Parallel Monte Carlo Computations on Gillespie's Algorithm

UPPSALA
UNIVERSITET

Gustav Malmsten

Parallel and Distributed Programming 1TD070
M.Sc. Engineering Physics, Uppsala Universitet
May 25, 2023

Contents

1 Introduction

Epidemic modeling has always been of interest to researchers, providing a powerful tool to predict outcomes of outbreaks and thus unveil effective intervention strategies. However, it was the onset of the COVID-19 pandemic that captured the public eye, captivating the attention and curiosity of people worldwide.

One popular approach for simulating ecological systems, such as the spreading of epidemics, is Gillespie's direct method or SSA (Stochastic Simulation Algorithm)(1). Since this method is stochastic, it is beneficial to conduct many independent simulations and draw conclusions about the distribution of the results, instead of single results prone to much variance. This approach is in this paper referred to as the *Monte Carlo algorithm*(2). Since all of these computations are independent, the algorithm should be adapted to a parallel algorithm to utilise the full potential of the multi core computers of today.

Algorithm 1 Gillespie's direct method (SSA)

Set a final simulation time T, current time $t = 0$, initial state $x = x_0$
while $t < T$ **do**
    Compute $w = \text{prop}(x)$
    Compute $a_0 = \sum_{i=1}^{R} w(i)$
    Generate two uniform random numbers u_1, u_2 between 0 and 1
    Set $\tau = -\ln(u_1)/a_0$
    Find r such that $\sum_{k=1}^{r-1} w(k) < a_0 u_2 \leq \sum_{k=1}^{r} w(k)$
    Update the state vector $x = x + P(r, :)$
    Update time $t = t + \tau$
end while

The function $prop(x)$ is given in the c source file *prop.c*. T is set to 100, $x_0 = [900, 900, 30, 330, 50, 270, 2]$, and P is the following matrix:

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

Algorithm 2 The MC algorithm

1: Choose the number of MC experiments N
2: **for** $i = 1$ to 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.

2 Problem description

The aim of this paper is to implement the Monte Carlo [MC] method on Gillespie's algorithm for simulation of a malaria epidemic in a parallel and distributed memory environment using MPI in c. The implementiation is confined to work for $N = n \cdot p$ MC experiments where p is the number of processors and $n \in \mathbb{N}$. After each MC experiment, the first element of the state vector, corresponding to the number of susceptible people, will be stored in the result vector. The distribution of these results will be presented in a histogram which will be prepared in parallel. Subsequently, this implementation will be optimized in terms of serial as well as parallel performance. The performance will be evaluated in terms of both strong and weak scaling on the UPPMAX cluster snowy. In addition, each process will store the mean time spent in each sub time interval, $(0\% - 25\%]T$, $(25\% - 50\%]T$, $(50\% - 75\%]T$ and $(75\% - 100\%]T$. These sub timings will be gathered by the root processes using MPI one sided communication, and thereafter plotted in a table. Example outputs for three runs on 32 processes and $N = 2.5''$, $5''$ and $10''$ are provided in appendix A.1.

3 Solution approach

3.1 Serial solution

The serial implementation of the Gillespie algorithm is simply the pseudo code (1) translated to c code, with some slight optimisations. The state vector x is stored on the stack for optimised reading and writing performance. The random numbers u_1 and u_2 are used directly in the expression instead of pre storing them as u_1 and u_2, which increases performance by approximately 2%. Apart from this, the serial implementation of algorithm (1) follows the pseudo code.

3.2 Parallel solution

The parallel implementation uses a random seed corresponding to the rank of the process, to make sure that all Monte Carlo experiments are done independently. When the processes have seeded their random number generator, they allocate a results vector on the heap. This will slightly worsen the performance as compared to allocation of the stack for small experiment sizes N, but since this vector is only referenced once per MC experiment, which take significantly longer and random time, this will not affect the performance in total.

Inside the MC experiment loop, each process continuously checks whether to store a timing for a sub-interval, and which. The timings for each sub-interval are all accumulated and later divided by $local_N = n$ to obtain the average time spent per process in each of the four sub-intervals. After the first timing is updated, after passing $0.25T$, the process is told to start checking for when to update the next timing, $0.5T$, by changing the value of a variable *timing* from 0 to 1, and so on for the other timings. The ordering of the conditions checking the above is optimised by making the compiler check the value of *timing* before checking t, thus minimising the number of comparisons done per time step

After the all the MC experiments are done, the sub timings are scaled to the mean timings per sub-interval. This could have been done inside the MC experiment loop instead, but that would take $4 \cdot local_N$ as many floating-point operations and is thus much worse performance wise.

After the sub timings are calculated, each processor puts its local results in the root processor's memory using MPI_Put. This is a non-blocking call and the program thus need to be synchronised before these values are used by the root process. This synchronization is achieved by the use of a blocking MPI call, which acts like a barrier to ensure that all the processes have put their data in the root process' memory before it tries to access it. The blocking call used in this implementation is the call that is used to wait for the global minimum and maximum ranges to be synchronized, MPI_Waitall, which is further described in the paragraphs below.

Another synchronization point, or *fence*, is set up before the beginning of the MC experiments, just after the Root Memory Access [RMA] window initialisation, to ensure that the root process has properly initialised the window before the other processes try to put data in its memory. This barrier is of the type MPI_Win_fence.

The sub timings could also have been gathered by the root process using the one sided communication call MPI_Get. That would however make only one of the processes, the root process, do all the work gathering results and thus decrease the parallelism.

When a process has put the sub timing results in the memory of the root process, it continues the execution by calculating the minimum and maximum value of the results, before synchronising with the other processes to obtain the global minimum and maximum values. This synchronisation is done in two steps. First, the process uses two non-blocking calls to MPI_Allreduce, synchronising both the global minimum and maximum values. After this, each process sorts its local array of results while waiting for the other processes to finish the reduction call. The waiting for completion is done using the MPI_Waitall call mentioned above. This wait implicitly waits for both the put and reduction to complete, which may in the best of worlds decrease the load balance from the different completion times of the MC experiments. But, this largely depends on the time spent sorting the local results list, which all are equally distributed and of equal size among the processes, and are thus, on average, the same.

Once all processes are aware of the result range, they can in parallel sort their local results into the appropriate bin. This is done by iterating through the sorted list and incrementing the variable *bin* every time the result is larger than the limit of the current bin. The values of the bins in the array *bins* are concurrently being incremented.

The local updating of the array *bins* could have been done in another way, such as a bucket sort-like insertion in buckets corresponding to the bins. This would completely remove the need for sorting the list, but would instead make the updating of the bins computationally heavier. This is because that for all elements, all bins lower than the correct bin will still need to be checked, using two logical expressions. On average, each number will be tested on 5 superfluous ranges, thus making the sorted approach better since the built-in sorting function qsort is so fast for sufficiently large arrays in combination with the sorting of the list being done while waiting for the non-blocking calls to finish.

After all processes have sorted their results in the correct bins, all local results are summed in the root process using MPI_Reduce, the optimised way of gathering the results. Thereafter, the root process prints the results to the output file and prints the largest execution time to the terminal, that is, the largest recorded execution time among the processes.

4 Experiments

All timings are done on the UPPMAX computing cluster snowy. Snowy consists of 228 compute nodes [1], each consisting of two 8-core Xeon E5-2660 processors. We can thus expect the parallelism to decrease when passing the limits of cores, both per processor, 8, and per node, 16, due to increased communication time. The timing will be done on the interesting part of the program, that is, from the start of the first MC experiment, to just after the local results have been summed in the root process.

For the weak scaling experiments, the code was run conducting 125000000 MC experiments per process, while for the strong scaling experiments, the total number of experiments was fixed to $N = 1000000$. Both the weak and strong scaling experiments will be conducted using three different versions of the code, one disregarding the timings of the sub-intervals, and the other two using MPI_Put and MPI_Get, respectively.

4.1 Strong scaling

Table 1: Strong scaling without sub timings

Number of MC experiments	Number of Processes	Execution Time [s]	Speedup
1000000	1	1280.643192	1.000000
1000000	2	647.833336	1.977845
1000000	4	322.400997	3.972218
1000000	8	166.709949	7.6818427
1000000	16	90.690750	14.116504
1000000	32	45.504842	28.143009

Table 2: Strong scaling when using MPI_Get

Number of MC experiments	Number of Processes	Execution Time [s]	Speedup
1000000	1	1303.646810	1.000000
1000000	2	657.268489	1.983731
1000000	4	325.151893	4.009256
1000000	8	176.456672	7.389585
1000000	16	90.313188	14.4347336
1000000	32	45.539060	28.6270031

Table 3: Strong scaling when using MPI_Put

Number of MC experiments	Number of Processes	Execution Time [s]	Speedup
1000000	1	1306.772393	1.000000
1000000	2	651.575493	2.005188
1000000	4	327.725878	3.986466
1000000	8	173.943214	7.504101
1000000	16	90.721104	14.404282
1000000	32	46.198120	28.286268

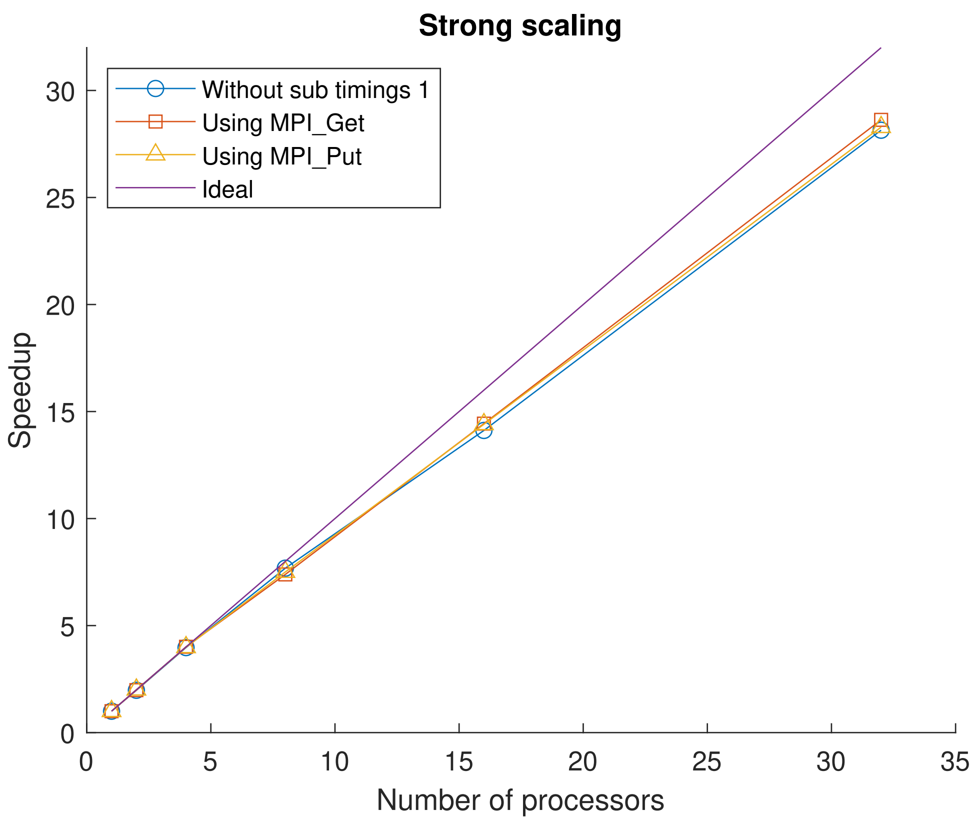

Figure 1: Strong scaling for different solution approaches

As can be seen in figure 1, the strong scaling is almost perfectly linear, with the significant parallelism decrease starting at 8 processes and therafter decreasing further. The figure also shows that the choice of one sided communication call does not significantly affect the parallelism, and neither does the omitting of the sub timings.

4.2 Weak scaling

Table 4: Weak scaling without sub timings

Number of MC experiments	Processes	Execution Time [s]	Weak Scaling
125000	1	161.799318s	1
250000	2	162.580660s	1.0048
500000	4	162.906400s	1.0068
1000000	8	168.441780s	1.0413
2000000	16	180.379975s	1.1152
4000000	32	181.539433s	1.1229

Table 5: Weak scaling when using MPI_Get

Number of MC experiments	Processes	Execution Time [s]	Weak Scaling
125000	1	161.703252s	1
250000	2	162.849568s	1.0069
500000	4	164.588902s	1.0185
1000000	8	171.075371s	1.0579
2000000	16	181.707286s	1.1238
4000000	32	182.152280s	1.1267

Table 6: Weak scaling when using MPI_Put

Number of MC experiments	Processes	Execution Time [s]	Weak Scaling
125000	1	162.068777	1
250000	2	162.115208	1.0003
500000	4	164.177195	1.0121
1000000	8	169.891534	1.0489
2000000	16	181.474822	1.1274
4000000	32	182.591559	1.1294

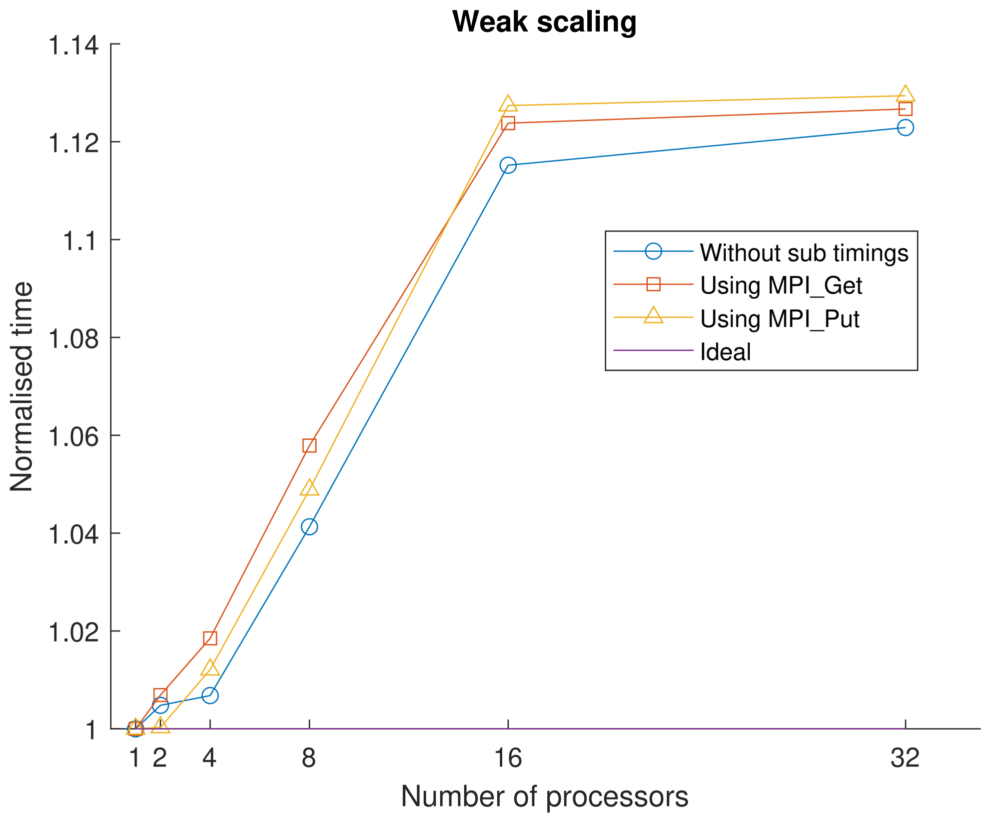

Figure 2: Weak scaling for different solution approaches

As can be seen in figure 2, the weak scaling is almost perfect for 1 – 4 processes, and thereafter grows linearly until a plateau is reached at 16 processors. Just as in the strong scaling experiments, all three approaches exhibit similar results, although the approach disregarding the sub-timings performs the best.

4.3 Profiling

The program was profiled in Allinea MAP using 16 processes on $N = 16000$ and $N = 500000$ MC experiments to demonstrate the uneven load balancing as well as the communication overhead.

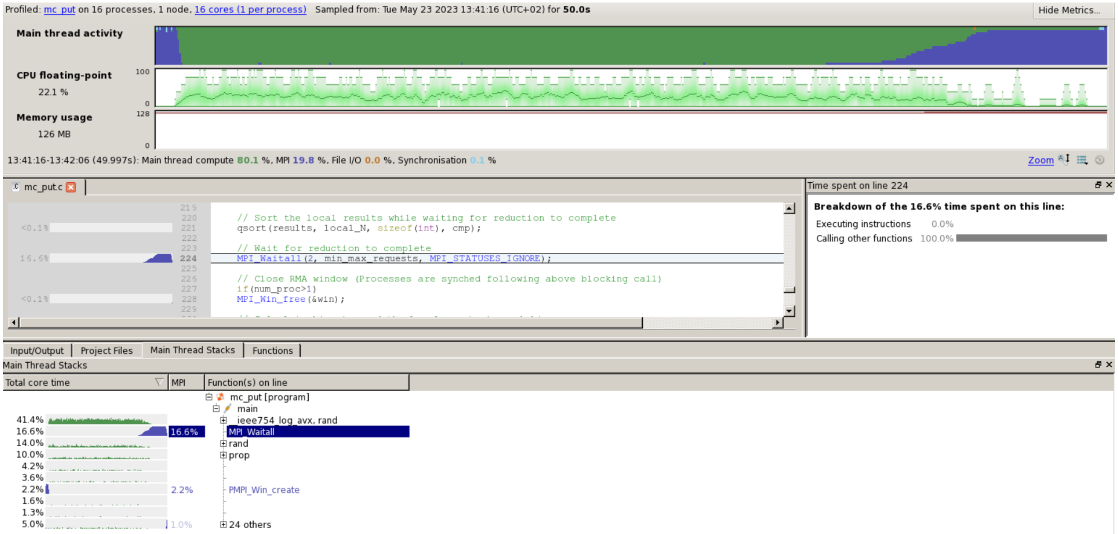

Figure 3: Profiling of program for $N = 16000$ MC experiments

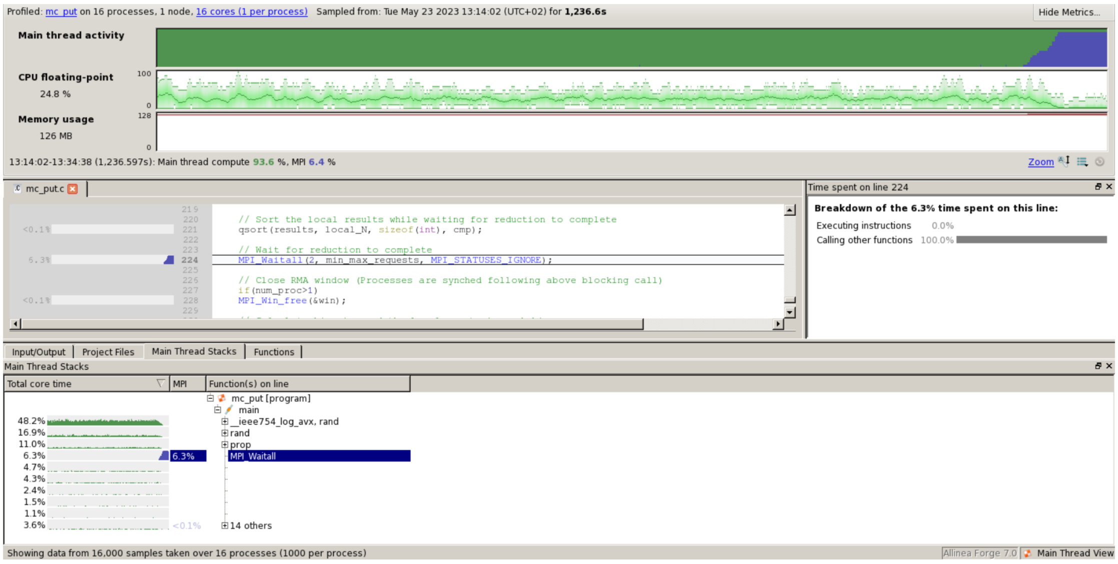

Figure 4: Profiling of program for $N = 500000$ MC experiments

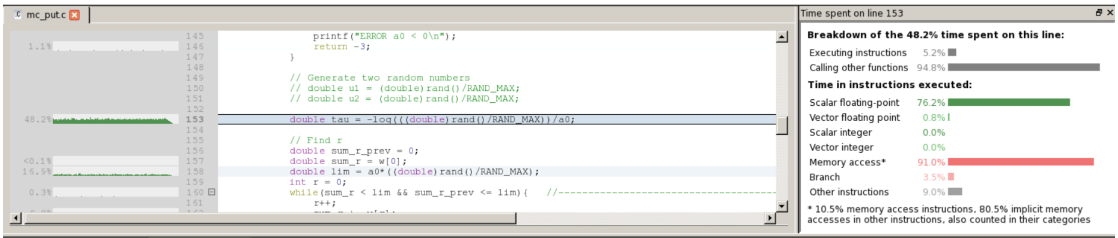

Figure 5: Breakdown of the time spent calculating the succeeding time step

The figures 3 and 4 show the fraction of total execution time spent on different lines of the code. These figures show that mostly all of the communication overhead stems from the synchronisation in MPI_Waitall, almost 17% and roughly 6%, respectively for the small and large experiment runs. The other communication calls account for less than a tenth of a percent each and are thus negligible. It is also apparent in the figures that the sorting of the list containing the local results is also negligible, totaling less than a tenth of a percent, as well, independently of the problem size. The absolute biggest fraction of the time is spent in the function __ieee754_log_avx, rand. This line is the one calculating the next time step, τ. Further analysis of the time spent in this function, as seen in figure 5, shows that almost all of the time spent in this function, almost 95%, is spent calling other functions.

5 Conclusion

The possible reasons for not achieving perfect linear parallel speedup are a few.

The first, obvious reason, is the fact that the total amount of work increases linearly with the number of processes for the two approaches that store sub-timings. This is an example of bad redundancy which will worsen the performance when compared to the approach that does not store sub-timings. Since the significance of this additional work decreases with the number of MC experiments, in combination with the non-blocking communications, this is not the major cause for sub-optimal parallel speedup.

The fact that the two different approaches exhibit negligible differences in terms of speedup indicates that the choice of one-sided communication call is insignificant for this problem. This is also shown in figures 3 and 4, which show that the significant inter-process communication overhead originates from the synchronization of the threads. The reason for this synchronization taking so long stems from the load unbalances in the processes, due to the asynchronous completion times of the Monte Carlo experiments. The non-blocking calls optimise the execution of the code after the experiments, but optimising the inter-process communications further without altering the functionality of the code would be difficult.

One possible solution to this problem would be to let the processes that finishes first continue conducting MC experiments until the total amount of experiments reach N, with some processes conducting more than, and some less than, $n = \frac{N}{p}$ experiments. This would however introduce the need for more inter-process communication, and thus possibly worsen the performance if the additional parallel overhead is greater than the gain of implementing this strategy.

Regarding the reordering of the logical conditions, the optimisation yielded no significant performance increase. There are likely a few reasons for this. Firstly, the execution time of this part of the loop is rather negligible compared to the rest of the loop. Additionally, the compiler is able to reorder conditions in the most efficient way, rendering manual reordering of conditions redundant.

One final potential optimization would involve reordering the code blocks responsible for synchronizing the global range of the histogram and storing the sub-timings in the memory of the root process. Since the workload after the Monte Carlo experiments are equal for all processes and thus should take the same time to execute, this would not improve the performance much.

All in all, this parallel implementation of Gillespie's Algorithm is performing well in terms of both weak and strong scalability. The major downfall is the bad load balancing originating from the completion times being random, which is nothing we can affect without altering the functionality of the code. Optimising the inter-process communication further would not enhance the performance greatly before the load balancing issue has been dealt with, for example using dynamic workload redistribution or task stealing. Furthermore, serially optimising the line updating the next time step would provide a huge performance improvement, following the results presented in section 4.3.

References

[1] *SNOWY User Guide*. `https://www.uppmax.uu.se/support/user-guides/snowy-user-guide/`. Accessed on May 23, 2023.

A Appendix

A.1 Output

All example runs were run on 32 processes. All subsections will include the output generated by running the program; the bounds of the intervals in the histogram, the sub timings for different processes and also a plot of the histogram itself.

A.1.1 2 500 000 MC Experiments

Table 7: Bins

Bin	Interval
1	$[426, 457)$
2	$(457, 488]$
3	$(488, 519]$
4	$(519, 550]$
5	$(550, 581]$
6	$(581, 612]$
7	$(612, 643]$
8	$(643, 674]$
9	$(674, 705]$
10	$(705, 736]$
11	$(736, 767]$
12	$(767, 798]$
13	$(798, 829]$
14	$(829, 860]$
15	$(860, 891]$
16	$(891, 922]$
17	$(922, 953]$
18	$(953, 984]$
19	$(984, 1015]$
20	$(1015, 1053]$

Table 8: Average time spent in sub intervals

Process	$t \in [0, 25]$	$t \in (25, 50]$	$t \in (50, 75]$	$t \in (75, 100]$
0	0.000723	0.000268	0.000217	0.000219
1	0.000723	0.000268	0.000217	0.000219
2	0.000724	0.000268	0.000218	0.000220
3	0.000729	0.000269	0.000218	0.000220
4	0.000763	0.000276	0.000224	0.000226
5	0.000723	0.000268	0.000217	0.000219
6	0.000733	0.000271	0.000220	0.000222
7	0.000725	0.000269	0.000217	0.000220
8	0.000739	0.000268	0.000217	0.000220
9	0.000724	0.000269	0.000218	0.000220
10	0.000770	0.000279	0.000227	0.000229
11	0.000773	0.000280	0.000228	0.000231
12	0.000723	0.000268	0.000217	0.000219
13	0.000724	0.000269	0.000217	0.000220
14	0.000726	0.000269	0.000218	0.000220
15	0.000723	0.000268	0.000217	0.000220
16	0.000723	0.000268	0.000217	0.000220
17	0.000723	0.000268	0.000218	0.000220
18	0.000731	0.000271	0.000220	0.000222
19	0.000723	0.000268	0.000218	0.000220
20	0.000723	0.000268	0.000218	0.000220
21	0.000722	0.000268	0.000217	0.000219
22	0.000725	0.000269	0.000218	0.000220
23	0.000728	0.000270	0.000219	0.000222
24	0.000739	0.000275	0.000223	0.000226
25	0.000723	0.000268	0.000217	0.000220
26	0.000729	0.000268	0.000218	0.000220
27	0.000723	0.000268	0.000218	0.000220
28	0.000723	0.000268	0.000217	0.000220
29	0.000723	0.000268	0.000217	0.000220
30	0.000722	0.000268	0.000217	0.000219
31	0.000723	0.000268	0.000217	0.000220

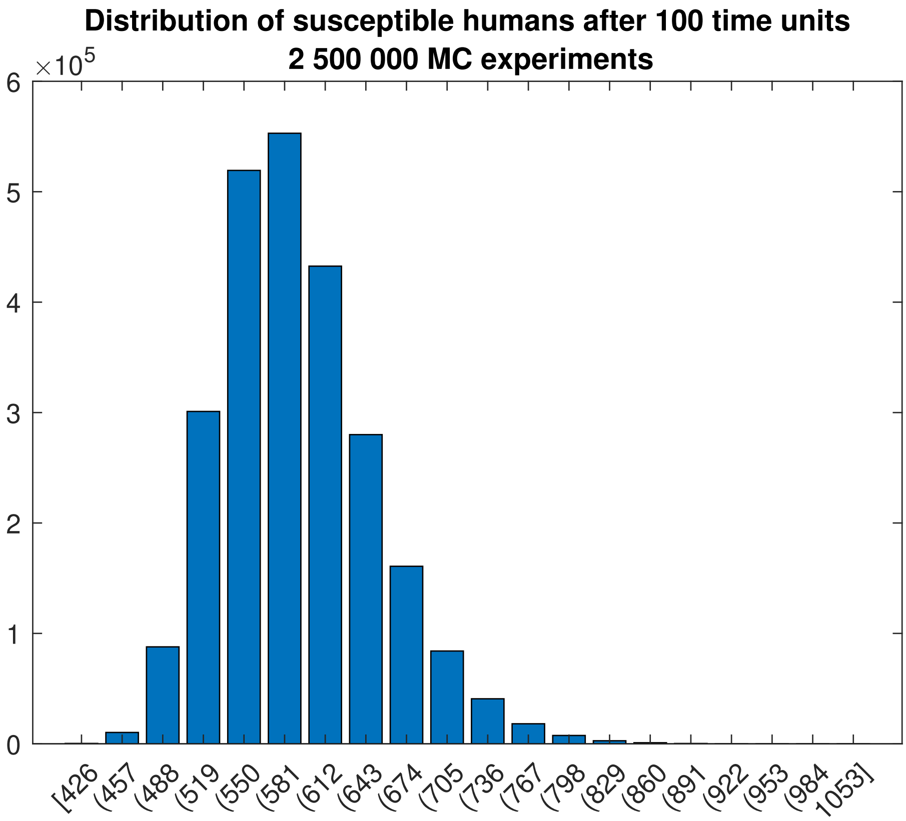

Figure 6: Histogram of susceptible people after 100 time units

A.1.2 5 000 000 MC Experiments

Table 9: Bins

Bin	Interval
1	$[423, 455)$
2	$(455, 487]$
3	$(487, 519]$
4	$(519, 551]$
5	$(551, 583]$
6	$(583, 615]$
7	$(615, 647]$
8	$(647, 679]$
9	$(679, 711]$
10	$(711, 743]$
11	$(743, 775]$
12	$(775, 807]$
13	$(807, 839]$
14	$(839, 871]$
15	$(871, 903]$
16	$(903, 935]$
17	$(935, 967]$
18	$(967, 999]$
19	$(999, 1031]$
20	$(1031, 1075]$

Table 10: Average time spent in sub intervals

Process	$t \in [0, 25]$	$t \in (25, 50]$	$t \in (50, 75]$	$t \in (75, 100]$
0	0.000725	0.000270	0.000218	0.000221
1	0.000723	0.000268	0.000217	0.000219
2	0.000723	0.000268	0.000217	0.000220
3	0.000736	0.000269	0.000218	0.000221
4	0.000722	0.000268	0.000217	0.000219
5	0.000723	0.000268	0.000217	0.000219
6	0.000723	0.000268	0.000217	0.000220
7	0.000732	0.000271	0.000220	0.000222
8	0.000863	0.000324	0.000264	0.000267
9	0.000731	0.000271	0.000220	0.000222
10	0.000729	0.000272	0.000220	0.000223
11	0.000725	0.000269	0.000218	0.000220
12	0.000740	0.000275	0.000223	0.000225
13	0.000723	0.000268	0.000217	0.000220
14	0.000724	0.000268	0.000217	0.000219
15	0.000724	0.000268	0.000218	0.000220
16	0.000724	0.000268	0.000218	0.000220
17	0.000738	0.000272	0.000220	0.000223
18	0.000723	0.000268	0.000217	0.000219
19	0.000762	0.000275	0.000224	0.000226
20	0.000724	0.000268	0.000217	0.000219
21	0.000723	0.000268	0.000217	0.000219
22	0.000726	0.000268	0.000218	0.000220
23	0.000731	0.000271	0.000220	0.000222
24	0.000727	0.000270	0.000219	0.000221
25	0.000723	0.000268	0.000217	0.000220
26	0.000724	0.000268	0.000218	0.000220
27	0.000723	0.000268	0.000217	0.000219
28	0.000723	0.000268	0.000217	0.000220
29	0.000723	0.000268	0.000217	0.000219
30	0.000766	0.000276	0.000225	0.000227
31	0.000724	0.000268	0.000217	0.000220

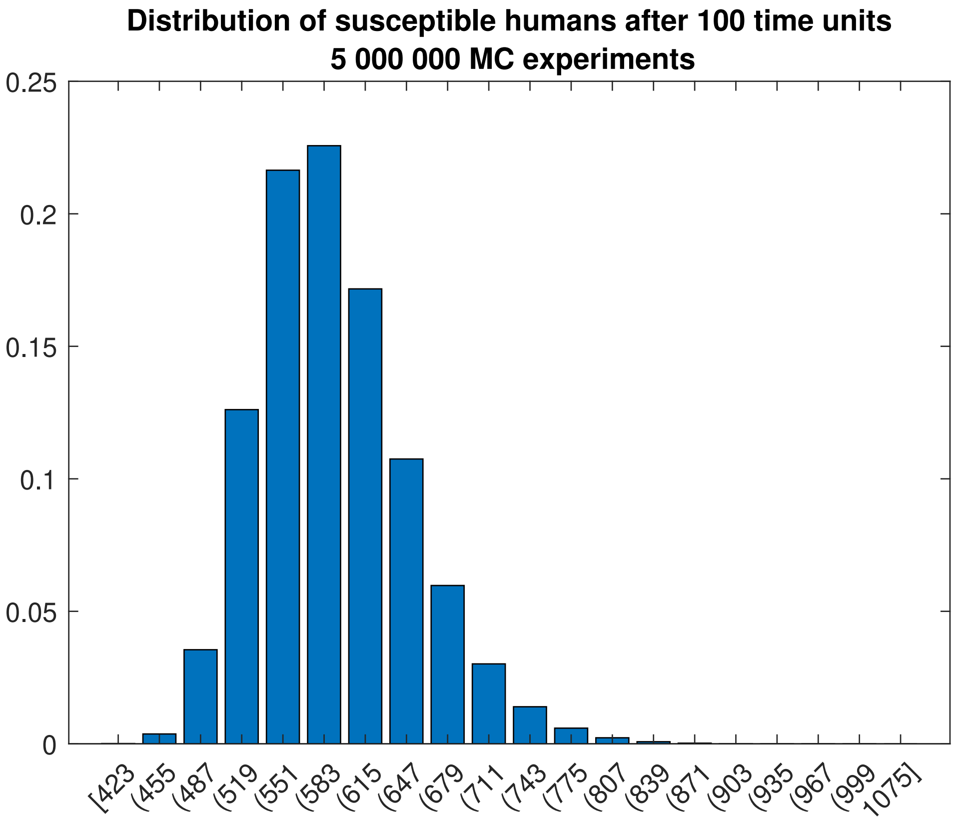

Figure 7: Histogram of susceptible people after 100 time units

A.1.3 10 000 000 MC Experiments

Table 11: Bins

Bin	Interval
1	$[415, 448)$
2	$(448, 481]$
3	$(481, 514]$
4	$(514, 547]$
5	$(547, 580]$
6	$(580, 613]$
7	$(613, 646]$
8	$(646, 679]$
9	$(679, 712]$
10	$(712, 745]$
11	$(745, 778]$
12	$(778, 811]$
13	$(811, 844]$
14	$(844, 877]$
15	$(877, 910]$
16	$(910, 943]$
17	$(943, 976]$
18	$(976, 1009]$
19	$(1009, 1042]$
20	$(1042, 1093]$

Table 12: Average time spent in sub intervals

Process	$t \in [0, 25]$	$t \in (25, 50]$	$t \in (50, 75]$	$t \in (75, 100]$
0	0.000723	0.000268	0.000217	0.000220
1	0.000739	0.000268	0.000218	0.000220
2	0.000723	0.000268	0.000217	0.000219
3	0.000723	0.000268	0.000217	0.000219
4	0.000723	0.000268	0.000217	0.000219
5	0.000732	0.000272	0.000220	0.000222
6	0.000724	0.000268	0.000218	0.000220
7	0.000722	0.000268	0.000217	0.000219
8	0.000726	0.000269	0.000218	0.000220
9	0.000738	0.000268	0.000217	0.000219
10	0.000770	0.000279	0.000227	0.000230
11	0.000723	0.000268	0.000217	0.000219
12	0.000722	0.000268	0.000217	0.000219
13	0.000724	0.000269	0.000218	0.000220
14	0.000723	0.000268	0.000218	0.000220
15	0.000730	0.000270	0.000219	0.000221
16	0.000731	0.000271	0.000220	0.000222
17	0.000722	0.000268	0.000217	0.000219
18	0.000723	0.000268	0.000217	0.000219
19	0.000724	0.000268	0.000217	0.000220
20	0.000726	0.000270	0.000219	0.000222
21	0.000762	0.000276	0.000224	0.000226
22	0.000728	0.000270	0.000219	0.000221
23	0.000730	0.000270	0.000219	0.000221
24	0.000739	0.000273	0.000221	0.000224
25	0.000731	0.000272	0.000220	0.000222
26	0.000727	0.000270	0.000218	0.000220
27	0.000723	0.000268	0.000217	0.000220
28	0.000723	0.000268	0.000217	0.000220
29	0.000738	0.000273	0.000222	0.000224
30	0.000723	0.000268	0.000217	0.000220
31	0.000722	0.000268	0.000217	0.000219

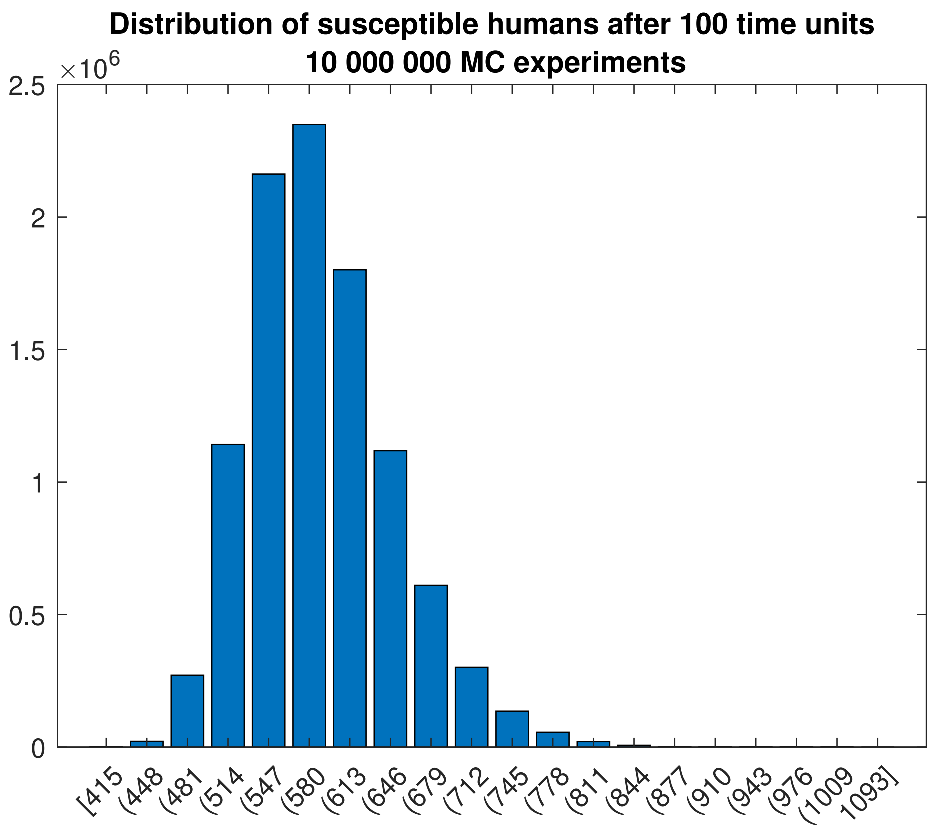

Figure 8: Histogram of susceptible people after 100 time units

A.2 Code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <mpi.h>
#include "prop.h"
#include <string.h>

#define ROWS 15
#define COLS 7
#define T 100
#define b 20

#define PRODUCE_OUTPUT

void print_d_vec(double *vector, int lim){
    /*Print a vector consisting of lim doubles*/
```

```
    printf("[");
    for(int i = 0; i < lim; i++){
        printf("%lf, ", vector[i]);
    }
    printf("]\n");
}

void print_i_vec(int *vector, int lim, FILE * restrict fp){
    /*Print a vector consisting of lim integers*/
    fprintf(fp, "[");
    for(int i = 0; i < lim - 1; i++){
        fprintf(fp, "%d, ", vector[i]);
    }
    fprintf(fp, "%d]\n", vector[lim - 1]);
}

int cmp (const void *num1, const void *num2) {
    /*Comparison function for qsort*/
   return ( *(int*)num1 > *(int*)num2 );
}

void print_i_vec_term(int *vector,int lim)
{
    /*Print a vector consisting of lim integers*/
    printf("[");
    for(int i = 0; i < lim - 1; i++){
        printf("%d, ", vector[i]);
    }
    printf("%d]\n", vector[lim - 1]);
}

int main(int argc, char *argv[]){

    if(argc != 3){
        printf("Usage %s N output_file\n", argv[0]);
        return -1;
    }

    // Arguments
    const int N = atoi(argv[1]);
    const char *output_file = argv[2];

    int rank, num_proc;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
    const int local_N = N/num_proc;

    // Assumption: N is divisible by num_proc
```

```
    if(N%num_proc){
        if(rank == 0){
            printf("ERROR N%%p != 0\n");
            return -2;
        }
    }


    // Transition matrix
    const int P[ROWS*COLS] = {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};

    // Seed
    time_t seed = time(NULL);
    // int seed = 1;
    MPI_Bcast(&seed, 1, MPI_INT, 0, MPI_COMM_WORLD);
    srand(seed + rank);


    int results[local_N];
    double sub_times[4] = {0};
    double all_sub_times[4*num_proc];
    MPI_Win win;
    if(num_proc > 1)
    {
        MPI_Win_create(all_sub_times, 4*num_proc*sizeof(double), sizeof(double), MPI_INFO_NULL, MPI_COMM_WORLD, &win);
        MPI_Win_fence(0, win);
    }
    // Start timer
    double start_time = MPI_Wtime();
    for(int epoch = 0; epoch < local_N; epoch++){
        // Initialize new simulation
        double t = 0;
        int x[COLS] = {900, 900, 30, 330, 50, 270, 20};

```

```c
    char timing = 0;  // Current sub time to store, 0 - 25, 1 - 50, 2 - 75, 3 - 100
    double sub_start_time = MPI_Wtime();
    while(t<T){
        // Store sub times
        if(timing == 0 && t > T/4){
            sub_times[0] += (MPI_Wtime() - sub_start_time);
            sub_start_time = MPI_Wtime();
            timing = 1;
        }
        if(timing == 1 && t > T/2){
            sub_times[1] += (MPI_Wtime() - sub_start_time);
            sub_start_time = MPI_Wtime();
            timing = 2;
        }
        if(timing == 2 && t > 3*T/4){
            sub_times[2] += (MPI_Wtime() - sub_start_time);
            sub_start_time = MPI_Wtime();
            timing = 3;
        }

        // Compute w
        double w[ROWS];
        prop(x, w);

        // Compute a0
        double a0 = 0;
        for(int i = 0; i < ROWS; i++){
            a0+=w[i];
        }
        if(a0<0){
            printf("ERROR a0 < 0\n");
            return -3;
        }

        // Generate two random numbers
        // double u1 = (double)rand()/RAND_MAX;
        // double u2 = (double)rand()/RAND_MAX;

        double tau = -log(((double)rand()/RAND_MAX))/a0;

        // Find r
        double sum_r_prev = 0;
        double sum_r = w[0];
        double lim = a0*((double)rand()/RAND_MAX);
        int r = 0;
        while(sum_r < lim && sum_r_prev <= lim){   // ------------------------------------------------------------------------//
            r++;
```

```
                sum_r += w[r];
                sum_r_prev = sum_r;
                if(r>ROWS){
                    printf("Error: r exceeds the bounds of the w array\n");
                    return -1;
                }
            }


            // Update x
            for(int i = 0; i < COLS; i++){
                x[i] += P[r*COLS + i];
            }

            // Step time
            t+=tau;
        }
        // Store sub time and result
        sub_times[3] += (MPI_Wtime() - sub_start_time);
        results[epoch] = x[0];
    }

    // Rescale sub_times to mean sub_times
    for(int i = 0; i < 4; i++){
        sub_times[i] /= (double)local_N;
    }

    // Put the sub timings in the root process memory
    if(rank == 0)
    {
        memcpy(&all_sub_times[0], sub_times, 4*sizeof(double));
    }
    else
    {
        MPI_Put(sub_times, 4, MPI_DOUBLE, 0, rank * 4, 4, MPI_DOUBLE, win);
    }
    // Calculate local and global min and max
    int global_min, global_max, local_min, local_max;

    local_min = results[0];
    local_max = results[0];
    for(int i = 1; i < local_N; i++)
    {
        int tmp = results[i];
        if(tmp < local_min){
            local_min = tmp;
        }
```

```c
        if(tmp > local_max){
            local_max = tmp;
        }
    }

    MPI_Request min_max_requests[2];

    // Reduce and broadcast global min and max to all processes
    MPI_Iallreduce(&local_min, &global_min, 1, MPI_INT, MPI_MIN,
MPI_COMM_WORLD, &min_max_requests[0]);
    MPI_Iallreduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX,
MPI_COMM_WORLD, &min_max_requests[1]);

    // Sort the local results while waiting for reduction to complete
    qsort(results, local_N, sizeof(int), cmp);

    // Wait for reduction to complete
    MPI_Waitall(2, min_max_requests, MPI_STATUSES_IGNORE);

    // Close RMA window (Processes are synched following above
blocking call)
    if(num_proc>1)
    MPI_Win_free(&win);

    // Calculate bin size and the local counts in each bin
    int bin_size = (global_max - global_min)/b;
    int bins[b] = {0};

    int bin = 0;    // Current bin
    for(int i = 0; i < local_N; i++){
        if(results[i] > global_min + bin_size*(bin+1)){
            bin++;
        }
        if(results[i] > global_min + bin_size*b) // Let last bin
contain all elements larger than global_min + 20*bin_size
        {
            bins[b-1]++;
        }
        else
        {
            bins[bin]++;
        }
    }

    // Sum all local results in root process (0)
    int global_bins[b];
    MPI_Reduce(bins, global_bins, b, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

```

```c

    // Stop timer
    double local_time = MPI_Wtime() - start_time;
    double global_time;
    MPI_Reduce(&local_time, &global_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);


    // Produce output
    if(rank == 0)
    {
        #ifdef PRODUCE_OUTPUT
        FILE *fp;
        fp = fopen(output_file, "w");

        fprintf(fp, "Sub-timings:\n");
        fprintf(fp, "Process\t[0, 25]\t\t(25, 50]\t(50, 75]\t(75, 100]\n");
        for(int p = 0; p<num_proc; p++)
        {
            fprintf(fp, "\t%d\t%lf\t%lf\t%lf\t%lf\n", p, all_sub_times[4*p], all_sub_times[4*p+1], all_sub_times[4*p+2], all_sub_times[4*p+3]);
        }
        fprintf(fp, "\nRange of histogram: [%d, %d]\n", global_min, global_max);
        fprintf(fp, "Bins:\n");
        fprintf(fp, "Bin %d [%d %d]\n", 1, global_min, global_min + bin_size);
        for(int i = 1; i < b - 1; i++)
        {
            fprintf(fp, "Bin %d (%d %d]\n", i+1, global_min + bin_size*i, global_min + bin_size*(i + 1));
        }
        fprintf(fp, "Bin %d (%d %d]\n", 20, global_min + bin_size*19, global_max);
        fprintf(fp, "Counts:\n");
        print_i_vec(global_bins, b, fp);
        fclose(fp);
        #endif

        printf("%lf\n", global_time);
    }


    MPI_Finalize();
    return 0;
}
```