fairly obvious goal is to make the period (the time before the generator repeats the sequence) long, this implies that m should be as large as possible. This means that 16 bit random numbers generated by this method have at most a period of 65,536, which is not nearly long enough for serious applications. Hence, in huge simulations 32, 64 or 128 bit random number generators are used. Another common use is to combine multiple random generators to provide a random number generator of very large period.

By combining two or more linear congruential generators may increase the length of the period and results in other better statistics. Such kind of generators are termed as

combined generators. This paper presents the parallel implementation of the combined generator by L'Ecuyer and Andres. Generally LCGs are best parallelized by parameterizing the iteration process, either through the multiplier or the additive constant. Based on the modulus, different parameterizations have been tried.

Wichman and Hill in [12], and L'Ecuyer [5] proposed two different methods for combining LCGs with distinct prime moduli. L'Ecuyer and Tezuka [9] later found that a combined generator by Wichman and Hill is truely an LCG with modulus equal to the product of the moduli of the individual components. To understand the combination method by L'Ecuyer and Andres consider J LCGS based on the recurrences

$$x_{j,n} = a_j x_{j,n-1} \; mod \; m_j \tag{2}$$

For j=1....J L'Ecuyer and Andres [6] assume that the moduli m_j are distinct primes and that the j^{th} LCG has maximal period length $\rho_j = m_j - 1$. $\delta_1 .... \delta_J$ are arbitrary integers such that for each **j**, δj and m_j have no common factor. They define the two combinations as follows:

$$z_n = (\sum_{j=1..J} \delta_j x_{j,n}) \; mod \; m_{1_i} \; u_n = z_n \; / \; m_1 \tag{3}$$

$$and \; w_n = (\sum_{j=1..J} \frac{\delta_j x_{j,n}}{m_j}) \; mod \; 1 \tag{4}$$

Let

$$m = \Pi_{j=1..J} \; m_j \tag{5}$$

The following theorm is provided in [9]

- "The sequences $\{u_n\}$ and $\{w_n\}$ both have period length $\rho = lcm(\rho_1 .... \rho_J)$ (the least common multiple of ρ_j) each ρ_j is even and the maximum possible value of ρ is $\rho = (m_1 - 1)....(m_J - 1)/2^{J-1}$
- The w_n obey the recurrence

  $x_n = ax_{n-1}$ mod m; $w_n = x_n/m$

  where a can be computed by a formula given in [9] and does not depend on the δ_j
- One has $u_n = w_n + \varepsilon_n$ with $\Delta^- \leq \varepsilon_n \leq \Delta^+$ where Δ^- and Δ^+ can be computed as explained in [9] and are very small when the m_j are very close to each other."

Therefore, the combinations of equation 3 and 4 are a practical way of implementing an LCG with large modulus m and multiplier a. In that case the structural analysis of the corresponding LCG define the criteria to choice equation parameters. The parameters then do not depend on the analysis of individual components.

3.1 Finding a Good Combination

The combined generator based on four LCGs (i.e. four components) should have the following properties [6].

- Each component should be an LCG with prime modulus m_j which is slightly smaller than 2^{31}, multiplier a_j so that $(m_j \quad mod \quad a_j) < m_j/a_j$, and full period length $\rho_j = m_j - 1$
- The moduli should satisfy $m_j < m_{j-1} - 100$ for j > 1 and $(m_1 - 1)/2....(m_4 - 1)/2$ should have no common factor to keep period length of the combined generator equal to the product of the ρ_j divided by 8
- **The combined generator must have a significant quality of its lattice structure [8]. Hence, should have the best possible figure of merit [8]**.

L'Ecuyer and Andres made an intensive computer search to find a combined generator with these properties and with the best possible value of figure of merit. They came up with the following parameters [6].

j	m_j	a_j
1	2147483647	45991
2	2147483543	207707
3	2147483423	138556
4	2147483323	49689

The LCG corresponding to the combined generator has modulus, multiplier and period length

m = 21267641435849934371830464348413044909
a = 5494569482908716143153333426731027229
ρ = (2^{31} - 2)(2^{31} - 106)(2^{31} - 226)(2^{31} - 326) $\approx 2^{121}$

4 Sequential Algorithm

The algorithm works in two phase initialize phase and generation phase. In initialize phase, seed values of all the generators are generated. In generate phase, random numbers are generated. Each generator g has an initial seed Ig (initially generated or assigned), a last seed Lg (the starting point of its current segment) and a current seed Cg (the current state of the generator). When a random number is produced in generation phase by this generator, C_g is advanced to the next state in the generators sequence by changing the segment number. C_g jumps back and forth between the different segments of any given generator and take the next offset value in that segment as the next random number. The sequential algorithm is provided in Figure 1.

5 Parallelization of Pseudo-random number Generator

We used OpenMP for parallelization of the random number generator for a shared memory architecture workstation. OpenMP provides multiple thread parallelism. We distribute the same code to all threads and all threads work as a stand alone program concurrently. The threads run on different processors of a SMP machine. It is also the case

- **Initialize**
  - Initialize segment and offset count. Segment = V = 2^v and Offset = W = 2^w where v >= 30 and w >= 41 and v + w <= 100.
  - Initial seed of generator 0 is set to the vector {11111111, 22222222, 33333333, 44444444}
  - Initialize seed for each generator. For the first generator I_0 the seed is the vector {s[0]…s[3]) following the rule
    1<=s[0]<= 2147483646, 1<=s[1]<= 2147483542,
    1<=s[2]<= 2147483422, 1<=s[3]<= 2147483322
  - The initial seed for the other generators are computed from the seed of the first generator so that seeds of different generators are VW values apart
  - Initialize generators to their initial or any other arbitrary seed to start from
- **Generation**

```
for each simulation/set/generator
    for each variant/item/parameter
        Generate an uniform random number within [0,1]
        Switch to different segment of the same generator
    end for
end for
```

Figure 1. Sequential Algorithm

- **Initialize**
  - Initialize segment and offset count. Segment = V = 2^v and Offset = W = 2^w where v >= 30 and w >= 41 and v + w <= 100.
  - Initial seed of generator 0 is set to the vector {11111111, 22222222, 33333333, 44444444}
  - For the first generator I_0 the seed is the vector {s[0]…s[3]) following the rule
    1<=s[0]<= 2147483646, 1<=s[1]<= 2147483542,
    1<=s[2]<= 2147483422, 1<=s[3]<= 2147483322
  - Initialize generator I_0 to its initial or any other arbitrary seed to start from
- **Generation**

```
for each simulation/set/generator pardo
  if first iteration
      o Initialize seed for each generator based on
        generator/simulation/set ID
      o The initial seed for the other generators are computed from
        the seed of the first generator I0 so that seeds of different
        generators are VW values apart
    for each variant/item/parameter [pardo]
      Generate an uniform random number within [0,1]
      Switch to different segment of the same generator
    end for
end for
```

Figure 2. Parallel Algorithm

the thread number can be more than the number of processors then a processor handles multiple threads in time sharing basis. If it is computation intensive application than if number of threads equals to the number of processors then we get better performance. Otherwise context switching causes more execution time. Our pseudo-random number generator is computation intensive. Ours is a parallelizable problem as at a time we want to generate random numbers for multiple simulations and there are a large number of variables/variates in each simulation.

We assign a thread to generate the variables for a particular simulation/set. We assign a different generator for each simulation. Hence, all threads in different processors run the same code where each thread work for a different simulation/set concurrently. As a result, efficiency increases a lot. The parallel algorithm is provided in Figure 2.

6 Experimental Results

We used C to develop the sequential program and OpenMP library for the multi-threaded parallel program. The methodologies followed for the experiment are given below:

- For every instance of graphs five runs were conducted the runtime is average of these five runs.
- Runs on the same set of inputs were done with number of processors ranging from 1 to 8.
- The experiments were conducted for static scheduling of OpenMP.

We tested our implementation in two different hardware platforms. Experimental results and discussions for both of the platforms are provided below:

6.1 Pentium-3, Linux Platform

The experiments were conducted on an eight processor i686 (Pentium-3) Symmetric Multiprocessor (SMP) Machine. The processor speed is 700.011 MHz. with a cache size 0f 1024 KB and a total memory space of 6 GB.

The speed up we get is optimistic. We calculated the speed up using the following equation:

$$Speedup = \frac{Time\ \ Required\ \ Using\ \ Sequential\ \ Code}{Time\ \ Required\ Using\ Parallel\ Code}$$

We ran our implementation for 1024 to 3072 simulations where each simulation has 1024 variate or input parameters. We vary the thread number from 1 to 8. For 2 threads we got the speed up around 1.8 when the number of the simulation is 3072. But for 4 threads we get speed up near to 3(2.90). And for 8 threads we get a speed up more than 3.6. This shows the scalability of our parallel implementation.

Figure 3 shows the speed up result for 1024, 2048 and 3072 simulations where each simulation has 1024 input parameters. We see for 2048 simulations when thread number is below 4 speedup is lower than 1024 simulations however after that the speedup for 2048 simulations exceeds the speedup of 1024 simulation though it remains lower

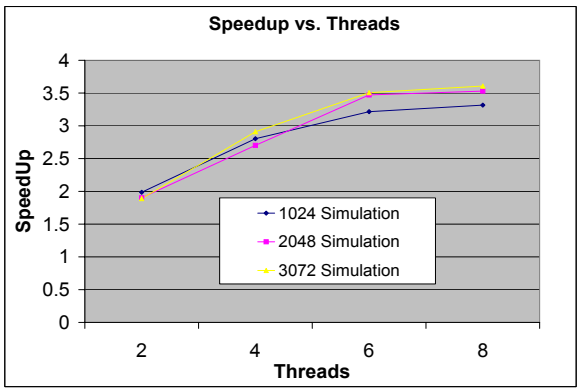

Figure 3. Speed up results for different threads

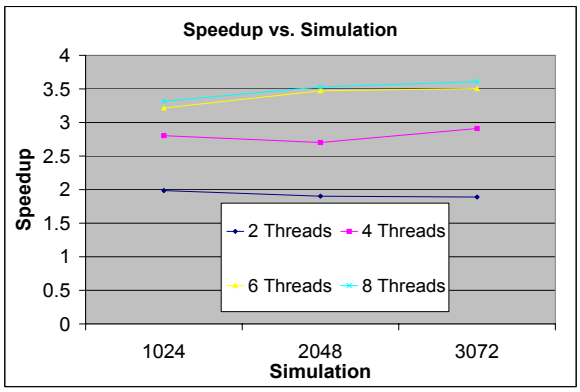

Figure 4. Speed up results for different number of Simulations for fixed threads

than the speedup of 3072 simulations. We get maximum speedup of 3.6 when the number of thread is 8 and simulation number is 3072. This is the highest load in our case. The reason is that, we experiment our algorithm on 8 multiprocessor machine. When there are more work load, the processors can utilize their full capacity. Therefore, it takes relatively less time for larger problems when compared to the sequential version of the same problem. Now it is inferred that the load is balanced to each thread evenly.

Table 1: Efficiency

Simulation Count/Thread	2	4	6	8
1024	0.99	0.70	0.53	0.42
2048	0.95	0.67	0.57	0.44
3072	0.94	0.72	0.58	0.45

Table 1 shows the efficiency of our implementation. We measure efficiency as follows:

$$efficiency = \frac{Speedup}{Thread\ Count}$$

For all of the simulations the efficiency is greater than 0.5 until thread number 6. However, the efficiency goes below 0.5 when the number of threads is 8. The reason behind this when there are more threads thread management becomes important which takes some time and reduces efficiency. Besides, sharing common data structure among threads become more conflicting hence reducing overall efficiency.

Table 2: Execution Time in Sec.

Simulation/Thr.	1	2	4	6	8
1024	15.77	7.94	5.63	4.90	4.75
2048	33.40	17.56	12.36	9.62	9.46
3072	48.49	25.66	16.66	13.83	13.44

From the Table 2, it is clear that the execution time decreases as we increase the number of threads. We can explain the behavior in such a way. When we use only one thread the thread is overloaded. When the number of threads increased the workload is distributed among the threads hence reducing execution times.

Figure 4 shows the improvement of efficiency for larger number of simulations. For simulation count 1024 we got the maximum speed up for 8 threads is 3.31. On the other hand, for 2048 simulations, the maximum speed up is 3.53. The speed up for 3072 simulations is 3.6. This phenomena clearly indicates that the parallel random number generator works better when more works are distributed among the threads. The reason is that when more work is given, the amount of computation increases and the load is distributed among the processors.

6.2 SUN Ultra Sparc-III

The experiments were conducted on machines containing 24, 1050 MHz UltraSparc-III CPUs, 48 gigabytes of memory, a terabyte of disk storage, L1 Cache, and L2 Cache. L1 cache consists of 64 KB 4-way data, 32 KB 4-way instruction, 2 KB Write, and 2 KB Prefetch. L2 Cache consists of 8 MB External On-chip controller and address tags.

The speed up we get for SUN Ultra Sparc-III is highly optimistic. We ran our implementation for 1024, 2048, 3072, 4096, 8192, 10240 simulations where each simulation has 1024 variate or input parameters. We varied the thread number from 1 to 16.

Figure 3 shows the speed up result for 1024, 2048, 3072,4096, 8192 and 10240 simulations where each simulation has 1024 input parameters. We see for the same problem size with the increase of the thread numbers speedup increases significantly. Also with the increase of problem size for the same number of threads speedup also increases in most cases. The speedup indicates our implementation is quite suitable for parallel implementation and pretty usable in parallel simulations to provide significant efficiency. We get maximum speedup of 10.57 when the number of thread is 16 and the number of simulations is 8192. This is the second highest load in our case. The reason is that, we experiment our algorithm on 16 multiprocessor machine. When there are more work load, the processors can utilize their full capacity. Therefore, it takes rela-

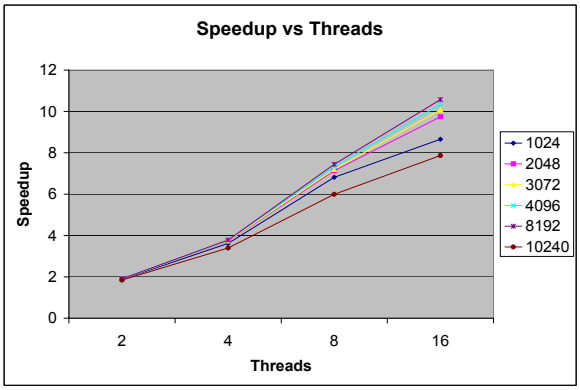

Figure 5. Speed up results for different threads

tively less time for larger problems when compared to the sequential version of the same problem. Now it is inferred that the load is balanced to each thread evenly. However, for the problem size 10240 simulations, speedup somewhat decreases. This might be because of the L2 cache size. Until 8192 simulations cache hit rate is significant. After that when the number of simulations increases cache failure rate increases and speedup decreases.

Table 3: Efficiency

Simulation Count/Thread	2	4	8	16
1024	0.94	0.91	0.85	0.54
2048	0.95	0.93	0.89	0.61
3072	0.95	0.94	0.90	0.63
4096	0.95	0.94	0.92	0.65
8192	0.96	0.95	0.93	0.66
10240	0.92	0.84	0.75	0.50

For simulation number until 8192 and thread number 8 efficiency is almost greater than 90%. With the increase of thread number efficiency is reduced slightly because of the thread initialization, thread termination also due to the context switch. However, the reduction is not that significant. With the increase of problem size for a fixed thread number the efficiency increases as the processors are utilized much more. However, we see for thread number 16 the efficiency reduces much this might happen due to many number of threads and context switching. Also for 16 threads with the increase of problem size efficiency increases until problem size becomes 8192. But when problem size is 10,240 efficiency reduces this might be due to L2 cache size and increase in cache failure rate. Besides, sharing common data structure among threads becomes more conflicting hence reducing overall efficiency.

Table 4: Execution Time

Simulation/Thr.	1	2	4	8	16
1024	1790	952.8	494.4	262.6	206.8
2048	3573.2	1882.2	958.6	500.6	366.8
3072	5357.8	2823.6	1432	744.4	532
4096	7143.2	3758.6	1893.8	974.2	691.2
8192	14278.8	7483.4	3774.6	1918.4	1350.8
10240	18441	9984	5425.6	3076.6	2343.8

From the Table 4, it is clear that the execution time decreases as we increase the number of threads. When we use only one thread the thread is overloaded. When the number of threads increased the workload is distributed among the threads hence reducing execution times.

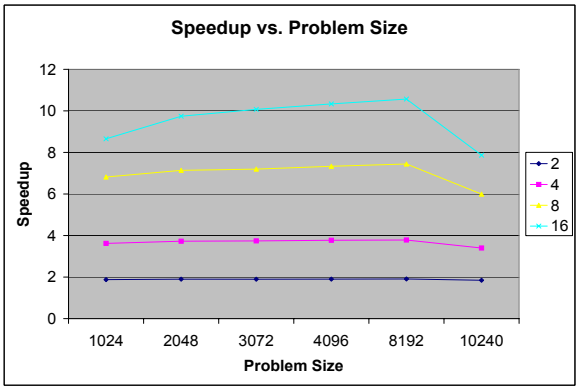

Figure 6. Speed up results for different number of Simulations for fixed threads

Figure 4 shows the improvement of efficiency for larger number of simulations. For simulation count 1024 we got the maximum speed up for 8 threads is 6.81. On the other hand, for 2048 simulations, the maximum speed up is 7.14. The speed up for 3072, 4096, 8192 simulations are 7.2, 7.34, 7.45 respectively. This phenomena clearly indicates that the parallel random number generator works better when more works are distributed among the threads. The reason is that when more work is given, the amount of computation increases and the load is distributed among the processors.

6.2.1 Scalability

Scalability is a measurement that indicates the capacity of a parallel procedure to provide the same performance level when the problem size and the number of available processors grow proportionally [4]. From the Figure 5 it is clearly seen that our implementation is scalable. We started with 1024 simulations and 2 threads. Then we doubled the problem size (2048 simulations) and thread number (4 threads) we get speedup 1.98 times than the speedup of 1024 customers which is almost double. Then we increased problem size to 4 times (4096 simulations) times and thread number to 4 times (8 threads) we get speed up 3.90 times than 32 customers which is almost 4 times. This measure definitely

indicates that our implementation is scalable. However, for 16 threads and 8192 simulations the speedup is 5.63 times than 1024 simulations. In ideal scalable system and algorithm it (5.63) should be almost 8. In this case the problem size is increased 8 times, system size is also increased to 8 times however, as now the cache size is not sufficient it needs some more secondary ram access increasing the execution time. In all the other cases intermediate ram access is absent. Hopefully, proper increase in system capacity would lead around 8 times gain here.

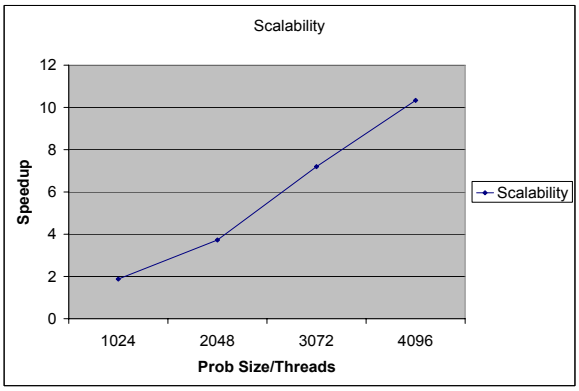

Figure 7. Scalability Mesures

7 Conclusions

In this paper, we presented our parallel implementation of the random number generator proposed in [6]. We found our solution to be scalable with the number of processors and input size. The implementation provides locality with significant speedup and efficiency also maintains all properties of its sequential implementation providing all the same random numbers as its sequential implementation. Future works may focus on the better parallelization to increase speedups. Some empirical tests can be performed to study the randomness of the output. Using our generator with Monte Carlo algorithms [10] used for simulating two dimensional Ising model [1], can be an effective empirical test. Besides, the implementation can be tested with workstations having more processors to better study the scalability issue.

References

[1] B. A. Cipra. An introduction to the ising model. In *Amer. Math*, volume 94, pages 937–959, 1987.

[2] P. D. Coddington. Random number generators for parallel computers. npac.syr.edu/users/paulc/papers/NHSEreview1.1/PRNGreview.ps, April 1997.

[3] C. Drake and A. Nicolson. A survey of pseudo-random number generators. Technical report, Department of Electrical Engineering and Computer Science, University of Michigan.

[4] Crainic T. G. and M. Toulouse. Parallel metaheuristics. In *Kluwer Academic Publishers*, pages 205–251, 1998.

[5] P. L'Ecuyer. Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–749, 1988.

[6] P. L'Ecuyer and T. H. Andres. A random number generator based on the combination of four lcgs. *Mathematics and Computer Simulations*, 44(1):99–107, 1997.

[7] P. L'Ecuyer and S. Cote. Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software*, 17(1):98–111, 1991.

[8] P. L'Ecuyer and R. Coutre. An implementation of the lattice and spectral tests for multiple recursive linear random number generators. *INFORMS Journal on Computing, Circa*, 1997.

[9] P. L'Ecuyer and S. Tezuka. Structural properties for two classes of combined random number generators. *Mathematics of Computation*, 57(196):735–746, 1991.

[10] N. Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

[11] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface*, version 1.0 edition, 1998.

[12] B. A. Wichmann and I. D. Hill. An efficient and portable pseudo-random number generator. *Applied Statistics*, 31:188–190, 1982.