UPPSALA
UNIVERSITET

Monte Carlo Computations

Yuanjing Yang

June 27, 2025

1 Introduction

This project implements a MPI parallel version of SSA algorithm to simulate the spread of malaria epidemic, which is primarily through the bite of infected female Anopheles mosquitoes. With parallel computing, Monte Carlo simulations can be independent and executed concurrently , enabling efficient execution.

2 Algorithm

2.1 Sequential Algorithm

The sequential component is based on the SSA algorithm, which simulating the time evolution of a system with probabilistic reaction events. In this simulation:
(1) The state of the system is represented by a 7-dimensional integer vector, describing different compartments of human and mosquito populations.
(2) The state evolves over time, resulting 15 reactions.
(3) At each step, the algorithm calculates reaction propensities, determines the time to the next event, and selects which reaction occurs.
SSA Algorithm [1]:

Algorithm 1 Gillespie's direct method (SSA)

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

2.2 Parallel Algorithm

Here I used master-worker patterns [3] with dynamic load balancing [2], the procedures is below:
1. Master process(Rank 0) distributes simulation tasks to workers.
2. worker process receive task assignments from master and execute SSA simulations independently.
MPI Function used:
MPI_Send and MPI_Recv for point to point communication.
MPI_ANY_SOURCE and MPI_ANY_TAG [4]for flexible communication.

3 Experiments and Results

3.1 Distribution of X after time T

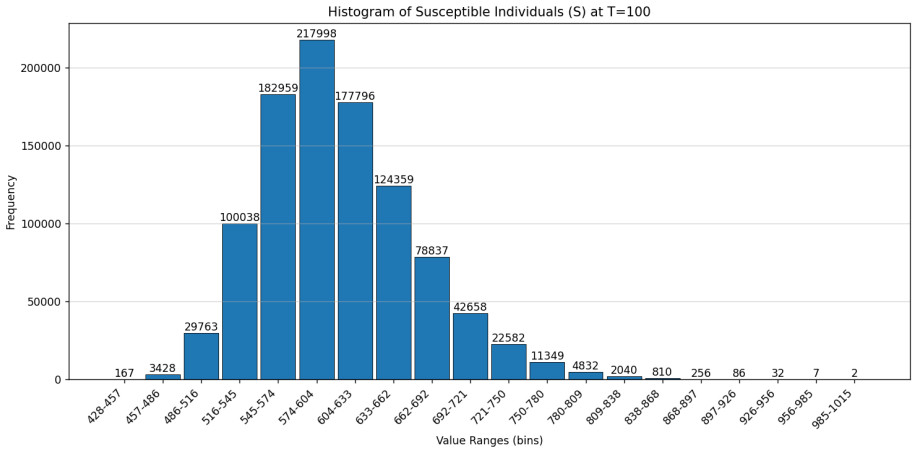

Figure 1: Distribution of X after time T

The figure above shows the histogram plot of the distribution of X at time step T given the same input x0 = 900, 900, 30, 330, 50, 270, 20, and it fits our expectation that is a normal distribution and it peaks at [573,604].

4 Performance

4.1 Strong scalability

Fixed-size scalability was tested with $N\_total = 10^6$ and varying p = 1, 2, 4, 8, 16.

Table 1: Strong Scalability Results

Processes	**Runtime (s)**	**Speedup (%)**	**Efficiency (%)**	**Ideal Speedup (%)**
Ideal Efficiency (%)				
1	1244.614	100%	100%	100%
2	1240.079	100%	50%	200%
4	446.136	279%	70%	400%
8	211.300	589%	74%	800%
16	99.826	1247%	78%	1600%

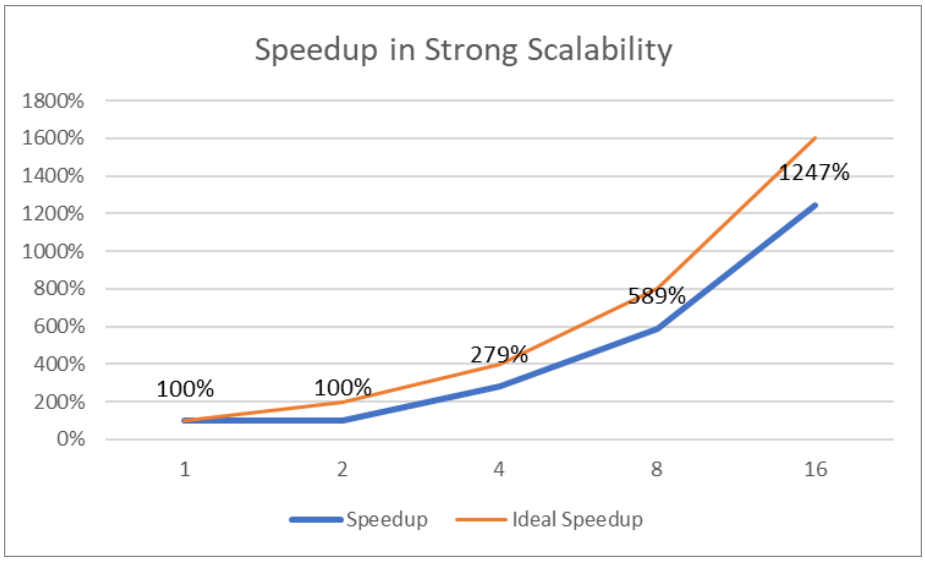

Figure 2: The Speedup in strong scalability

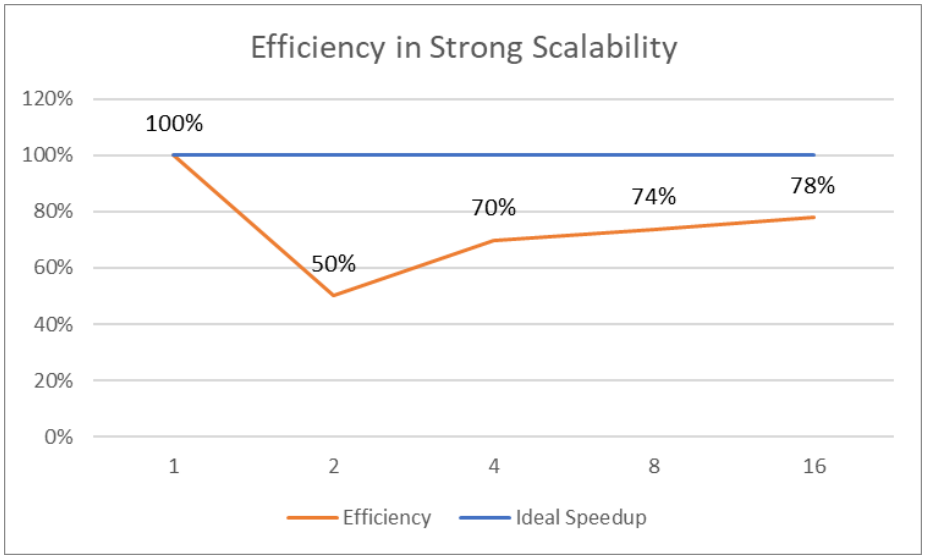

Figure 3: The Efficiency in strong scalability

From the result above, the runtime of 2 processes is almost equal to 1 process, but with the increase of processes (2, 4, 8, 16), we can find the improvement of speedup as we expected. For 16 processes, the runtime is reduced to 99.826 seconds, it acheives the speedup of 12.47.

4.2 weak scalability

Weak scalability was tested by increasing N_total proportionally with p, keeping the workload per process constant (n = 250,000).

Table 2: Weak Scalability Results

Processes	Runtime (s)	Speedup (%)	Ideal Speedup (%)
1	284.663	100%	100%
2	585.387	49%	100%
4	428.362	66%	100%
8	422.527	67%	100%
16	398.103	72%	100%

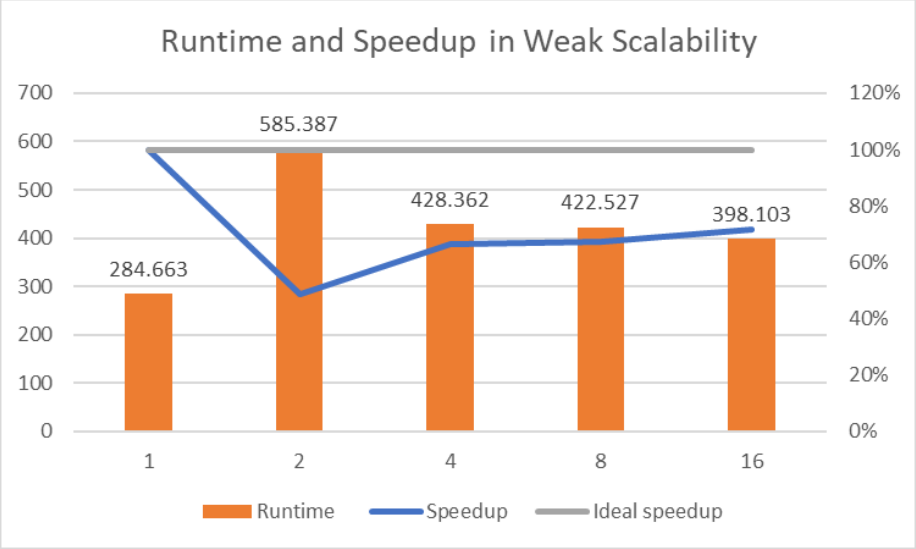

Figure 4: The Runtime and Speedup in strong scalability

The execution time increase significantly when moving from 1 process to 2 processes, in the work per process remaining constant, beyond 2 process, the runtime and speedup remains almost constant.

5 Discussion

In strong scalability, from 1 process to 2 processes, the runtime even not change, it indicates the overhead of communication and master-worker coordination for small number of processes. However, we can see an obvious improvement beyond 2 processes, this proof that the benefit of distributing tasks handle outweigh the

communication overhead. With the use of dynamical load balancing, the larger numbers of workers can keep them busy and reduce idle time.

In weak scalability, we met same problem, the significant increase of execution time when we moved from 1 process to 2 processes, it can be attributes to the master-worker framework, because when running on more processes, the master must begin sending tasks and receiving results, which adds the computational burden and leads to overweight the benefit of distribution. On contrast, when we run a single process, the overheads of message passing are completely disappear.But beyond 2 processes, the runtime is almost constant, and this suggests that for a larger number of processes, the dynamic load balancing becomes more efficient.

6 Conclusion

Both strong and weak scaling tests shows a significant overhead when introduced it to an MPI-parallelized master-worker architecture, and it is prominent for small number of processes. But for larger process counts(beyond 2 processes), the dynamic load balancing effectively utilizes the available cores. And there has a lot of way of improvement(e.g. a manager-worker model where the manager also participates in computation but when tried to add master(Rank 0) into computation as worker, it increases the execution time due to increase burden of rank 0).

7 Declaration of AI

In my project, I used AI to write some repeated and basic codes, debugging and make a improvement of my code. I write the original code frame and tell AI my the ideas, from the materials from AI's search, I found a way to improve performance and looked for the bibliographic on online library, it increases the efficiency a lot, I don't need to waste a lot time in searching method. AI also helped me debug my code when I met some errors on the UPPMAX computing cluster, and modifying my code based on my results, I always get some useful feedback from AI.

References

[1] Myles R. Allen and Leonard A. Smith. Monte carlo ssa: Detecting irregular oscillations in the presence of colored noise. *Journal of climate*, 9(12):3373–3404, 1996.

[2] Rosa Filgueira, Jesús Carretero, David E. Singh, Alejandro Calderón, and Alberto Núñez. Dynamic-compi: dynamic optimization techniques for mpi parallel applications. *The Journal of supercomputing*, 59(1):361–391, 2012.

[3] Mats Rynge, Scott Callaghan, Ewa Deelman, Gideon Juve, Gaurang Mehta, Karan Vahi, and Philip J. Maechling. Enabling large-scale scientific workflows on petascale resources using mpi master/worker. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond*, pages 1–8, New York, NY, USA, 2012. ACM.

[4] Anh Vo, Sarvani Vakkalanka, Jason Williams, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, Jan Westerholm, Jack Dongarra, and Matti Ropo. Sound and efficient dynamic verification of mpi programs with probe non-determinism. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science, pages 271–281. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

8 Appendix

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #include <mpi.h>
6  #include <time.h>
7  #include <string.h>
8  #define R_ssa 15
9  #define X_LEN 7
10 #define T_FINAL 100.0
11
12
13 int P[R_ssa][X_LEN] = {
14     { 1,  0,  0,  0,  0,  0,  0},
15     {-1,  0,  0,  0,  0,  0,  0},
16     {-1,  0,  1,  0,  0,  0,  0},
17     { 0,  1,  0,  0,  0,  0,  0},
18     { 0, -1,  0,  0,  0,  0,  0},
19     { 0, -1,  0,  1,  0,  0,  0},
20     { 0,  0, -1,  0,  0,  0,  0},
21     { 0,  0, -1,  0,  1,  0,  0},
22     { 0,  0,  0, -1,  0,  0,  0},
23     { 0,  0,  0, -1,  0,  1,  0},
24     { 0,  0,  0,  0, -1,  0,  0},
25     { 0,  0,  0,  0, -1,  0,  1},
26     { 0,  0,  0,  0,  0, -1,  0},
27     { 1,  0,  0,  0,  0,  0, -1},
28     { 0,  0,  0,  0,  0,  0, -1}
```

```
};

void prop(int *x, double *w) {
    const double LAMBDA_H = 20;
    const double LAMBDA_M = 0.5;
    const double B = 0.075;
    const double BETA_H = 0.3;
    const double BETA_M = 0.5;
    const double MU_H = 0.015;
    const double MU_M = 0.02;
    const double DELTA_H = 0.05;
    const double DELTA_M = 0.15;
    const double ALFA_H = 0.6;
    const double ALFA_M = 0.6;
    const double R = 0.05;
    const double OMEGA = 0.02;
    const double NU_H = 0.5;
    const double NU_M = 0.15;

    w[0]  = LAMBDA_H;
    w[1]  = MU_H * x[0];
    w[2]  = (B * BETA_H * x[0] * x[5]) / (1 + NU_H * x[5]);
    w[3]  = LAMBDA_M;
    w[4]  = MU_M * x[1];
    w[5]  = (B * BETA_M * x[1] * x[4]) / (1 + NU_M * x[4]);
    w[6]  = MU_H * x[2];
    w[7]  = ALFA_H * x[2];
    w[8]  = MU_M * x[3];
    w[9]  = ALFA_M * x[3];
    w[10] = (MU_H + DELTA_H) * x[4];
    w[11] = R * x[4];
    w[12] = (MU_M + DELTA_M) * x[5];
    w[13] = OMEGA * x[6];
    w[14] = MU_H * x[6];
}

void SSA(double T, int *x0, int *x_final) {
    double t = 0.0;
    int x[X_LEN];
    double w[R_ssa];
    double a0, tau;
    int i, r;
    double u1, u2, sum;

    for (i = 0; i < X_LEN; i++) x[i] = x0[i];

    while (t < T) {
        prop(x, w);
        a0 = 0.0;
        for (i = 0; i < R_ssa; i++) a0 += w[i];
```

```
        if (a0 == 0.0) break;

        u1 = (double) rand() / RAND_MAX;
        u2 = (double) rand() / RAND_MAX;
        tau = -log(u1) / a0;
        t += tau;
        if (t >= T) break;

        sum = 0.0;
        for (r = 0; r < R_ssa; r++) {
            sum += w[r];
            if (sum >= u2 * a0) break;
        }

        for (i = 0; i < X_LEN; i++)
            x[i] += P[r][i];
    }

    for (i = 0; i < X_LEN; i++)
        x_final[i] = x[i];
}

void build_histogram(int *values, int N, int bins) {
    int min_val = values[0];
    int max_val = values[0];
    for (int i = 1; i < N; i++) {
        if (values[i] < min_val) min_val = values[i];
        if (values[i] > max_val) max_val = values[i];
    }

    int *hist = calloc(bins, sizeof(int));
    double bin_width = (double)(max_val - min_val) / bins;

    for (int i = 0; i < N; i++) {
        int bin_index = (int)((values[i] - min_val) /
            bin_width);
        if (bin_index == bins) bin_index = bins - 1;
        hist[bin_index]++;
    }

    printf("Histogram bins and counts:\n");
    for (int i = 0; i < bins; i++) {
        double bin_start = min_val + i * bin_width;
        double bin_end = bin_start + bin_width;
        printf("Bin %2d: [%6.2f - %6.2f) Count: %d\n", i,
            bin_start, bin_end, hist[i]);
    }

    free(hist);
}
```

```
127 void run_simulation(double *result) {
128     int x0[X_LEN] = {900, 900, 30, 330, 50, 270, 20};
129     int x_final[X_LEN];
130     SSA(T_FINAL, x0, x_final);
131     for (int i = 0; i < X_LEN; i++) {
132         result[i] = (double)x_final[i];
133     }
134 }
135 
136 int main(int argc, char *argv[]) {
137     MPI_Init(&argc, &argv);
138     int rank, size;
139     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
140     MPI_Comm_size(MPI_COMM_WORLD, &size);
141     srand(time(NULL) + rank);
142     if (argc < 2) {
143         if (rank == 0) {
144             printf("Usage: %s <total_runs>\n", argv[0]);
145         }
146         MPI_Finalize();
147         return 1;
148     }
149 
150     int total_runs = atoi(argv[1]);
151 
152 
153     double start_time = MPI_Wtime();
154 
155     /*Here, I used the dynamic loading balancing, so each
        woker gets a new tasks
156     only when it finishes the previous one, I learn this way
         from the chagpt and the course of HPP*/
157     double *all_results = NULL;
158     if (rank == 0) {
159         all_results = malloc(total_runs * X_LEN * sizeof(
            double));
160         if (all_results == NULL) {
161             fprintf(stderr, "Master: Failed to allocate
                memory for results.\n");
162             MPI_Abort(MPI_COMM_WORLD, 1);
163         }
164     }
165      if (size == 1) {
166         // If only one process, run all simulations
            sequentially on rank 0
167         if (rank == 0) {
168             printf("Running in single-process mode (rank 0
                performs all %d simulations).\n", total_runs)
                ;
169             for (int i = 0; i < total_runs; ++i) {
```

```
170                 double result[X_LEN];
171                 run_simulation(result);
172                 memcpy(&all_results[i * X_LEN], result,
                        X_LEN * sizeof(double));
173             }
174         }
175     } else {
176         // Master-worker with dynamic load balancing
177         if (rank == 0) {
178             int num_sent = 0;
179             int num_results = 0;
180 
181             // 1. Master sends initial tasks to all workers
                    (excluding itself for now)
182             for (int i = 1; i < size && num_sent <
                    total_runs; i++) {
183                 MPI_Send(&num_sent, 1, MPI_INT, i, 0,
                        MPI_COMM_WORLD);
184                 num_sent++;
185             }
186 
187             // 2. Master continues to receive results and
                    send new tasks.
188             while (num_results < total_runs) {
189                 double result[X_LEN];
190                 MPI_Status status;
191 
192                 // 3. Receive results from any worker
193                 MPI_Recv(result, X_LEN, MPI_DOUBLE,
                        MPI_ANY_SOURCE, MPI_ANY_TAG,
                        MPI_COMM_WORLD, &status);
194                 int sender = status.MPI_SOURCE;
195                 memcpy(&all_results[num_results * X_LEN],
                        result, X_LEN * sizeof(double));
196                 num_results++;
197 
198                 // 4. Send next task if available
199                 if (num_sent < total_runs) {
200                     MPI_Send(&num_sent, 1, MPI_INT, sender,
                            0, MPI_COMM_WORLD);
201                     num_sent++;
202                 } else {
203 
204                     int stop_signal = -1;
205                     MPI_Send(&stop_signal, 1, MPI_INT,
                            sender, 0, MPI_COMM_WORLD);
206                 }
207             }
208 
```

```
209             // After all results are collected, send stop
                    signals to any remaining idle workers
210             for (int i = 1; i < size; ++i) {
211                 int stop_signal = -1;
212                 MPI_Send(&stop_signal, 1, MPI_INT, i, 0,
                        MPI_COMM_WORLD);
213             }
214
215         } else {
216             int task_index;
217             while (1) {
218                 MPI_Recv(&task_index, 1, MPI_INT, 0, 0,
                        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
219                 if (task_index == -1) break;
220
221                 double result[X_LEN];
222                 run_simulation(result);
223                 MPI_Send(result, X_LEN, MPI_DOUBLE, 0, 0,
                        MPI_COMM_WORLD);
224             }
225         }
226     }
227     if (rank == 0) {
228         FILE *fp = fopen("histogram_data.csv", "w");
229         if (fp != NULL) {
230             for (int i = 0; i < total_runs; i++) {
231                 double *x = &all_results[i * X_LEN];
232                 fprintf(fp, "%.2f,%.2f,%.2f\n", x[0], x[1],
                        x[2]);
233             }
234             fclose(fp);
235         } else {
236             printf("Error␣opening␣file␣for␣writing.\n");
237         }
238
239         int *S_values = malloc(total_runs * sizeof(int));
240         for (int i = 0; i < total_runs; i++) {
241             S_values[i] = (int) all_results[i * X_LEN];
242         }
243         build_histogram(S_values, total_runs, 20);
244         free(S_values);
245         free(all_results);
246
247         double end_time = MPI_Wtime();
248         printf("Total␣simulation␣time:␣%.3f␣seconds\n",
                end_time - start_time);
249     }
250
251     MPI_Finalize();
252     return 0;
```

```
253 }
```