HW 8
Game of Life

Philip Nelson

2018 October 26

Introduction

The purpose of this assignment is to simulate Conway's Game of Life taking advantage of parallelization through MPI. The program begins by populating the world such that there is a one in five chance of a cell being alive. Then it splits the world up groups of rows and sends them to the other processes. Each process then exchanges information with the processes to the "north" ((rank−1)+world_size)%world_size and to the "south" (rank+1)%world_size. They each determine the state of their strip for the next generation and send the results to the master to be displayed or saved as pngs. This is repeated until the maximum number of generations is reached. The world can be displayed in ascii on the terminal, or saved as a png and combined into an animated gif. The command line arguments are the world width, world height, number of generations to simulate whether or not to output and output type (0 - ascii, 1 - png/gif).

Code

The code is broken up into seven files, main.cpp, communication.hpp, rules.hpp, cell.hpp, random.hpp, output.hpp, and writePNG.hpp. The files are included below.

0.1 main.cpp

```
1  #include "cell.hpp"
2  #include "communication.hpp"
3  #include "output.hpp"
4  #include "random.hpp"
5  #include "rules.hpp"
6  #include <algorithm>
7  #include <fstream>
8  #include <iomanip>
9  #include <iostream>
10 #include <mpi.h>
11 #include <unistd.h>
```

```
12 #include <vector>
13
14 inline void help(std::string msg, int rank)
15 {
16   if (rank == 0)
17   {
18     std::cout << msg << std::endl;
19
20     std::cout
21       << "usage gameOfLife width height generations makeOutput
           outputType"
22       << std::endl;
23   }
24 }
25
26 inline void end()
27 {
28   MPI_Finalize();
29   exit(EXIT_SUCCESS);
30 }
31
32 int main(int argc, char** argv)
33 {
34   MPI_Init(&argc, &argv);
35
36   int rank, world_size;
37   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
38   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
39
40   if (argc < 4)
41   {
42     help("", rank);
43     end();
44   }
45
46   auto shouldOutput = (bool)std::stoi(argv[4]);
47   Output output_type = Output::ASCII;
48   if (shouldOutput)
49   {
50     if (argc > 5)
51     {
52       output_type = static_cast<Output>(std::stoi(argv[5]));
53     }
54     else
55     {
56       help("Not enough arguments provided", rank);
```

```
57        end();
58      }
59    }
60
61    if (output_type > 1)
62    {
63      help("invalid outputType:\n  0 - term ascii\n  1 - gif\n", rank);
64      end();
65    }
66
67    auto width = std::stoi(argv[1]);
68    auto hight = std::stoi(argv[2]);
69    auto iters = std::stoi(argv[3]);
70    auto rpp = hight / world_size;
71    auto aspectRatio = (double)width / hight;
72    width = rpp * world_size;
73    hight = width / aspectRatio;
74
75    //
          ============================================================================//

76    //                                    Master
                                          //
77    //
          ============================================================================//

78    if (0 == rank)
79    {
80      std::cout << "Simulating:\n"
81                << "--------------------\n"
82                << "\n"
83                << width << " x " << hight << " world\n"
84                << "Processes: " << world_size << "\n"
85                << "Rows Per Process: " << rpp << "\n"
86                << "Make output: " << std::boolalpha << shouldOutput << "
                      \n"
87                << "--------------------\n"
88                << "\n";
89
90      /* setup world */
91      World world(hight);
92      std::for_each(begin(world), end(world), [&](std::vector<Cell>& row)
            {
93        for (auto i = 0; i < width; ++i)
94        {
```

```
95          row.push_back(random_int(0, 6) == 0 ? Cell::ALIVE : Cell::DEAD)
                ;
96        }
97      });
98
99      /* print the first generation */
100     if (shouldOutput) print_world(world, output_type);
101
102     /* send strips to other processes */
103     for (int dest = 1, row = rpp; dest < world_size; ++dest)
104     {
105       for (int tag = 0; tag < rpp; ++tag, ++row)
106       {
107         MPI_Send(world[row].data(), width, MPI_INT, dest, tag,
                    MPI_COMM_WORLD);
108       }
109     }
110
111     /* copy out master's strip */
112     World strip(rpp + 2);
113     std::for_each(begin(strip), end(strip), [&](std::vector<Cell>& row)
            {
114       row.resize(width);
115     });
116     for (int row = 0; row < rpp; ++row)
117     {
118       strip[row + 1] = world[row];
119     }
120
121     /* Run the simulation */
122     auto simulationTime = 0.0;
123     auto imageTime = 0.0;
124     double t1, t2, t3;
125     for (auto i = 0; i < iters; ++i)
126     {
127       t1 = MPI_Wtime();
128
129       send_recv(strip, rank, world_size);
130       natural_selection(strip);
131       gatherMaster(world, strip, rpp, world_size);
132
133       t2 = MPI_Wtime();
134
135       if (shouldOutput)
136       {
137         print_world(world, output_type);
```

```
138         std::cout << "generation " << i << " complete\n";
139       }
140
141       t3 = MPI_Wtime();
142
143       simulationTime += t2 - t1;
144       imageTime += t3 - t2;
145       MPI_Barrier(MPI_COMM_WORLD);
146     }
147
148     /* convert the images in ./images/ to a gif using imagemagick */
149     auto t4 = MPI_Wtime();
150     if (shouldOutput)
151       system("convert -loop 0 -delay 25 `ls images | sort -g | sed "
152              "'s-^-images/-'` out.gif");
153     auto t5 = MPI_Wtime();
154     std::cout << "Simulation Time: " << simulationTime
155               << "\nImage Write Time: " << imageTime
156               << "\ngif Creating time: " << t5 - t4 << std::endl;
157     std::ofstream fout;
158     // fout.open("benchmark.csv", std::fstream::app);
159     // fout << world_size << ',' << simulationTime << std::endl;
160   }
161   //
      ============================================================================//
162   //                                  Slave
                                        //
163   //
      ============================================================================//
164   else
165   {
166     World strip(rpp + 2);
167
168     /* resize rows to receive from neighbors */
169     std::for_each(begin(strip), end(strip), [&](std::vector<Cell>& row)
          {
170       row.resize(width);
171     });
172
173     /* receive strip from master */
174     MPI_Status stat;
175     for (auto row = 0; row < rpp; ++row)
176     {
177       MPI_Probe(0, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
```

```
178       MPI_Recv(strip[stat.MPI_TAG + 1].data(),
179                width,
180                MPI_INT,
181                0,
182                MPI_ANY_TAG,
183                MPI_COMM_WORLD,
184                MPI_STATUS_IGNORE);
185     }
186
187     /* run the simulation */
188     for (auto i = 0; i < iters; ++i)
189     {
190       send_recv(strip, rank, world_size);
191       natural_selection(strip);
192       gatherSlave(strip, rpp);
193
194       MPI_Barrier(MPI_COMM_WORLD);
195     }
196   }
197
198   MPI_Finalize();
199
200   return EXIT_SUCCESS;
201 }
```

0.2 communication.hpp

```
1  #ifndef COMMUNICATION_HPP
2  #define COMMUNICATION_HPP
3
4  #include "cell.hpp"
5  #include <mpi.h>
6  #include <vector>
7
8  using World = std::vector<std::vector<Cell>>;
9
10 /**
11  * Share border information with neighbors
12  *
13  * @param world      The representation of the world known to the
       process
14  * @param rank       The process rank
15  * @param world_size The number of processes
16  */
17 void send_recv(World& world, int rank, int world_size)
18 {
19   auto destN = ((rank - 1) + world_size) % world_size;
```

```
20    auto destS = (rank + 1) % world_size;
21    auto ct = world[1].size();
22    /* clang-format off */
23    MPI_Request request1, request2;
24    MPI_Isend(
25      world[1].data(), ct, MPI_INT, destN, 0, MPI_COMM_WORLD, &request1);
26    MPI_Recv(
27      world[world.size()-1].data(), ct, MPI_INT, destS, 0, MPI_COMM_WORLD
          , MPI_STATUS_IGNORE);
28    MPI_Isend(
29      world[world.size()-2].data(), ct, MPI_INT, destS, 0, MPI_COMM_WORLD
          , &request2);
30    MPI_Recv(
31      world[0].data(), ct, MPI_INT, destN, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
32    /* clang-format on */
33    int flag1, flag2;
34    MPI_Test(&request1, &flag1, MPI_STATUS_IGNORE);
35    if (!flag1) std::cout << "Request 1 not finished\n";
36    MPI_Test(&request1, &flag2, MPI_STATUS_IGNORE);
37    if (!flag2) std::cout << "Request 2 not finished\n";
38  }
39
40  /**
41   * Gather all the strips to the master, (called by the master)
42   *
43   * @param world      The representation of the whole world
44   * @param strip      The representation of the world known to the
       process
45   * @param rpp        The number of rows given to each process
46   * @param world_size The number of processes
47   */
48  void gatherMaster(World& world, World& strip, int rpp, int world_size)
49  {
50    MPI_Status stat;
51    for (auto src = 1, row = rpp; src < world_size; ++src)
52    {
53      for (auto recvd = 0; recvd < rpp; ++recvd, ++row)
54      {
55        MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
56        auto r = stat.MPI_SOURCE * rpp + stat.MPI_TAG;
57        MPI_Recv(world[r].data(),
58                 world[r].size(),
59                 MPI_INT,
60                 stat.MPI_SOURCE,
61                 stat.MPI_TAG,
```

```
62                MPI_COMM_WORLD,
63                MPI_STATUS_IGNORE);
64      }
65    }
66    for (auto row = 0; row < rpp; ++row)
67    {
68      world[row] = strip[row + 1];
69    }
70  }
71
72  /**
73   * Gather all the strips to the master (called by the slaves)
74   *
75   * @param strip      The representation of the world known to the
       process
76   * @param rpp        The number of rows given to each process
77   */
78  void gatherSlave(World& strip, int rpp)
79  {
80    for (auto row = 0; row < rpp; ++row)
81    {
82      MPI_Send(strip[row + 1].data(),
83               strip[row + 1].size(),
84               MPI_INT,
85               0,
86               row,
87               MPI_COMM_WORLD);
88    }
89  }
90
91  #endif
```

0.3 rules.hpp

```
1   #ifndef RULES_HPP
2   #define RULES_HPP
3
4   #include "cell.hpp"
5   #include <vector>
6
7   /**
8    * Count the number of neighbors for a cell in the world
9    *
10   * @param world The representation of the whole world
11   * @param i     The row of the cell to check
12   * @param j     The column of the cell to check
13   * @return      The number of alive neighbors
```

```
14   */
15  int get_neighbors(std::vector<std::vector<Cell>> const& world,
16                    unsigned int i,
17                    unsigned int j)
18  {
19    int neighbors = 0;
20    if (j != world[i].size() - 1 && world[i][j + 1] == Cell::ALIVE)
21    {
22      ++neighbors;
23    }
24    if (j != 0 && world[i][j - 1] == Cell::ALIVE)
25    {
26      ++neighbors;
27    }
28    if (i != world.size() - 1 && j != world[i].size() - 1 &&
29        world[i + 1][j + 1] == Cell::ALIVE)
30    {
31      ++neighbors;
32    }
33    if (i != world.size() - 1 && world[i + 1][j] == Cell::ALIVE)
34    {
35      ++neighbors;
36    }
37    if (i != world.size() - 1 && j != 0 && world[i + 1][j - 1] == Cell::
          ALIVE)
38    {
39      ++neighbors;
40    }
41    if (i != 0 && j != world[i].size() - 1 && world[i - 1][j + 1] == Cell
          ::ALIVE)
42    {
43      ++neighbors;
44    }
45    if (i != 0 && world[i - 1][j] == Cell::ALIVE)
46    {
47      ++neighbors;
48    }
49    if (i != 0 && j != 0 && world[i - 1][j - 1] == Cell::ALIVE)
50    {
51      ++neighbors;
52    }
53    return neighbors;
54  }
55
56  /**
57   * determines if a cell is alive or dead based on it's current state
```

```
      and the
58   * number of neighbors
59   *
60   * @param neighbors The number of living neighbors
61   * @param state     The current state of a cell
62   * @return          The state of the cell in the next generation
63   */
64  Cell live_die(int neighbors, Cell state)
65  {
66    if (state == Cell::ALIVE)
67    {
68      if (neighbors < 2)
69      {
70        return Cell::DEAD;
71      }
72      else if (neighbors > 3)
73      {
74        return Cell::DEAD;
75      }
76      else
77      {
78        return Cell::ALIVE;
79      }
80    }
81    if (state == Cell::DEAD)
82    {
83      if (neighbors == 3)
84      {
85        return Cell::ALIVE;
86      }
87      else
88      {
89        return Cell::DEAD;
90      }
91    }
92    return Cell::DEAD;
93  }
94
95  /**
96   * determines the state of every cell in the world for the next
      generation
97   *
98   * @param world The representation of the whole world
99   */
100 void natural_selection(std::vector<std::vector<Cell>>& world)
101 {
```

```
102   std::vector<std::vector<Cell>> next_gen = world;
103   for (auto i = 0u; i < world.size(); ++i)
104     for (auto j = 0u; j < world[i].size(); ++j)
105       next_gen[i][j] = (live_die(get_neighbors(world, i, j), world[i][j
          ]));
106   world = next_gen;
107 }
108 #endif
```

0.4 cell.hpp

```
1  #ifndef CELL_HPP
2  #define CELL_HPP
3
4  /**
5   * The enumeration of cell states
6   */
7  enum Cell
8  {
9    ALIVE, /**< The alive state */
10   DEAD   /**< The dead state */
11 };
12
13 #endif
```

0.5 random.hpp

```
1  #ifndef RANDOM_HPP
2  #define RANDOM_HPP
3
4  #include <algorithm>
5  #include <functional>
6  #include <optional>
7  #include <random>
8
9  /**
10  * Generate a random number from [low, high]
11  *
12  * @param low  The lower bound
13  * @param high The upper bound
14  * @return     A random number on the range [low, high]
15  */
16 template <typename T>
17 int random_int(int low, int high, T seed)
18 {
19   static std::mt19937 mt(seed);
20   std::uniform_int_distribution<> dist(low, high);
```

```
21    return dist(mt);
22  }
23
24  int random_int(int low, int high)
25  {
26    static std::random_device rd;
27    static std::mt19937 mt(rd());
28    std::uniform_int_distribution<> dist(low, high);
29    return dist(mt);
30  }
31
32  /**
33   * Generate a random number from [low, high)
34   *
35   * @param low  The lower bound
36   * @param high The upper bound
37   * @return     A random number on the range [low, high)
38   */
39  double random_double(double low, double high)
40  {
41    static std::random_device rd;
42    static std::mt19937 mt(rd());
43    std::uniform_real_distribution<> dist(low, high);
44    return dist(mt);
45  }
46
47  /**
48   * Fill a container from [first, last) with random numbers from [low,
        high]
49   *
50   * @param first Iterator to beginning of range to fill
51   * @param last  Iterator to end of range to fill
52   * @param low   The lower bound
53   * @param high  The upper bound
54   */
55  template <typename it>
56  void random_int_fill(it first, it last, const int low, const int high)
57  {
58    static std::random_device rd;
59    static std::mt19937 mt(rd());
60    std::uniform_int_distribution<> dist(low, high);
61    std::generate(first, last, std::bind(dist, mt));
62  }
63
64  /**
65   * Fill a container from [first, last) with random numbers from [low,
```

```
     high)
66   *
67   * @param first Iterator to beginning of range to fill
68   * @param last  Iterator to end of range to fill
69   * @param low   The lower bound
70   * @param high  The upper bound
71   */
72  template <typename it>
73  void random_double_fill(it first, it last, const double low, const
        double high)
74  {
75    static std::random_device rd;
76    static std::mt19937 mt(rd());
77    std::uniform_real_distribution<double> dist(low, high);
78    std::generate(first, last, std::bind(dist, mt));
79  }
80
81  #endif
```

0.6 output.hpp

```
1   #ifndef OUTPUT_HPP
2   #define OUTPUT_HPP
3
4   #include "cell.hpp"
5   #include "writePNG.hpp"
6   #include <algorithm>
7   #include <iomanip>
8   #include <iostream>
9   #include <unistd.h>
10  #include <vector>
11
12  enum Output
13  {
14    ASCII, /**< print in ascii to the terminal */
15    GIF    /**< save as sequentially named pngs and converted to a gif */
16  };
17
18  using World = std::vector<std::vector<Cell>>;
19
20  /**
21   * Print a row of the world in ascii to the terminal
22   *
23   * @param row The row of the world to print
24   */
25  void print_ascii_row(std::vector<Cell> const& row)
26  {
```

```
27    for (auto i = 0u; i < row.size(); ++i)
28    {
29      std::cout << (row[i] == Cell::ALIVE ? "*" : ".");
30    }
31    std::cout << '\n';
32  }
33
34  /**
35   * Print the world in ascii to the terminal
36   *
37   * @param world The representation of the whole world
38   */
39  void print_ascii_world(World const& world)
40  {
41    std::cout << "\033[2J\033[1;1H";
42    for (auto i = 0u; i < world.size(); ++i)
43    {
44      for (auto j = 0u; j < world[0].size(); ++j)
45      {
46        std::cout << (world[i][j] == Cell::ALIVE ? "*" : ".");
47      }
48      std::cout << '\n';
49    }
50    std::cout << std::setw(world[0].size()) << std::setfill('-') << '-'
51              << std::endl;
52  }
53
54  /**
55   * Save the world as a png image
56   *
57   * @param world The representation of the whole world
58   */
59  void print_png_world(World const& world)
60  {
61    static int image_num = 0;
62    int scale = 3;
63    std::vector<uint8_t> image;
64    std::for_each(begin(world), end(world), [&](auto row) {
65      for (int l = 0; l < scale; ++l)
66      {
67        std::for_each(begin(row), end(row), [&](auto c) {
68          auto color = c == Cell::ALIVE ? 0 : 255;
69          for (auto i = 0; i < scale; ++i)
70          {
71            image.push_back(color);
72            image.push_back(color);
```

```
73            image.push_back(color);
74          }
75        });
76      }
77    });
78    std::string filename = "images/" + std::to_string(++image_num) + ".
          png";
79    save_png_libpng(
80      filename, image.data(), world[0].size() * scale, world.size() *
          scale);
81  }
82
83  /**
84   * Print the world in the given output type
85   *
86   * @param world       The representation of the whole world
87   * @param output_type The output type
88   */
89  void print_world(World world, Output output_type)
90  {
91    switch (output_type)
92    {
93    case Output::ASCII:
94      print_ascii_world(world);
95      sleep(1);
96      break;
97    case Output::GIF:
98      print_png_world(world);
99      break;
100   }
101 }
102
103 #endif
```

0.7 writePNG.hpp

```
1  #ifndef WRITE_PNG_HPP
2  #define WRITE_PNG_HPP
3
4  #include <png.h>
5  #include <string>
6
7  /**
8   * Saves a pixel buffer as a png using libpng
9   *
10  * @param filename The name to save the image to
11  * @param pixels   The buffer of pixel data
```

```
12   * @param w         The width of the image
13   * @param h         The height of the image
14   */
15  bool save_png_libpng(const std::string filename, uint8_t* pixels, int w
       , int h)
16  {
17    png_structp png =
18      png_create_write_struct(PNG_LIBPNG_VER_STRING, nullptr, nullptr,
          nullptr);
19    if (!png)
20    {
21      return false;
22    }
23
24    png_infop info = png_create_info_struct(png);
25    if (!info)
26    {
27      png_destroy_write_struct(&png, &info);
28      return false;
29    }
30
31    FILE* fp = fopen(filename.c_str(), "wb");
32    if (!fp)
33    {
34      png_destroy_write_struct(&png, &info);
35      return false;
36    }
37
38    png_init_io(png, fp);
39    png_set_IHDR(png,
40                 info,
41                 w,
42                 h,
43                 8 /* depth */,
44                 PNG_COLOR_TYPE_RGB,
45                 PNG_INTERLACE_NONE,
46                 PNG_COMPRESSION_TYPE_BASE,
47                 PNG_FILTER_TYPE_BASE);
48    png_colorp palette =
49      (png_colorp)png_malloc(png, PNG_MAX_PALETTE_LENGTH * sizeof(
          png_color));
50    if (!palette)
51    {
52      fclose(fp);
53      png_destroy_write_struct(&png, &info);
54      return false;
```

```
55    }
56    png_set_PLTE(png, info, palette, PNG_MAX_PALETTE_LENGTH);
57    png_write_info(png, info);
58    png_set_packing(png);
59
60    png_bytepp rows = (png_bytepp)png_malloc(png, h * sizeof(png_bytep));
61    for (int i = 0; i < h; ++i)
62    {
63      rows[i] = (png_bytep)(pixels + (h - i) * w * 3);
64    }
65
66    png_write_image(png, rows);
67    png_write_end(png, info);
68    png_free(png, palette);
69    png_destroy_write_struct(&png, &info);
70
71    fclose(fp);
72    delete[] rows;
73    return true;
74  }
75
76  #endif
```

Output

```
# mpic++ -O3 -lpng main.cpp -o gameOfLife.out

# mpiexec --oversubscribe -n 5 gameOfLife.out 1024 1024 25 0 0

Simulating:
--------------------

1020 x 1020 world
Processes: 5
Rows Per Process: 204
Make output: false
--------------------

Simulation Time: 0.243867
Image Write Time: 5.0079e-06
gif Creating time: 5.90226e-08

# mpiexec --oversubscribe -n 5 gameOfLife.out 10 10 10 1 0

Simulating:
--------------------

10 x 10 world
Processes: 5
Rows Per Process: 2
Make output: true
--------------------

..........
.....*....
........*.
*.*....*..
........*.
.......*..
.......**.
..........
.....**...
....*.....
----------
..........
..........
..........
.......**.
```

```
.......**.
.......*..
.......**.
......**..
.....*....
.....*....
----------
generation 0 complete
..........
..........
..........
.......**.
......*...
......*...
........*.
......***.
.....*....
..........
----------
generation 1 complete
..........
..........
..........
.......*..
......*...
.......*..
......*.*.
......***.
......**..
..........
----------
generation 2 complete
..........
..........
..........
..........
......**..
......**..
......*.*.
.....*..*.
......*.*.
..........
----------
generation 3 complete
..........
..........
```

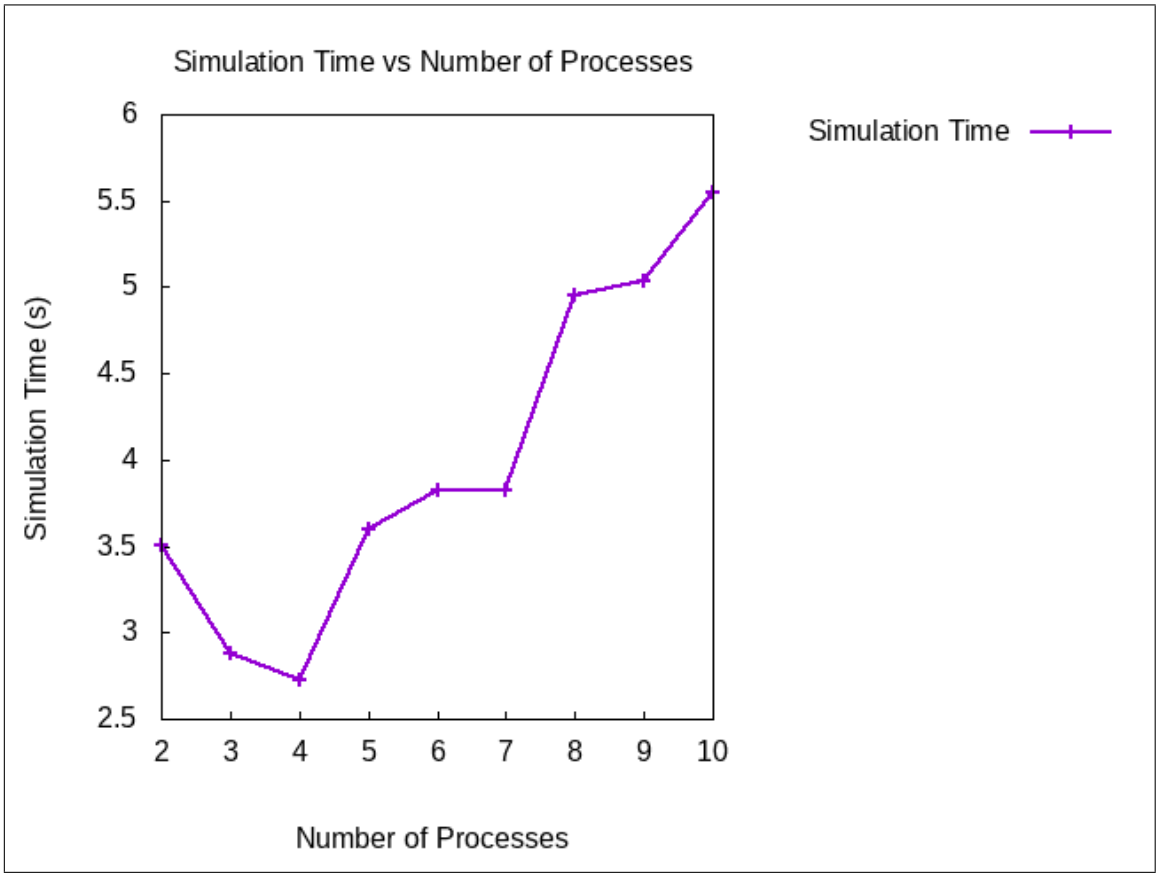
```
..........
..........
......**..
.....*..*.
.....**.*.
.....**.**
.......*..
..........
----------
generation 4 complete
..........
..........
..........
..........
......**..
.....*..*.
....*...*.
.....*..**
......***.
..........
----------
generation 5 complete
..........
..........
..........
..........
......**..
.....**.*.
....**.**.
.....**..*
......****
.......*..
----------
generation 6 complete
..........
..........
..........
..........
.....***..
....*...*.
....*...**
....*....*
.....*...*
......**..
----------
generation 7 complete
```

```
..........
..........
..........
......*...
.....***..
....*.*.**
...***..**
....**...*
.....**.*.
......*...
----------
generation 8 complete
..........
..........
..........
.....***..
........*.
...*.....*
...*..**..
...*...*.*
....*.**..
.....***..
----------
generation 9 complete

Simulation Time: 0.0599119
Image Write Time: 10.0065
gif Creating time: 0.439408
```

Findings

The findings were fairly unsurprising. My computer has two cores with hyper-threading and it is clear to see that performance improves as the number of processes increases until it surpasses the hardware. The graph below shows a 1024 x 1024 world simulated for 500 generations.


Simulation Time vs Number of Processes
Simulation Time
Simulation Time (s)
Number of Processes
6
5.5
5
4.5
4
3.5
3
2.5
2
3
4
5
6
7
8
9
10

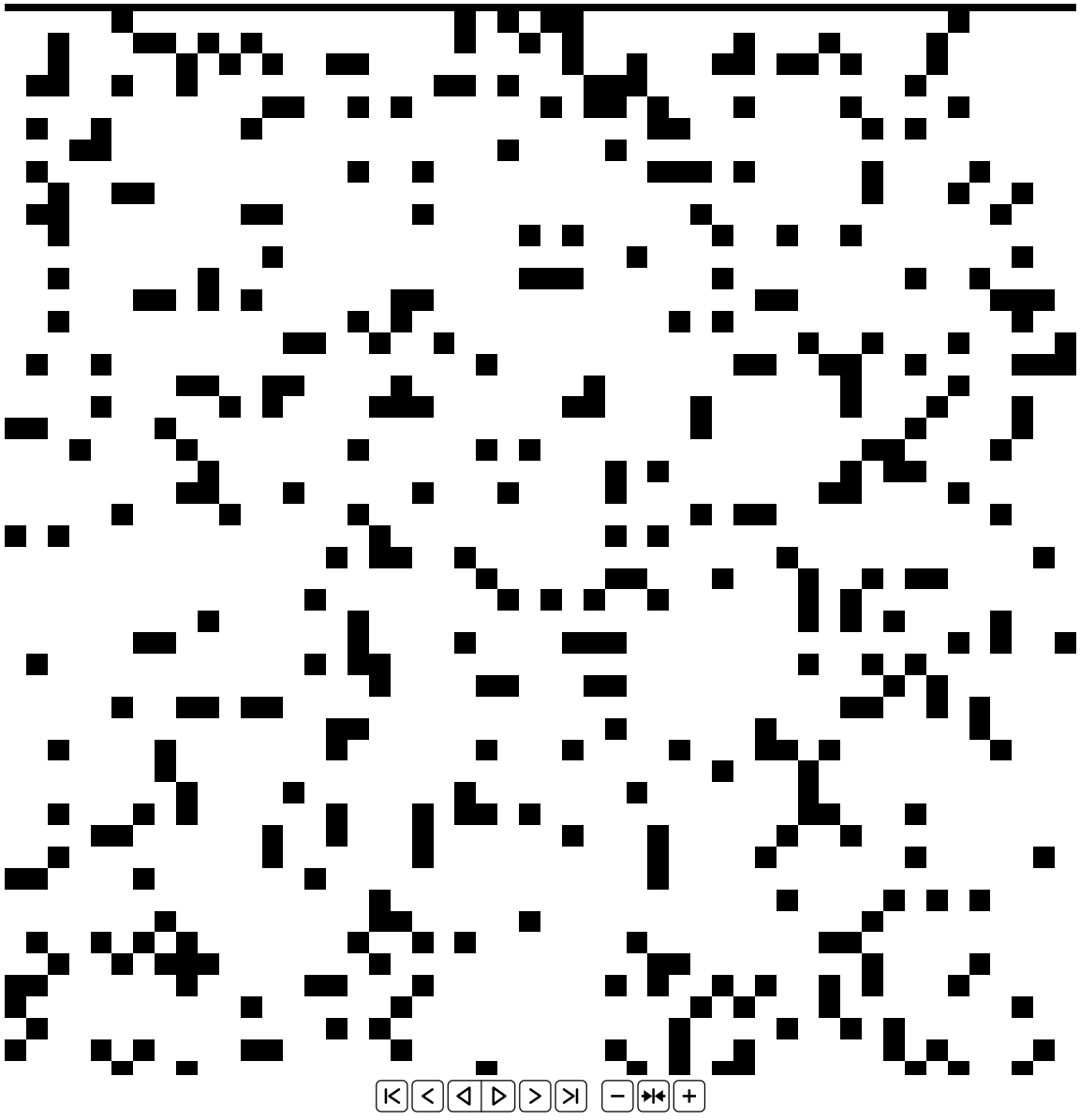


Figure 1: Conway's Game of Life GIF