Uppsala University

Parallel & Distributed Programming

1TD070

Individual Project

Monte Carlo computations, combined with the Stochastic Simulation Algorithm to simulate malaria epidemic

Student:
Junjie Chu

Teachers:
Maya Neytcheva, Jarmo Rantakokko

May 25, 2021

1 Introduction

1.1 Background

We name a simulation of a system in which different variables change randomly with individual probabilities as a stochastic simulation. The random variables are generated and inserted into the model of the system during the simulation. Outputs of the model will be recorded as well as the process is repeated using a new set of random values. The steps are repeated until we get the amount of the data we want. The last step is to perform some statistical calculations. At last, the distribution of the outputs should the most probable estimates as well as a frame of expectations regarding what ranges of values the variables are more or less likely to fall in[1].

Monte Carlo is an estimation procedure. The main idea is that if it is necessary to know the average value of some random variable and its distribution cannot be stated, and if it is possible to take samples from the distribution, we can estimate it by taking the samples, independently, and averaging them. If there are sufficient samples, then the law of large numbers says the average must be close to the true value. The central limit theorem says that the average has a Gaussian distribution around the true value[2].

With the above two tools, we could do some practical simulation. Malaria is a mosquito-borne infectious disease that affects humans and other animals. Malaria remains a major public health problem with 109 countries declared as endemic to the disease. Mathematics and mathematical models play an important role in simulating the spread of diseases. By using appropriate tools and methods, we can use mathematical models or mathematical expressions to express the internal relationships of some known clinical information or biological information. Through such research, we can find important factors and characteristics of the spread of the disease. The models will help us to prevent and control the spread of malaria.

1.2 Motivation

This is the second topic of the project. In 2020 and 2021, covid-19 spreads throughout the world. Therefore, I want to study how to use mathematical models and computer technology to simulate the spread of diseases and find ways to control the spread of diseases. Although this topic does not directly use the relevant data of covid-19, the transmission models and factors of the disease are similar and universal, and I can get some enlightenment from it. In addition, MC and SSA can be perfectly parallel in this case. I think choosing this topic can exercise my MPI skills.

2 Problem description

The project is aimed to simulate the development of malaria epidemic with Monte Carlo method, combined with the Stochastic Simulation Algorithm.

In the program, n processors will run in parallel and for each processor, p individual stochastic simulations will be done. Thus, the number of total simulation is $n \times p$. All the test results will be collected by the root processor. All data for attribute "susceptible humans" will be saved as well. Statistics-based visualization (histograms) and analysis will be done on the as-obtained data.

The performance of the program will be evaluated in the following aspects: running time analysis, strong scalability and weak scalability.

3 Solution approach

3.1 Overall algorithm

In this project, to do a Monte Carlo simulation for malaria epidemic combined with the stochastic simulation algorithm in parallel, MPI will be used. In each individual simulation, Gillespie's direct method is used. As the computation in each for iteration is independent, the whole computation is perfectly parallelizable. Thus, for N experiments,if p nodes are given, each node will perform $n = N/p$ experiments. This also achieves perfect load balancing. After each node finishes their own task, they should send their local maximum value and local minimum value to the root processor. The root will find the global maximum value and global minimum value as the upper bound and lower bound of histograms. Then each bin's upper bound and lower bound are set. Each processor counts the number of each new interval and the root processor aggregates the data.

Because the computation cost for each simulation is constant k, time complexity for all N simulations is linear to the number, $O(N)$. If p processors are given, time complexity for the computation is $O(N/p)$.

Therefore, the following algorithm is designed.

Algorithm 1: Gillespie's direct method-based MC simulation

```
   Input: The number $N$ of MC experiments
1  for $i = 1; i \leq N; i = i + 1$ do
2      Set final simulation time $T$;
3      Set current time t = 0;
4      Set initial state x = x0;
5      while $t < T$ do
6          Compute $w$ = prop($x$);
7          Compute $a_0 = \sum_{i=1}^{R} w(i)$;
8          Generate two uniform random numbers $u1$ and $u2$ between 0 and 1;
9          Set $\tau = -ln(u_1)/a_0$;
10         Find $r$ that $\sum_{k=1}^{r-1} w(k) < a_0 u_2 \leq \sum_{k=1}^{r} w(k)$;
11         Update the state vector $x = x + P(r, :)$;
12         Update time $t = t + \tau$;
13     Save the final state;
14 Collect the susceptible samples in each process(the first element in each row of final state);
15 Find the max and min of samples in each process;
16 Summary the max and min in all processes;
17 Find the global intervals(width of each bin),Upper bound(global max) and lower
     bound(global min);
18 Set upper bound and lower bound of each interval;
19 According to the new range, count the number of each interval in each process;
20 Rank 0 collects all data;
```

3.2 Communication and other points of attention

3.2.1 Design of communication

The communication between the processors happens in two situations: 1. after each processor gets their local maximum and minimum values, the local maximum and minimum value need to be sent to the root processor; 2. after each processor finishes their local statistics, the result needs to be sent to the root processor.

There are three methods to realize the first communication. 1. Use MPI allreduce function. The function will reduce all local maximum and minimum value to global maximum and minimum value and saving them in each process. 2. Use MPI send in each processor and make the root processor receive the data using MPI recv. Then the root processor calculates global minimum and maximum locally and broadcast them. 3. Use MPI allgather. These functions could gather all maximum and minimum values in each processor and each processor calculates global ones locally.

Compared with these two methods, MPI allreduce function is the easiest way. Because it makes use of its built-in MPI MAX and MPI MIN operation. In addtion, there is only one line. Also it is the most efficient way: we only call one system communication function once. The third method has the lowest performance. In the method, many data are exchanged and each processor does the same operation(find the global maximum and minimum values). This is a waste of the CPU.

To realize the second communication, MPI reduce function is used. The function takes advantage of MPI SUM operation as well. Use MPI gather function is another choice. In the way, all arrays are saved in a longer array in one process. When counting the number of each bin, data from each process need to be summed and stored in another array. Compared with the second one, MPI reduce is much more easier and more efficient. The reasons are very similar to those of MPI allreduce.

For the above reasons, I used reduce and allreduce in the program. They are the most efficient of the methods I have come up with.

3.2.2 Other points of attention

In the step 19 of the above algorithm, we use $Lowerbound \leq value < Upperbound$. And the maximum value of each processor may not be counted. We need to count that value manually.

When generating the random numbers, we should set the seed of the rand() function and must control the precision of floating point numbers. If we want all the processors to use different seeds, we could apply 'srand(rank)'. If you want the result to be closer to true randomness rather than pseudo-random, we could use 'srand(time(NULL)+rank)'. In the case, the simulation situation will be different for each processor. When the number of experiments is large, using '(float)rand()/RAND_MAX' and '(double)rand()/RAND_MAX', the simulation results are slightly different. In the simulation program, all integers are stored and calculated as "int" and all real numbers are stored and calculated as "double" in C. In order to maintain the accuracy of the data and the consistency of the data type, I choose to use '(double)rand()/RAND_MAX'.

4 Experiments

4.1 Test environment

All tests are conducted on UPPMAX system Rackham. The detailed information of hardware is listed below:

1. CPU model: Intel(R) Xeon(R) Xeon V4
2. Operating system: CentOS Linux 7 (Core)
3. 486 nodes and 9720 cores
4. The majority of the nodes has 128 GB memory. There are four nodes with 1TB memory and 32 nodes - with 256 GB memory

4.2 Performance experiments

To evaluate the performance evaluation, time analysis is introduced. The measured time includes the time cost on message passing and local calculation in each processor (time spent on writing data in output file not included). A MPI reduce function is used to get maximum time cost of all processes. The maximum time cost is used to represent the time cost of the parallel program. The optimization flag is O3 of gcc.

In strong scalability, the size of problem size is fixed while the number of processors increases[3]. The input argument is n which is the local data size of each processor. So if we keep the global data size N a constant number, each processor will get a reduced workload per processor. In our experiments, several cases are tested, and the fixed problem size is $N = 10^5$. I test each case for 5 times and the average time cost is used. Numbers of processors are chosen to be 1, 2, 4, 8, 16 and 32. The detailed information and analysis of the strong scalability test is shown in Analysis part.

In weak scalability, the workload of one processor is fixed[3]. In our experiments, we fix the problem size of each process as 10^4. To increase the total number, just increase the number of processors used. Same as strong scalability tests, each case is also done 5 times and the average time cost is used. We also need to take the time complexity into consideration. The detailed information and analysis of the weak scalability test is shown in Analysis part.

4.3 Analysis

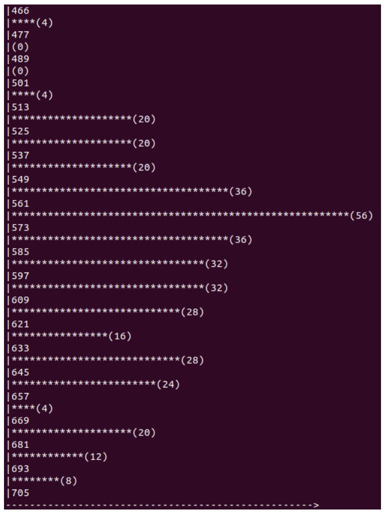
4.3.1 Analysis of results

Let N be 1.0×10^6, $1.1 \times 10^6$and 1.2×10^6. Some basic statistic calculation are done based on the above values of N. The results are shown in the following table:

The output data is written into the output file and the file contains two lines: the 1st line is 21 integers indicating the range of each bin of the histogram; the 2nd line is 20 integers indicating the number of susceptible cases that belongs to each bin respectively. In addition, when the total number of experiments is no larger than 400, a simple histogram will be printed on the command line. An example could be seen in the following picture.

N	Average value	Standard deviation	Variance
1.0×10^6	605.7342	57.9918	3363.0489
1.1×10^6	606.0162	57.9961	3363.5476
1.2×10^6	605.8384	58.0028	3364.3248

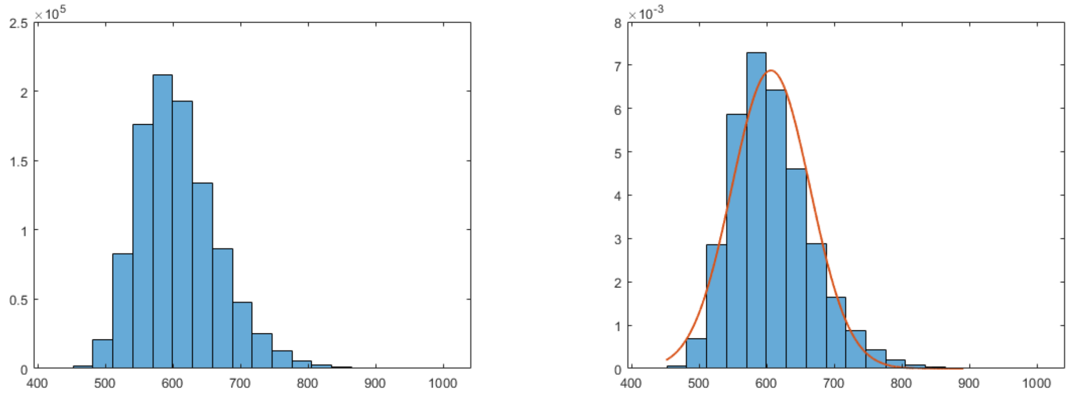
Table 1: Basic statistic results



```
|466
|****(4)
|477
|(0)
|489
|(0)
|501
|****(4)
|513
|********************(20)
|525
|********************(20)
|537
|********************(20)
|549
|************************************(36)
|561
|********************************************************(56)
|573
|************************************(36)
|585
|********************************(32)
|597
|********************************(32)
|609
|****************************(28)
|621
|****************(16)
|633
|****************************(28)
|645
|************************(24)
|657
|****(4)
|669
|********************(20)
|681
|************(12)
|693
|********(8)
|705
-------------------------------------------------->
```

Figure 1: A simple histogram on the command line

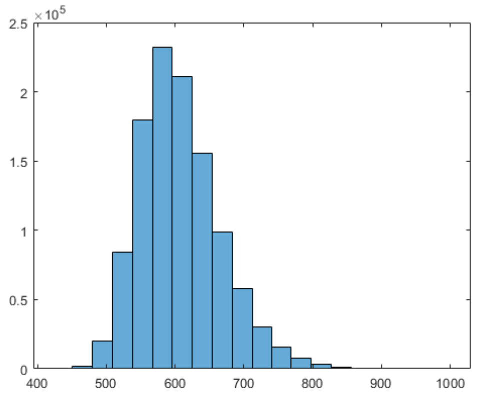
When $N = 10^6$, I plot the picture via Matlab using the data stored in the output file. Also, I use the fitted curve of the normal distribution for comparison. About the fitted curve, the average value is 606 and the standard deviation is 58.



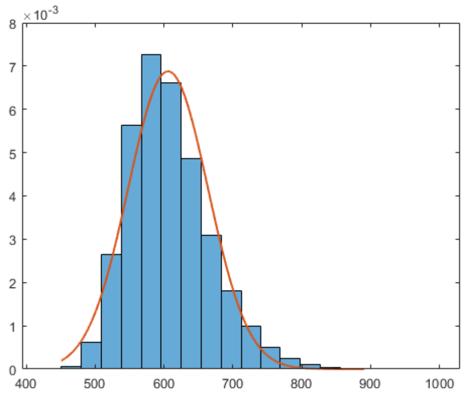
(a) $N = 10^6$: histogram

(b) $N = 10^6$: Matlab pdf histogram with a fitted curve

Figure 2: $N = 10^6$

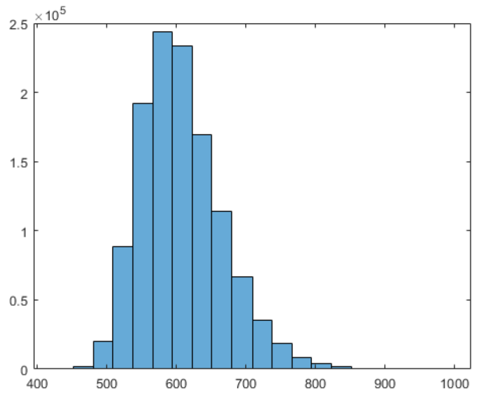


(a) $N = 1.1 \times 10^6$: histogram

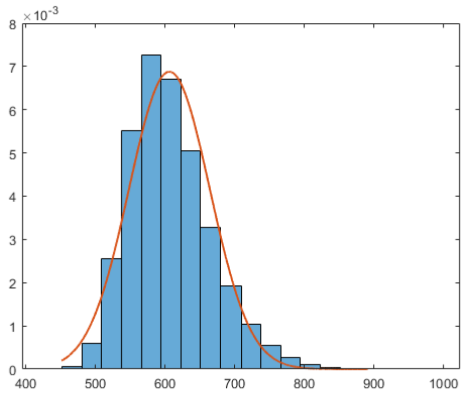


(b) $N = 1.1 \times 10^6$: Matlab pdf histogram with a fitted curve

Figure 3: $N = 1.1 \times 10^6$



(a) $N = 1.2 \times 10^6$: histogram



(b) $N = 1.2 \times 10^6$: Matlab pdf histogram with a fitted curve

Figure 4: $N = 1.2 \times 10^6$

As we can see from the histogram, the distribution of the data are almost normal. This is consistent with what is expected in the tutorial of the project.

4.3.2 Analysis of strong scalability

The results for strong scalability test is shown in Table(2). Also, the relationship between the speedup ratio and the number of processors is also plotted.

Total size N	Number of processors p	Local size n	Time $/s$	Speedup
1.0×10^5	1	100000	122.7514	1.0000
1.0×10^5	2	50000	64.8721	1.8922
1.0×10^5	4	25000	33.4567	3.6690
1.0×10^5	8	12500	15.9558	7.6932
1.0×10^5	16	6250	8.2022	14.9657
1.0×10^5	32	3125	4.4737	27.4385

Table 2: Strong scalability analysis results

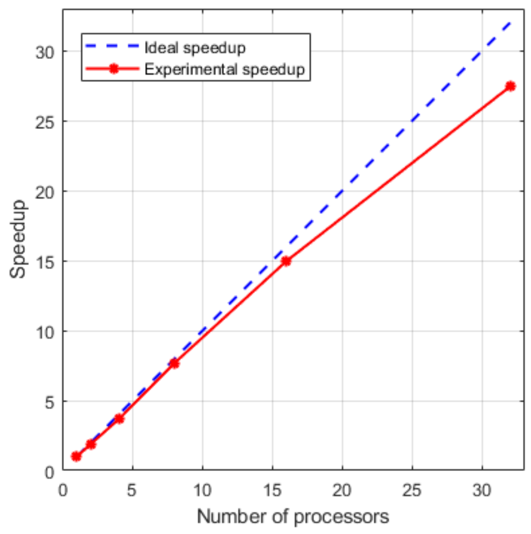


Figure 5: Strong scalability: speedup

According the figure and table, if the total size is fixed, when the number of processors increases, the speedup of the program increases correspondingly. According to Amanda's Law, due to the time spent on communication and system, it is not possible to reach the ideal speedup. In my case, the experimental speedup is close to the ideal speedup line, which indicates that our program run in parallel perfectly and the program has good strong scalability for fix-sized problem. When the number of processors increases, the local size decreases. Most of the work are completed in parallel, and MPI functions are only called twice, which mean the percentage of serial part is very small. So, the running time could decrease and the speedup is very close to the ideal one.

We can see from the figure that as the number of processors increases, the distance between the actual curve and the ideal straight line increases. This is because the amount of local data decreases (the percentage of the parallel part decreases) and the communication overhead increases (the percentage of the serial part increases). If the number of processors is large enough and the amount of local data is small enough, the speedup ratio will remain unchanged or even smaller.

4.3.3 Analysis of weak scalability

Since the time complexity is $O(N)$, the selection of the number of processors, the corresponding number of experiments and the results for weak scalability are shown in the table. The relationship between the efficiency(ideal time/experimental time) and the number of processors is plotted.

Total size N	Number of processors p	Local size n	Time $/s$	Efficiency
1.0×10^4	1	10^4	10.7842	1.0000
2.0×10^4	2	10^4	10.8997	0.9894
4.0×10^4	4	10^4	11.0096	0.9795
8.0×10^4	8	10^4	12.2873	0.8777
16.0×10^4	16	10^4	12.4509	0.8661
32.0×10^4	32	10^4	12.9764	0.8311

Table 3: Weak scalability analysis results

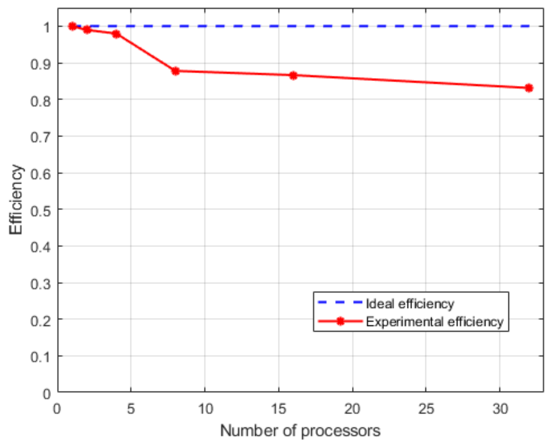


Figure 6: Weak scalability: efficiency

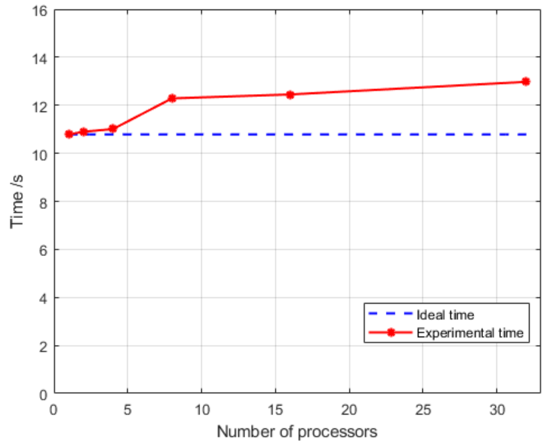


Figure 7: Weak scalability: time

According the figure and table, if the workload of each processor is the same, when the number of processors increases, the efficiency of the program decreases very slightly. According to Gustafson's Law, due to the time spent on gathering the data and system, it is not possible to keep the efficiency constant. In my case, the experimental efficiency is close to 1, which indicates that our program has very good weak scalability. This is because in the case, the problem is perfectly parallelizable. Each individual simulation is absolutely independent. And because of the selection of the communication functions, the communication between processors happens only in two situations:1.when we try to find the global maximum and minimum values; 2. when we collect the global statistic results. And in each situation, only one communication function is called. Many synchronization points and potential waiting times are avoided. This is the main reason why the program is of very great weak scalability.

We can see from the figure that when the number of processors increases, the parallel efficiency decreases and the total time spent increases. This is because the amount of local data is constant, but the communication overhead becomes larger. This causes the parallel percentage to decrease and the serial percentage to increase. If the number of processors is large enough, when the communication time is much longer than the processing time of 10,000 local experiments, the parallel efficiency will be extremely low.

5 Conclusions

In the project, the simulation of malaria epidemic is realized based on Monte Carlo computations, combined with the Stochastic Simulation Algorithm. As we could see in the analysis part, the program has a reasonable output as well as good strong scalability and weak scalability. These come from my optimization of communication. A small number of MPI functions are called, which also means a lot of synchronization points could be avoided! And another reason is that the algorithm could run in parallel perfectly. The above two mean that the percentage of serial part is small and that of parallel part is large. According to the two famous laws in scalability theory, we could get both good strong and weak scalability. Because of the limited source and the tight deadline, I have not tried more processors and larger data size to find the limit of strong and weak scalability. In the future, I will do more tests to investigate the scalability.

Usually, the global input data needs to be collected by the root processor and the root processor will delivery the data to different processors. But it is not that case in the topic. Since all the individual experiments are independent, we could make all processors begin the work at the same time with the same size of experiments. The load balance is realized here and perfect. The time and memory used for distributing data could be saved.

In addition to the existing optimizations, we can further optimize the program. If each node has more than 1 threads, we could use OpenMP in the code. Not all the nodes have the same number of threads, so using pthread is a bit hard here. A lot of vector and matrix operations are involved in the program. This means that automatic vectorization technology and SIMD can be used to optimize performance. Also, there are many C language high-performance computing libraries available. We could use high performance functions to replace the user-defined functions we use now.

Because of the content of this course, most of my energy is spent on the selection and optimization of MPI communication. In fact, we could combine the contents of parallel and distributed programming and high performance programming when doing parallel computation! I will try to use both of them in the future.

References

[1] DLOUHÝ, M.; FÁBRY, J.; KUNCOVÁ, M.. Simulace pro ekonomy. Praha : VŠE, 2005.

[2] Cosma Rohilla Shalizi, Monte Carlo, and Other Kinds of Stochastic Simulation, [online] available at http://bactra.org/notebooks/monte-carlo.html

[3] Xin Li, Scalability: strong and weak scaling, [online] available at https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/