UPPSALA UNIVERSITY

COURSE NAME: PARALLEL AND DISTRIBUTED PROGRAMMING

COURSE CODE: 1TD070

Malaria simulation using Monte Carlo computations combined with Stochastic Simulation Algorithm

---

Author:
Alhassan JAWAD

Supervisor: Roman Iakymchuk

21-05-2024

Malaria simulation using Monte Carlo computations combined with Stochastic Simulation Algorithm

Abstract

Using MPI libraries, this project simulates the spread of a malaria epidemic using the Monte Carlo method in conjunction with the stochastic simulation algorithm. This project leads to the conclusion that the created software is perfectly parallel and scalable, and that it can generate accurate simulations of malaria epidemics. There is still room for improvement. For example, thread-level optimization utilizing the pthread or OpenMP library to further parallelize the program may be carried out.

1 Introduction

1.1 Background

Malaria is a prehistoric disease that has been investigated and researched for hundreds and hundreds of years. Despite that, it still has an enormous social, economic, and health impact, with it still remaining a major public health issue in close to 85 nations designated as endemic to the disease in 2022 [1]. Nowadays, as technology has advanced with the power to simulate different phenomena, the precise framework that mathematics and mathematical models provide for comprehending the dynamics of disease transmission inside and between hosts and parasites is crucial.

Mathematical models are a way to represent complex disease data with simpler equations. Researchers select the most relevant biological and clinical information to capture how the disease develops and spreads. With that said, an approximation of the complicated reality is called a model, and the processes under study and the intended extrapolation determine the model's structure [2].

Monte Carlo Computations/Simulation

A large family of computing algorithms known as Monte Carlo techniques, or Monte Carlo experiments, relies on iterative random sampling to get numerical results. The fundamental idea is to leverage randomness to find solutions to issues that, in theory, may be deterministic. They are particularly helpful when using alternative ways is difficult or impossible, and they are frequently utilized in mathematics and physical issues. Some of the well-known applications of Monte Carlo methods include estimating pi and approximating integrals [3] [4].

Stochastic Simulation Algorithm

A system containing variables that have the potential to vary stochastically (i.e., randomly) with discrete probability is simulated using stochastic simulation. These random variables' realizations are created and added to a system model. After recording the model's outputs, the procedure is repeated using a fresh set of random data. Until a sufficient quantity of data is collected, these procedures are repeated. Ultimately, the distribution of the outputs provides a framework of expectations for the ranges of values that the variables are more or less likely to fall within, in addition to the most likely predictions [5].

1.2 Problem Description

The project's goal is to use Monte Carlo (MC) Computations in addition to the Stochastic Simulation Algorithm (SSA) to model the spread of a malaria epidemic. The program will execute in parallel on n processors, and p separate stochastic simulations will be performed for each CPU. All nxp test results will be gathered, and all data pointing to susceptible humans will be stored for later usage.

Moreover, statistically grounded visualization (such as histograms) and analysis of the acquired data will be carried out. Tests such as strong scaling and weak scaling will be carried out on the parallelized code.

1.3 Hardware Specifications

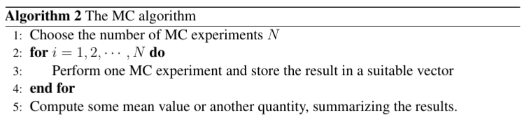
All performance tests were conducted on the *RACKHAM* cluster, which is part of *UPPMAX*, a high-performance computing resource at Uppsala University [6]. The Rackham cluster is configured to have an 486 nodes of 10-core Intel Xeon V4 CPU's each at 2.20 GHz/core, a minimum of 128 GB memory and a CentOS 7 operating system (OS) [7].

2 Algorithms

According to the project instructions given, N parallel simulations should be completed and gathered Monte Carlo simulation for a malaria outbreak in conjunction with the stochastic simulation algorithm.

This means that the simulation function will have 2 input parameters The first is the number of simulations to run while the second parameter I put as the name of the output file for where to write the results of the simulations (values to be used for histogram computations). The output file will contain 2 lines where the first line contains 21 integer values which are the range of the histogram. The second line will be containing the histogram integer values of susceptible humans per intervals.

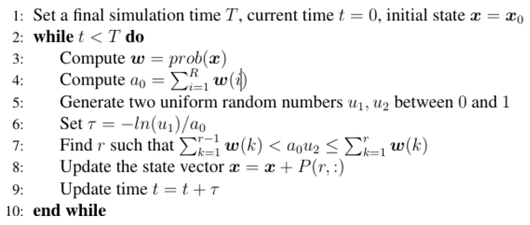
The following 2 figures will show the given pseudo-codes for the MC algorithm together with SSA.



Algorithm 2 The MC algorithm

```
1: Choose the number of MC experiments $N$
2: for $i = 1, 2, \cdots, N$ do
3:     Perform one MC experiment and store the result in a suitable vector
4: end for
5: Compute some mean value or another quantity, summarizing the results.
```

Figure 2.0.1: The MC algorithm as pseudo-code



```
1: Set a final simulation time $T$, current time $t = 0$, initial state $\boldsymbol{x} = \boldsymbol{x}_0$
2: while $t < T$ do
3:     Compute $\boldsymbol{w} = prob(\boldsymbol{x})$
4:     Compute $a_0 = \sum_{i=1}^{R} \boldsymbol{w}(i)$
5:     Generate two uniform random numbers $u_1, u_2$ between $0$ and $1$
6:     Set $\tau = -ln(u_1)/a_0$
7:     Find $r$ such that $\sum_{k=1}^{r-1} \boldsymbol{w}(k) < a_0 u_2 \leq \sum_{k=1}^{r} \boldsymbol{w}(k)$
8:     Update the state vector $\boldsymbol{x} = \boldsymbol{x} + P(r,:)$
9:     Update time $t = t + \tau$
10: end while
```

Figure 2.0.2: Stochastic Simulation Algorithm

Regarding each simulation, Gillespie's direct method is used which is shown in figure 2.0.2. Furthermore, we are using the MPI library, making the calculations' parallelizible. And as each iterations computations are independent of each other, we can make the whole code script perfectly parallelized. Another consequence of the code being perfectly parallelizble is that we will not need to take into consideration the case where N is not divisible by p. As a result, if n nodes/processes are provided for N experiments, each node will do p = N/n experiments, and the final data will be gathered and stored by this node.

What we also have been given regarding the algorithms and the used variables are:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}. \tag{2.0.1}$$

with length of vector x = 7 and length of w = 15. R = 15. The initial $x_0 = [900; 900; 30; 330; 50; 270; 20]$, T=100 and the function *prop* is given so we don't need to write it on our own.

Our first task was to "perform the necessary collection of the results (within the MPI program) and plot a histogram".

2.1 Sequential part of the code

The only part of the code that cannot be parallelized are the writing to the output as it needs to be done at the final stage of the code after all different calculations are done across all processes.

We also have the helper functions such as step 3-4 and step step 7 of the SS algorithm (figure 2.0.2). Those function's will not be parallelized as they are not something that can be done across multiple processes to make the code faster.

2.2 Parallelized part of the code

Inside the main part of the code, we start by initializing the MPI environment, and getting the size of the process pool as well as initializing a variable to keep track of the rank of the current processor.

Afterwards, we made some variable declarations as well as memory declarations to be able to run the SSA and MC algorithm. After starting a custom random number generator and writing the state of the matrix shown in 2.0.1, I started the MC algorithms by creating a for-loop. Inside it I wrote the code for the Gillespie's direct method (SSA).

After writing the algorithms and collecting the susceptible human data that we are after, I used **MPI_Allreduce** to get the global minimum and maximum histogram intervals from across all processes.

Finally came the part of the histogram value calculations where we computed e.g. the range of the histogram and then stored the values in the resulting array. At last, we send the values of the resulting array to a helper function that will write the values to an output file in the format stated above.

And to not get any segmentation errors, I didn't forget to free the resources that I allocated for some of the variables.

The time complexity of all N simulations is proportional to the number O(N), as the cost of each simulation is constant => *The computation's time complexity is O(N/n) if n processors are available*

2.3 Revision 1 Changes

It was decided, after conferring with one of the teachers, that changes to the random number generator seed were required in order to guarantee that every processor utilized a distinct seed. The goal of parallel computing is defeated if the random number generator in each process uses the same seed, as this would result in identical random number sequences in all of the processes. So for each processor, we start the random number generator with a distinct seed, improving the simulation findings' statistical validity and resilience.

To make each processors seed different, I modified the *srand()* and the parameters I used where time(NULL) + rank. time(NULL) returns the current time in seconds since the Epoch (January 1, 1970) and it introduces a type of randomness as each seed will be different from all others [8].

3 Experiments and results

Note that in the performance tests, the time spent on message passing and local computation on each processor was included in the counted time in our performance test (output file writing was excluded). All tests are one on a parallelized code with optimization flags -std=c99 and -O3 enabled when compiling the code.

According to the instructions, for 3 simulation values that are larger than 10^6, provide the bounds of the histogram intervals and plot the histogram. The histogram interval bounds are provided by running the simulation code named *main.c* that I parallelized using MPI. The fallowing table shows the values gotten:

Table 3.0.1: Statistics for 3 simulation values

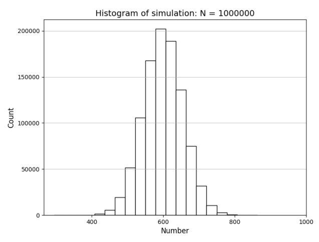
N	Mean Time (s)	Standard Deviation (s)
10^6	606.89	58.54
$12 * 10^5$	606.87	58.72
$15 * 10^5$	606.74	58.59

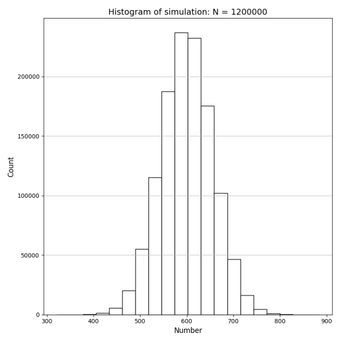
Regarding the table above, the mean time is taken from 5 reruns of each test case while the standard deviation is computed using eq. 3.0.1.

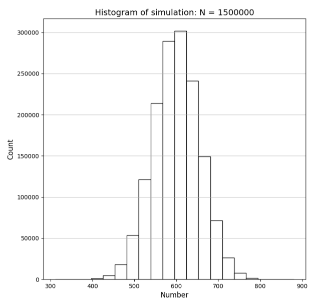
$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} \tag{3.0.1}$$

where s represents the sample standard deviation, x_i is the individual time value from each rerun, $\bar{x}$ is the mean value across all reruns and n is the number of reruns.

To make sure that the obtained data will reasonable a normal distribution for large datasets N, I extracted the contained data values from the output files and plotted a histogram using python.







As we can see, all the figures show a normal distribution which means that for large datasets, this Malaria simulations code gives us data that have a normal distribution framework, according to what the instructions assumed.

3.1 Strong Scalability

Regarding Strong Scalability, the problem size, number of simulations in this case, remains fixed while the numbers of processes are changed. This leads to a situation where we get reduced workload per processor as when increase the number of processes used. I tested several situations with a fixed problem size of $N = 10^5$. Every test case is run5 times, and the time cost is presented using the lowest time required. The number of processes was varied between 1-16 processes and the following table shows the results of these experiments:

Table 3.1.2: Strong Scalability Performance tests

Total N	Processors	Time (s)	Speedup	Efficiency
$N = 10^5$	1	130.8	1.00	1
$N = 10^5$	2	66.5	1.97	0.99
$N = 10^5$	4	34.5	3.79	0.95
$N = 10^5$	5	27.3	4.81	0.96
$N = 10^5$	8	17.1	7.65	0.96
$N = 10^5$	10	13.5	9.62	0.96
$N = 10^5$	16	8.74	14.95	0.93

What the table shows is that when the number of processors increases, the program's performance increases proportionally. Regrettably, the optimal speedup cannot be achieved because of the time spent on the system's communication etc. On the bright side, the actual speedup is really near to the optimal speedup line, suggesting that the program may scale well for problems of fixed sizes. The following figure shows the actual speedup presented above versus the optimal speedup.

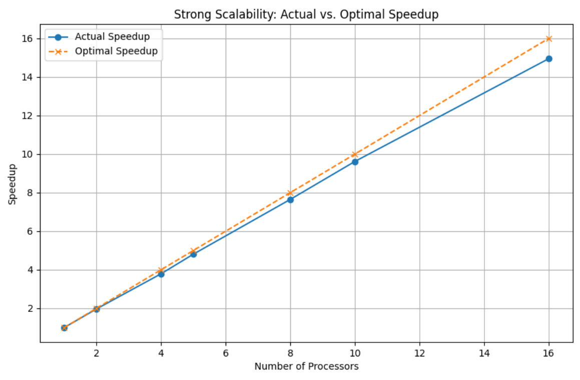


Figure 3.1.1: Actual Speedup vs Optimal Speedup

3.2 Weak Scalability

Regarding Weak Scalability, the problem size and the number of processors are both changed proportionally to each other. As a result, each processor has a constant workload no matter if we decrease or increase the number of processes. For the following performance tests, we fixed the problem size per process as $N = 10^4$ simulations. Similar to strong scalability testing, every test case is run 5 times, and the test case's time cost is displayed using the minimal time cost.

Table 3.2.3: Weak Scalability Performance Tests

Workload per processes	Processors	N	Time (s)	Efficiency
10^4	1	10000	12.1	1
10^4	2	20000	12.5	0.97
10^4	4	40000	12.8	0.94
10^4	5	50000	13.3	0.91
10^4	8	80000	14.2	0.85
10^4	10	100000	14.2	0.85
10^4	16	160000	14.2	0.85

As the table shows, when the calculation quantity (problem size) for each processor is fixed, the program's efficiency typically stays steady and extremely close to 1, even though it does somewhat decline as the number of processors grows. The following figure shows efficiency plotted in a graph for better readability:

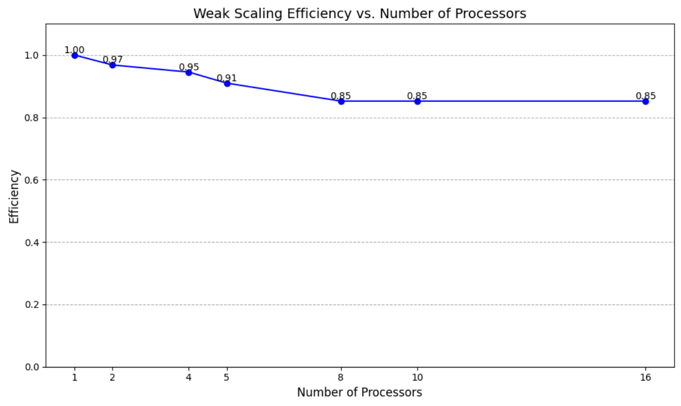


Figure 3.2.2: Efficiency of weak scaling

From what we can see from figure 3.2.3, after an initial decrease up to 5 processes, efficiency keeps a steady pace at 0.85 for different number of processes.

The initial decrease of efficiency is due to many reasons, but for my case I suspect that these 2 are the most probable reasons:

- Communication Overhead
- Amdahl's Law that says that even perfectly parallelized code that have small serial code will have decrease in efficiency.

The reason for it not decreasing more is that the code is perfectly parallelized where every single simulation is completely autonomous, and processor-to-processor communications is only needed once, upon completion of the simulation and data collection.

4 Discussion

There are 2 times in my code where there is a need for communication between processes.

Regarding the first communication occasion:

It was following the computation of local maximum and minimum values in each process. Inter-process communication was necessary to enable each process to get the global maximum and minimum value. There, I used the MPI_Allreduce() function to reduce all local min/max values and remake them in global min/max values that are "shared" everywhere between the processes.

An alternative method was to compute the global minimum and maximum locally using a Send/Recv pair, and then broadcast the results to other processes. Another method is to use the MPI_Allgather() function to collect all maximum and lowest values, then locally compute the global ones.

When comparing all these functions, I chose to use MPI_Allreduce() as it was a single line command that did multiple steps and it's also the most efficient method because it just requires one system communication call.

Regarding the second communication occasion:

It was following the completion of local statistics, where local data collecting is necessary. There I used MPI_Reduce() to gather all data and reduce them until I got a global data to be "shared" between all processes.

An alternative method was to use the MPI_Gather() function to store all arrays into a single, larger array inside a single process. Afterwards, I needed to sum data from each process and store it in another array and redo the previous step again. In contrast, MPI_Reduce is far simpler and allows for data collection and summation in a single function call.

Further improvements can be made like for example thread-level optimization using Pthreads or OpenMP to parallelize the program further. Even in-line functions can hopefully be used to replace the place of some function callings thus improving e.g. efficiency.

5 Referencens

[1] - P. Venkatesan, *The 2023 WHO World malaria report*, The Lancet Microbe, vol. 5, no. 2, pp. E51-E52, Feb. 2024. doi: 10.1016/S2666-5247(24)00016-8 [Accessed: May 21, 2024]

[2] - S. Mandal, R. R. Sarkar, and S. Sinha, *Mathematical Models of Malaria—A Review*, Malaria Journal, vol. 10, Jul. 2011, Art. no. 202. doi: 10.1186/1475-2875-10-202 [Accessed: May 21, 2024]

[3] - N. Singh Chauhan, *Introduction to Monte Carlo Simulation in Statistics*, The AI Dream, Sep 19, 2021. [Accessed: May 22, 2024]

[4] - *Monte Carlo method*, Wikipedia, The Free Encyclopedia, [Online]. Available: https://en.wikipedia.org/wiki/Monte_Carlo_method. [Accessed: May 22, 2024].

[5] - *Stochastic Simulation*, Wikipedia, The Free Encyclopedia, [Online]. Available: https://en.wikipedia.org/wiki/Stochastic_simulation. [Accessed: May 22, 2024].

[6] - *Uppmax*, Uppsala University, [Online]. Available: https://www.uu.se/en/centre/uppmax. [Accessed: May 23, 2024].

[7] - *Rackham Cluster*, Uppsala University, [Online]. Available: https://www.uu.se/centrum/uppmax/resurser/kluster/rackham. [Accessed: May 23, 2024].

[8] - *srand*, GeeksForGeeks, [Online]. Available: https://www.geeksforgeeks.org/rand-and-srand-in-ccpp/. [Accessed: June 5, 2024].