Uppsala University

Parallel and Distributed Programming

1TD070

Individual Project: Parallel Monte Carlo Malaria Simulations

By:
Ture Hassler

May 25, 2023

Contents

1 Introduction

Malaria is a life threatening disease that spreads through mosquitoes and is especially prevalent in tropical countries. Simulating the spread of this disease, and many others like it, is a very valuable tool in predicting and preventing outbreaks. However it can be hard to simulate due to the stochastic nature of the spread of disease. One way to perform these simulations is by using a stochastic simulation algorithm (SSA) to simulate the spread of a single outbreak. One can then perform and average over many of these independent, random simulations in what is called the Monte Carlo method. From this a good approximation of the mean, median and distribution of possible outcomes can be created.

2 Problem description

The goal of this project was first to implement the Monte Carlo SSA algorithm in parallel, using C and MPI. Then to calculate a histogram of the possible outcomes, including the min and max values. In addition to this, the time splits when each processor reached the simulated time of 25, 50 and 100 was also to be recorded for each process.

3 Solution method

The code takes the total number of Monte Carlo simulations N as input, it then divides them equally between each process $n = N/p$. Each process then performs their simulations independently and saves the number of susceptible humans in a local array.

The simulation consists of a state vector x and the propensities w for different reactions to occur. The simulation is performed in a while loop until 100 simulated time units have occurred. For each iteration through the while loop two random numbers are generated which are then used to calculate the timestep and what reaction occured. Here we also check if the simulated time has passed a certain milestone, after which we record the wall time and add it to a local array. One simple serial optimization performed here is that all memory allocations are performed outside the function and are instead sent as input arguments with a pointer to each array, reducing unnecessary repeated memory allocations.

After that they need to compute the histogram bins. Each process goes through their local results array and finds the min and max values of that process, after which the global min and max values are found using MPI_Allreduce together with MPI_Min and MPI_Max. From this each process finds the histogram bins and counts how many occurrences they had in each bin. After that the results are summed up again using MPI_Reduce and MPI_Sum. The allreduce is no longer needed on this step since only process 0 needs the results from the bins. Furthermore each process counts the occurrences on a local array and then sums them up once, reducing the messaging overhead.

The timesplits uses MPI one sided communications using a shared memory window. This is initialized before the simulations are performed, and after the simulations each process calls MPI_Put to save their timesplits, which can then be accessed by process 0.

After the simulations and timesplits are calculated the maximum time is found using yet another MPI_Reduce and MPI_Max and the ouput is printed to the terminal. The results of the histogram is also saved in a file and printed to the terminal, depending on the input arguments.

4 Experiments

The performance of the code was evaluated with both strong and weak scaling experiments on the Uppmax cluster Snowy with the following system specifications:

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

4.1 Simulation results

The histogram results when running a total of $N = 10^6$ simulations can be seen in figure 1.

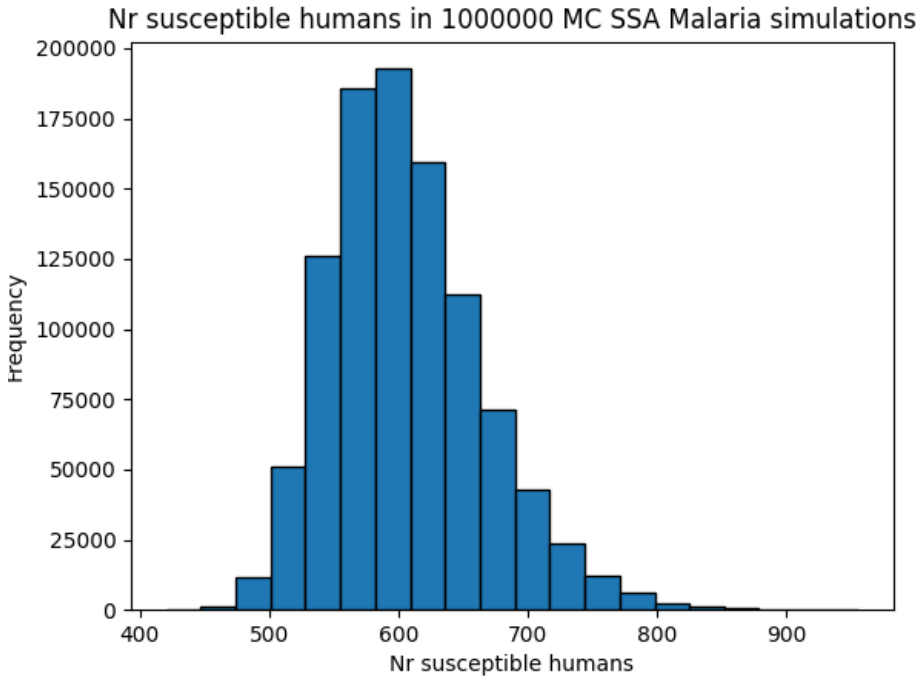

Figure 1: Histogram of the number of susceptible humans in 1m Monte Carlo stochastic simulation algorithm malaria simulations

As we can see from the figure, the distribution of outcomes is somewhat similar to a normal

distribution, with a mean somewhere around 600 susceptible humans. The max and min values of three simulations with large N can be seen below:

Table 1: Minimum and maximum number of susceptible humans after varying number of Monte Carlo simulations

Nr Simulations	Min	Max
$1 \cdot 10^6$	420	956
$2 \cdot 10^6$	440	956
$4 \cdot 10^6$	440	956

4.2 Strong scaling

For the strong scaling, the total number of iterations was kept constant at $N = 10^5$, while the number of processes was increased from 1 to 64. The execution times can be seen in figure 2 and the speedup can be seen in figure 3.

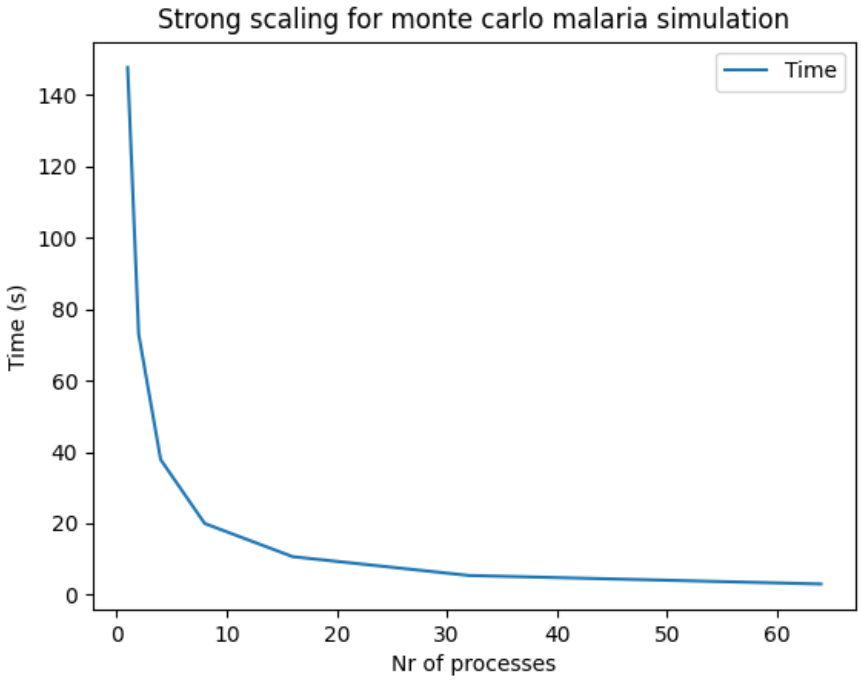

Figure 2: Strong scaling times for Monte Carlo malaria simulations. N=100000

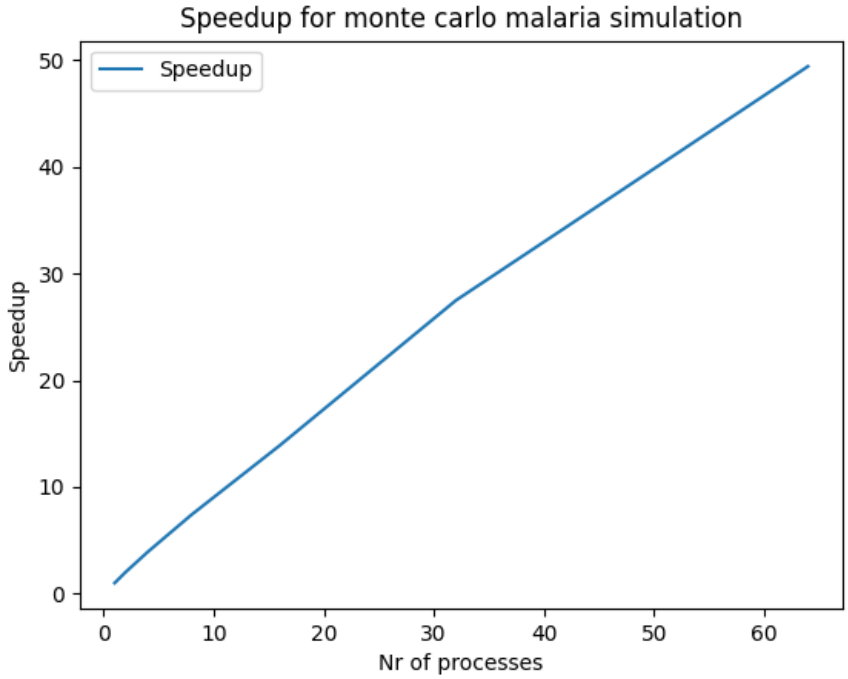

Figure 3: Strong scaling speedups for Monte Carlo malaria simulations. N=100000

4.3 Weak scaling

For the weak scaling experiments, the total number of work per process was kept constant at $n = 10^5$. Since the workload increases linearly with the number of simulations performed, aside from possible synchronization and messaging delays, the number of simulations was increased from $n = 1 \cdot 10^5$ to $n = 64 \cdot 10^5$ while the number of processes was increase from 1 to 64. The speedups achieved can be seen in figure 4.

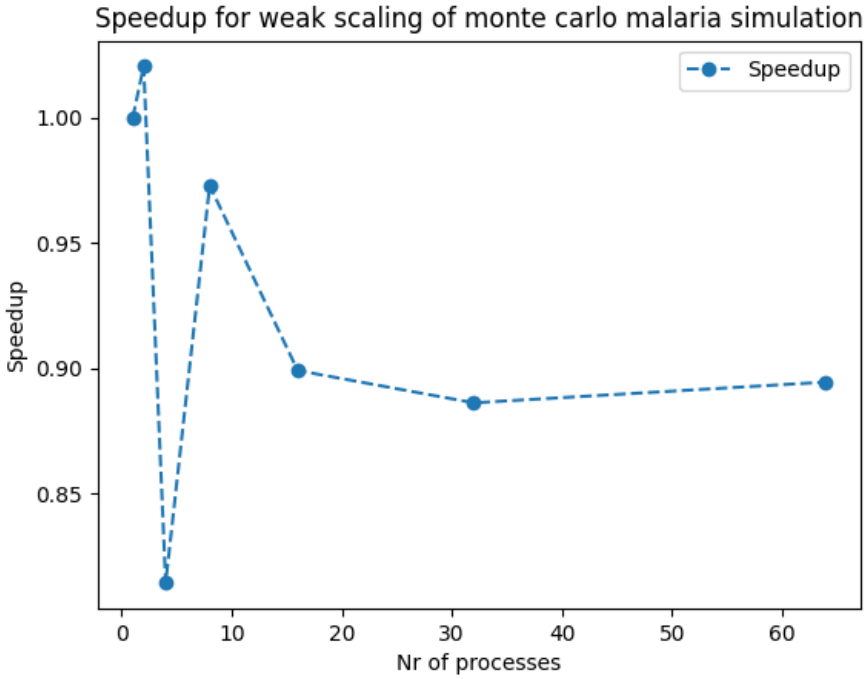

Figure 4: Weak scaling speedups for Monte Carlo malaria simulations. n=100000

5 Conclusions & Discussion

The code achieved a speedup very close to linear, see figure 2. I think these results are very good, however they can mostly be attributed to the large simulation portion of the code that is completely parallel. The only big causes of slowdowns are when all the processes need to synchronize, such as with the MPI_Reduceall call. I tried to keep this to a minimum, but it is unavoidable when for example the processes needs to find the bin limits.

The results of the weak scaling was harder to draw meaningful conclusions from, see figure 4. There appeared to be a large variance when increasing from 1 to 16 processes, especially noticeable was the large slowdown with 8 processes. I do not have any explanation of what caused this. With a larger number of processes the results seemed to stabilize at around 90% of time with a constant amount of work per process.

One thing I tried that ended up performing very badly, was a MPI_Accumulate call inside the simulation for the timesplits instead of a local array. For a single process this performed around equally compared to a local array, which is expected if the shared memory resides on that process. However when the code was run on multiple processes it performed significantly slower. I believe this is due to two factors, firstly the additional overhead of sending a small message several times, and the non-uniform memory access problem. Accessing memory that resides on another process is significantly slower, especially for large distributed memory systems such as most supercomputers.

What I instead ended up using was a local array and a single MPI_Put to gather the results

for process 0 to access.

The load balancing between processes should be approximately even. Since the biggest part of the time consists of performing the simulations I do not believe the program has any load balancing issues. The exact time to reach a certain time point in a simulation is stochastic, however since the simulations are repeated many times these time differences should, on average, cancel each other out with large n.

Another choice I made was to let each process create their own histograms, before adding up the final results. This can be compared to for example sending all the simulation results to process 0 and only calculating the intervals of the bins once. By letting each process calculate their own bin intervals and then find their own bin counts, we ensure better load balancing and reduce the amount of data that needs to be sent between processes.

One big impact on the simulation time was the timesplits calculations. Since this is inside the innermost loop, this is repeated many times and even small performance gains will have an effect on the total runtime. I chose to use a list to save the checkpoints, to reduce the amount of if statements. Another possible improvement would be to "unroll" the while loop, and split it into several parts and record the wall time inbetween, from time 0 to 25, 25 to 50 etc, to avoid having to check the value of the time too many times.

The one sided communications with MPI_Put and MPI_Get uses remote direct memory access (RDMA), which is supported by infiniband and allows the processes to access and update the memory of another process without interrupting it. This means that we avoid unnecessary synchronization delays compared to using a normal send or receive. For the code this means that process 0 is not slowed down by having to receive the timesplits from all processes which could lead to uneven load balancing and therefore slowdowns.

5.1 Future improvements

One possible improvement could be to improve load balancing by having dynamic load balancing, and let each process "grab" iterations from a shared variable or queue. This approach would work better if the code is run on unequal processes, where not all processes have the same computing capacity. This would also deal with random fluctuations in load balancing due to the stochastic nature of the simulation time, however the law of large numbers will cause these fluctuations to have a smaller impact on the total time for large n and I noticed approximately even load balancing during my experiments.