Individual project:

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

Parallel and Distributed programming, 5 hp

Oskar Holmes

May 25, 2023

UPPSALA
UNIVERSITET

Contents

1 Introduction

Malaria is one of the most deadly diseases that still affect humans. To help fight its spread and devastating effects scientist can simulate it and try to predict outcomes to be able to react easier in real life. In this report i will present one of those simulation algorithms and use it to simulate a malaria epidemic.

2 Problem description

The task is to use a Monte Carlo(MC) method to simulate the development of a malaria epidemic. More specifically, *Gillespie's direct method*, a stochastic simulation algorithm(SSA) is used.

3 Solution approach

Implementing a MC algorithm is relatively simple. The core concept is using randomness and a large enough sample size to yield significant results. In this case the MC part of the algorithm is the SSA. The SSA manipulates the state vector $x()$, which contains a variety of quantities relevant to the simulation. Since this is a epidemic simulation we have quantities such as susceptible humans, exposed humans, infected humans, etc. The SSA randomizes a reaction and and a time step, which overtime changes the quantities in the state vector. At an end time T, we have a final state vector X. In this task we are only interested in the amount of susceptible humans, which is the only quantity we are studying. // To keep results relevant, we run the SSA algorithm over and over again to generate a large sample of final state vectors to show a distribution of outcomes. The parallelisation comes in when we just divide the amount of runs we want over several PE:s, with each PE just running the SSA and generating its own state vector a certain amount of times.

4 Expreiments

Firstly the results were checked with large numbers, $N = 10^6$, $N = 1.6 \cdot 10^6$ and $N = 2 \cdot 10^6$ respectively.

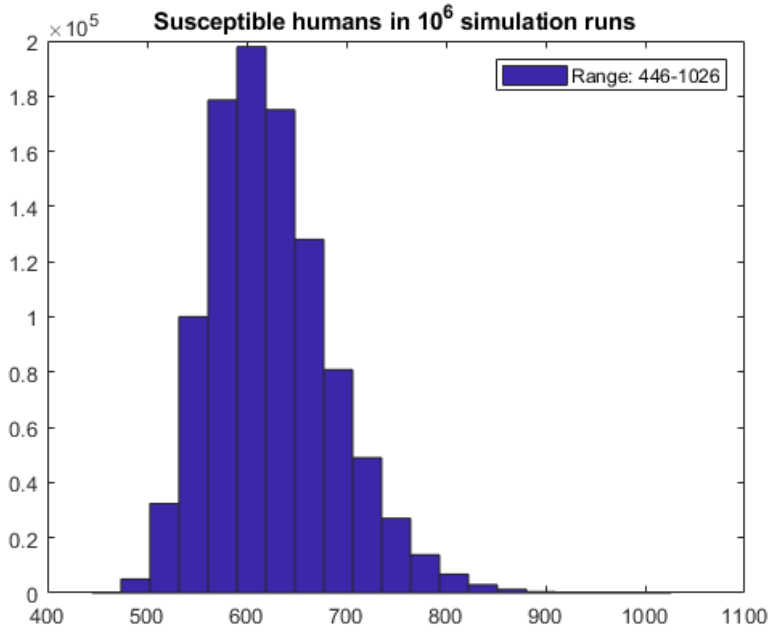

Figure 1: Distribution of the final amount of susceptible humans in 10^6 runs

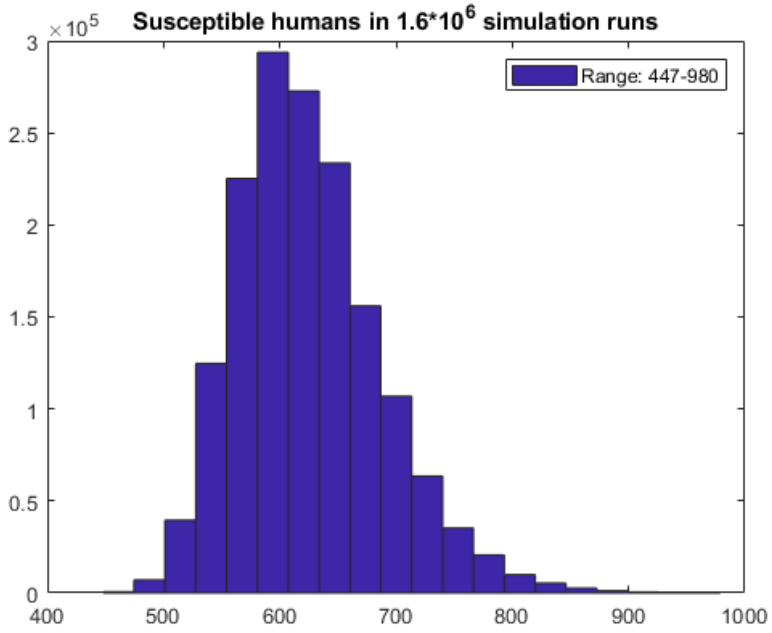

Figure 2: Distribution of the final amount of susceptible humans in $1.6 \cdot 10^6$ runs

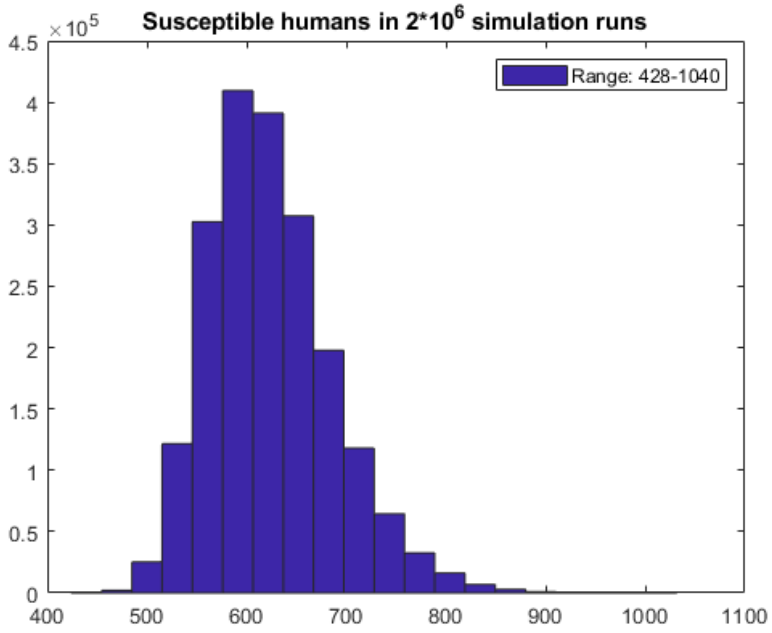

Figure 3: Distribution of the final amount of susceptible humans in $2 \cdot 10^6$ runs

4.1 Performance

The measure the performance of this implementation a few tests were performed. To test strong scaling tests were only run up $N = 10^6$ times. The strong scaling was tested by running the simulations $N = 10^6$ times with 1, 2, 4, 8, 16 and 32 PEs. The result can be seen in figure 4.

it will only be sorted by the built in `qsort()` which makes the results difficult to compare.

4.2 Strong scaling

Table 1: Average execution times for 10^6 computations and varying processing elements

PEs	Execution time(s)
1	116.6843
2	119.6849
4	35.3548
8	17.9493
16	14.2703
32	8.6913

Table 2: Execution times for the final large computations, all conducted on 16 PEs

N	Execution time(s)
10^6	152.9229
$1.6 \cdot 10^6$	245.9641
$2 \cdot 10^6$	213.0677

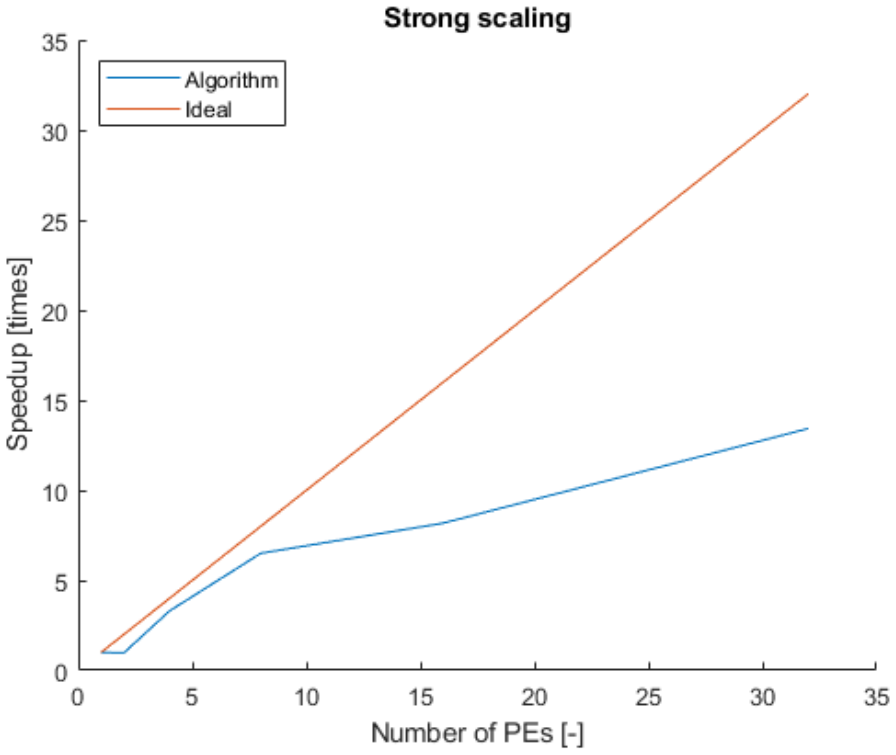

Figure 4: Strong scaling

4.3 Weak scaling

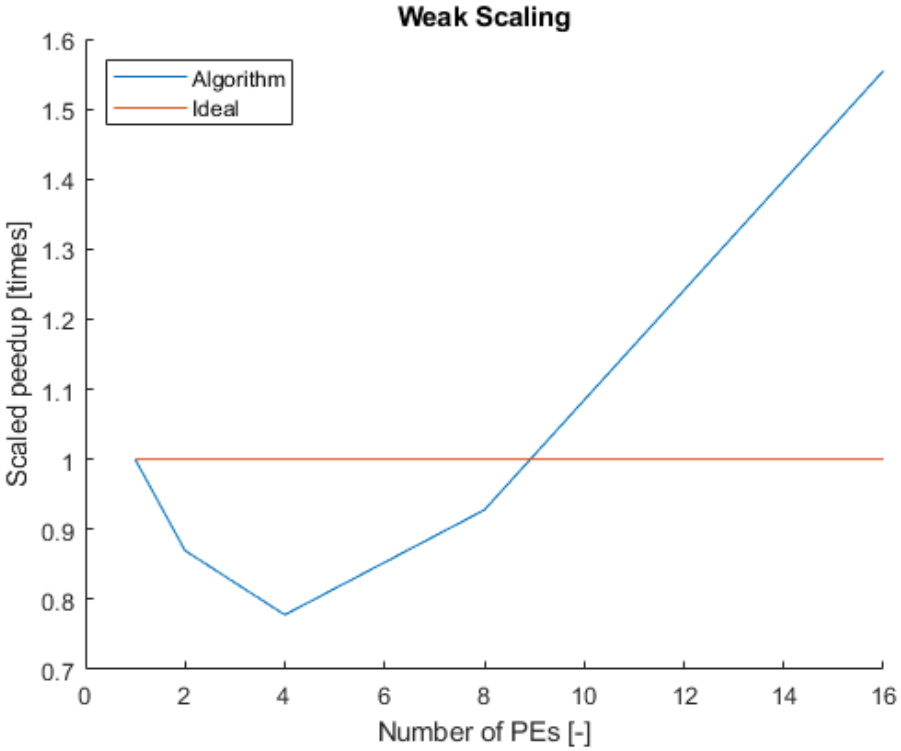

Figure 5: Weak scaling

4.4 Variance

There was quite a bit of variance in the computation times. Here i present some more values to show differences between runs, and also how it could vary between different computers.

Table 3: Multiple runs on Vitsippa

PEs	Execution time(s)		
1	180.9899	143.2432	69.0342
2	179.6639	114.6116	113.6574
4	37.7090	33.0558	36.0763
8	17.9802	18.0932	17.8022
16	9.9717	14.8115	15.6672
32	8.3604	8.8604	8.3404

Table 4: Multiple runs on Tussilago

PEs	Execution time(s)		
1	64.3685	133.6954	108.7749
2	80.1661	113.8797	116.1309
4	35.5937	35.7601	33.934
8	17.932	17.9957	17.8927
16	14.989	14.8536	15.329
32	8.7482	8.7804	8.7239

5 Conclusions

The results from the simulation itself i would say speak for themselves. They are of a normal type distribution and seem to corroborate each other even with very large numbers. As to how reasonable the results are i don't know, but we expect a normal distribution, so at least something went right.

Regarding scaling, the implementation show somewhat good scaling. In figure 4 we can see that it starts of ideal at least, even if it becomes worse with more elements. I'm not sure why it worsens. Usually it would have to do with the overhead of threading, but here i only measure the execution times, which does not take that into account. The weak scaling in figure 5 is less consistent than the strong scaling. As it both gets better and worse with more elements. This shows a bit better the variance of the runs timewise.

Lastly i would touch on the variance. I would say it is most clear in the time it took for the long runs, in table 2. Where 2 million runs went faster than 1,6 million. I show this in tables 4 and 3 as well, where different runs would take very different times. When determining weak and strong scaling i tried to take this into account by taking the average computing times of several tests.