UPPSALA
UNIVERSITET

Parallel and Distributed Programming

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

Jinglin Gao

May 25, 2023

1 Introduction

1.1 Background

Malaria is a life-threatening disease spread to humans by some types of mosquitoes. It is mostly found in tropical countries. The infection is caused by a parasite and does not spread from person to person.[1]In 2021 there were 247 million cases of malaria worldwide resulting in an estimated 619,000 deaths. Approximately 95% of the cases and deaths occurred in sub-Saharan Africa. Rates of disease decreased from 2010 to 2014 but increased from 2015 to 2021. It is a serious epidemic since it causes both losses of life and losses of money. In Africa, it is estimated to result in losses of US$12 billion a year due to increased healthcare costs, lost ability to work, and adverse effects on tourism.[3]

Stochastic Simulation Algorithm, in probability theory, generates a statistically correct trajectory of a stochastic equation system for which the reaction rates are known. Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The underlying concept is to use randomness to solve problems that might be deterministic in principle. However, the computational cost associated with a Monte Carlo simulation can be staggeringly high. In general, the method requires many samples to get a good approximation, which may incur an arbitrarily large total runtime if the processing time of a single sample is high.

Monte Carlo computations, combined with the Stochastic Simulation Algorithm, can be used to simulate the malaria epidemic by generating random samples of the state of the system over time and computing statistics of interest from these samples. We can use a Monte Carlo SSA that can simulate the dynamics of malaria transmission among human and mosquito populations using a set of reactions and propensities that describe the infection process. This method is almost perfectly parallelizable, so we can use parallel computing to speed up the simulation and analyze large-scale scenarios.

1.2 Motivation

The worldwide pandemic can affect our lives a lot. Especially after 2019 the Covid-19 outbreak, I realized that the control of diseases is the most important thing. The simulation of the malaria epidemic is the most attractive topic in this individual project for me. Although it is just a very simple simulate method it also can be a good start and interesting experience. Moreover, the parallel in this topic is almost perfect. I can learn how to distribute a mathematical model using parallel methods.

2 Problem description

In this problem, we are aiming to simulate a stochastic model of the development of an epidemic. We will perform a parallel experiment by using p processes and each process will do n experiments locally. Thus, the number of experiments $N = n * p$ in total. So basically we distribute the Monte Carlo experiments into p processes and each process shall execute the algorithm SSA n times, and the result from each local group of n SSA runs at the final time would form a matrix $X(7, n)$ locally. In the end, the test results will be collected by the root processor. We are interested in the distribution of "susceptible humans", which is the first component of x, so we will perform a statistics-based visualization and analysis based on the result we get from the experiments.

The performance of the program is expected to be close to perfect parallelized. We will evaluate the performance in the following aspects: running time analysis, fixed-size scalability, and weak scalability.

3 Solution approach

3.1 Overall Algorithm

Based on the general scheme of the Monte Carlo (MC) method and Gillespie's direct method, we can achieve an overall algorithm like this1:

Algorithm Stochastic Simulation Algorithm based Monte Carlo Simulation

Choose the number of MC experiments N
Set a final simulation time T
for $i = 1, 2, ..., N$ **do**
    Set current time $t = 0$
    Set initial state $x = x_0$
    while $t < T$ **do**
        Compute $\boldsymbol{w} = prop(\boldsymbol{x})$
        Compute $a_0 = \sum_{i=1}^{R} \boldsymbol{w}(i)$
        Generate two uniform random numbers u_1, u_2 between 0 and 1
        Set $\tau = -ln(u_1)/a_0$
        Find r such that $\sum_{k=1}^{r-1} \boldsymbol{w}(k) < a_0 u_2 \leq \sum_{k=1}^{r} \boldsymbol{w}(k)$
        Update the state vector $\boldsymbol{x} = \boldsymbol{x} + P(r, :)$
        Update time $t = t + \tau$
    end while
    Form the local result matrix $X(7, n)$
Calculate the bin width
Get the lower and upper bounds of each bin in the histogram
Calculate the frequency in each bin

Table 1: Overall Algorithm

3.2 Parallel Implementation

Technically after the MC experiments and we get the local results from each process, we have two ways to implement how we get the final result. The first way is to collect the data we need to root process and then do the rest of the statistical calculation. The second way is keeping the data locally and just performing the local statistical calculation, we will collect the results at the very end of the program. Considering the load-balancing problem, I choose the second way to implement my simulator. If not after the MC experiments all the computational missions will concentrate on the root process and other processes have nothing to do but wait.

In this case, the computation part does not need any communication between processes. Communication only happens in two places, one is when we get the global minimum and maximum value of the first component, and the other one is when we collect the frequency of each bin from each process.

There are several ways to collect data from each process. In the first place where communication is needed, I choose **MPI_Allreduce** since this method can also spread the result to every process. It is more simple when we just have the global minimum and maximum values locally to compute **bin_width** than we calculate it in the root process and then spread it to everyone.

In the second place where communication is needed, I choose **MPI_Reduce** to achieve the data collection. Here we have part of the frequency of each bin in each process, they are stored in one array (locally). We need to collect those arrays from every process and sum them up to get the final result. **MPI_Reduce** has the build-in **MPI_SUM** method and we can directly get the final array. This is the most efficient way to reach our goal.

4 Performance experiments

When we compile the program we use the -O3 flag to do the optimization of the code. We will record the time spent in each experiment and do the analysis to check if the program has good scalability or not. Here we use two communication methods **MPI_Allreduce** and **MPI_Reduce**, both of which are blocking operations.

For the fixed-size scaling experiment, we set a constant total size N, with the increasing number of processes, the local size n will decrease. Then we measure if the program has good strong scalability by checking its speedup and compare to the ideal speedup curve.

For the weak scaling experiment, we increase the total size N and the number of processes at the same time, so the local size n keeps constant. Again we measure if the program has good weak scalability by checking its speedup.

Since the time steps are chosen randomly, the processes proceed to reach the final time asynchronously. In order to compute the average time per processor for each time sub-interval, we output the every quarter time spent per process.

4.1 Fixed-size Scalability

In the fixed-size scaling performance experiments, the max execution time for the MC experiments and also the communication part is measured. Writing the results to the output file is not included in the time measurement. The experiments are managed with the Gullviva machine, the detailed information about this machine is as follows2:

Component	**Specification**
CPU	AMD Opteron (Bulldozer) 6274, 2.2 GHz, 16-cores, dual socket
Memory	128 GB
Operating System	Ubuntu 22.04
Server Name	gullviva.it.uu.se

Table 2: Server Specifications

The number of experiments N for fixed-size scaling experiment is 100000. The scaling is tested by running the code with different numbers of processes while keeping the total number of experiments N constant. As we increase the number of processes, the workload of each process is reduced. The result for fixed-size scalability is shown in the table below3.

Total size N	Number of processes	local size n	Time(s)	Speedup
100000	1	100000	288.2678	1.0
100000	2	50000	201.4705	1.43
100000	4	25000	71.2426	4.05
100000	8	12500	38.7406	7.44
100000	16	6250	24.9353	11.56

Table 3: Execution time of the fixed-size scalability experiments on gullviva

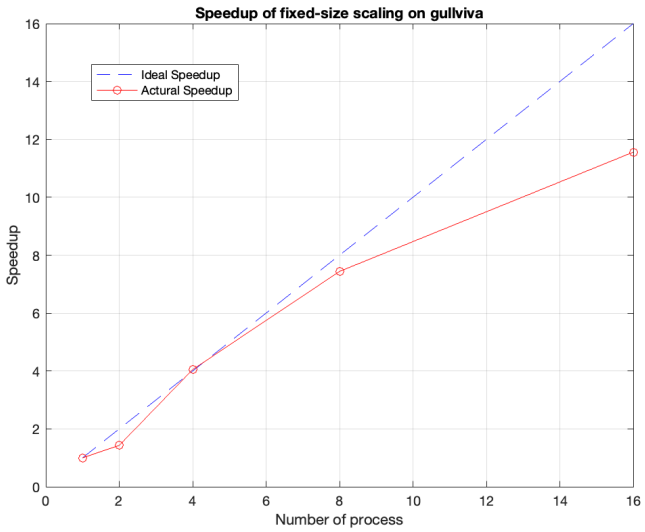


Figure 1: Fixed-size scaling speedup on Gullviva

I also test the fixed-size scalability on UPPMAX system Rackham. The detailed information of hardware is listed below4:

Component	**Specification**
CPU	Intel(R) Xeon(R) Xeon V4
Operating system	CentOS Linux 7 (Core)
Resources	486 nodes and 9720 cores
Memory	The majority of the nodes have 128 GB memory

Table 4: Rackham Specifications

The experiment result is shown in the following table6.

Total size N	Number of processes	local size n	Time(s)	Speedup
100000	1	100000	150.8592	1.0
100000	2	50000	76.3659	1.98
100000	4	25000	39.2677	3.84
100000	8	12500	21.1972	7.12
100000	16	6250	10.5628	14.28
100000	32	3125	5.3245	28.33

Table 5: Execution time of the fixed-size scalability experiments on Rackham

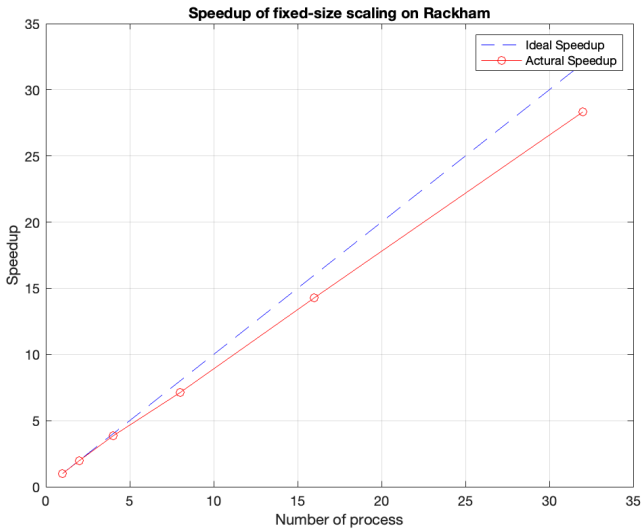


Figure 2: Fixed-size scaling speedup on Rackham

Analysis

We can see that the program works both on UPPMAX (Snowy) and the Scientific Linux system. It achieves an almost perfect speedup on the UPPMAX system. With the increase in usable resources, the execution time has significantly decreased. The speedup line is very close to the ideal speedup curve. Obviously according to Amdahl's law one program cannot be perfectly parallelized since its serial part will give an upper limit of speedup. But we can still see that our simulator has very good strong scalability for fixed-size problems. Most of the work is done in parallel and the communication overhead gets controlled. If we use more processes the speedup will eventually convert to a constant since more communications are done in the program and that might cause a serious communication overhead.

4.2 Weak Scalability

The weak scaling is measured by running the code with different numbers of processes and with a correspondingly scaled number of experiments. The experiments are managed with the UPPMAX machine. We submit the job by submitting the bash file. The given total sizes of experiments are 10000, 20000, 40000, 80000 160000, and 320000. Gustafson's law indicated that the parallel part scales linearly with the amount of resources, and that the serial part does not increase with respect to the size of the problem. So theoretically we should get a linearly scaled speedup line from the experiment. The results show in the table below**??**

Total size N	Number of processes	local size n	Time(s)	Speedup
10000	1	10000	15.1712	1.0
20000	2	10000	15.2771	1.98
40000	4	10000	15.8901	3.84
80000	8	10000	16.9501	7.12
160000	16	10000	16.8580	14.28
320000	32	10000	17.5116	28.33

Table 6: Execution time of the fixed-size scalability experiments on Rackham

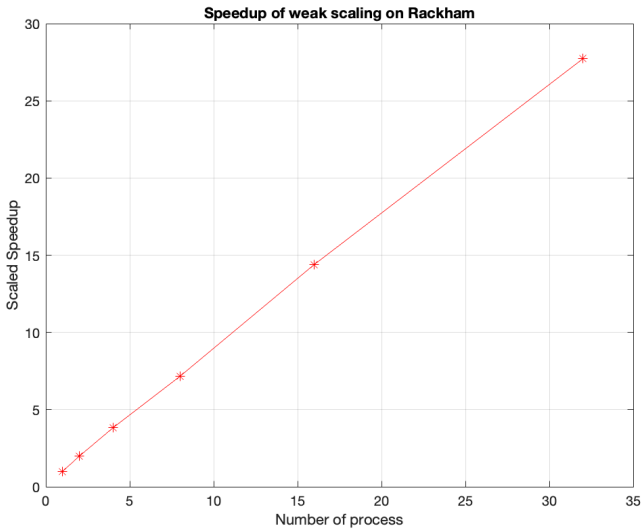


Figure 3: Weak scaling speedup on Rackham

Analysis

The graph shows that we have relatively good weak scalability in our program. When we perform the weak scaling experiments, we focus on the speedup of the parallel part. Theoretically, the serial part should remain constant, and the speedup should be ideal.[2] But in reality, the overhead of parallelization also increase with the job size. In this case, we keep the local workload constant, but the difference between the ideal speedup and the actual speedup slowly becomes larger. It is not hard to understand that with the increasing of number of processes, we need to configure more communication between each process, so the overhead also increases, which ends in the increase of the serial portion.

4.3 Running time analysis

Here I use 8 processes and choose the local size n as 10000. The result shows in the table below:

Process N	Quarter time spent	Half time spent	Three quarter time spent	Final time spent
0	16.9884	16.9888	16.9891	16.9893
1	17.0087	17.0090	17.0093	17.0096
2	16.9668	16.9671	16.9674	16.9677
3	17.1632	17.1636	17.1639	17.1641
4	16.9914	16.9917	16.9920	16.9923
5	17.0575	17.0578	17.0581	17.0584
6	16.9821	16.9824	16.9827	16.9829
7	16.9679	16.9683	16.9686	16.9689

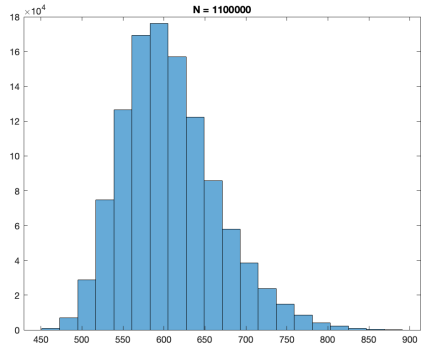
Table 7: Running time per process

The results do not directly show each process spent how long time to passing time 25, 50, 75, 100. but if we use the later result minus the front result we can get the time this process spent going through the time interval. By calculation, the asynchronous is not obvious. This might be because the local size n I choose to perform the experiment is not large enough. If the number of local experiments increases, then there might be a more significate difference for each process to finish the experiment.

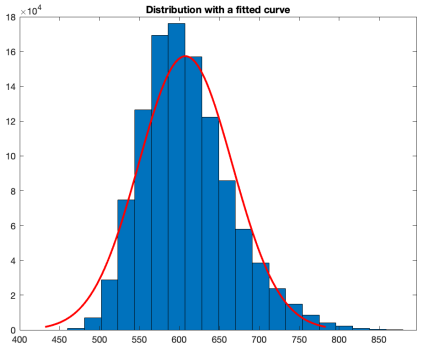
4.4 Result from the program

Here we are interested in the distribution of the susceptible humans, which is the first component of the vector $\boldsymbol{x}$. We write the bounder of each bin and the corresponding frequency into the output file and draw the histogram in Matlab.

We evaluate three values of N, 1100000, 1500000, and 1800000 respectly. The result histograms show below456.

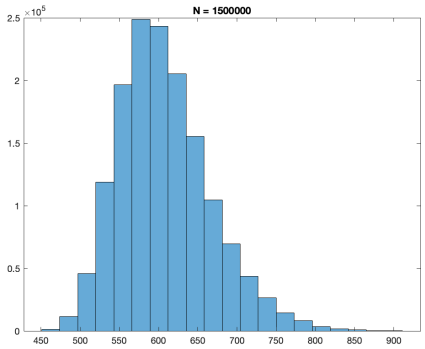


(a) Distribution of $N = 1100000$

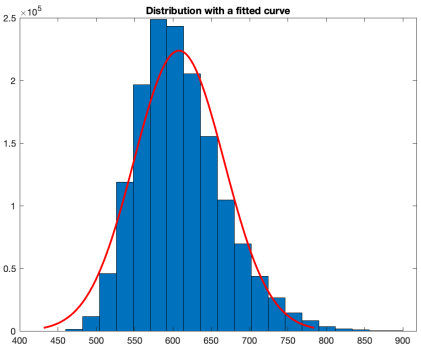


(b) Distribution with fitted curve

Figure 4: Result of $N = 1100000$

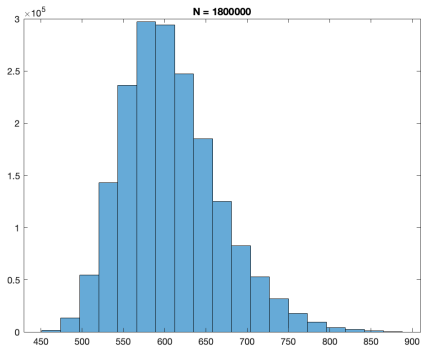


(a) Distribution of $N = 1500000$

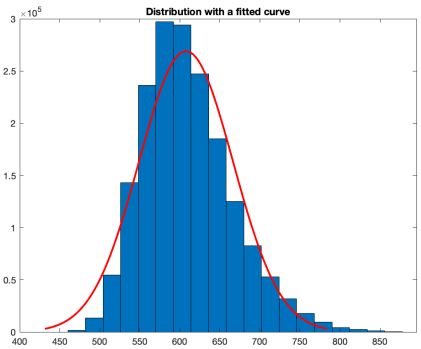


(b) Distribution with fitted curve

Figure 5: Result of $N = 1500000$



(a) Distribution of $N = 1800000$



(b) Distribution with fitted curve

Figure 6: Result of $N = 1800000$

As we can see from these figures, the distributions of susceptible humans almost follow a normal distribution. The results meet the expectation of the introduction.

5 Conclusions

In this project, we use Monte Carlo computations combined with the Stochastic Simulation Algorithm to simulate the malaria epidemic. The method requires generating many samples to get a good approximation. If we perform in a single node then it will be an extremely heavy computation mission. In this case, we choose to scale the experiments into p processes and speed up the program.

There are some possible optimizations that can be done in my implementation. The way I choose to store the state vectors into local matrix $X(7, n)$ is very inefficient. Since the matrix uses a row-wise way to be stored in the memory, every time we need to go through n more memory places to store that specific component of the vector $\boldsymbol{x}$ into matrix $X(7, n)$. One way to fix this is to choose a Column-wise partitioning method to store matrix X. Then when we need to get all the first components from every single experiment, we can use the loop-unrolling method to fetch the first "row" in the matrix. In addition, we can use some C language high-performance libraries to optimize the program.

According to the performance result, we get both good strong scaling and weak scaling. We can actually calculate the proportion of serial parts by using the Amdahl's law and Gustafson's law. The algorithm itself is kind of perfectly parallelizable, so the only thing we need to care about is choosing the correct communication method and make the structure of the program efficient.

References

[1] *Malaria*. World Health Organization, **may** 2023. URL: `https://www.who.int/news-room/fact-sheets/detail/malaria`.

[2] KTH Royal Institute of Technology. *Scalability: Strong and Weak Scaling*. [Accessed: May 24, 2023]. 2018. URL: `https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/`.

[3] Wikipedia. *Malaria*. [Online; accessed 23-May-2023]. 2023. URL: `https://en.wikipedia.org/wiki/Malaria`.