

Introduction

Three different tests are examined in this writeup: serial runtime, parallel runtime, and parallel communication time. It should be noted that the code was compiled on ACB-I using the command "mpicc main.c" for all 3 configurations: serial, parallel, parallel with communication timers. For all implementations, the number of time steps (k) was varied from 10 to 90 in increments of 10 and the grid size (m) was varied from 1,000 to 10,000 in steps of 1,000. For the parallel implementations, the number of processors (p) was varied from 2 to 24 in steps of 2.

The C code for the simulation, Python code used to parse data, and Matlab code used to create the plots are all included as appendices.

Serial Runtime

As one can see from the serial runtimes, when the time steps are fixed but grid size is increased, there is an approximate parabolic increase in time for calculations. This behavior is expected as the algorithm implemented for Conway's game of life is $O(n^2)$. Minor anomalies were present in $T = 50$ and $T = 60$ at a grid size of 9000x9000, as can be seen in the following figure.

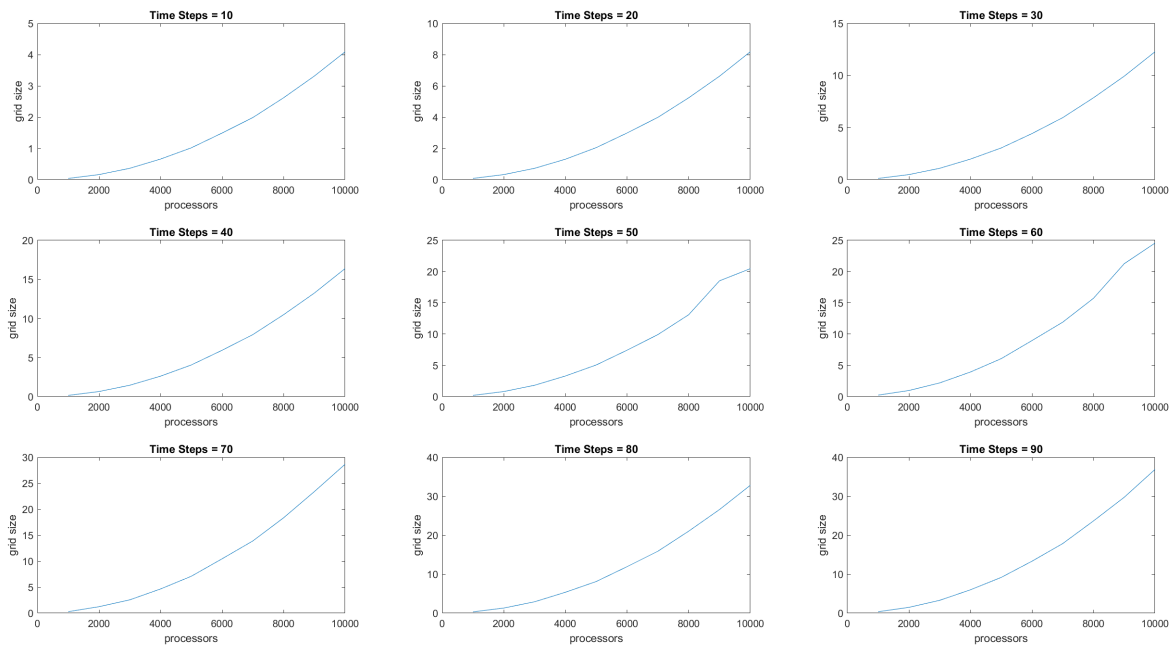


Figure 1: Serial Runtimes for Given Time Steps

Parallel Runtime

The figures within this section contain 3-dimensional meshes with the number of processors as the X axis, the grid size as the Y axis and time as the Z axis. In order to gain a perspective on the impact of the number of time steps on computation time, an adjusted version of each set of plots is included where the time axis is fixed above the maximum time value at 30.

As can be seen in the first set of graphs, the computation time when using MPI follows a mostly parabolic slope, but as the number of processors increases, the algorithm is not quite $O(n^2)$.

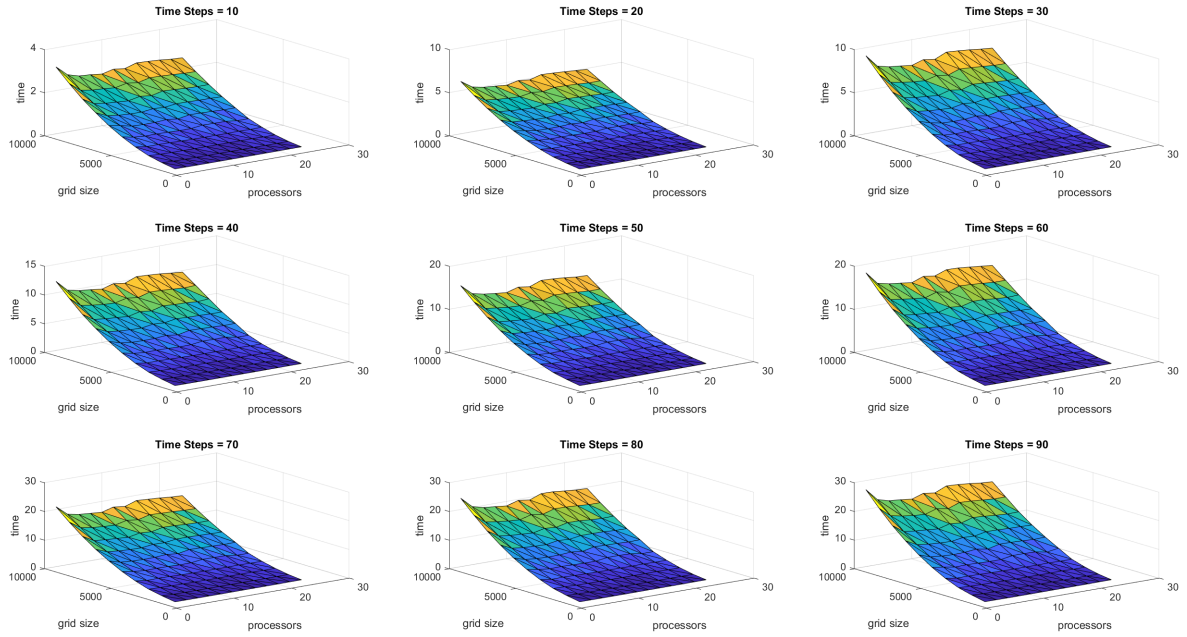


Figure 2: Parallel Runtimes for Given Time Steps

This figure contains the fixed z-axis height in order to provide a visual perspective on how detrimental the number of time steps is to the computation time.

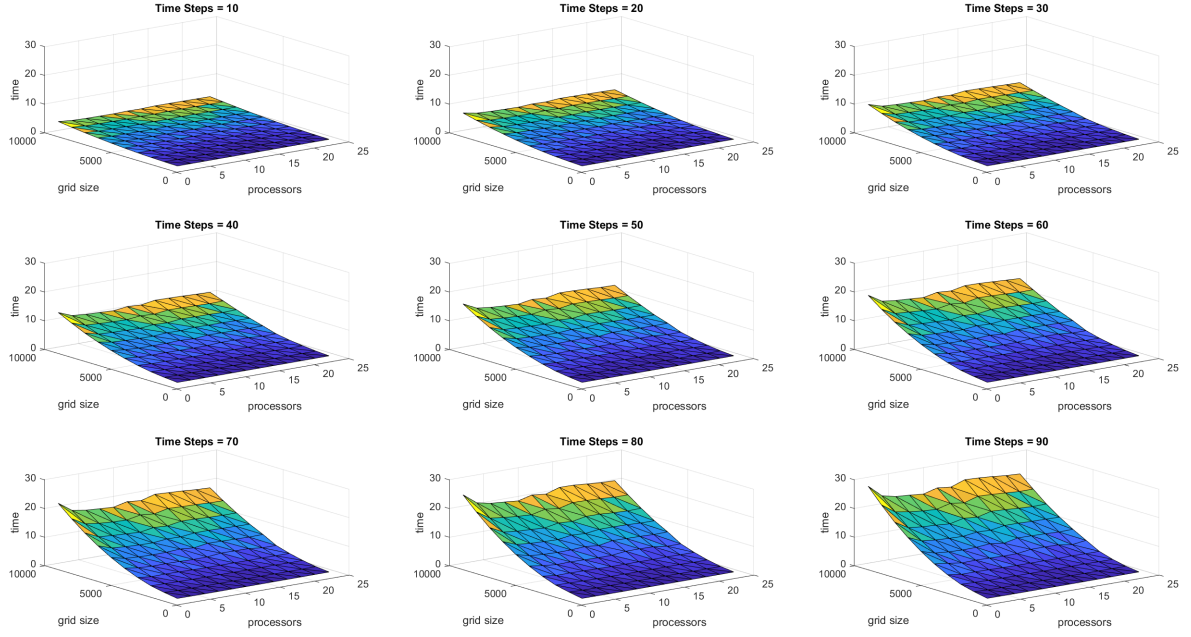


Figure 3: Parallel Runtimes for Given Time Steps (Scaled)

Parallel Communication Time

For calculating the communication time, each processor created a timer before and after the MPI_Sendrecv function calls, computed that time and printed it out into a file corresponding to it's rank. The data was then parsed using a python script in order to find the total communication time for each combination of:

- Number of processors
- Grid size
- Time steps

For example, the communication time for rank 0 and rank 1 were added when $p = 2, k = 10, m = 1000$ and so on.

Two pairs of graphs were created from this data. The first pair of graphs includes the absolute total communication time for each of the combinations, meaning that the time data point corresponding to $k = 90, m = 10,000, p = 24$ contains the total communication time for all 24 processors. This data is presented below in the following two figures.

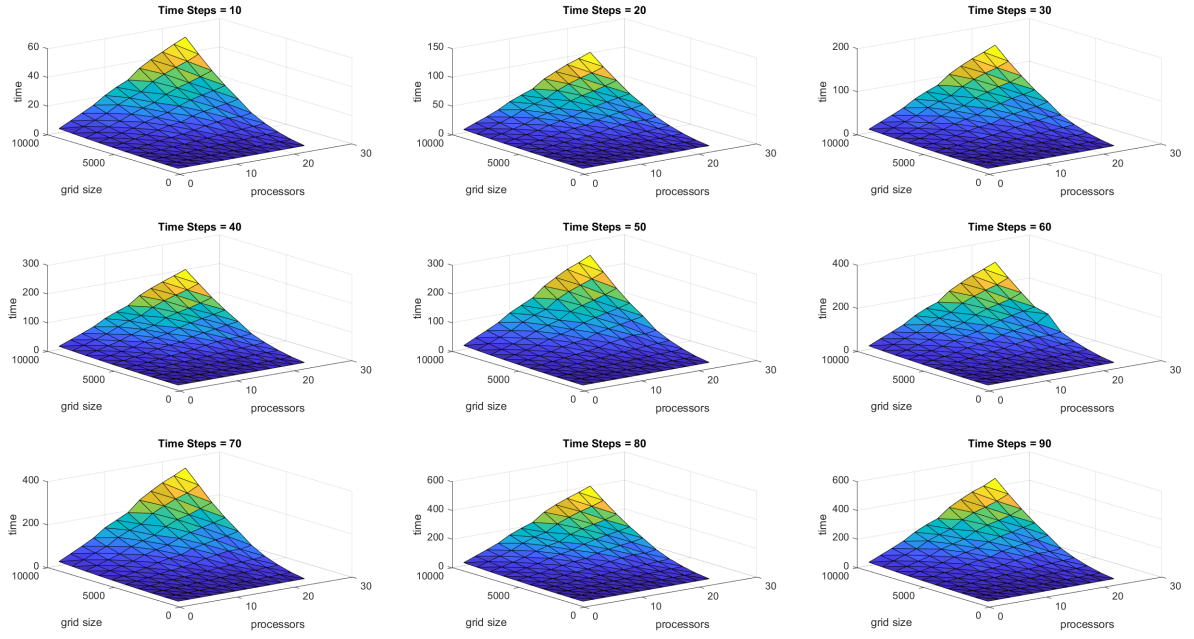


Figure 4: Communication Time for Given Time Steps

The following set of graphs includes the same information as above, but with the time axis fixed at 500 seconds.

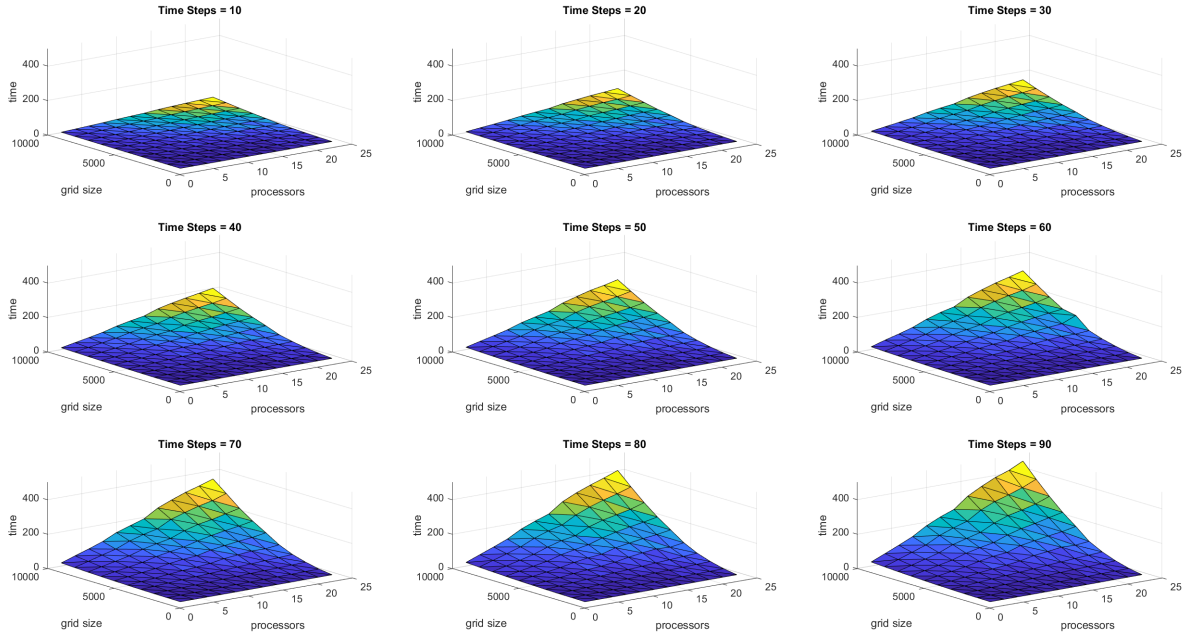


Figure 5: Communication Time for Given Time Steps (Adjusted)

The second pair of graphs contain the average computation time per processor for each

combination listed at the start of this section. It is interesting to note that, although the way the times were gathered is not perfect, at $k=90$, the communication times account for nearly $2/3$ of the total run time.

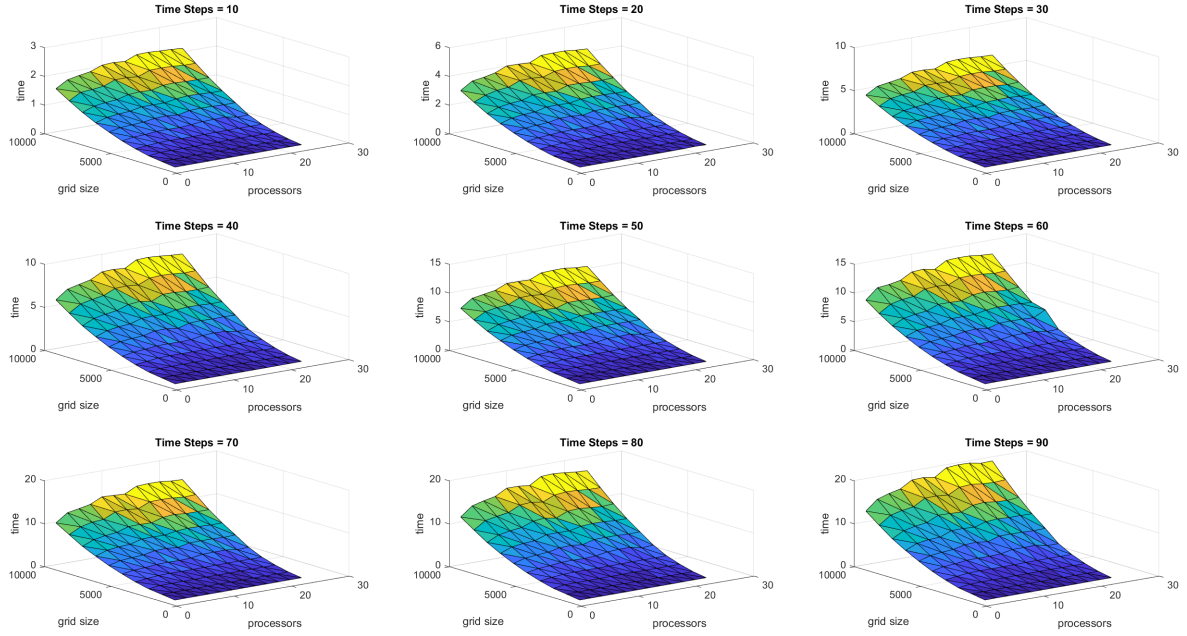


Figure 6: Average Communication Time for Given Time Steps

Again, in the following figure, the z axis is fixed in order to provide a perspective on the relative cost of increasing time steps.

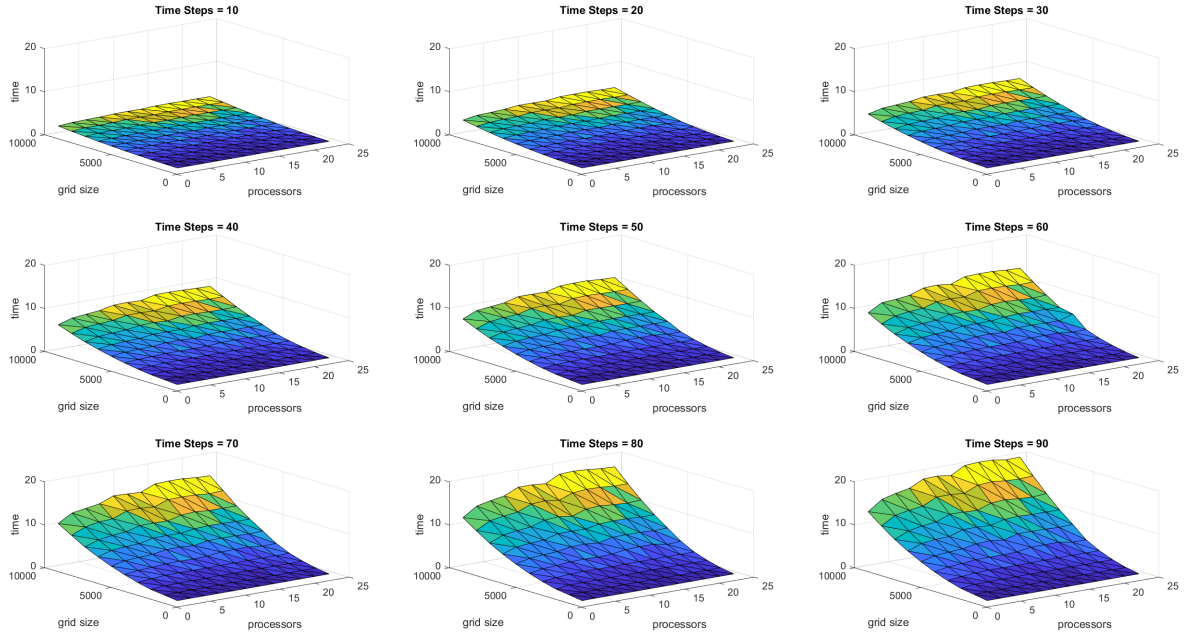


Figure 7: Average Communication Time for Given Time Steps (Adjusted)

MPI Code Appendix

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4 #include <sys/time.h>
5 #include <string.h>
6 #include <time.h>
7
8
9 #define USE_MPI 0
10 #define COMM_TIMER 0
11
12 #if USE_MPI
13
14     #include <mpi.h>
15
16 #endif
17
18 static double timer()
19 {
20     struct timeval tp;
21     gettimeofday(&tp, NULL);
22     return ((double) (tp.tv_sec) + 1e-6 * tp.tv_usec);
23 }
24
25 int main(int argc, char **argv)
26 {
27
28     int rank, num_tasks;
29
30     /* Initialize MPI */
31     #if USE_MPI
32     MPI_Init(&argc, &argv);
33     MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);
34     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
35     MPI_Status topsendrcv, botsendrcv;
36     int max_thread = num_tasks - 1;
37
38     #else
39     rank = 0;
40     num_tasks = 1;
41     #endif
42
43     if (argc != 3)
44     {
45         if (rank == 0)
46         {
47             fprintf(stderr, "%s <m> <k>\n", argv[0]);
48             fprintf(stderr, "Program for parallel Game of Life\n");
49             fprintf(stderr, "with 1D grid partitioning\n");
50             fprintf(stderr, "<m>: grid dimension (an mxm grid is created)\n");
51             fprintf(stderr, "<k>: number of time steps\n");
52             fprintf(stderr, "(initial pattern specified inside code)\n");
53
54             #if USE_MPI
55             MPI_Abort(MPI_COMM_WORLD, 1);
56             #else
57             exit(1);
58             #endif
59         }
60     }
```

```

59     }
60 }
61
62 int m, k;
63
64 m = atoi(argv[1]);
65 assert(m > 2);
66 assert(m <= 10000);
67
68 k = atoi(argv[2]);
69 assert(k > 0);
70 assert(k <= 1000);
71
72 /* ensure that m is a multiple of num_tasks */
73 m = (m / num_tasks) * num_tasks;
74
75 int m_p = (m / num_tasks);
76
77 /* print new m to let user know n has been modified */
78 char *std_filename = calloc(14, sizeof(char));
79 FILE *std_log;
80 if (rank == 0)
81 {
82     strcat(std_filename, "std_times.txt");
83     std_log = fopen(std_filename, "a+");
84     fprintf(stderr, "Using m: %d, m_p: %d, k: %d\n", m, m_p, k);
85     fprintf(stderr, "Requires %3.6lf MB of memory per task\n",
86             ((2 * 4.0 * m_p) * m / 1e6));
87 }
88
89 /* Linearizing 2D grids to 1D using row-major ordering */
90 /* grid[i][j] would be grid[i*n+j] */
91 int *grid_current;
92 int *grid_next;
93 int *boundary_top;
94 int *boundary_bottom;
95
96 grid_current = (int *) calloc((size_t) m_p * m, sizeof(int));
97 assert(grid_current != 0);
98
99 grid_next = (int *) calloc((size_t) m_p * m, sizeof(int));
100 assert(grid_next != 0);
101
102 boundary_bottom = (int *) calloc((size_t) m, sizeof(int));
103 assert(boundary_bottom != 0);
104
105 boundary_top = (int *) calloc((size_t) m, sizeof(int));
106 assert(boundary_top != 0);
107
108 int i, j, t;
109
110 /* static initialization, so that we can verify output */
111 /* using very simple initialization right now */
112 /* this isn't a good check for parallel debugging */
113 #ifdef _OPENMP
114     #pragma omp parallel for private(i,j)
115 #endif
116 for (i = 0; i < m_p; i++)
117 {
118     for (j = 0; j < m; j++)

```



```

119     {
120         grid_current[i * m + j] = 0;
121         grid_next[i * m + j] = 0;
122     }
123 }
124
125 /* initializing some cells in the middle */
126 assert((m * m_p / 2 + m / 2 + 3) < m_p * m);
127 grid_current[m * m_p / 2 + m / 2 + 0] = 1;
128 grid_current[m * m_p / 2 + m / 2 + 1] = 1;
129 grid_current[m * m_p / 2 + m / 2 + 2] = 1;
130 grid_current[m * m_p / 2 + m / 2 + 3] = 1;
131
132 #if USE_MPI
133 MPI_Barrier(MPI_COMM_WORLD);
134 #endif
135
136 double elt = 0.0;
137 if (rank == 0)
138     elt = timer();
139
140 double comm_start = 0.0, comm_time = 0.0;
141 char *time_filename = calloc(25, sizeof(char));
142 FILE *time_log;
143
144 if (COMM_TIMER == 1)
145 {
146     sprintf(time_filename, "rank_%02d_comm_times.txt", rank);
147     time_log = fopen(time_filename, "a+");
148 }
149
150 #if USE_MPI
151 int *full_grid, *full_grid_next;
152 if (rank == 0)
153 {
154     full_grid = (int *) calloc(m_p * m * num_tasks, sizeof(int));
155     full_grid_next = (int *) calloc(m_p * m * num_tasks, sizeof(int));
156 }
157
158 for (t = 0; t < k; t++)
159 {
160     if (COMM_TIMER == 1) comm_start = timer();
161     if (rank > 0)
162     {
163         MPI_Sendrecv(&grid_current[0], m, MPI_INT, rank - 1, 0,
164                     &boundary_top[0], m, MPI_INT, rank - 1, 0,
165                     MPI_COMM_WORLD, &topsendrcv);
166     } // Exchange Top
167     if (rank < max_thread)
168     {
169         MPI_Sendrecv(&grid_current[m * m_p + 1], m, MPI_INT, rank + 1, 0,
170                     &boundary_bottom[0], m, MPI_INT, rank + 1, 0,
171                     MPI_COMM_WORLD, &botsendrcv);
172     } // Exchange Bottom
173     if (COMM_TIMER == 1) comm_time += timer() - comm_start;
174
175     MPI_Barrier(MPI_COMM_WORLD);
176
177     for (i = 1; i < m_p; i++)

```

```

179     {
180         for (j = 1; j < m - 1; j++)
181         {
182             /* avoiding conditionals inside inner loop */
183             int prev_state = grid_current[i * m + j];
184             int num_alive = 0;
185             if (rank != 0 & i == 0)
186             {
187                 num_alive =
188                     grid_current[(i) * m + j - 1] +
189                     grid_current[(i) * m + j + 1] +
190                     grid_current[(i - 1) * m + j - 1] +
191                     grid_current[(i - 1) * m + j] +
192                     grid_current[(i - 1) * m + j + 1] +
193                     boundary_top[j - 1] +
194                     boundary_top[j] +
195                     boundary_top[j + 1];
196
197             } else if (rank != max_thread & i == m_p)
198             {
199                 num_alive =
200                     grid_current[(i) * m + j - 1] +
201                     grid_current[(i) * m + j + 1] +
202                     boundary_bottom[j - 1] +
203                     boundary_bottom[j] +
204                     boundary_bottom[j + 1] +
205                     grid_current[(i + 1) * m + j - 1] +
206                     grid_current[(i + 1) * m + j] +
207                     grid_current[(i + 1) * m + j + 1];
208
209             } else if (i != 0 && i != m_p)
210             {
211                 num_alive =
212                     grid_current[(i) * m + j - 1] +
213                     grid_current[(i) * m + j + 1] +
214                     grid_current[(i - 1) * m + j - 1] +
215                     grid_current[(i - 1) * m + j] +
216                     grid_current[(i - 1) * m + j + 1] +
217                     grid_current[(i + 1) * m + j - 1] +
218                     grid_current[(i + 1) * m + j] +
219                     grid_current[(i + 1) * m + j + 1];
220             }
221
222             grid_next[i * m + j] =
223                 prev_state * ((num_alive == 2) + (num_alive == 3)) + (1 -
224                 prev_state) * (num_alive == 3);
225
226         }
227     }
228     if (COMM_TIMER == 1) comm_start = timer();
229     MPI_Barrier(MPI_COMM_WORLD);
230     MPI_Gather(grid_current, m_p * m, MPI_INT, full_grid, m_p * m, MPI_INT, 0,
231     MPI_COMM_WORLD);
232     MPI_Barrier(MPI_COMM_WORLD);
233     MPI_Gather(grid_next, m_p * m, MPI_INT, full_grid_next, m_p * m, MPI_INT,
234     0, MPI_COMM_WORLD);
235     MPI_Barrier(MPI_COMM_WORLD);
236     if (COMM_TIMER == 1) comm_time += timer() - comm_start;
237     int *grid_tmp = grid_next;

```

```

236     grid_next = grid_current;
237     grid_current = grid_tmp;
238 }
239
240
241 #else
242 /* serial code */
243 /* considering only internal cells */
244 for (t = 0; t < k; t++)
245 {
246     for (i = 1; i < m - 1; i++)
247     {
248         for (j = 1; j < m - 1; j++)
249         {
250             /* avoiding conditionals inside inner loop */
251             int prev_state = grid_current[i * m + j];
252             int num_alive =
253                 grid_current[(i - 1) * m + j - 1] +
254                 grid_current[(i - 1) * m + j + 1] +
255                 grid_current[(i + 1) * m + j - 1] +
256                 grid_current[(i + 1) * m + j + 1] +
257                 grid_current[(i - 1) * m + j] +
258                 grid_current[(i + 1) * m + j] +
259                 grid_current[(i - 1) * m + j] +
260                 grid_current[(i + 1) * m + j];
261
262             grid_next[i * m + j] =
263                 prev_state * ((num_alive == 2) + (num_alive == 3)) + (1 -
264                 prev_state) * (num_alive == 3);
265         }
266     }
267     /* swap current and next */
268     int *grid_tmp = grid_next;
269     grid_next = grid_current;
270     grid_current = grid_tmp;
271 }
272 #endif
273
274 if (rank == 0)
275     elt = timer() - elt;
276
277 if (rank == 0 && COMM_TIMER == 0)
278 {
279     double updates = ((1.0 * m * m) * k / (elt * 1e9));
280     fprintf(std_log, "p\t%d\tm\t%d\tm_p\t%d\tk\t%d\ttime\t%f\tupdates\t%f\n",
281             num_tasks, m, m_p, k, elt, updates);
282 }
283 if (COMM_TIMER == 1)
284 {
285     fprintf(time_log, "rank\t%d\tcomm_time\t%f\n", rank, comm_time);
286 }
287
288 /* free memory */
289 free(grid_current);
290 free(grid_next);
291
292 /* Shut down MPI */
293 #if USE_MPI
294 MPI_Finalize();

```

```
294
295 #endif
296
297     if (rank == 0)
298     {
299         fclose(std_log);
300     }
301     if (COMM_TIMER == 1)
302     {
303         fclose(time_log);
304     }
305
306     return 0;
307 }
```

Parse Code Appendix

```
1 import csv
2
3 with open('data.csv', 'r') as f:
4     reader = csv.reader(f)
5     data = list(reader)
6
7 prev_proc = data[0][0]
8 prev_grid = data[0][1]
9 prev_step = data[0][2]
10 sum = 0.0
11 for i in data:
12     if (prev_proc == i[0] and prev_grid == i[1] and prev_step == i[2]):
13         sum = sum + float(i[4])
14     else:
15         print(prev_proc, prev_grid, prev_step, str(sum), sep='\t')
16         sum = float(i[4])
17         prev_proc = i[0]
18         prev_grid = i[1]
19         prev_step = i[2]
20 print(prev_proc, prev_grid, prev_step, str(sum), sep='\t')
21
22
23 input("Press Enter to continue...")
```

Plotting Code Appendix

```
1 clear;clc
2 x = csvread('comm_times.csv');
3
4 for i = 10:10:90
5     idx = (x(:,3)==i);
6     x_new = x(idx,:);
7     tx = x_new(:,1);
8     ty = x_new(:,2);
9     tz = x_new(:,4);
10    tri = delaunay(tx,ty);
11    plt_idx = i/10;
12    subplot(3,3,plt_idx);
13    h = trisurf(tri, tx, ty, tz);
14    t = strcat("Time Steps = ",num2str(i));
15    title(t);
16    xlabel("processors");
17    ylabel("grid size");
18    zlabel("time");
19    f = "comm_time.png";
20 end
21 print(f,"-dpng");
22
23 for i = 10:10:90
24     idx = (x(:,3)==i);
25     x_new = x(idx,:);
26     tx = x_new(:,1);
27     ty = x_new(:,2);
28     tz = x_new(:,4);
29     tri = delaunay(tx,ty);
30     plt_idx = i/10;
31     subplot(3,3,plt_idx);
32     h = trisurf(tri, tx, ty, tz);
33     axis([0 25 0 10000 0 500]);
34     t = strcat("Time Steps = ",num2str(i));
35     title(t);
36     xlabel("processors");
37     ylabel("grid size");
38     zlabel("time");
39     f = "comm_time_scaled.png";
40 end
41 print(f,"-dpng");
42
43 for i = 10:10:90
44     idx = (x(:,3)==i);
45     x_new = x(idx,:);
46     tx = x_new(:,1);
47     ty = x_new(:,2);
48     tz = x_new(:,4)./tx;
49     tri = delaunay(tx,ty);
50     plt_idx = i/10;
51     subplot(3,3,plt_idx);
52     h = trisurf(tri, tx, ty, tz);
53     t = strcat("Time Steps = ",num2str(i));
54     title(t);
55     xlabel("processors");
56     ylabel("grid size");
57     zlabel("time");
58     f = "avg_comm_time.png";
```

```

59 end
60 print(f,"-dpng");
61
62 for i = 10:10:90
63     idx = ( x(:,3)==i );
64     x_new = x(idx,:);
65     tx = x_new(:,1);
66     ty = x_new(:,2);
67     tz = x_new(:,4)./tx;
68     tri = delaunay(tx,ty);
69     plt_idx = i/10;
70     subplot(3,3,plt_idx);
71     h = trisurf(tri, tx, ty, tz);
72     axis([0 25 0 10000 0 20]);
73     t = strcat("Time Steps = ",num2str(i));
74     title(t);
75     xlabel ("processors");
76     ylabel ("grid size");
77     zlabel ("time");
78     f = "avg_comm_time_scaled.png";
79 end
80 print(f,"-dpng");
81
82 y = csvread ('parallel_times.csv');
83 for i = 10:10:90
84     idx = ( y(:,3)==i );
85     y_new = y(idx,:);
86     tx = y_new(:,1);
87     ty = y_new(:,2);
88     tz = y_new(:,4);
89     tri = delaunay(tx,ty);
90     plt_idx = i/10;
91     subplot(3,3,plt_idx);
92     h = trisurf(tri, tx, ty, tz);
93     axis([0 25 0 10000 0 30]);
94     t = strcat("Time Steps = ",num2str(i));
95     title(t);
96     xlabel ("processors");
97     ylabel ("grid size");
98     zlabel ("time");
99     f = "parallel_time.png";
100 end
101 print(f,"-dpng");
102
103 for i = 10:10:90
104     idx = ( y(:,3)==i );
105     y_new = y(idx,:);
106     tx = y_new(:,1);
107     ty = y_new(:,2);
108     tz = y_new(:,4);
109     tri = delaunay(tx,ty);
110     plt_idx = i/10;
111     subplot(3,3,plt_idx);
112     h = trisurf(tri, tx, ty, tz);
113     t = strcat("Time Steps = ",num2str(i));
114     title(t);
115     xlabel ("processors");
116     ylabel ("grid size");
117     zlabel ("time");
118     f = "parallel_time_scaled.png";

```

```

119 end
120 print(f,"-dpng");
121
122 z = csvread ('serial_times.csv');
123 for i = 10:10:90
124     idx = ( z(:,2)==i );
125     z_new = z(idx,:);
126     tx = z_new(:,1);
127     ty = z_new(:,3);
128     plt_idx = i/10;
129     subplot(3,3,plt_idx);
130     plot(tx,ty);
131     t = strcat("Time Steps = ",num2str(i));
132     title(t);
133     xlabel ("processors");
134     ylabel ("grid size");
135     zlabel ("time");
136     f = "serial_time.png";
137 end
138 print(f,"-dpng");

```