Uppsala Universitet

Parallel and Distributed Programming

1TD070

Project: Monte Carlo Stochastic Simulation Algorithm

By:
Jacob K. Malmenstedt

May 25, 2023

Contents

1 Introduction

A Monte Carlo method is a method of approximating solutions to complex systems in a stochastic way. In contrast to analytical solutions the stochastic methods does not give a precise solution but when a sufficient number of simulations are ran the solutions become more and more precise. For this reason Monte Carlo methods are often more computationally expensive than analytical solutions but does not require that a deterministic solution exist to work. These stochastic methods are also suited to problems with some inherit uncertainty or complex dependencies.

One problem that can be solved using the Monte Carlo method is the Stochastic Simulation Algorithm (SSA), also known as Gillespies algorithm. This algorithm is often used to simulate complex biological systems in a stochastic way.

The SSA works by simulating the system's behavior over time by repeatedly selecting and executing individual reactions or events based on their probabilities. These probabilities are calculated using the rates of change and the current state of the system. The time at which the next event occurs is also determined stochastically.

2 Problem description

The aim of this project is to write code to run the SSA in parallel, using OpenMPI, on Uppsala University's compute cluster UPPMAX.

The code will run N number of simulations of the SSA and then collect the results and write the data for a histogram of the susceptible number of humans at the final simulation time to an output file.

3 Solution approach

The general steps for my solution is the following:

1. Initialize variables and seed the random number generator.
2. Run the SSA in parallel and save the results to a local array.
3. Find the local max and min values.
4. Find global max and min values.
5. Create 20 bins between the global min and max values.
6. Count the number of results in each bin locally.
7. Add local results to the global result.
8. Print timings and time statistics for the SSA
9. Print results to file

In the code the different processors will take an equal amount n, where $n = N/size$, of the simulations and run them completely in parallel. This was a requirement in the assignment instruction but since the SSA is stochastic the processes will complete these simulations at different times and some processes will therefore wait before they can move on. The only thing that they can do while they wait is to find their local max and min values but until global max and min values exist this is as far as the code can run before all processes give their local max and min values. This can be a significant limitation to the performance of the code but one that is rather hard to remove due to the unpredictability of the runtime for the SSA.

Since the simulations take up the majority of the time it is important to optimize this part of the code as much as possible. Therefore a lot of time has been spent on trying to optimize the $gillespieSSA()$ function and it's subfunctions. Even with these optimizations that makes the simulations significantly faster the sheer volume of calculations that needs to be done still contributes to the large time spent on this part of the code.

Some of the attempts at optimizing the code further was unsuccessful. Among these attempts were attempts at substituting the $rand()$ function for a faster, less accurate, pseudo random generator. After no statistically significant benefit was found the choice was made to use the standard $rand()$ function since it is a proven method of generating pseudo random numbers.

Some further potential improvements to this version of the code are discussed in Section 5.2

4 Experiments and Results

The code prints out both the total execution time as well as the communication time, where the communication time refers to the part of the code that processes the simulation results and calculates the output data. For both these values the maximum, minimum and average times are printed.

The code was ran with 2, 4, 8 and 16 cores for 1 000 000, 2 000 000, 4 000 000 and 8 000 000 total simulations. The results are presented in tables 1, 2 and figures 1,2, 3, 4.

Table 1: The average total runtime (in seconds) for different number of simulations and cores.

	Number of cores			
Total simulations	16	8	4	2
1000000	93.1167	175.2665	347.6363	721.4737
2000000	185.4496	347.5732	687.0017	1417.798
4000000	375.6268	708.4514	1361.734	2871.611
8000000	736.7305	1383.735	3474.439	5571.847

Table 2: The minimum communication time (in seconds) for different number of simulations and cores.

	Number of cores			
Total simulations	16	8	4	2
1000000	0.0038	0.0066	0.0129	0.0267
2000000	0.0073	0.0133	0.0247	0.0528
4000000	0.0138	0.0265	0.0492	0.1051
8000000	0.0270	0.0513	0.1037	0.2003

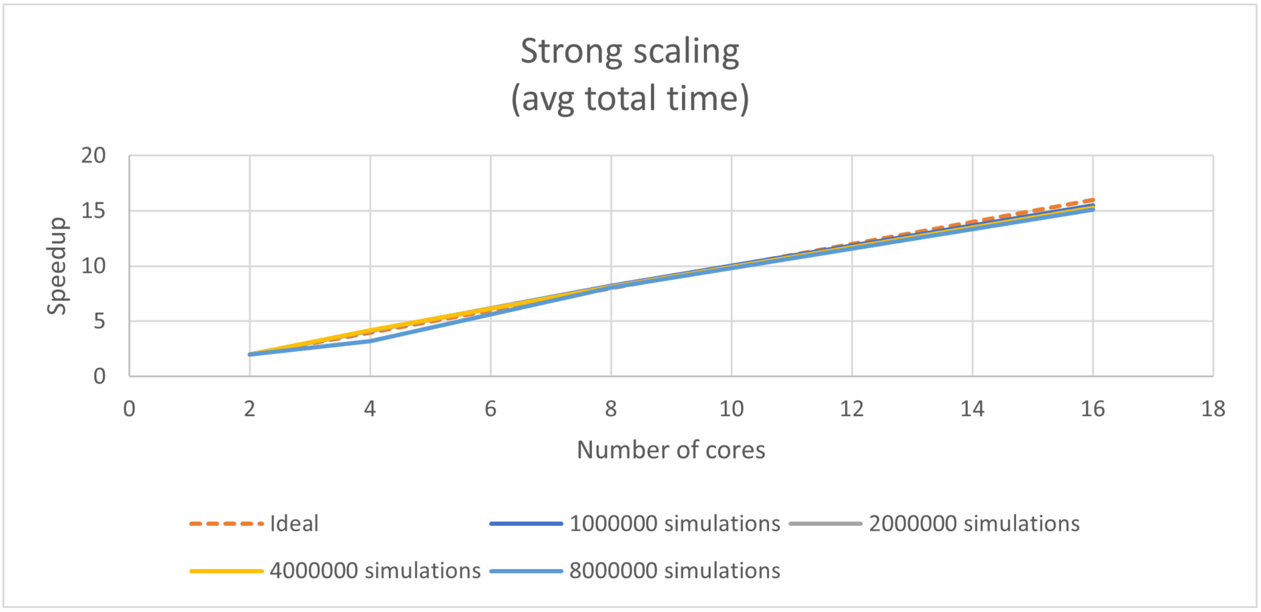

Figure 1: Plot of the strong scalability for different number of simulations when considering the total runtime.

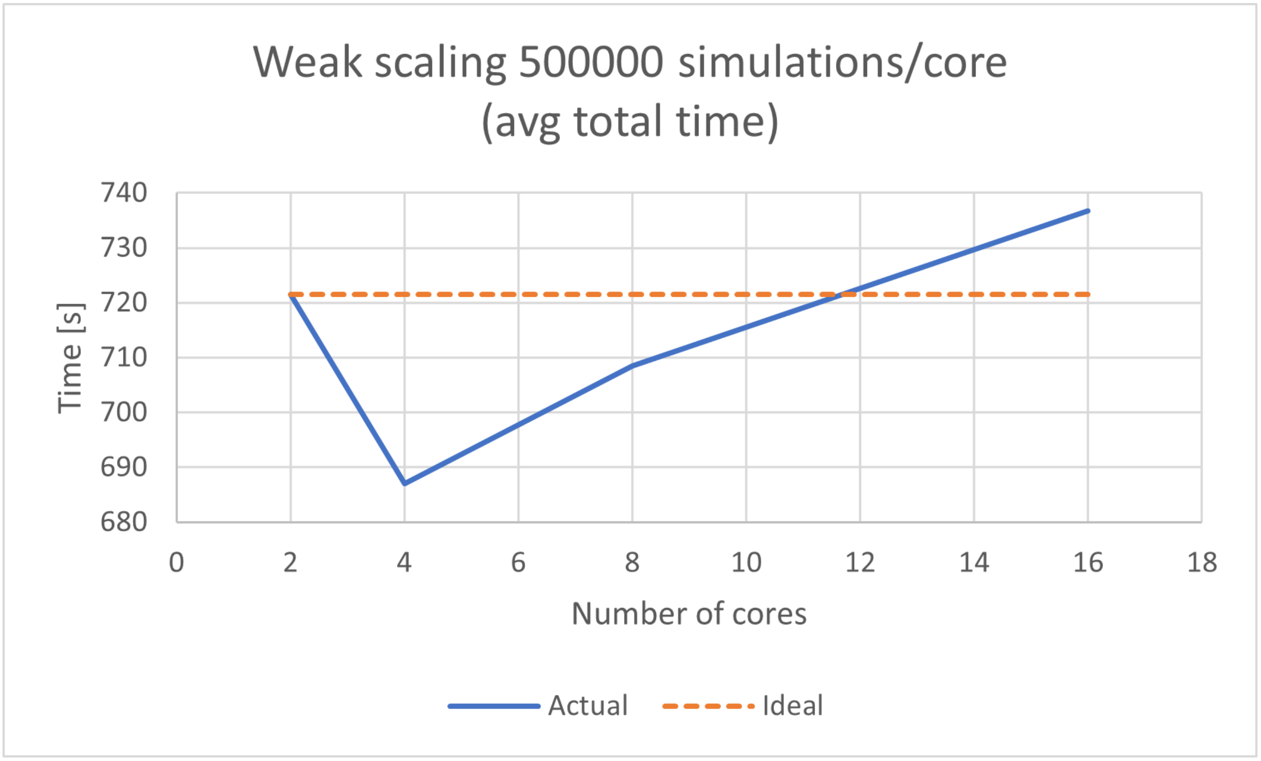

Figure 2: Plot of the weak scalability with 500 000 simulations per core, when considering the total runtime.

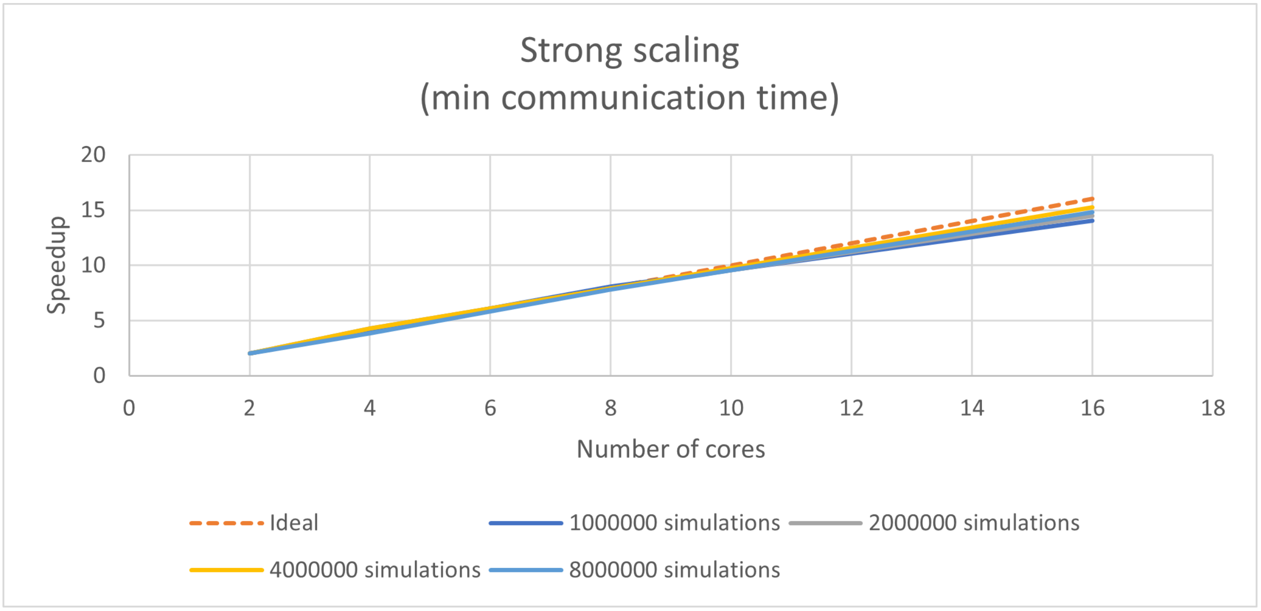

Figure 3: Plot of the strong scalability for different number of simulations when considering the minimum communication time.

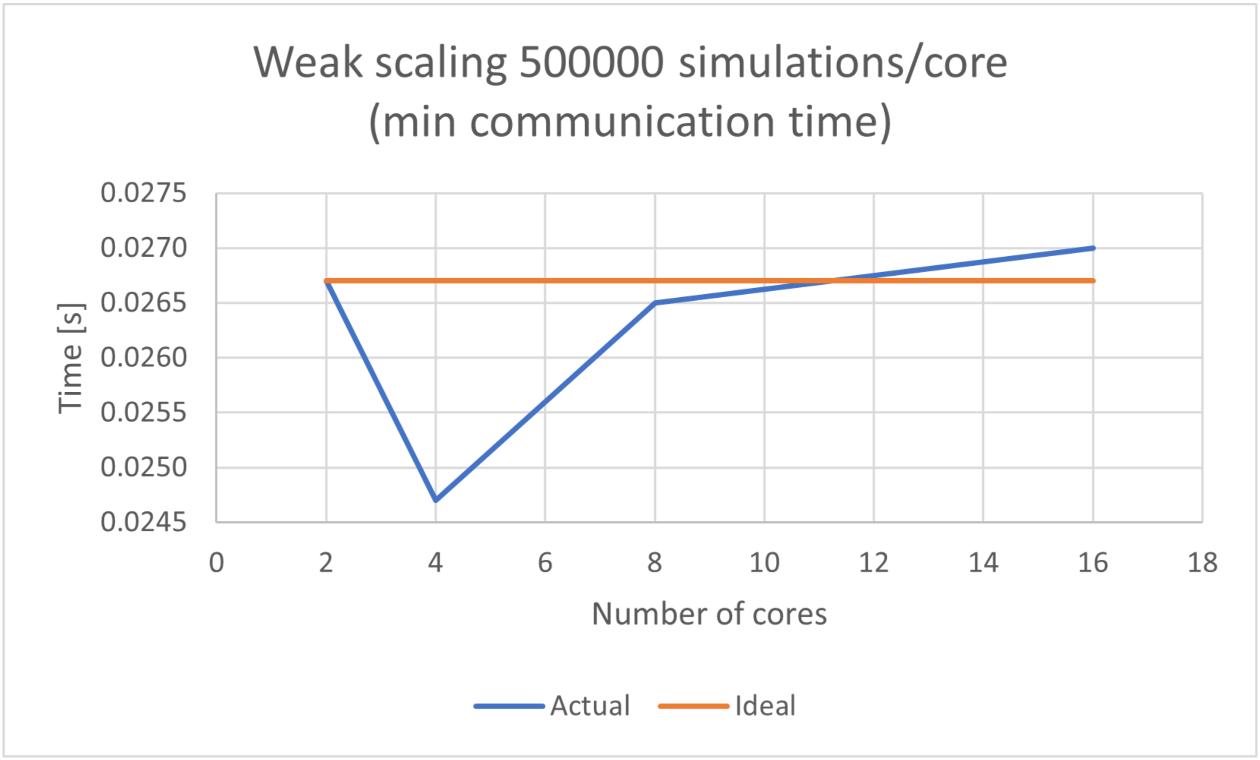

Figure 4: Plot of the weak scalability with 500 000 simulations per core, when considering the minimum communication time.

The choice was made to present plots of the strong and weak scalability both for the average total runtime for each core but also just the part of the code that handles the creation of the output data. This is because the simulation can take widely different times for the different processes to do since the number of iterations in each simulation is random. Since the processes can work asynchronously up until the exchange of the local max and min simulation values they can become widely unsynchronized which means that they start their timer for the communication time at different times but finish at roughly the same time. This then gives a slightly skewed picture of the actual time taken for the communication part and is the reason that the minimum time was chosen for this part of the code in the scalability plots. When using an $MPI\_Barrier()$ right before the timer for this part of the code it was validated that this gave a very close estimate to how long this part of the code would take even if all processes started at the same time.

The results of the simulations can be seen in figures 5, (6), 7 and the data for those histograms can be found in the appendix.

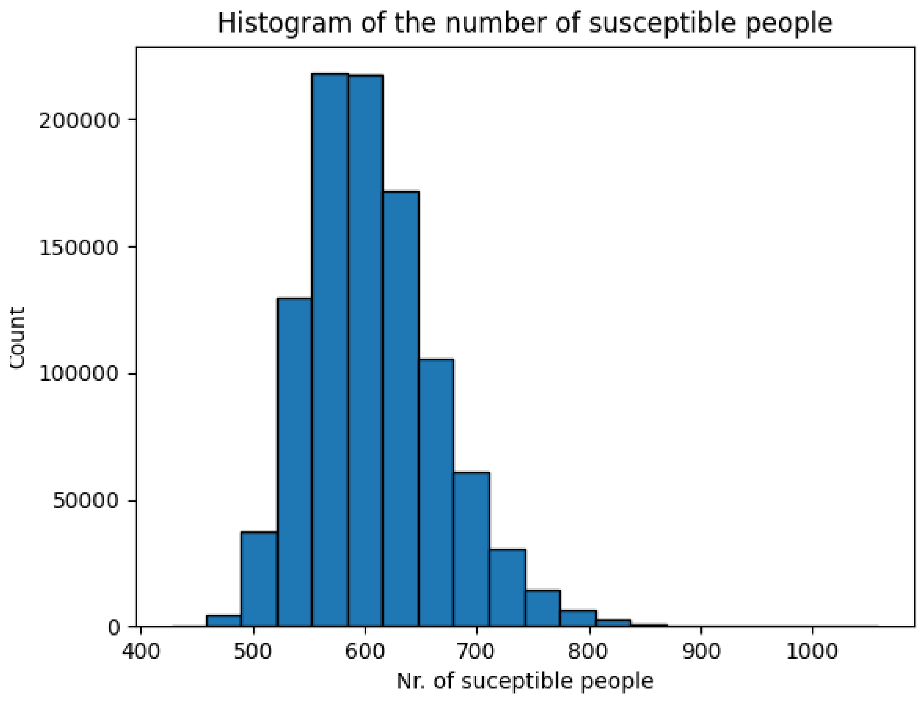

Figure 5: The histogram produced by the code for 1 000 000 simulations

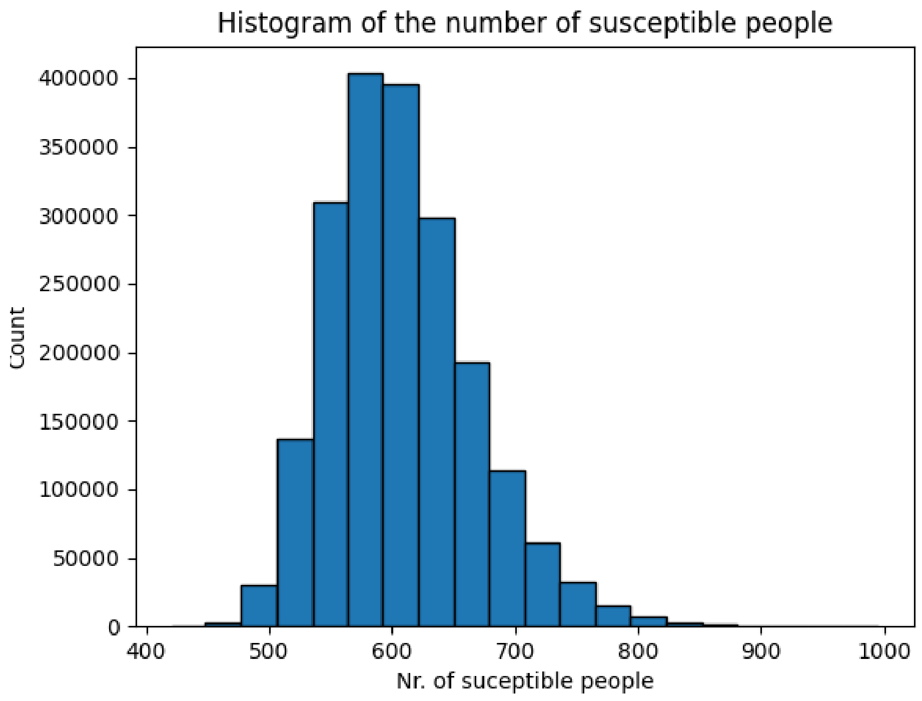

Figure 6: The histogram produced by the code for 2 000 000 simulations

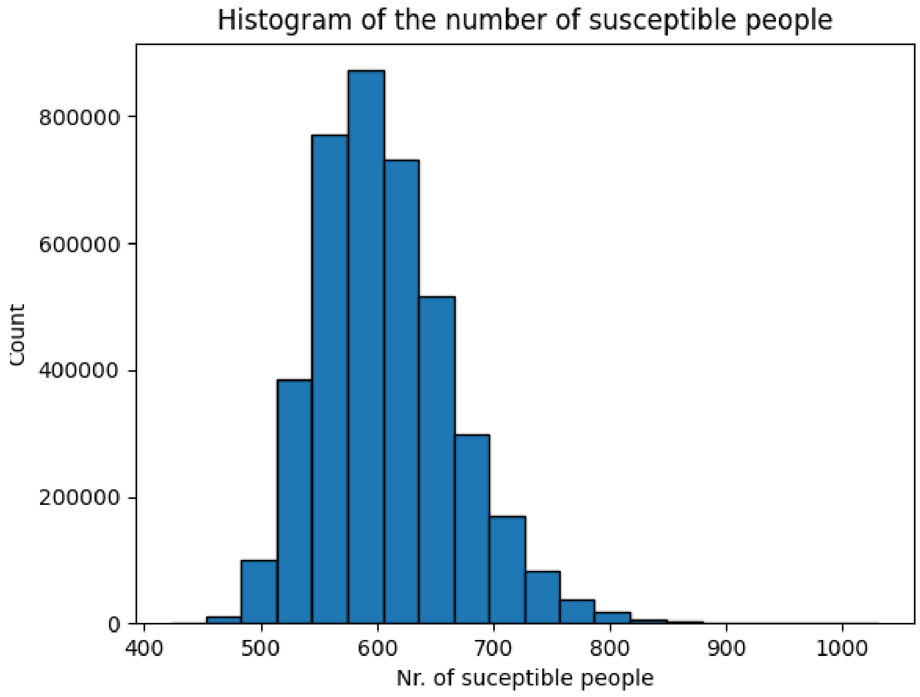

Figure 7: The histogram produced by the code for 4 000 000 simulations

5 Conclusion

5.1 Discussion of results

From the results one can clearly see that the code has an almost perfect strong and weak scalability both when considering the total time and when considering the communication part. The total time can be explained by the fact that almost all time is spent running the simulations and this is something that is done completely in parallel. For this reason the processes does not impact each others performance in any nameable sense so the code can run almost unobstructed except for the part when the processes needs to start communicating with each other. When all processes have finished their simulations and moved over to the second part of the code they have a relatively small amount of work to do but still have to have a rather frequent communication with each other. The reason I think that this still scales very well with an increasing number of cores is that the load on each core still is almost perfectly balanced and if all processes are loaded equally then they are mostly in sync and therefore they do not have to wait at the different communication steps and that part of the code still is able to scale well.

5.2 Potential improvements

The main performance bottleneck in this code is the fact that the processes are not fully utilized when they have to wait for the other processes to finish their simulations. The ability to change this is somewhat limited by the fact that the assignment specifies that the processes should all handle the same amount of simulations. If that was not the case the assignment of simulations could be handled in a similar way to how the OpenMP schedule dynamic or schedule guided handles assigning work to threads by giving them small chunks to work with and when they are done with one chunk they are assigned another chunk until all simulations have been run. This would decrease the likelihood that one process takes considerably longer than the others and therefore also decrease the wait time for all processes. This could give a significant boost of performance as I found that the processes had an average wait time of approx 525 seconds of the total runtime of 3 475 seconds when running 4 processes and 8 000 000 total simulations.

Another potential improvement could be to let the waiting processes start to process the already collected data of the process that is taking a long time. This way, once the slow process gets free it can almost instantly start counting the number of elements in each bin instead of first having to check for it's own local min and max. This is however a rather complicated way of processing the data and could add more time in all the communication time needed than the gain is in performance. Since this part of the code contributes to less than $1/10000$ of the total runtime this is not where the current bottleneck in the performance lies. For this reason my opinion is that this is only something worth considering when other parts of the code have been able to be optimized so that they are significantly faster or when every last microsecond matters. This is also the reason I chose to not implement this type of solution in my code as it would not make a difference in the total runtime of the code.

5.3 Final words

In conclusion the code scales very well with both increasing number of simulations and number of processes. The fundamental flaw with the code is the fact that the processes have a varying amount of work since they are assigned a fixed amount of simulations. Since that is given in the instructions and not possible to change I believe that this code performs it's duties very well.

6 Appendix

Table 3: Data for histogram for 1 000 000 simulations

Lower bound	Upper bound	Count
427	458	159
458	490	4540
490	521	37110
521	553	129540
553	585	218154
585	616	217390
616	648	171532
648	679	105652
679	711	60964
711	743	30788
743	774	14119
774	806	6369
806	837	2362
837	869	904
869	901	298
901	932	81
932	964	29
964	995	6
995	1027	2
1027	1059	1

Table 4: Data for histogram for 2 000 000 simulations

Lower bound	Upper bound	Count
420	448	78
448	477	2570
477	506	29632
506	535	136824
535	564	309496
564	592	403055
592	621	395062
621	650	297792
650	679	193017
679	708	113772
708	736	60755
736	765	32098
765	794	15151
794	823	6661
823	852	2661
852	880	938
880	909	323
909	938	85
938	967	21
967	996	1

Table 5: Data for histogram for 4 000 000 simulations

Lower bound	Upper bound	Count
423	453	293
453	483	9482
483	514	100153
514	544	384026
544	575	771666
575	605	872445
605	635	732330
635	666	515290
666	696	298675
696	727	168307
727	757	81870
757	787	38567
787	818	16966
818	848	6525
848	879	2328
879	909	771
909	939	210
939	970	69
970	1000	23
1000	1031	1