[[1]](#footnote-1)

Computer Programming 3 – Assignment 3

J. Francis

# INTRODUCTION

T

HIS report will document the speed differences between the single threaded and multithreaded implementations of the game of life and also discuss design decisions within each of the objectives.

# Objective 1

To begin the skeleton code was downloaded and observed while running to determine how the GUI could be made more responsive. It was found that within the play button action within the GameOfLifeGUI class, the function that processes life was running on the main thread. Upon discovering this, another class was made, GUI Worker, in order to move the life processing to another thread. After this was done, the GUI became responsive and as an additional surprise, the GUI began to dynamically update. It was left like this to serve as a “progress update” for the user and because its more interesting to look at rather than looking at a blank screen and only viewing the final result after x number of generations. Additional features were added like the ability to pause the simulation and clear the board.

# Objective 2

Objective 2 involved implementing the multithreaded solution for the game of life. This was achieved by firstly making another class that implements runnable to act as the worker thread. The overridden run method contains the logic for determining the number of alive surrounding squares of the current square and determining if it should stay alive, die or come to life. This means that the board initiation and updating the points are computed in a linear fashion while the “grunt” of the computational work is done in parallel. A thread pool and a countdown latch were also utilized to achieve this. The parallel computation works by dividing the board up into rows and giving each thread a number of rows to work on individually. To protect against race conditions, the points on the board are updated by the main thread after all the worker threads have called latch.countdown(). Additional thread safe precautions involve moving the isAlive and birth logic into their own methods within the LifeProcessWorker class and synchronizing on survivingCells before adding new points to the array list. The result of these changes will be discussed in objective 3.

# Objective 3

Objective 3 involves comparing the improvement in speed between the single and multi-threaded solutions. Some results can be seen below.

Generations: 1000 – Board Size: 160x64 – Block Size: 5

|  |  |  |
| --- | --- | --- |
| Fill Percentage  (%) | SingleThreaded Time (s) | MultiThreaded Time (s) |
| 10% | 2.600 | 0.991 |
| 20% | 2.629 | 0.821 |
| 30% | 2.832 | 0.809 |
| 40% | 2.760 | 0.790 |
| 50% | 2.645 | 0.770 |
| 60% | 2.708 | 0.801 |
| 70% | 2.705 | 0.761 |
| 80% | 2.658 | 0.801 |
| 90% | 2.707 | 0.820 |

Generations: 10000 – Board Size: 160x64 – Block Size: 5

|  |  |  |
| --- | --- | --- |
| Fill Percentage  (%) | SingleThreaded Time (s) | MultiThreaded Time (s) |
| 10% | 24.282 | 8.94 |
| 20% | 27.110 | 7.815 |
| 30% | 26.986 | 7.876 |
| 40% | 27.124 | 7.855 |
| 50% | 27.182 | 7.724 |
| 60% | 27.410 | 7.595 |
| 70% | 27.164 | 7.693 |
| 80% | 27.750 | 7.714 |
| 90% | 26.947 | 7.684 |

It can be seen from the first table that the multithreaded solution is approximately 2.6 times quicker than the single threaded solution. The second table shows that the multithreaded solution is approximately 3.4 times quicker. Obviously, as the number of generations increases, the difference in time between the single and multi-threaded solutions will also continue to increase. The theoretical maximum increase in speed is hypothesized to be an increase in a little less than 8 times faster with the multithreaded solution.

Generations: 1000 – Board Size: 374x175 – Block Size: 5

|  |  |  |
| --- | --- | --- |
| Fill Percentage  (%) | SingleThreaded Time (s) | MultiThreaded Time (s) |
| 10% | 14.944 | 5.302 |
| 20% | 14.324 | 5.211 |
| 30% | 14.323 | 5.310 |
| 40% | 14.594 | 5.196 |
| 50% | 16.614 | 5.404 |
| 60% | 16.880 | 5.314 |
| 70% | 15.994 | 5.117 |
| 80% | 14.244 | 5.710 |
| 90% | 16.877 | 4.934 |

Generations: 10000 – Board Size: 374x175 – Block Size: 5

|  |  |  |
| --- | --- | --- |
| Fill Percentage  (%) | SingleThreaded Time (s) | MultiThreaded Time (s) |
| 10% | 162.934 | 52.173 |
| 20% | 167.730 | 50.962 |
| 30% | 163.439 | 50.430 |
| 40% | 163.417 | 51.184 |
| 50% | 163.543 | 51.104 |
| 60% | 163.594 | 50.125 |
| 70% | 162.995 | 53.938 |
| 80% | 163.958 | 51.354 |
| 90% | 161.966 | 50.844 |

Interestingly it can be seen that the time for the multithreaded solution is more independent of the fill percentage than the single threaded solution is. Also, it is evident that there is not exactly an 8 times speed up. This is because the work is not evenly spread between the 8 threads i.e. linear initialization and point updating and also because the rows do not always divide evenly between the 8 threads. Other reasons include thread overheads, scheduling, other programs / processes running and context switching.

These results were captured on a machine with an Intel core i7-3740QM @ 2.7GHz with 32GB’s of DDR3 RAM utilizing 8 threads.

# Limitations and Design Decisions

When implementing the multithreaded solution, a thread pool was chosen as it makes it easier to shutdown threads properly after use. A countdown latch was also chosen as it is a non-blocking method and is simple to implement. As for the logic, the board was split up into rows for individual threads as the math was easier to implement rather than segmenting the board into sections. I chose to implement runnable rather than extend thread as I didn’t have much experience with runnables and wished to try and use them. Within the LifeProcessWorker, I chose to move the critical regions into private methods and synchronize them such that multiple threads could not add to the survivingCells array concurrently in order to avoid concurrency issues.

Some limitations include resizing issues where the game of life is not contained within the right and bottom borders. This was not fixed due to time limitations. Also, every now and then a null pointer is thrown when running the multithreaded solution for extended periods of time. Due to time constraints, I was unable to correct this minor issue.

1. [↑](#footnote-ref-1)