Parallel and Distributed Programing Individual Project

Fredrik Gustafsson

June 14, 2020

1. Introduction

Malaria is spread through the bite of female mosquitos of the specie Anopheles [1]. Every year an estimated 1.5 to 2 million deaths are caused by malaria with around half of the human population being at risk of getting infected [2]. Therefore it is of great interest to try and model the spread of malaria through mosquito-human and human-human infection to get greater insight into said dynamics.

2. Project Description

In this Project the spread of malaria through mosquito-human infection and human-human infection is modelled with Gillespie's direct method also called "*Stochastic Simulation Algorithm (SSA)*". The pseudocode below, Algorithm 1 describes how the algorithm works.

Algorithm 1: Gillespie's direct method (SSA)

```
Set final simulation time T, current time t = 0, and $\mathbf{x}=\mathbf{x}_0$
while $t<T$ do
    Compute $\boldsymbol{w}$ = prop($\mathbf{x}$)
    Compute $a_0 = \sum_{i=0}^{n} \boldsymbol{w}(i)$
    Generate two uniform random numbers $u_1, u_2$ between 0 and 1
    Set $\tau = -ln(\frac{u_1}{a_0})$
    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 $\mathbf{x} = \mathbf{x} + P(r,:)$
    update $t = t + \tau$
end
```

In Algorithm 1 **x** is a state vector containing the state of the system, in this case it is a one by seven vector, $\boldsymbol{w}$ is a one by 15 vector containing derived properties from the current state of the system, and P is a matrix whose rows describes how the different states are updated. Since we are interested in finding the distribution of susceptible humans, first component of $\mathbf{x}$ we perform monte carlo simulation of Algorithm 1. Monte carlo simulation basically mean that we perform a simulation of a stochastic model multiple times whith the result being a distribution of results, as described in Algorithm 2.

Algorithm 2: Monte Carlo simulation

```
1 Set number of simulations N
2 for $i$ = $1, 2, 3, ..., N$ do
3     Perform one execution of code
4     Store result of interest in suitable format
5 end
6 Calculate properties of interest
```

Since we are interested in the distribution the properties of interest are the bounds of the distribution when the result is plotted as a histogram and the histogram, distribution, itself.

3. Solution Approach

Since Algorithm 1 is stochastic, both state vector and the time update uses random variables, we can not parallelise it. However Algorithm 2 is highly parallelisable since no monte carlo experiment depends on any other hence one PE performing 10000 monte carlo experiments or 10000 PEs performing a single experiment each should not have any effects on the final result. This would also mean that the load balance between each PE should be near ideal, as long as we ensure that N is divisble by the number of PEs such that $N = n * p$.

When implementing Algorithm 1 and 2 each PE intialises the matrix P before starting the monte carlo experiments. Furthermore line 7 in Algorithm 1 is solved by incrementing the leftmost sum, $\sum_{k=1}^{r-1} \boldsymbol{w}(\mathrm{k})$ until the condition, <, is no longer fullfilled and then subtracting one. To ensure that each PE creates a unique serie of meassurement each PE sets the seed for drawing random numbers to $seed = CurrentTime + rank$.

Each PE performs $N/size$ monte carlo experiments and store their local results in an one by $N/size$ vector as well as the recording their smallest and largest values during the experiments. To prepare the data as a histogram the PEs then exchange the smallest and largest values using $MPI\_Allgather$ since the resulting list will be fairly small, $2{*}size$, each PE simply runs through it to find the globally smallest and largest value. Once these valuse have been found the interval can be calculated and finally the bin index for each monte carlo experiment can be found and the frequency for each bin can be calculated. Finally the PE with rank 0 collects the each locally made histogram using either *MPI_Reduce*, or with MPI_Send and MPI_Recv, in both approaches the sum for each bin is calculated. As output the smallest and largest value together with the frequency for each bin is printed to a file. Said file can then be plotted in a program of choice e.g. matlab.

4. Experiments

The first thing investigated was wheter or not the parallel implementation had a large performance impact compared to a purely sequential implementation. Then a test was performed to check if there was any major difference in performance when using MPI_Reduce compared to using MPI_Send and MPI_Recv. After that the strong scaling of the program was investigated and finally the weak scaling was meassured. All performance experiments were performed on the linux server *Gullviva*

4.1. Comparison of Sequential and single PE Implementation

To compare the performance of the sequential code with that of one PE the code was executed five times for 5000, 10000, and 20000 monte carlo experiments. The results are shown in Table 1 below. For the single PE case the code was tested when using MPI_Reduce to construct the final result or when using MPI_Send and MPI_Recv to see if there was any performance difference.

	5000	10000	20000	Average increase in time (%)
Sequential	16.1540	33.8640	64.4160	0%
Single PE using reduce	19.4981	39.0354	77.6663	18.85%
Single PE using Send/Recv	19.3974	39.0404	77.6540	18.64%

Table 1: Average time in seconds for five executions of sequential and single PE, either using MPI_Reduce or MPI_Send/MPI_Recv, program for 5000, 10000, and 20000 MC experiments. Last column is the average increase in time calculated as $\frac{1}{3}\sum_{i=1}^{3}\frac{singlePE_i}{Sequential_i}$

As we can see there is noticeable decrease in performance when comparing both MPI implementations with the sequential implementation. As a final test the end timer of the parallel program was moved to be just before doing the collection. Doing this resulted in virtually no change in performance hence the slowdon is not caused by the collection.

4.2. MPI_Reduce Compared to MPI_Send/MPI_Recv

One thing which might affect the parallel performance is the communication when collecting the final results when using different methods. One way to collect the data is using MPI_Reduce to sumate each bins into a final histogram while another way is to use a combination of memcpy, MPI_Send, and MPI_Recv to achieve the same result. To investigate if there is any difference in performance both methods where tested using one, two, four, eight, and 16 PEs performing $20 * 10^3$ MC experiments, the result is shown in Table 2.

nr. of PEs	1	2	4	8	16
MPI_Reduce	77.9730	39.1895	19.5710	9.8922	5.5384
MPI_Send/MPI_Recv	77.9331	39.0767	19.5393	9.8397	5.6257
$\frac{MPI}{MPI\_Send/MPI\_Recv}$	1.0005	1.0029	1.0016	1.0053	0.9845

Table 2: Average time in seconds for five executions of using program with either MPI_Reduce or MPI_Send and MPI_Recv using $20*10^3$ MC experiments tested on one, two, four, eight, and 16 PEs last row indicates which approach was faster with x<1 indicating MPI_Reduce being faster and x>1 indicating MPI_Send and MPI_Recv being faster.

From this, and in conjunction with the results in 1, we can see that there is virtually no difference in regards to performance between the two approaches however using a MPI_Send/MPI_Recv together with a loop to construct the total histogram appears to have a slight edge over MPI_Reduce and will therefore be used for the strong and weak scaling experiments.

4.3. Strong scaling

Strong scaling experiments were performed on *Gullviva* the code was tested using one, two, four, eight, and 16 PEs using three different amount of MC experiments, $1 * 10^4$, $5 * 10^4$, and $1 * 10^5$. The results are shown in Table 3 and the speedup as a function of number of PEs is shown in Figure 1 below.

N \ PEs	1	2	4	8	16
10^4	38.813	19.644	10.566	5.09	2.541
$5 * 10^4$	194.851	97.872	49.100	24.566	12.584
10^5	389.959	195.395	97.515	48.989	25.436

Table 3: Average time in seconds for five executions of program usin one, two, four, eight, and 16 PEs performing 10^4, $5 * 10^4$, and 10^5 MC experiments.

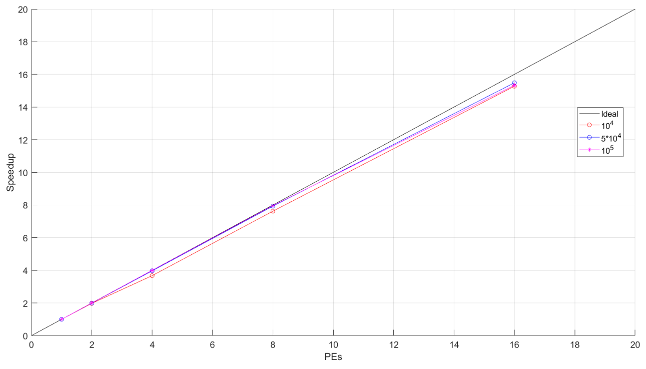


Figure 1: Plot of strong scaling speedup for three problem sizes, 10^4, $5 * 10^4$, and 10^5.

4.4. Weak Scaling

To test the weak scaling of the code we can start with noting, by examing Table 1, that the complexity of the program appears to be O(n) performing twice as many MC experiments doubled the time. Since we want to try and keep the workload constant a doubling in PEs should therefore mean that we perform twice as many experiments. The results are shown in Table 4 and Figure 2 below.

PEs	N	Average time
1	$6*10^3$	23.455
2	$12*10^3$	23.451
4	$24*10^3$	23.692
8	$48*10^3$	23.685
16	$96*10^3$	24.475

Table 4: Average time in seconds for weak scaling experiment where N is number of MC experiments.

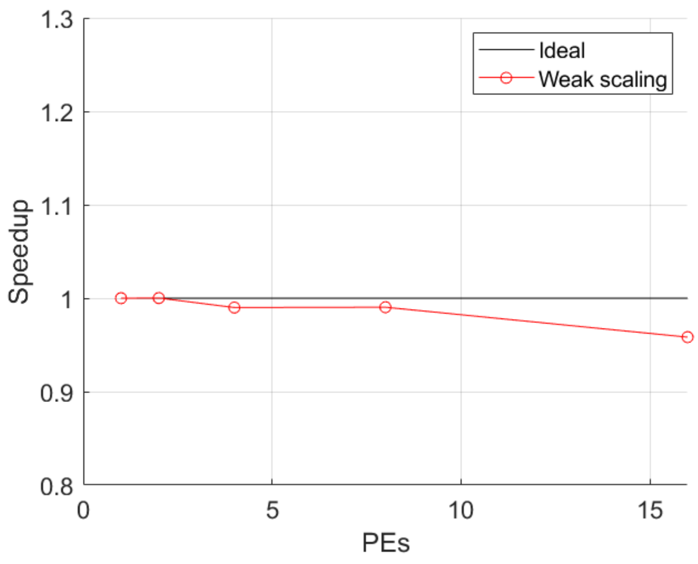


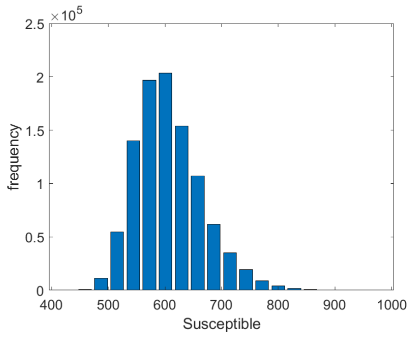
Figure 2: Plot of weak scaling speedup experiment.

4.5. Distribution of Susceptible Humans

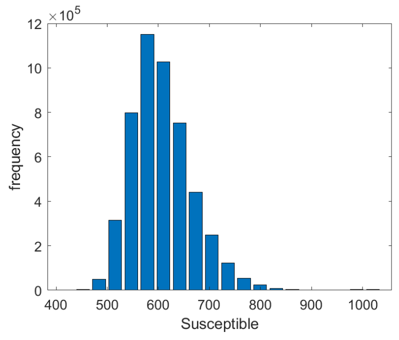
To study the distribution of susceptible humans at time $T = 100$ three experiments were performed using 10^6, $5 * 10^6$, and 10^7 MC experiments each. The resulting histograms, with 20 bins, are shown in Figure 3a-3c while information regarding the length of each bin and the upper and lower limits of the histogram are shown in table 5. Plotting of histogram was performed in matlab using the resulting data file from the program containing the min and max value and the frequency for each bin.

N	Lower limit	Upper limit	Interval width
10^6	416	985	28.45
$5*10^6$	405	1036	31.55
10^7	407	1045	31.90

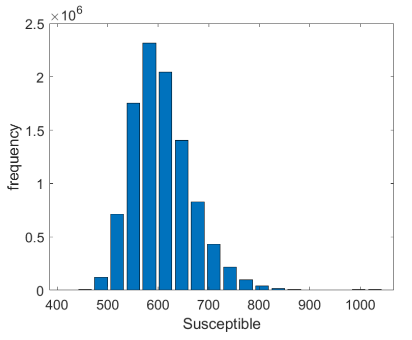
Table 5: Lower limit, upper limit, and interval width of histogram with 20 bins for susceptible humans at time T=100 when performing N MC experiments.



(a) Histogram with 20 bins of susceptible humans at time T=100 when performing 10^6 MC experiments.



(b) Histogram with 20 bins of susceptible humans at time T=100 when performing $5 * 10^6$ MC experiments.



(c) Histogram with 20 bins of susceptible humans at time T=100 when performing 10^7 MC experiments.

Figure 3: Histogram over suscecptible humans at time T=100 for 10^6, $5 * 10^6$, and 10^7 MC experiments.

5. Conclusion

Using MPI slows down the program as Table 1 shows. However this slowdown is not due to any inefficiencies when using MPI_Reduce, compared to MPI_Send/MPI_Recv as Table 2 indicates, although the later implementation has a slight edge in four out of five tests. Nonetheless the implemented algorithm has good strong scaling, as seen in Figure 1 where the problem with the least MC experiments displays the worst strong scaling. That the algorithm has a bit worse scaling when performing less MC experiments could be either due to the stochastic term used when updating the time, which means that more MC experiments should make the average number of iterations approach an expectation value, or more likely the decrease in performance has to do with time spent on communication compared to time spent performing experiments and binning the data. The decrease in parallel performance can also be seen in Figure 2 where the weak scaling speedup has decreased to 0.958 at 16 PEs. This decrease is not great especially seeing as it is the steepest decrease in performance. If the decrease would continue at the same rate as it did from two PEs to 16 then the weak scaling should be seen as decent.

If we look at the results of performing different number of MC experiments 3b-3c we can see that the result appears to follow a Gaussian or even more a Poisson distribution. Since, as shown in Table 5, the lower and upper limits, interval widths and mean value are similar for the three different numbers of experiments which is as expected under the assumption that the results indeed follows a distribution.

From these experiments we can then see that the algorithm in question was well suited for being parallelized achieving a good strong and weak scaling. Furthermore the mathematical model and parameter choice give a distribution with a peak between around 566-598 susceptible humans after 100 days of disease spread.

References

[1] Edoardo Beretta, Vincenzo Capasso, and Dario G. Garao. "A mathematical model for malaria transmission with asymptomatic carriers and two age groups in the human population". In: *Mathematical Biosciences* 300 (2018), pp. 87–101. ISSN: 0025-5564. DOI: `https://doi.org/10.1016/j.mbs.2018.03.024`. URL: `http://www.sciencedirect.com/science/article/pii/S0025556417304893`.

[2] C.P. McCormack, A.C. Ghani, and N.M. Ferguson. "Fine-scale modelling finds that breeding site fragmentation can reduce mosquito population persistence". In: *Communications Biology* 2 (2019), p. 273. ISSN: 2399-3642. DOI: `https://doi.org/10.1038/s42003-019-0525-0`.