Kyle Salitrik
CMPSC 450
Homework 4 Report

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.

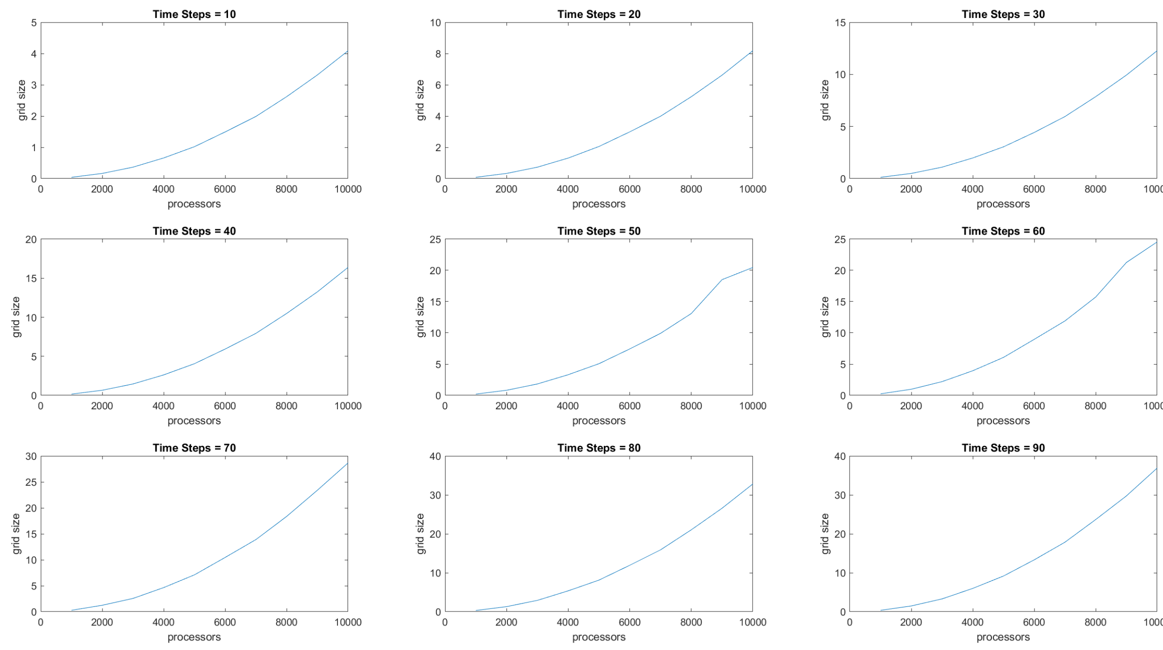

Figure 1: Serial Runtimes for Given Time Steps

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 $O(n^2)$.

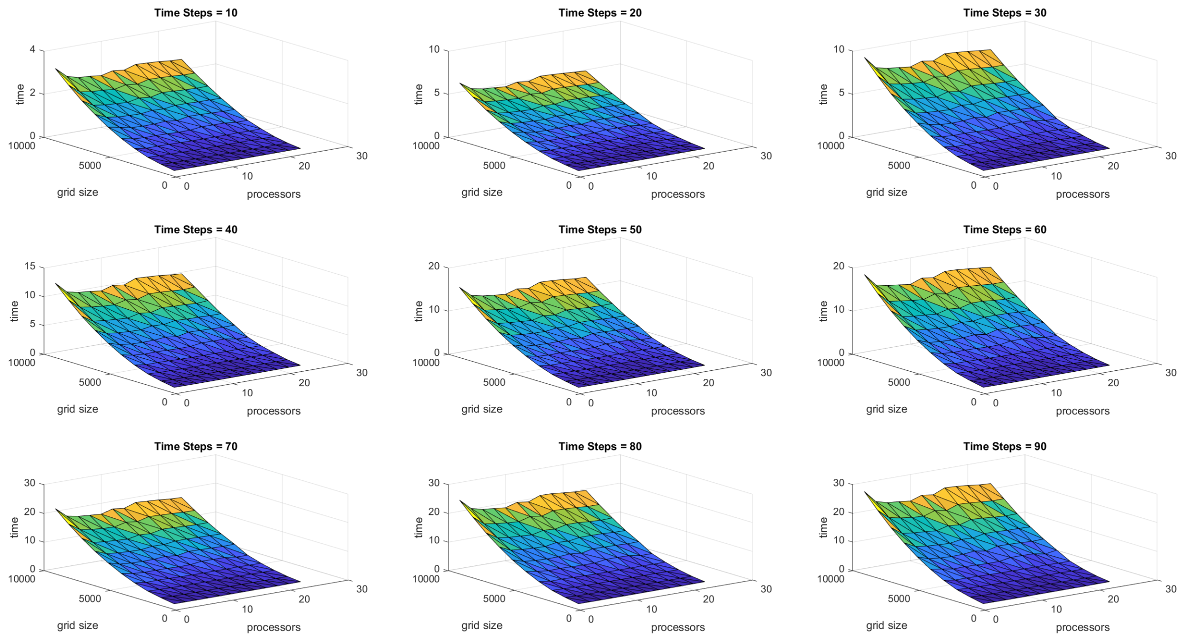

Figure 2: Parallel Runtimes for Given Time Steps

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.

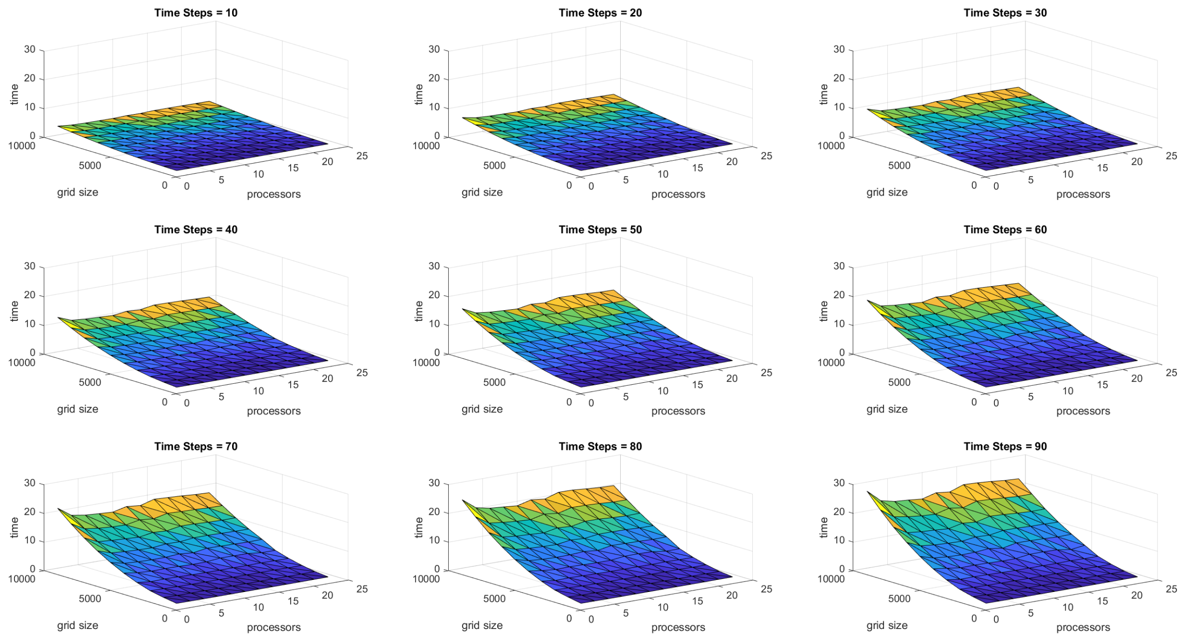

Figure 3: Parallel Runtimes for Given Time Steps (Scaled)

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.

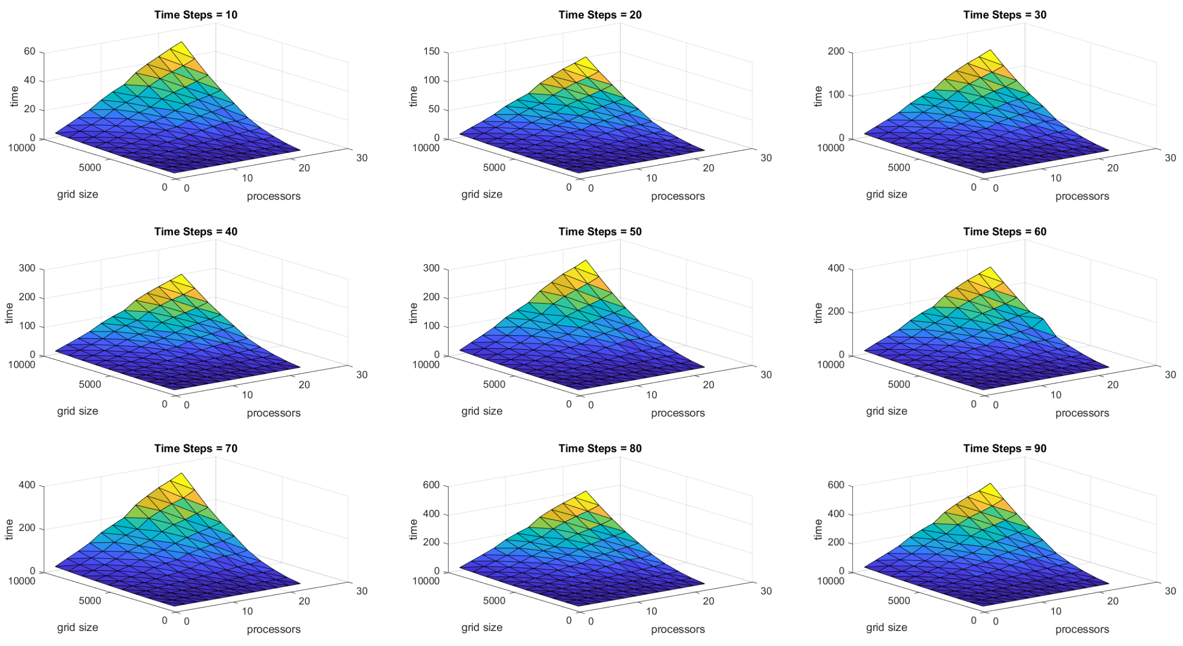

Figure 4: Communication Time for Given Time Steps

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

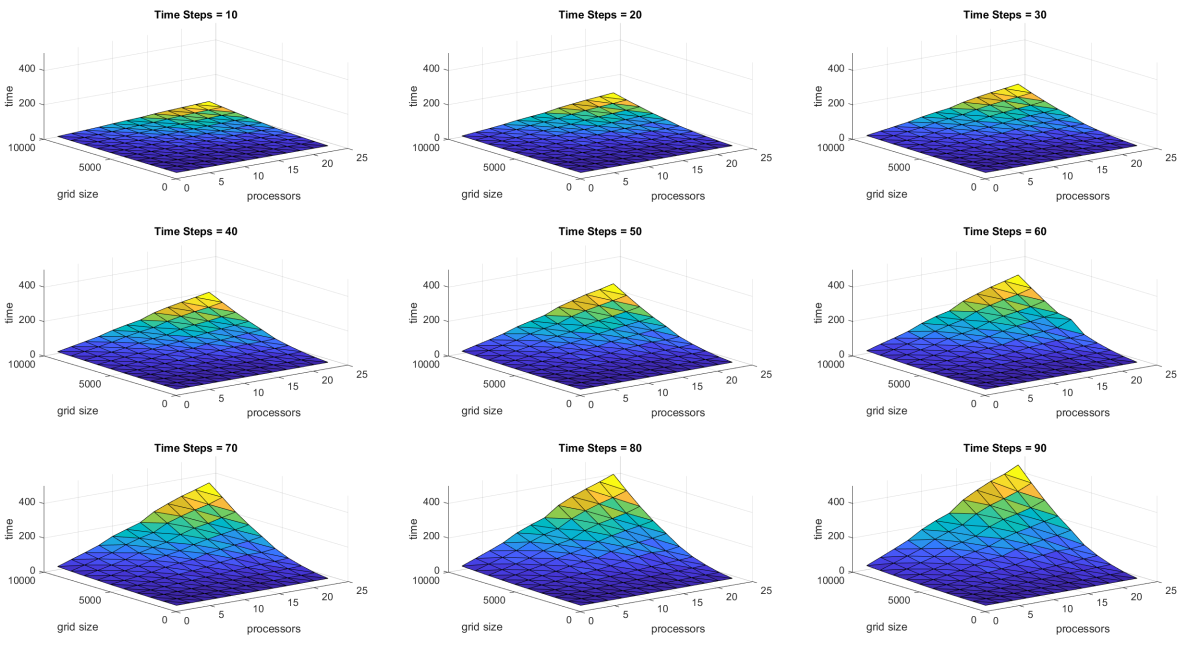

Figure 5: Communication Time for Given Time Steps (Adjusted)

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.

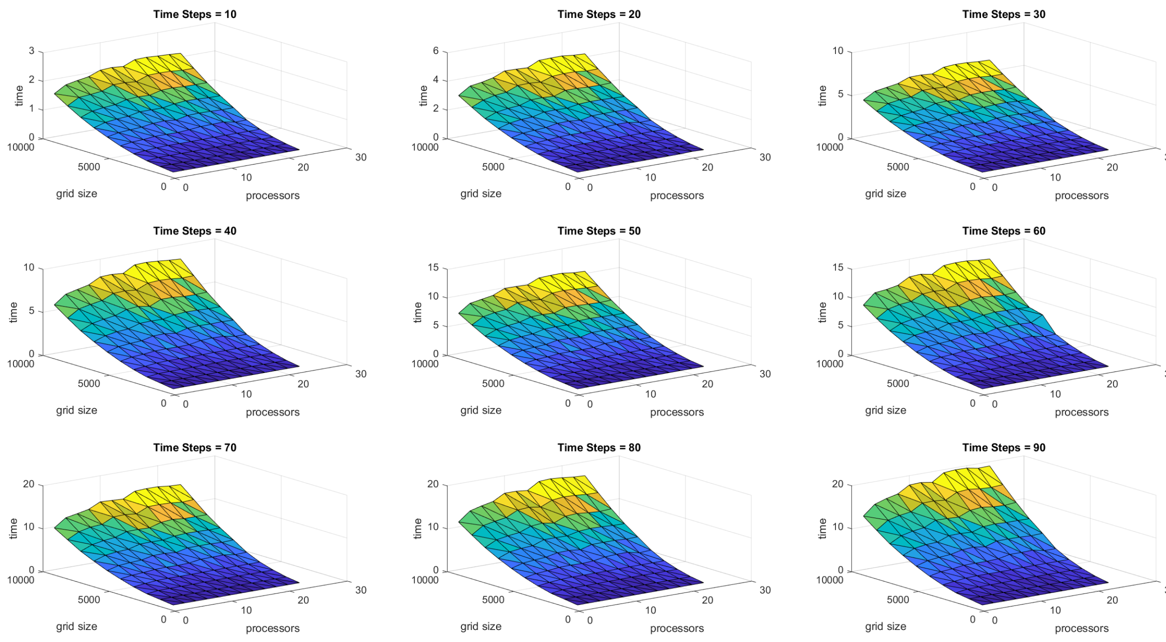

Figure 6: Average Communication Time for Given Time Steps

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

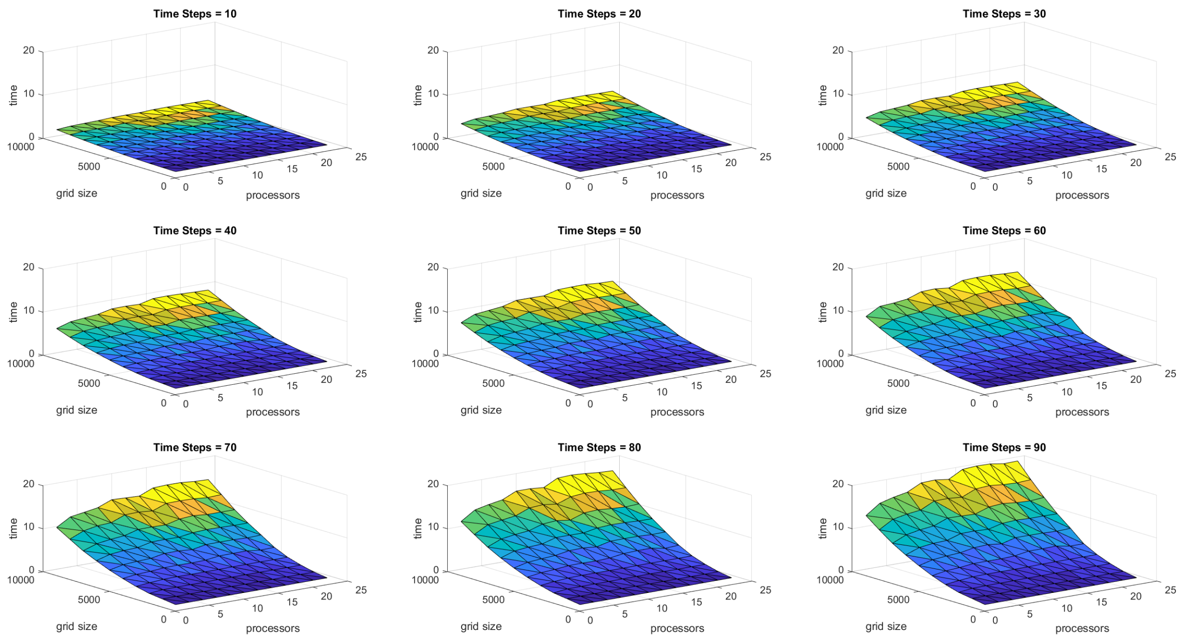

Figure 7: Average Communication Time for Given Time Steps (Adjusted)

MPI Code Appendix

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/time.h>
#include <string.h>
#include <time.h>


#define USE_MPI 0
#define COMM_TIMER 0

#if USE_MPI

    #include <mpi.h>

#endif

static double timer()
{
    struct timeval tp;
    gettimeofday(&tp, NULL);
    return ((double) (tp.tv_sec) + 1e-6 * tp.tv_usec);
}

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

    int rank, num_tasks;

    /* Initialize MPI */
    #if USE_MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &num_tasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Status topsendrcv, botsendrcv;
    int max_thread = num_tasks - 1;

    #else
    rank = 0;
    num_tasks = 1;
    #endif

    if (argc != 3)
    {
        if (rank == 0)
        {
            fprintf(stderr, "%s <m> <k>\n", argv[0]);
            fprintf(stderr, "Program for parallel Game of Life\n");
            fprintf(stderr, "with 1D grid partitioning\n");
            fprintf(stderr, "<m>: grid dimension (an mxm grid is created)\n");
            fprintf(stderr, "<k>: number of time steps\n");
            fprintf(stderr, "(initial pattern specified inside code)\n");

            #if USE_MPI
            MPI_Abort(MPI_COMM_WORLD, 1);
            #else
            exit(1);
            #endif
        }
    }

    int m, k;

    m = atoi(argv[1]);
    assert(m > 2);
    assert(m <= 10000);

    k = atoi(argv[2]);
    assert(k > 0);
    assert(k <= 1000);

    /* ensure that m is a multiple of num_tasks */
    m = (m / num_tasks) * num_tasks;

    int m_p = (m / num_tasks);

    /* print new m to let user know n has been modified */
    char *std_filename = calloc(14, sizeof(char));
    FILE *std_log;
    if (rank == 0)
    {
        strcat(std_filename, "std_times.txt");
        std_log = fopen(std_filename, "a+");
        fprintf(stderr, "Using m: %d, m_p: %d, k: %d\n", m, m_p, k);
        fprintf(stderr, "Requires %3.6lf MB of memory per task\n",
                ((2 * 4.0 * m_p) * m / 1e6));
    }

    /* Linearizing 2D grids to 1D using row-major ordering */
    /* grid[i][j] would be grid[i*n+j] */
    int *grid_current;
    int *grid_next;
    int *boundary_top;
    int *boundary_bottom;

    grid_current = (int *) calloc((size_t) m_p * m, sizeof(int));
    assert(grid_current != 0);

    grid_next = (int *) calloc((size_t) m_p * m, sizeof(int));
    assert(grid_next != 0);

    boundary_bottom = (int *) calloc((size_t) m, sizeof(int));
    assert(boundary_bottom != 0);

    boundary_top = (int *) calloc((size_t) m, sizeof(int));
    assert(boundary_top != 0);

    int i, j, t;

    /* static initalization, so that we can verify output */
    /* using very simple initialization right now */
    /* this isn't a good check for parallel debugging */
    #ifdef _OPENMP
        #pragma omp parallel for private(i,j)
    #endif
    for (i = 0; i < m_p; i++)
    {
        for (j = 0; j < m; j++)
        {
            grid_current[i * m + j] = 0;
            grid_next[i * m + j] = 0;
        }
    }

    /* initializing some cells in the middle */
    assert((m * m_p / 2 + m / 2 + 3) < m_p * m);
    grid_current[m * m_p / 2 + m / 2 + 0] = 1;
    grid_current[m * m_p / 2 + m / 2 + 1] = 1;
    grid_current[m * m_p / 2 + m / 2 + 2] = 1;
    grid_current[m * m_p / 2 + m / 2 + 3] = 1;

    #if USE_MPI
    MPI_Barrier(MPI_COMM_WORLD);
    #endif

    double elt = 0.0;
    if (rank == 0)
        elt = timer();

    double comm_start = 0.0, comm_time = 0.0;
    char *time_filename = calloc(25, sizeof(char));
    FILE *time_log;

    if (COMM_TIMER == 1)
    {
        sprintf(time_filename, "rank_%02d_comm_times.txt", rank);
        time_log = fopen(time_filename, "a+");
    }

    #if USE_MPI
    int *full_grid, *full_grid_next;
    if (rank == 0)
    {
        full_grid = (int *) calloc(m_p * m * num_tasks, sizeof(int));
        full_grid_next = (int *) calloc(m_p * m * num_tasks, sizeof(int));
    }

    for (t = 0; t < k; t++)
    {
        if (COMM_TIMER == 1) comm_start = timer();
        if (rank > 0)
        {
            MPI_Sendrecv(&grid_current[0], m, MPI_INT, rank - 1, 0,
                         &boundary_top[0], m, MPI_INT, rank - 1, 0,
                         MPI_COMM_WORLD, &topsendrcv);
        } // Exchange Top
        if (rank < max_thread)
        {
            MPI_Sendrecv(&grid_current[m * m_p + 1], m, MPI_INT, rank + 1, 0,
                         &boundary_bottom[0], m, MPI_INT, rank + 1, 0,
                         MPI_COMM_WORLD, &botsendrcv);
        } // Exchange Bottom
        if (COMM_TIMER == 1) comm_time += timer() - comm_start;


        MPI_Barrier(MPI_COMM_WORLD);

        for (i = 1; i < m_p; i++)
    {
        for (j = 1; j < m - 1; j++)
        {
            /* avoiding conditionals inside inner loop */
            int prev_state = grid_current[i * m + j];
            int num_alive = 0;
            if (rank != 0 & i == 0)
            {
                num_alive =
                        grid_current[(i) * m + j - 1] +
                        grid_current[(i) * m + j + 1] +
                        grid_current[(i - 1) * m + j - 1] +
                        grid_current[(i - 1) * m + j] +
                        grid_current[(i - 1) * m + j + 1] +
                        boundary_top[j - 1] +
                        boundary_top[j] +
                        boundary_top[j + 1];

            } else if (rank != max_thread & i == m_p)
            {
                num_alive =
                        grid_current[(i) * m + j - 1] +
                        grid_current[(i) * m + j + 1] +
                        boundary_bottom[j - 1] +
                        boundary_bottom[j] +
                        boundary_bottom[j + 1] +
                        grid_current[(i + 1) * m + j - 1] +
                        grid_current[(i + 1) * m + j] +
                        grid_current[(i + 1) * m + j + 1];

            } else if (i != 0 && i != m_p)
            {
                num_alive =
                        grid_current[(i) * m + j - 1] +
                        grid_current[(i) * m + j + 1] +
                        grid_current[(i - 1) * m + j - 1] +
                        grid_current[(i - 1) * m + j] +
                        grid_current[(i - 1) * m + j + 1] +
                        grid_current[(i + 1) * m + j - 1] +
                        grid_current[(i + 1) * m + j] +
                        grid_current[(i + 1) * m + j + 1];
            }


            grid_next[i * m + j] =
                    prev_state * ((num_alive == 2) + (num_alive == 3)) + (1 - prev_state) * (num_alive == 3);

        }
    }
    if (COMM_TIMER == 1) comm_start = timer();
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Gather(grid_current, m_p * m, MPI_INT, full_grid, m_p * m, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Gather(grid_next, m_p * m, MPI_INT, full_grid_next, m_p * m, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    if (COMM_TIMER == 1) comm_time += timer() - comm_start;
    int *grid_tmp = grid_next;
```

```
        grid_next = grid_current;
        grid_current = grid_tmp;
    }


    #else
    /* serial code */
    /* considering only internal cells */
    for (t = 0; t < k; t++)
    {
        for (i = 1; i < m - 1; i++)
        {
            for (j = 1; j < m - 1; j++)
            {
                /* avoiding conditionals inside inner loop */
                int prev_state = grid_current[i * m + j];
                int num_alive =
                        grid_current[(i  )*m+j-1] +
                        grid_current[(i  )*m+j+1] +
                        grid_current[(i-1)*m+j-1] +
                        grid_current[(i-1)*m+j  ] +
                        grid_current[(i-1)*m+j+1] +
                        grid_current[(i+1)*m+j-1] +
                        grid_current[(i+1)*m+j  ] +
                        grid_current[(i+1)*m+j+1];

                grid_next[i * m + j] =
                        prev_state * ((num_alive == 2) + (num_alive == 3)) + (1 - prev_state) * (num_alive == 3);
            }
        }
        /* swap current and next */
        int *grid_tmp = grid_next;
        grid_next = grid_current;
        grid_current = grid_tmp;
    }
    #endif

    if (rank == 0)
        elt = timer() - elt;

    if (rank == 0 && COMM_TIMER == 0)
    {
        double updates = ((1.0 * m * m) * k / (elt * 1e9));
        fprintf(std_log,"p\t%d\tm\t%d\tm_p\t%d\tk\t%d\ttime\t%f\tupdates\t%f\n", num_tasks, m, m_p, k, elt, updates);
    }
    if (COMM_TIMER == 1)
    {
        fprintf(time_log, "rank\t%d\tcomm_time\t%f\n", rank, comm_time);
    }

    /* free memory */
    free(grid_current);
    free(grid_next);

/* Shut down MPI */
#if USE_MPI

    MPI_Finalize();
```

```

#endif

    if (rank == 0)
    {
        fclose(std_log);
    }
    if (COMM_TIMER == 1)
    {
        fclose(time_log);
    }

    return 0;
}
```

Parse Code Appendix

```python
import csv

with open('data.csv', 'r') as f:
  reader = csv.reader(f)
  data = list(reader)

prev_proc = data[0][0]
prev_grid = data[0][1]
prev_step = data[0][2]
sum = 0.0
for i in data:
  if(prev_proc == i[0] and prev_grid == i[1] and prev_step == i[2]):
    sum = sum + float(i[4])
  else:
    print(prev_proc,prev_grid,prev_step,str(sum),sep='\t')
    sum = float(i[4])
    prev_proc = i[0]
    prev_grid = i[1]
    prev_step = i[2]
print(prev_proc,prev_grid,prev_step,str(sum),sep='\t')


input("Press Enter to continue...")
```

Plotting Code Appendix

```
clear;clc
x = csvread ('comm_times.csv');

for i = 10:10:90
  idx = ( x(:,3)==i );
  x_new = x(idx,:);
  tx = x_new(:,1);
  ty = x_new(:,2);
  tz = x_new(:,4);
  tri = delaunay(tx,ty);
  plt_idx = i/10;
  subplot(3,3,plt_idx);
  h = trisurf(tri, tx, ty, tz);
  t = strcat("Time Steps = ",num2str(i));
  title(t);
  xlabel ("processors");
  ylabel ("grid size");
  zlabel ("time");
  f = "comm_time.png";
end
print(f,"-dpng");

for i = 10:10:90
  idx = ( x(:,3)==i );
  x_new = x(idx,:);
  tx = x_new(:,1);
  ty = x_new(:,2);
  tz = x_new(:,4);
  tri = delaunay(tx,ty);
  plt_idx = i/10;
  subplot(3,3,plt_idx);
  h = trisurf(tri, tx, ty, tz);
  axis([0 25 0 10000 0 500]);
  t = strcat("Time Steps = ",num2str(i));
  title(t);
  xlabel ("processors");
  ylabel ("grid size");
  zlabel ("time");
  f = "comm_time_scaled.png";
end
print(f,"-dpng");

for i = 10:10:90
  idx = ( x(:,3)==i );
  x_new = x(idx,:);
  tx = x_new(:,1);
  ty = x_new(:,2);
  tz = x_new(:,4)./tx;
  tri = delaunay(tx,ty);
  plt_idx = i/10;
  subplot(3,3,plt_idx);
  h = trisurf(tri, tx, ty, tz);
  t = strcat("Time Steps = ",num2str(i));
  title(t);
  xlabel ("processors");
  ylabel ("grid size");
  zlabel ("time");
  f = "avg_comm_time.png";
```

```
end
print(f,"-dpng");

for i = 10:10:90
  idx = ( x(:,3)==i );
  x_new = x(idx,:);
  tx = x_new(:,1);
  ty = x_new(:,2);
  tz = x_new(:,4)./tx;
  tri = delaunay(tx,ty);
  plt_idx = i/10;
  subplot(3,3,plt_idx);
  h = trisurf(tri, tx, ty, tz);
  axis([0 25 0 10000 0 20]);
  t = strcat("Time Steps = ",num2str(i));
  title(t);
  xlabel ("processors");
  ylabel ("grid size");
  zlabel ("time");
  f = "avg_comm_time_scaled.png";
end
print(f,"-dpng");

y = csvread ('parallel_times.csv');
for i = 10:10:90
  idx = ( y(:,3)==i );
  y_new = y(idx,:);
  tx = y_new(:,1);
  ty = y_new(:,2);
  tz = y_new(:,4);
  tri = delaunay(tx,ty);
  plt_idx = i/10;
  subplot(3,3,plt_idx);
  h = trisurf(tri, tx, ty, tz);
  axis([0 25 0 10000 0 30]);
  t = strcat("Time Steps = ",num2str(i));
  title(t);
  xlabel ("processors");
  ylabel ("grid size");
  zlabel ("time");
  f = "parallel_time.png";
end
print(f,"-dpng");

for i = 10:10:90
  idx = ( y(:,3)==i );
  y_new = y(idx,:);
  tx = y_new(:,1);
  ty = y_new(:,2);
  tz = y_new(:,4);
  tri = delaunay(tx,ty);
  plt_idx = i/10;
  subplot(3,3,plt_idx);
  h = trisurf(tri, tx, ty, tz);
  t = strcat("Time Steps = ",num2str(i));
  title(t);
  xlabel ("processors");
  ylabel ("grid size");
  zlabel ("time");
  f = "parallel_time_scaled.png";
```

```
end
print(f,"-dpng");

z = csvread ('serial_times.csv');
for i = 10:10:90
  idx = ( z(:,2)==i );
  z_new = z(idx,:);
  tx = z_new(:,1);
  ty = z_new(:,3);
  plt_idx = i/10;
  subplot(3,3,plt_idx);
  plot(tx,ty);
  t = strcat("Time Steps = ",num2str(i));
  title(t);
  xlabel ("processors");
  ylabel ("grid size");
  zlabel ("time");
  f = "serial_time.png";
end
print(f,"-dpng");
```