Parallel and Distributed Programming – Project
Simulating Malaria Epidemic

Claude Carlsson

May 25, 2023

UPPSALA
UNIVERSITET

Contents

1 Introduction

In this project, we are going to simulate a malaria epidemic. Malaria is a mosquito-borne infectious disease that affects humans as well as other animals. Malaria, itself, is a parasite that when entering the human body travels to the liver and reproduces. The usual symptoms of malaria are fever, tiredness, vomiting, and headaches. In 2021, 619 000 people died from malaria, which is why it is important to try to model the disease. [1]

By simulating the epidemic, we can perhaps get an approximate distribution on how the disease will develop and get some insights. Furthermore, if there already is an outbreak, we can perhaps know where we are in the timeline of that outbreak by comparing the empirical distribution of the epidemic with the simulated one.

2 The problem

In order to simulate the disease, we are going to use Monte Carlo computations, combined with Gillespie's Stochastic Simulation Algorithm (SSA). This will be implemented in C, and parallelized with Open MPI. In this project, we are already supplied with a function that computes the propensities for the malaria model. In that function we can find different rates such as, mortality rate, recovery rate, and loss of immunity rate to name a few.

This problem is perfectly parallelizable since each process will be handling N/P runs, where N is the total of runs, or Monte Carlo experiments, and P is the amount of processes. Furthermore, to ensure that we have good load balancing, which should be good in the first place since the Monte Carlo runs are independent, I have implemented four checkpoints evenly distributed across the run. Since the final time is 100, these checkpoints are placed at: 25, 50, 75, and 100. After the runs, we will be creating a histogram over the results, displaying how many susceptible humans there were in each run. We need to somehow combine the local result into a synthesized global result.

Furthermore, we are also going to study how efficient our implemented algorithm is and perform scaling tests. In this project, we are going to perform both strong and weak scaling to understand how our program performs. These results will be presented in both tables and in figures.

3 The solution and parallelization

The algorithm that will be used in this implementation is the common Monte Carlo algorithm and Gillespie's direct method (SSA).

Algorithm 1: Monte Carlo

1. Choose the number of MC experiments N
2. for i = 1, 2, ... , N do
   (a) Perform on MC experiment and store the results
3. Gather the result

Algorithm 2: Gillespie's direct method (SSA)

1. Restart timers, set initial state $x = x_0$
2. while $t < T$
   (a) Compute $w = prob(x)$
   (b) Compute $a_o = \sum_{i=1}^{R} w(i)$
   (c) Generate two random numbers u_1, u_2
   (d) Set $\tau = -ln(u_i)/a_0$
   (e) Find r such that $\sum_{k=1}^{r-1} w(k) < a_0 u_2 \leq \sum_{k=1}^{r} w(k)$
   (f) Update state vector $x+ = P(r, :)$
   (g) Update time $t = t + \tau$
3. end while

Note here, on **(a)**, the function prop is already supplied, which was mentioned before.

3.1 General optimization

In my application, all the larger arrays are allocated on the heap, this ensures that we can handle memory requirements for when we have large amount of runs. Furthermore, we use the "-O3" flag when compiling, this enables optimization such as loop unrolling and function inlining.

3.2 The simulation and creating the histogram

Since each Monte Carlo experiment is independent of each other, each process gets N/P runs. This makes parallelization of the runs trivial. However, after the run when we should gather the results, some work needs to be done. First, after the runs, we find the local min and max of the amount of susceptible humans of each process, then we use **MPI_Allreduce** to find the global min and max. This will be used when we create a histogram later. Then, for each process, a local histogram is created, this will be combined into one global histogram with **MPI_Reduce**. Finally, this histogram will get printed to an output file.

3.3 The checkpoint timings

For each Monte Carlo experiment, we did four checkpoints. These checkpoint timings were then collected after all the runs, and we averaged the timing for each process. To do this, I created a local timing array and a global one. During the runs, we check if we have met the checkpoint condition, that is, if our time is either: 25, 50, 75 or 100. We add these timings to the local array during the run. After the runs, the timings are converted the timings to milliseconds and averages them by dividing by the amount of runs per process. Finally, we use **MPI_Gather** to sum all of these result and place them in the global array.

3.4 Limitations and the output of the application

The application assumes that the number of runs is divisible by the number of processes. The program only outputs one number in the standard output, and that is the execution time of the algorithm and the gathering of the results. Furthermore, the program writes two files: the first file contains the histogram data, and the second file contains the timings for the checkpoint for each process.

4 Performance experiments and results

In this section, we are going to talk about our experiment's methodology and present the results.

4.1 Experiments setup

The experiments are going run on UPPMAX. The system specification for the cluster we are running is the following:
System: Uppmax Cluster "snowy"
Processor: Intel Xeon E5-2660
Operating System: CentOS Linux 7 (Core)
Compiler version: gcc 12.2.0
MPI version: openmpi 4.1.4

For strong scaling, we are testing: 1, 2, 4, 8, and 16 processes with the numbers of simulations set to $N = 50000$. For weak scaling, we are trying to hold the problem size constant per process. In other words, for weak scaling we are testing: 1, 2, 4, 8, and 16 processes with the numbers of simulations set to $N = 50000, 100000, 200000, 400000, 800000$. Furthermore, we are presenting some result from the checkpoint timings and the peak memory usage of this application. After the checkpoint timings has been presented, we are presenting the histograms for three different runs: $N = 1.0 * 10^6$, $N = 2.0 * 10^6$, and $N = 3.0 * 10^6$. Finally, the raw data from the histogram can be found in the appendix 6.

The timings include the Monte Carlo runs with the simulation, gathering the information from the runs, and creating a histogram. It does not include generating the files for the histogram or checkpoints.

4.2 Strong scaling

Table 1: The results for strong scaling, both measured time and ideal time

Processes	Measured Time [s]	Ideal Time [s]
1	103,32	103,32
2	52,18	51,66
4	26,17	25,83
8	15,75	12,92
16	7,28	6,46

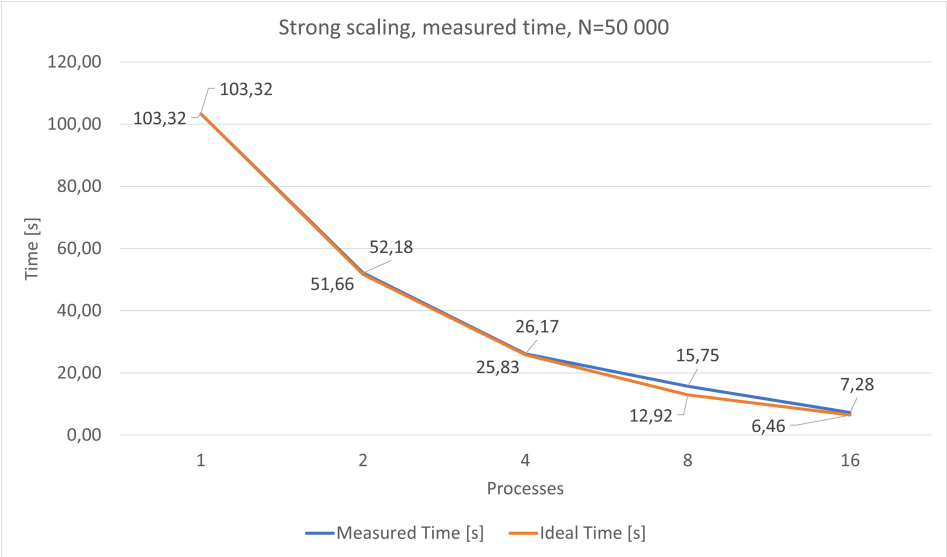

Figure 1: The results for strong scaling, both measured time and ideal time

Table 2: The results for strong scaling speedup, both measured speedup and ideal speedup

Processes	Measured speedup	Ideal Speedup
1	1,00	1,00
2	1,98	2,00
4	3,95	4,00
8	6,56	8,00
16	14,19	16,00

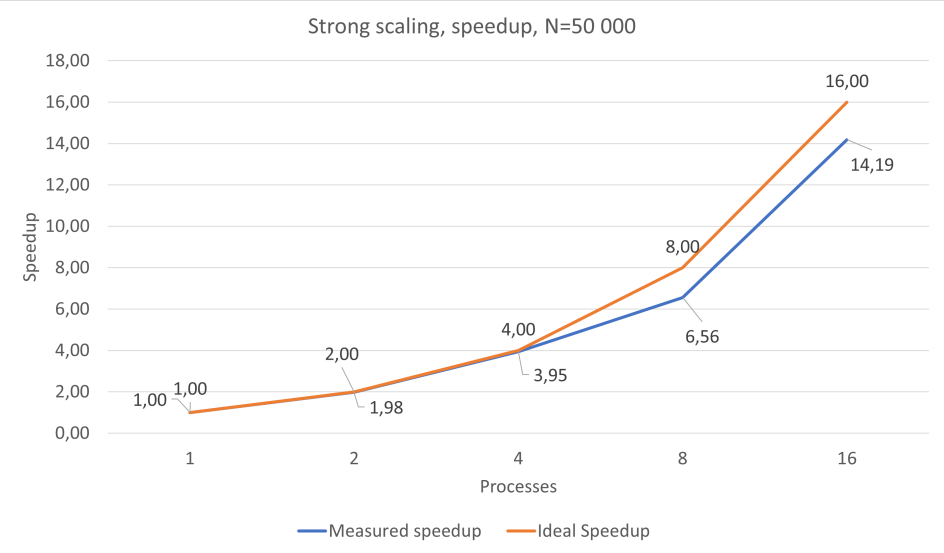

Figure 2: The results for strong scaling speedup, both measured speedup and ideal speedup

4.3 Weak scaling

Table 3: The result for weak scaling, both measured and ideal time

Processes	Measured Time [s]	Ideal Time [s]
1	103,80	103,57
2	103,85	103,57
4	105,33	103,57
8	108,28	103,57
16	120,09	103,57

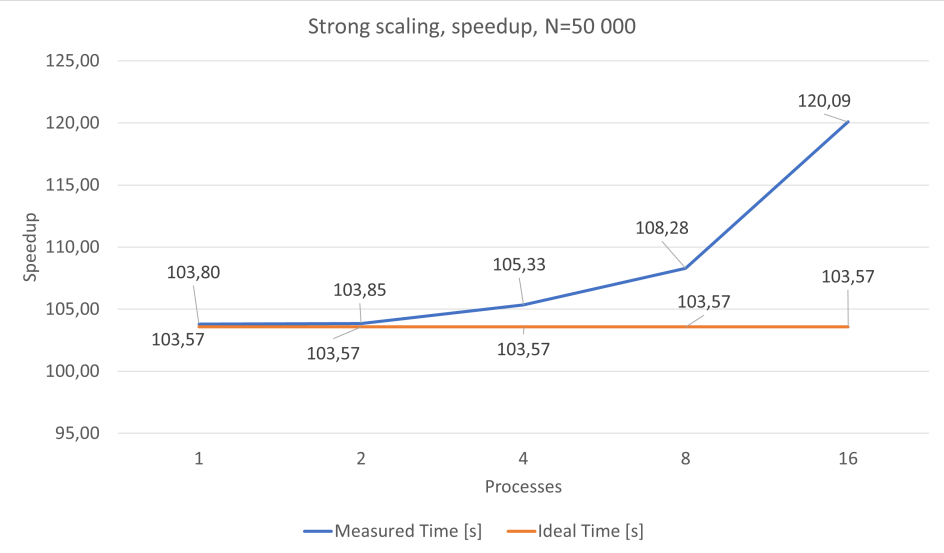

Figure 3: The results for weak scaling, both measured time and ideal time

4.4 Memory usage

With 16 processes and the numbers of simulation set to $N = 50000$, by using this command:

```
sacct -j <job_id> --format=JobID,JobName,MaxRSS
```

we can see what our peak memory usage was during the execution of the job, our job used approximately 36 320 KB.

4.5 Checkpoint timings

Here we will present checkpoint timings for when we run the application on eight processes and have set the numbers of simulation to $N = 50000$ and $N = 500000$.

Table 4: Checkpoint timings for 8 processes and when $N = 50000$

	checkpoint-25	checkpoint-50	checkpoint-75	checkpoint-100
rank-0	1,128715	1,525098	1,839196	2,153984
rank-1	1,47181	1,885682	2,202464	2,519719
rank-2	1,129093	1,524717	1,838661	2,153589
rank-3	1,126871	1,523994	1,838622	2,153995
rank-4	1,129605	1,528567	1,844039	2,16051
rank-5	1,12776	1,523988	1,837661	2,153031
rank-6	1,139623	1,540157	1,857978	2,176854
rank-7	1,125661	1,520634	1,833881	2,149512

Table 5: Checkpoint timings for 8 processes and when $N = 500000$

	checkpoint-25	checkpoint-50	checkpoint-75	checkpoint-100
rank-0	1,175449	1,574885	1,884976	2,196123
rank-1	1,154089	1,550476	1,862871	2,176403
rank-2	1,112692	1,503394	1,813298	2,124082
rank-3	1,123053	1,517327	1,829678	2,143416
rank-4	1,121687	1,515497	1,827847	2,141094
rank-5	1,112715	1,50345	1,813199	2,123848
rank-6	1,122441	1,516588	1,829072	2,142603
rank-7	1,117755	1,51014	1,821317	2,133614

4.6 Histogram

All of these runs were made with 16 processes.

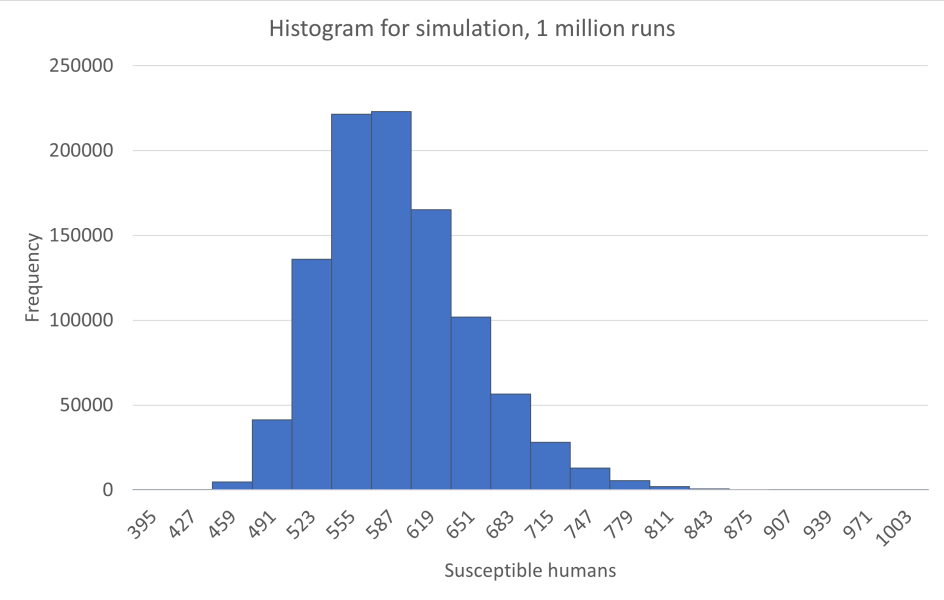

Figure 4: The histogram for one million runs

The lowest bound for this experiment with one million runs is 395 susceptible humans, with the most frequent being 587, and the highest bound was 1003.

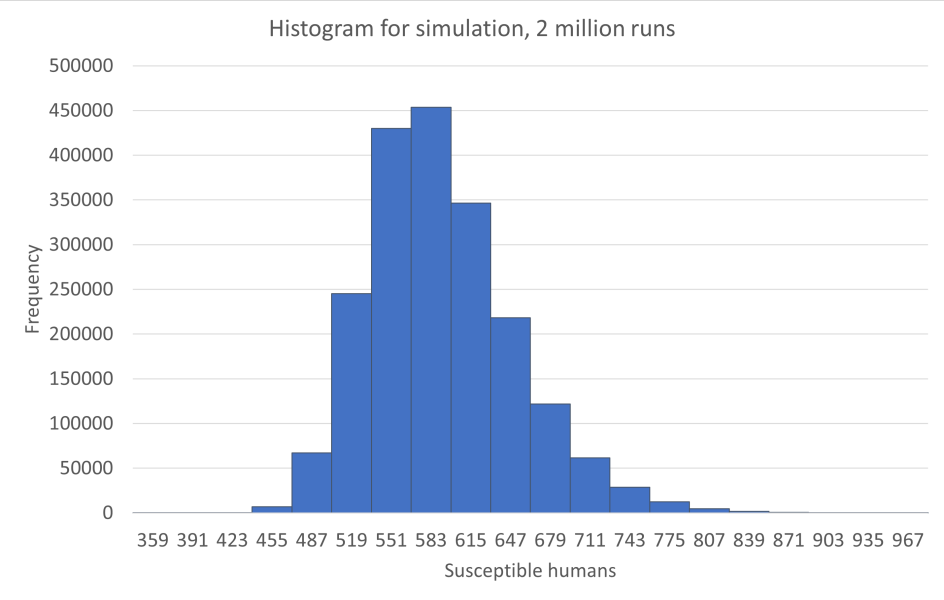

Figure 5: The histogram for two million runs

The lowest bound for this experiment with two million runs is 359 susceptible humans, with the most frequent being 583, and the highest bound was 967.

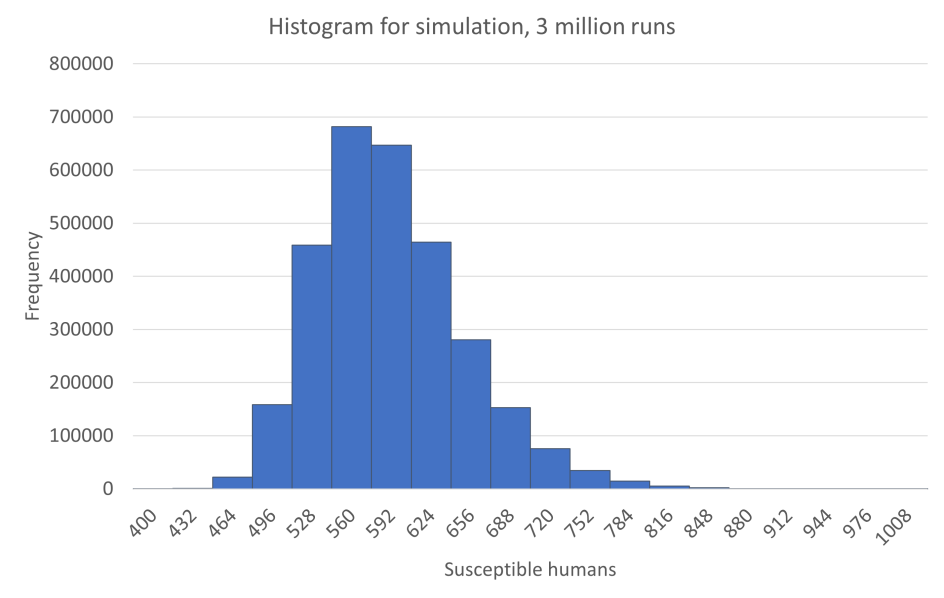

Figure 6: The histogram for three million runs

The lowest bound for this experiment with three million run is 400 susceptible humans, with the most frequent being 560, and the highest bound was 1008.

5 Discussion and analysis

From our results, we can see that we achieved good scaling results. Looking at table 1 or figure 1 we can see that the difference between our measured time and the ideal time are close. This was expected, since we previously mentioned that this simulation is almost trivial to parallelize, and since each Monte Carlo experiment is independent of one another, close to ideal scaling was expected. If we look close to table 2 or figure 2, we can see that we got virtually ideal scaling up to four processes, after four we got some diminishing returns. This is also notable in the weak scaling section, see table 3 or figure 3. For the weak scaling results, the biggest jump in worse performance was between 8 and 16 processes, at 16 processes it takes approximately 20% longer than ideal to execute the algorithm and gather the data. One possible explanation for the degradation in the performance is the communication overhead, as we increase the number of runs, so does the size of the arrays. These arrays are then communicated between the processes.

We can exclude the possibility that communication overhead occurs inside the algorithm by using the checkpoints to calculate the average time per processor for each time sub-interval for two widely different numbers of MC experiments. Looking at table 4 and 5 we can see that even if we have one order of magnitude higher, each sub-interval average time stays virtually the same. In other words, the diminishing returns from scaling happens when we are gathering the results from the runs. However, the storing of the timings during the run might degrade the performance somewhat, but it should not be a disproportionately more when increasing the numbers of MC experiments.

Looking at the histograms, see figure 4, 5, and 6. We can see that the histogram reassembles a normal distribution with some skew in it. Furthermore, if we assume it is a normal distribution, we see that the distribution seems to converge with a mean at around 560-580 susceptible humans, and it has a standard deviation of about 40-50 susceptible humans. However, it has some skew towards the right for all the histograms, this can give some insights about a malaria outbreak.

In conclusion, we have successfully implemented a Gillespie's Stochastic Simulation Algorithm in combination with Monte Carlo to simulate a malaria epidemic. The application scales well, and achieves a speedup of 14.19 compared to the ideal of 16 for when using 16 processes and $N = 50000$. Finally, we have implemented checkpoints so that we can observe how long each sub-interval takes within the algorithm. We have made histograms to visualize the results, and these histograms reassembles a normal distributions.

5.1 Future improvements and alternative solutions

In my implementation, I have used **MPI_Gather** to gather the timing results from the checkpoint sub-intervals. One advantage of using this method is that it is easier to implement since we do not have to keep track of indexes across the processes as we would have needed with the method of **MPI_Win**. However, since the method **MPI_Gather** uses collective communication it might introduce some extra overhead compared with **MPI_Win** which is one-sided communication.

Finally, in my code, I use a 2D array for simulation states, this array could be converted into a 1D array to improve cache locality. Since we use this array inside the algorithm, this might increase the speed somewhat since we perform so many operations on this array.

References

[1] Wikipedia contributors. Malaria.
https://en.wikipedia.org/wiki/Malaria, 2023. Accessed: May 2023.

6 Appendix

6.1 Histogram with one million runs

```
bin ,amount
395,3
427,168
459,4777
491,41342
523,136037
555,221443
587,223025
619,165143
651,101878
683,56581
715,28167
747,12824
779,5546
811,2040
843,716
875,227
907,62
939,13
971,4
1003,4
```

6.2 Histogram with two million runs

```
bin ,amount
359,1
391,7
423,207
455,6950
487,67264
519,245369
551,430019
583,453639
615,346591
647,218264
679,121955
711,61627
743,28756
775,12301
807,4722
839,1577
871,546
903,152
935,39
967,14
```

6.3 Histogram with three million runs

```
bin ,amount
400,11
432,893
464,21985
496,158666
528,458944
560,682074
592,646884
624,464659
656,280606
688,153143
720,75602
752,34316
784,14275
816,5298
848,1853
880,606
912,126
944,41
976,16
1008,2
```