Uppsala University

Parallel & Distributed Programming

1TD070

Individual Project

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

Student:
Panagiotis Stefanos Aslanis

Teachers:
Maya Neytcheva, Course Director
Jarmo Rantakokko, Teacher

June 5, 2021

1 Introduction

This project aims to implement and evaluate the performance of parallel solution to run a series of Monte Carlo (MC) experiments combined with Gillespie's stochastic simulation algorithm (SSA) to simulate the developement of malaria epidemic; For the parallel part I use the OpenMPI library[1]. In general Monte Carlo simulations repeat the same procedure(i.e., repeated sampling) over and over producing a series of results that we then data mine in order to get some desired insight from our data. The main idea is to use the randomness in simulating problems that might be deterministic and as such obtain solutions that we would not be able to with other methods for the same problem[2]. The SSA method we follow which was first introduced by Daniel T. Gillepsie on January of 1976 is an exact method for numerically calculating, taking into consideration the stochastic formulation of chemical kinetics, the time evolution of species within their environment following a set of rules that describes their interactions[3]. SSA is a stochastic compact computer oriented Monte Carlo simulation procedure and it takes into account the fluctuations and correlations of the problem while a deterministic approach does not[3].

2 Problem description

In this simulation the data of interest is the number of humans susceptible to the malaria disease and its evolution over time; phenomenons that evolve over time are usually studied through partial differential equations. In this case all required variable values(e.g.,initial values for vector x and evolution matrix P) as well as a C language programming ready function that calculates the required probability quantities is provided. In Table 1 the constant variables are denoted and described and Table 2 displays the relation between those constants and the values that evolve during the simulation which are nothing but a vector of seven elements each denoting a different integer value, more details will not be presented since it is out of context to explore the different mathematical equations that constitute the simulation rather to implement it and parallelize it, then someone with expertise in epidemiology modeling can take over and put it to use. Vector x is initialized with values $[900, 900, 30, 330, 50, 270, 20]$, the value that we study in this project is $x[0]$

Table 1: Simulation Constant values

Constant Variables	Description
$\lambda_\eta = 20$	Birth number of humans
$\lambda_\mu = 0.5$	Birth number of mosquito
$\beta = 0.075$	Biting rate of mosquito
$\beta_\eta = 0.3$	Probability that a bite results in transmission of disease to human
$\beta_\mu = 0.5$	Probability for mosquito transition
$\mu_\eta = 0.015$	Mosquito mortality rate
$\alpha_\mu = 0.6$	Rate of progression from exposed to infectious state, mosquitoes
$\mu_\mu = 0.02$	Mosquito mortality rate
$\delta_\eta = 0.05$	Disease induced death rate, humans
$\delta_m = 0.15$	Disease induced death rate, mosquitoes
$\alpha_\eta = 0.6$	Rate of progression from exposed to infectious state, humans
$R = 0.05$	Recovery rate, humans
$\omega = 0.02$	Loss of immunity rate, humans
$n_\eta = 0.5$	Proportion of an antibody produced by human
$b_\mu = 0.15$	Proportion of mosquito antibody in response to infection by human

Table 2: Probability Density Function Vector

$w[0] = \lambda_\eta$	$w[1] = \mu_\eta * x[0]$	$w[2] = \frac{\beta * \beta_\eta * x[0] * x[5]}{1 + n_\eta * x[5]}$
$w[3] = \lambda_\mu$	$w[4] = \mu_\mu * x[1]$	$w[5] = \frac{\beta * \beta_\eta * x[1] * x[4]}{1 + n_\eta * x[4]}$
$w[6] = \mu_\eta * x[2]$	$w[7] = \alpha_\eta * x[2]$	$w[8] = \mu_\mu * x[3]$
$w[9] = \alpha_\mu * x[3]$	$w[10] = (\mu_\eta + \delta_\eta) * x[4]$	$w[11] = R * x[4]$
$w[12] = \mu_\mu$	$w[13] = \omega * x[6]$	$w[14] = \mu_\eta \times x[6]$

The P matrix describes the updates that happen in the x vector according to the SSA algorithm included in Table 3.

Table 3: P Matrix(15,7) presented as a Table

1	0	0	0	0	0	0
-1	0	0	0	0	0	0
-1	0	1	0	0	0	0
0	1	0	0	0	0	0
0	-1	0	0	0	0	0
0	-1	0	1	0	0	0
0	0	-1	0	0	0	0
0	0	-1	0	1	0	0
0	0	0	-1	0	0	0
0	0	0	-1	0	1	0
0	0	0	0	-1	0	0
0	0	0	0	-1	0	1
0	0	0	0	0	-1	0
1	0	0	0	0	0	-1
0	0	0	0	0	0	-1

3 Solution

3.1 Serial Implementation

The solution is a straight forward implementation of the MC method combined with Gillepsies's SSA as it was described in the project instructions; each MC experiment executes a run of the SSA for a number of steps which is unknown and whose duration depends among others on variable generated through a random distribution. For the serial procedure followed the pseudocode is displayed in Algorithm 1.

Algorithm 1 Monte Carlo Simulation following Gillepsie's SSA procedure(Serial)

```
**Parameters:** [$N$] Number of MC experiments, [$T$] SSA max runtime, [$t$] SSA initial time, [$w$] density probability vector, [$\alpha_0$], $u1, u2$ $\epsilon(0,1)$ initialized randomly, [$\tau$] time update value, [$r$] stopping condition variable, [$x$] state vector, $R$ length of w vector, [$P$] state update vector, $b$ histogram bins number, $interval$ the bins interval value,
1: Set a random seed(diff each time)
2: for $i = 0; i < N; i = i + 1$ do
3:     Initialize $x0 = [900, 900, 30, 330, 50, 270, 20]$
4:     set t = 0, t=100, x=x0
5:     while $t < T$ do
6:         Compute $w = prob(x)$                          ▷ see Table 2 for $prob(x)$
7:         Compute $\alpha_0 = \sum_{i=1}^{R} w(i)$
8:         Set $\tau = -ln(u1)/\alpha_0$
9:         Find r such that $\sum k = 1^{r-1} w(k) < \alpha_0 u2 \leq \sum k = 1^r$
10:        Update the state vector $x = x + P(r, :)$
11:        Update time $t = t + \tau$
12:    end while
13:    Store the results of the experiment to a suitable container
14: end for
15: Collect all the susceptible humans values $x[0]$ from each run of the MC from 13 :
16: Find the min and max among the values stored in previous step
17: Set the $interval = (max - min)/bins$
18: Count how many elements fall in each bin
19: Output the required data to plot a histogram for $x[0]$
```

3.2 Parallel Implementation

It is fortunate that biggest part of the problem is fully parallelizable since it is easy to split the number of MC experiments equally among a number of p processors, that way each processor can do n runs of the MC simulation and the relation that describes that is $N = n \times p$ where N denotes the total number of elements. For the parallel implementation not much change since the code at it's most part is perfectly parallelizable. In fact three more extra steps are required to be added to the serial Algorithm 1 to parallelize it with OpenMPI. For the sake of inclusiveness the full parallel algorithm is also included 2.

Algorithm 2 Monte Carlo Simulation following Gillepsie's SSA procedure(Parallel)

```
    Parameters: [$N$] Number of MC experiments, [$T$] SSA max runtime, [$t$] SSA initial time, [$w$] density probability vector, [$\alpha_0$], $u1, u2 \ \epsilon(0,1)$ initialized randomly, [$\tau$] time update value, [$r$] stopping condition variable, [$x$] state vector, $R$ length of w vector, [$P$] state update vector, $b$ histogram bins number, $interval$ the bins interval value
    OpenMPI Parameters: [$size$] the size of our communicator, [$rank$] current processor rank, [$n$] number of experiments per processor, [$global\_max$] [$global\_min$] the global max and min from all the processors
 1: Initialize MPI Communicator
 2: Set $n = N/size$
 3: Set random seed modifiable by current rank so each processor receives a different seed
 4: for $i = 0; i < n; i = i + 1$ do
 5:     Initialize $x0 = [900, 900, 30, 330, 50, 270, 20]$
 6:     set t = 0, t=100, x=x0
 7:     while $t < T$ do
 8:         Compute $w = prob(x)$                                  ▷ see Table 2 for $prob(x)$
 9:         Compute $\alpha_0 = \sum_{i=1}^{R} w(i)$
10:         Set $\tau = -ln(u1)/\alpha_0$
11:         Find r such that $\sum k = 1^{r-1} w(k) < \alpha_0 u2 \leq \sum k = 1^r$
12:         Update the state vector $x = x + P(r, :)$
13:         Update time $t = t + \tau$
14:     end while
15:     Store the results of the experiment to a suitable container
16: end for
17: Collect all the susceptible humans values $x[0]$ from each run of the MC from 15
18: Find the global min and global max from all the processors
19: Collect all the results from 17 and store them to a single container on ROOT process▷ Parallel part end here
20: Set the $interval = (global\_max - \_globalmin)/bins$
21: Count how many elements fall in each bin
22: Output the required data to plot a histogram for $x[0]$
```

To outline the few additions to the parallelized code, it is worth noting that now after the MC loop finishes for each processor the results are stored locally from each processor and then gathered using the $MPI\_Gather()$ function, a thread safe routine from MPI that gathers values from a group of processes. Then we find the global max and global min using MPI_Reduce which finds and returns the biggest max and smallest min from all the processors and we set those to be the upper and lower bound of the histogram.

4 Experiments

To evaluate the performance of the solution I perform a scalability study, it refers to the ability of the software to deliver greater computational power when the amount of resources is increased. Strong and weak scaling are the two most common types of scaling and indicate the ability of the software to scale.

4.1 Strong Scaling

The strong scaling is in simple terms how does the software run time behave when we have fixed input size but increase the number of processing elements (PEs).

Table 4: Strong Scalability Study for 10^6 Simulations

PE load	# PEs	Time(s)	Speedup
10^6	1	1289.2445	1.0
500000	2	753.5889	1.71
250000	4	345.9602	3.72
125000	8	177.5642	7.26
62500	16	101.7660	12.66
31250	32	49.1400	26.23

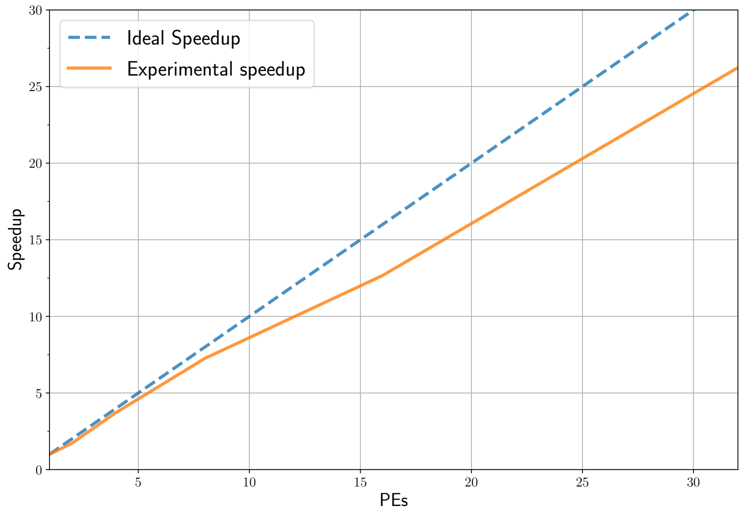


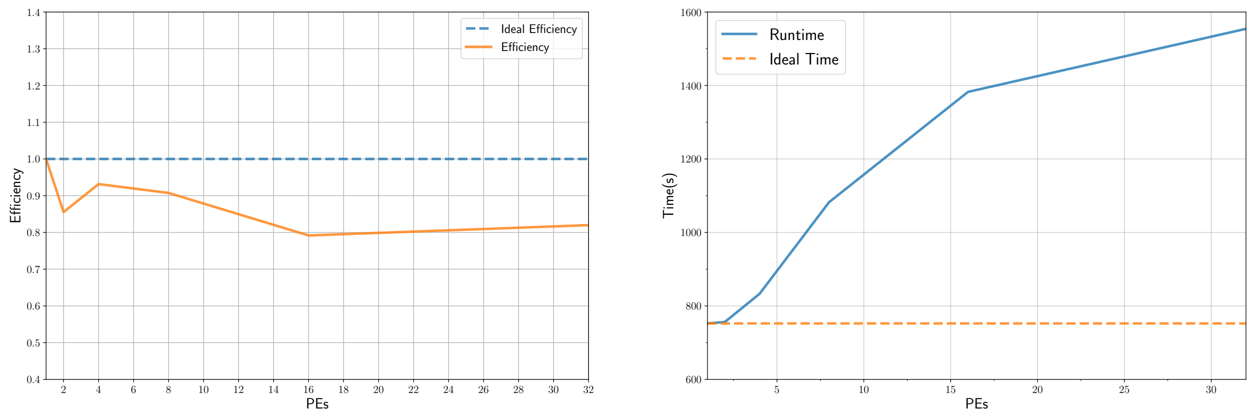
Figure 1: Strong Scalability, speedup

4.2 Weak Scaling

In this subsection the results of weak scalability study are shown, the time complexity of the solution is $O(N)$. Now the load per PE is scaled as we increase their number such that a fixed ratio is kept throughout the experiment.

Table 5: Weak Scalability Study

Total Size	PE load (n)	# PEs	Time(s)	Efficiency
$1 \times 5 \times 10^5$		1	752.3224	1.0
$2 \times 5 \times 10^5$		2	756.2786	0.8554
$4 \times 5 \times 10^5$	500000	4	833.0628	0.9316
$8 \times 5 \times 10^5$		8	1082.3322	0.9075
$16 \times 5 \times 10^5$		16	1382.5938	0.7917
$32 \times 5 \times 10^5$		32	1554.3005	0.8198



(a) Weak scalability, Efficiency

(b) Weak scalability, time

Figure 2: Weak scalability study

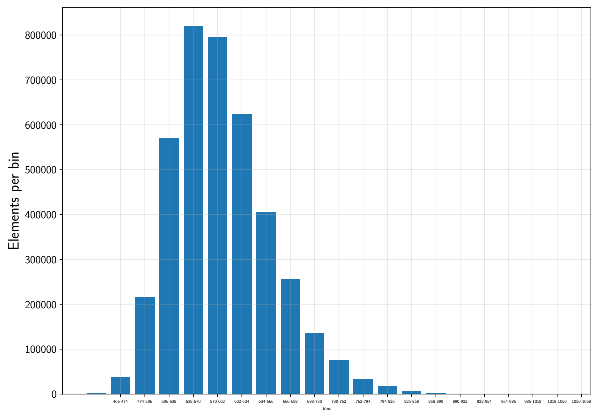
5 Discussion of Results & Output

Overall the performance evaluation produced the expected results. Strong scaling seems to be satisfying enough, it seems that after 4 PEs the experimental time deviates from the ideal speed, as such it would be better to run this code using around 4 PEs. In terms of weak scaling we can safely say that the sweet spot for our code lies at equal workload per processor for 4 PEs meaning $4 \times 5 \times 10^5$ total simulations as this is where we observe the closest point to the ideal constant runtime. The serial part in the code is minimal therefore both strong and weak scaling produce good enough results!
There is room for improvement, for example a hybrid version with OpenMP could be realized, this could potentially improve the run time of the code. In some initial version I was using qsort to find the max and the min this is bad practice since its $O(nlogn)$ while iterating the container the traditional way is just $O(n)$ this gave a good amount of improvement to the code especially when the number of elements is bigger than 10^6. As a final thing, the output histogram plots along with their accompanying data (Table.6) that is requested as an output for our program, is presented.

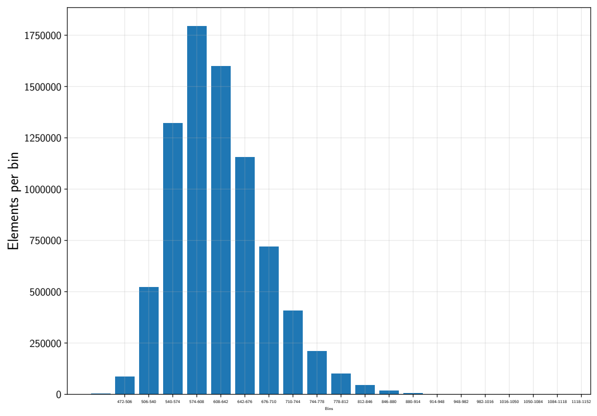
Table 6: Histogram Data

	16×10^6		8×10^6		4×10^6	
Bin	Lower Bound	# Elements Per Bin	Lower Bound	# Elements Per Bin	Lower Bound	# Elements Per Bin
1	475	16339	472	4367	474	2078
2	512	309898	506	85993	506	36872
3	549	1642201	540	522274	538	215961
4	586	3463711	574	1321035	570	570387
5	623	3782464	608	1794704	602	820506
6	660	2982739	642	1600741	634	795706
7	697	1846212	676	1155883	666	622746
8	734	1056081	710	720175	698	405659
9	771	508146	744	408179	730	255638
10	808	238010	778	209973	762	136606
11	845	99522	812	102159	794	75779
12	882	36977	846	45687	826	34365
13	919	13026	880	18599	858	17120
14	956	3282	914	7068	890	6488
15	993	940	948	2204	922	2807
16	1030	221	982	668	954	845
17	1067	69	1016	161	986	285
18	1104	21	1050	54	1018	88
19	1141	8	1084	27	1050	25
20	1178	3	1118	8	1082	9

(a) Histogram of 4×10^6 elements



(b) Histogram of 8×10^6 elements



(c) Histogram of 16×10^6 elements

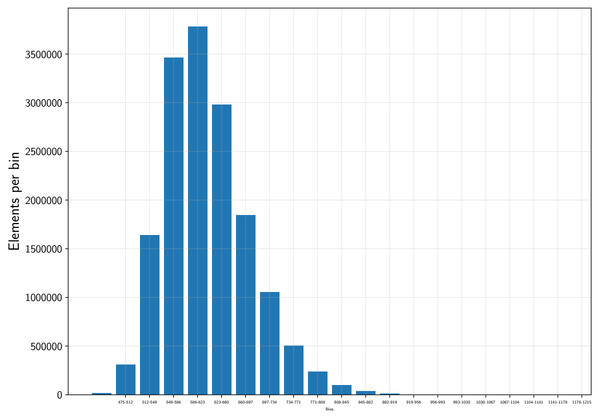


Figure 3: Histogram plots for 4 (3a), 8 (3b), and 16 (3c) million elements respectively

References

[1] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[2] Ghassan Hamra, Richard MacLehose, and David Richardson. Markov Chain Monte Carlo: an introduction for epidemiologists. *International Journal of Epidemiology*, 42(2):627–634, 04 2013.

[3] Daniel T Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4):403–434, 1976.