

Cellular Automata

**Title**

The simulation of the spread of Ebola across the population of Dublin, Ireland using Cellular Automata.

**Authors**

Christopher Slattery B00092939

Joseph Tierney B00092923

Derek McCarthy B00007439

Jenny O'Connell B00086441

**Abstract**

This paper was written to describe the implementation and visualisation of the spread of ebola in county Dublin, Ireland. It will also describe the technologies used to achieve this. The paper will also describe what a Cellular Automata(CA) is, what a CA is used for and how it is implemented.The paper will be describing the difference between a serial CA program and a parallel CA program. In this paper we will also be showing flow charts on both a serial and parallel program. We will also have an IPO chart for the serial and parallel programs to show the design of the both programs. The paper will also show the user a flow chart of the serial program and parallel programs,and the neighbourhood model we will use. It will also show the rules we are hoping to implement into the CA. It will also show the conclusion we got from this project, testing, the results and whatever future work we could add to the CA.

**Introduction**

The purpose of this report is to show how a computer program can be used to simulate the spread of infectious diseases such as Ebola. This is achieved by using Cellular Automata and a predefined set of rules. During this report we will talk about the two programming techniques we used for the simulation Sequential programming and Parallel programming. we will also discuss the tests we carried out and the results we got from the simulation. And finally, we will talk about what improvements we could make to the program if any.

**Background**

The Ebola virus is very severe disease and in many cases fatal illness to humans if left untreated. The first reported cases of the disease were reported in Africa in 1976[3]. Between 2014 and 2016 saw the largest outbreak of the disease since its discovery. This led to many deaths in west Africa as it started to spread from country to country.

It is though that its transmitted to humans who have come into close contact with the blood or other bodily floods of infected animals. These infected animals being chimpanzees, gorillas, fruit bats and monkeys. Scientist believe that fruit bats are natural hosts of the Ebola virus.

It can be spread from human to human contact as well. It is spread by humans coming into direct contact with an infected person. It can also be spread by coming into direct contact with an infected persons blood or other bodily floods. As long as the virus remains in an infected persons blood they will remain infected.

Throughout this report we will discuss how we simulated the spread of the Ebola virus using Cellular Automata.

**Cellular Automata**

**What is Cellular Automata?**

A Cellular Automaton is a model used for simulating something from the natural world in the virtual world. It is made up of a grid of cells. Each cell can have different states. An example of different states would be on or off. More states can be added to a CA depending on what the CA is being implemented for. The states start off having a certain state.These states can change over time, depending on what rules are implemented into the program. The rules for the changing of the states will usually apply to every cell in the grid although there are some exceptions.

**What is it used for?**

A Cellular Automaton(CA) is a framework used for modelling things form the physical world with a computer program. It is used for grasping an idea of anomalies in the real world. They are also used for grasping an accurate prediction for anomalies in the real world. After understanding this, you can get an idea of how a CA can be very useful to the physical world. It can be used for simulating the spread of a disease like ebola, which we will be doing in this project. It can also be used to simulate the spread of less harmful things like the flu or a disease like foot and mouth disease. A CA is something that is used by major health organizations around the world, including the World Health Organization(WHO).

**How it's implemented**

A CA is implemented by either having a 2 dimensional grid of cells or two 2 dimensional grids of cells. Each cells has a state and that state can change over the course of the run through of the program depending on what rule are in place. CA’s are usually implemented through a computer program. They can be implemented in many different programming languages. The changing of a cell’s state is decided by its surrounding neighbour states. When looking at the rules part of this project, you can get a much better understanding of how a cell’s state can be changed depending on its surrounding cell states.

**Overview**

In this paper, we will talk about Cellular Automata, what they are, what they are used for and how they are implemented. We will also talk about why they are useful t the physical world and the scientific world. We will also give readers an introduction on cellular automata. We also give visual representation of how the program works. This is done using graphs and flowcharts. This is done for both serial and parallel programs. We also go through the rules for the programs, the initial state of cells, and the number of different states we incorporated into our program. We talk about the program design and the model we are going to use for our program, including the neighbourhood we chose for the program. We also show the a visual representation of the CPU usage during the running of both programs and compare both of them. We also talk about conclusions to the project and what future work we would implement if we had more time.

**Program Design <HEADING 1>**

**States <HEADING 2>**

For our program we used three states. The states we used in our program were Susceptible Infected Recovered based on the SIR model. Each state in our program is represented by a numeric value 1,2 and 3 respectively.

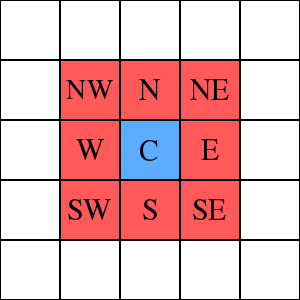
**Rules <HEADING 2>**

Once we decided on the states we would use in the program we needed to devise a ruleset to govern the cellular automata.After some deliberation on what rules we should apply we decided to use the following,

1. A cell transitions from susceptible to infected if 2 or more neighbours are infected.
2. A cell transitions from infected to recovered if 2 or fewer neighbours are infected.
3. A cell infected with 3 neighbours infected has a 25% chance of recovering.
4. A cell infected with 4 neighbours infected has a 12.5% chance of recovering.
5. A cell recovered stays recovered.

**Neighbourhood** **<HEADING 2>**

After we had devised our ruleset and states we needed to come up with an algorithm which would enable us to check a cells(index of the 2D array) state. Having carried out some research on such algorithms we found the 2 most popular ones were the Morre model named after the american professor Edward F. Moore and the Von Neumann model named after the hungarian mathematician, physicist, and computer scientist John von Neumann. After weighing up the pros and cons of both algorithms we decided on the the Moore model. We chose the Moore model as we feel it gives a more accurate result, because it looks at all surrounding cells unlike the Von Neumann algorithm which only looks at the cells above, below and on either side.



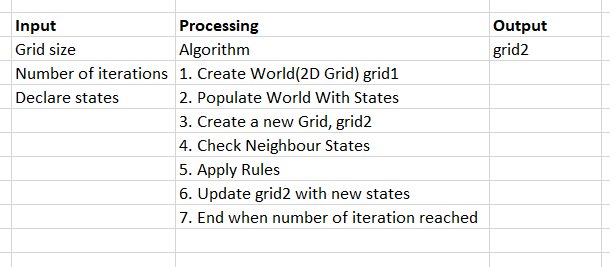
**[1] Moore neighbourhood model**

**Serial Program <HEADING 2>**

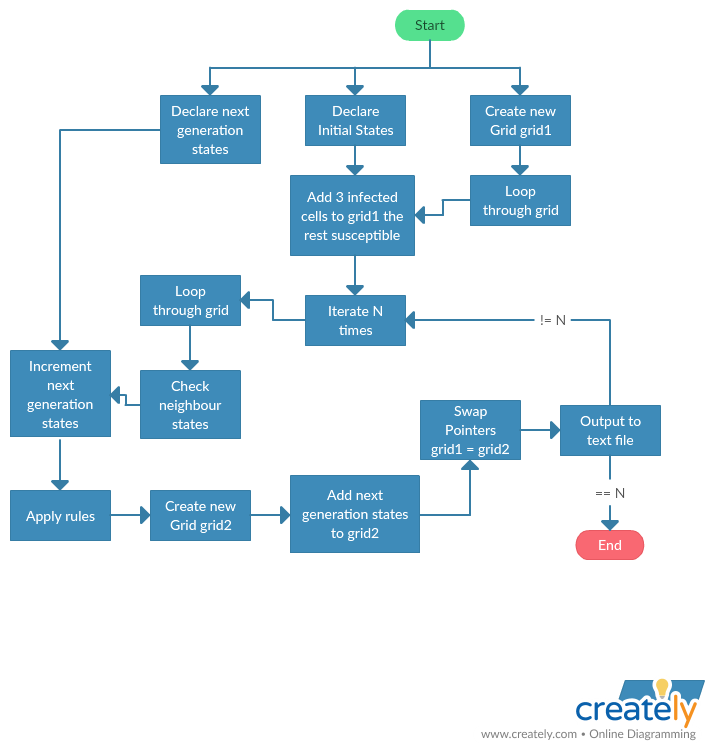
For the serial program we we firsted created a 2D array to store the cells(index of array) state. As we were using a large 2D array 1000x1000 we needed a form of memory management as otherwise we would have had issues with the memory of the stack. For the memory management we had a number of options calloc or malloc.

For our program we used malloc as we felt it better suited our program and what we were trying to achieve. Once we had the array created and malloc implemented we then needed to populate the array with states. At first we randomly allocated states to the array. But after some testing we chose to place three infected cells into the center and fill the rest of the array with susceptible states. After populating the array we the then needed to check the neighbour states. For this we used a nested for loop within a nested for loop. Within the inner nested for loop we had a number of if and else statements to count the number of each state types. Once we have counted the cells states we then applied the rules which we had devised which will add the new cells state to a second array. After this we needed to swap the pointers of the two arrays for this we needed to add a third array. We swapped the pointers by assigning the third array with the contents of the first, the first array with the contents of the second and finally the second array with the contents of the thirds. We did this with in a for loop which looped one thousand times. One of the last things we had to worry about with the program was memory leaks. We faced the possibility of memory leaks because of the fact we were using malloc. So in order to address this issue we needed to free the memory at the end of the program.

**Serial Program IPO Chart <HEADING 3>**



**Serial Program Flowchart <HEADING 3>**



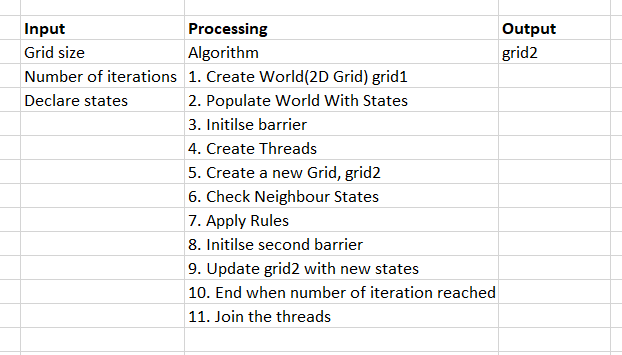
**Parallel Program <HEADING 2>**

A parallel program is a program which divides multiple tasks between multiple processors. To create a parallel program in a shared memory multiprocessor machine threads are used. A thread is an independent stream of instructions. This allows a process or program to be split into many threads that all execute concurrently which gives a huge speed increase. In our program we will be using POSIX (Portable Operating System Interface) threads aka pThreads to implement program parallelism.

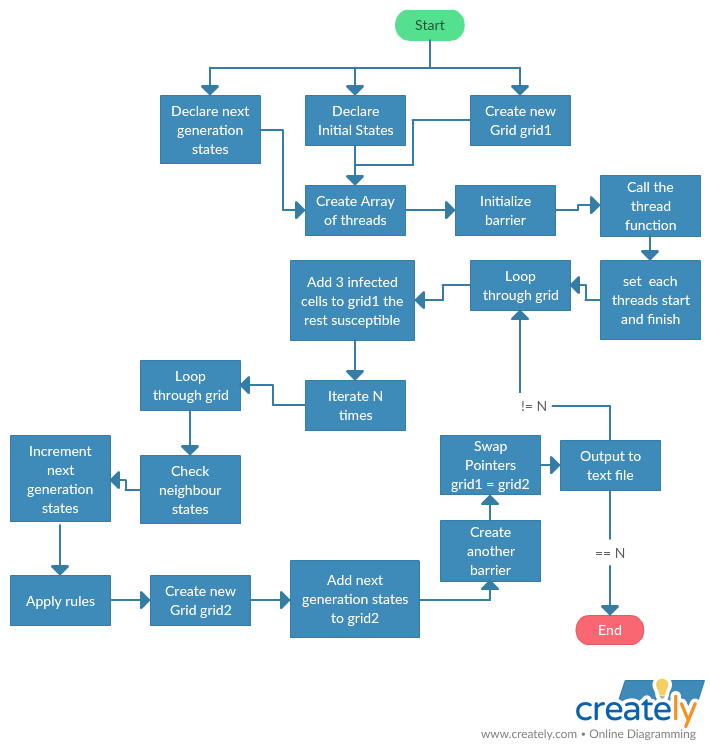
For our parallel program we had the same initial setup as our serial program. The parallel program obviously differs from the serial program as it is much faster. This is because the threads in the program are being shared among more than one processor core(The whole point of doing a parallel program). The way we created the parallel program was we created an array of threads (4 threads) after we had populated the array with states. For the thread synchronization we will be using technique known as Barriers. A barrier a point in our program where the threads will wait until all the threads have reached before continuing. This helps to avoid things like data corruption and unexpected program behaviour.

After implementing the barrier we created the threads within a for loop. Within the thread function we set the threads starting position and ending position. This insured that each thread did the same amount of work. Also within the thread function we used a second barrier which ensured that each thread got to execute without interruption. After the threads had finished executing they joined back in the main function.

**Parallel Program IPO Chart <HEADING 3>**

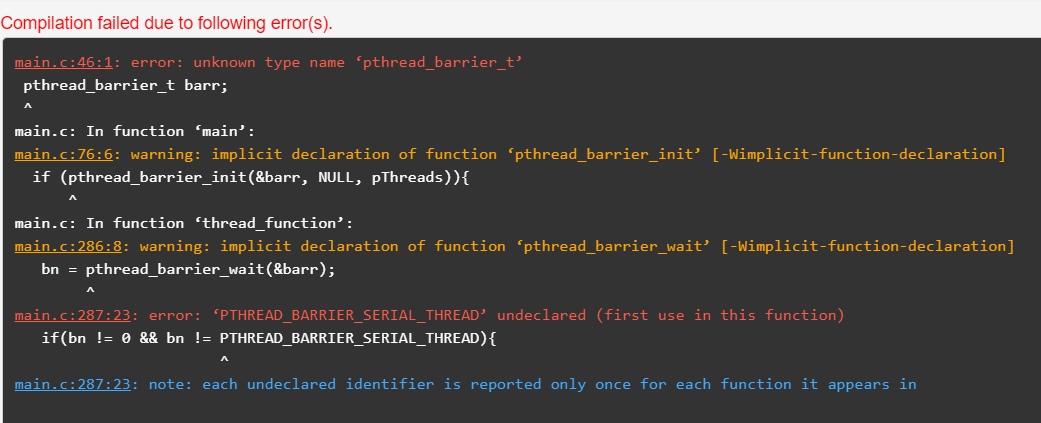


**Parallel Program Flowchart <HEADING 3>**



**Testing <HEADING 1>**

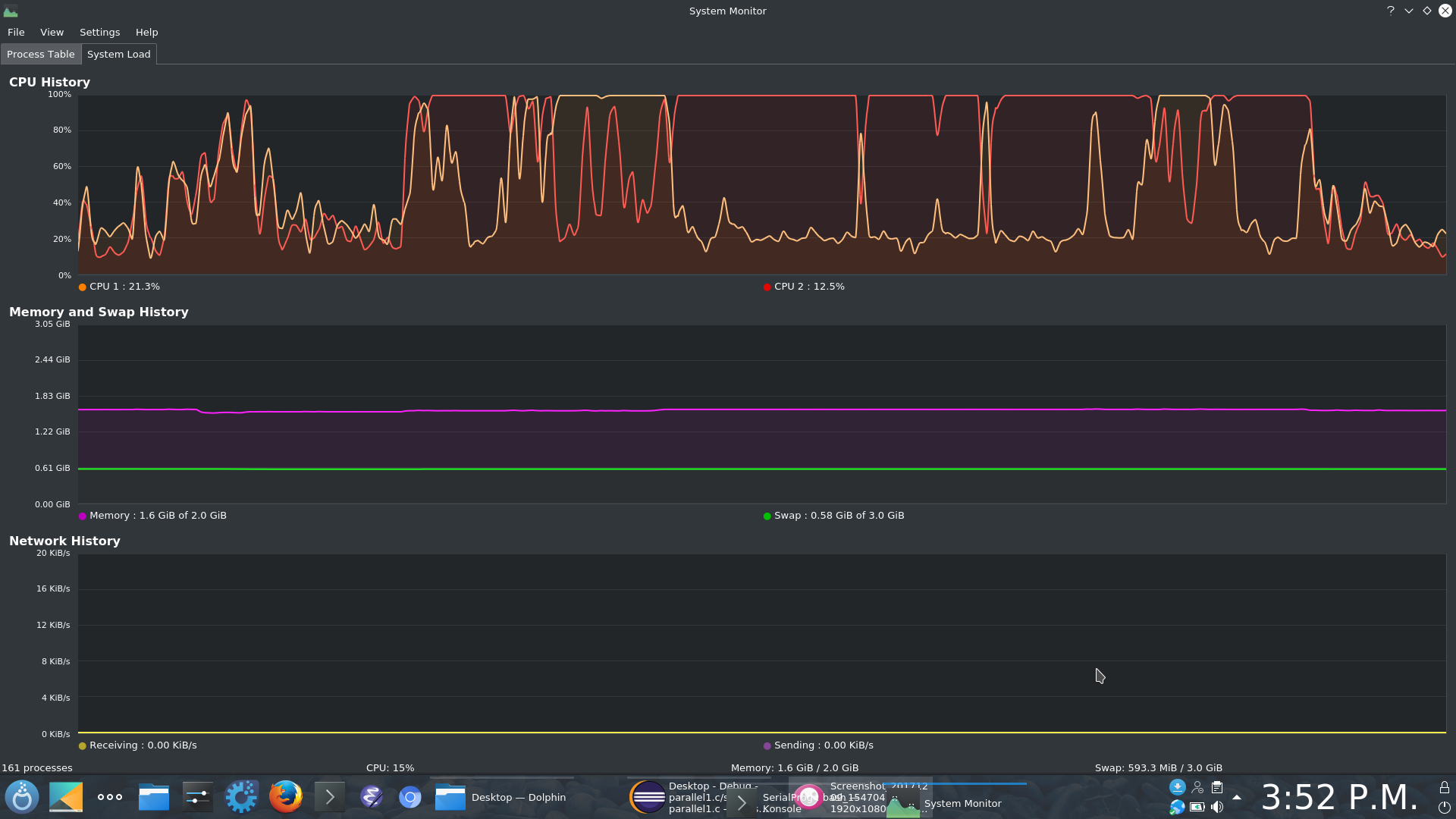
The test we have carried out on the program was we ran the program through two debuggers one built into the eclipse IDE and the other an online C program debugger [2]. For the Serial program we had no issues and it passed both test. Where as with the Parallel program we had found no issues in the program within the eclipse built in debugger. But in the same online debugger we used for the serial program we found an issue with pThread barrier which we could not rectify A screenshot of the errors we received is below.



**Results <HEADING 1>**

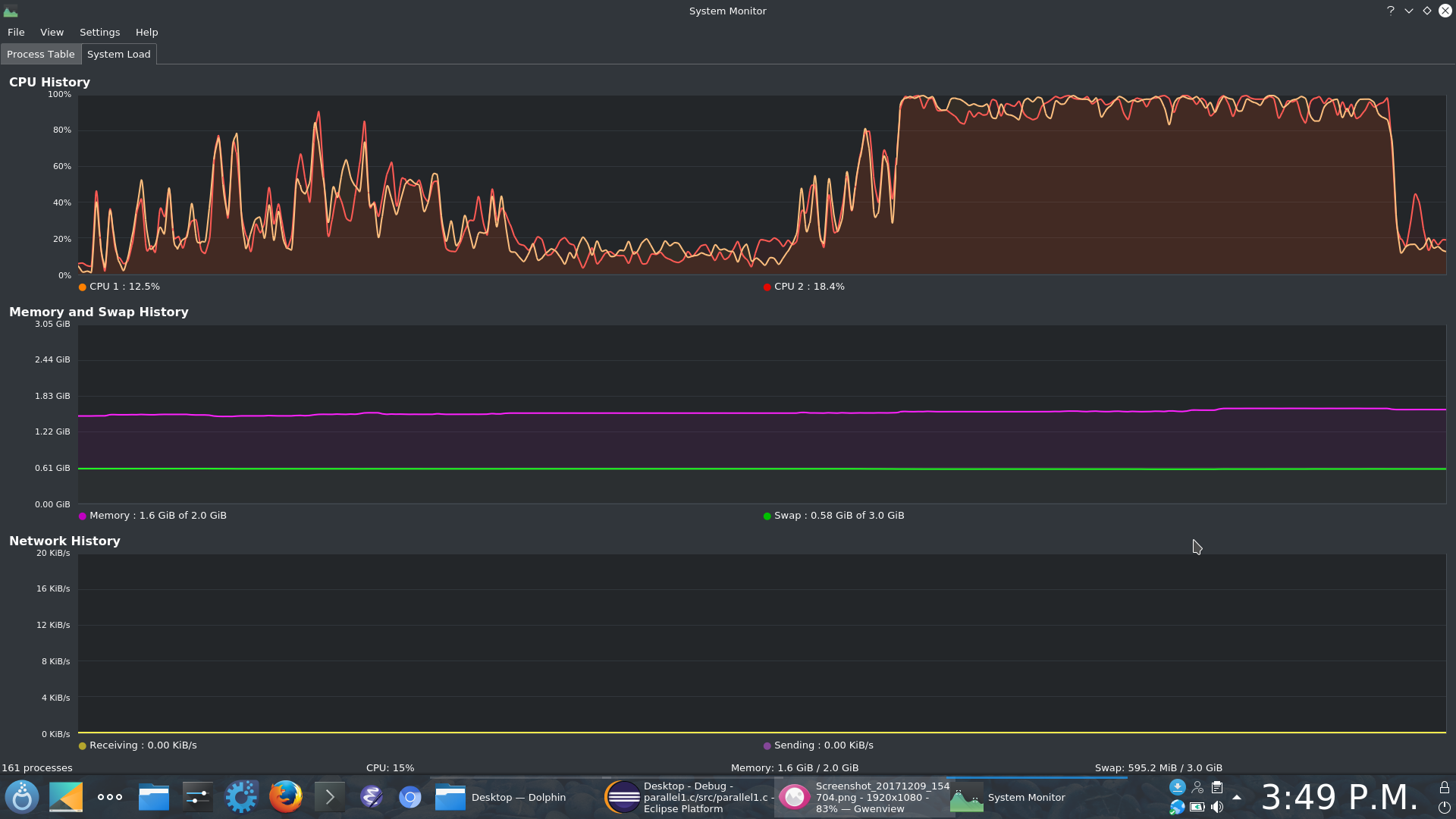
During our running of the program we were writing the outcome of the results to a file. We did this for the first iteration and for every 10th iteration thereafter. From these run throughs the average time to run the serial program was 52.46 seconds. Where as the parallel programs average time was 35.18 seconds. This is an increase of just under 33% not the 50% we were expecting. This could because of a number of factors such as writing to file sequentially which we did. As we were getting inconsistent results when writing to a text file using the parallel method.

**CPU Usage of Serial Program Running**



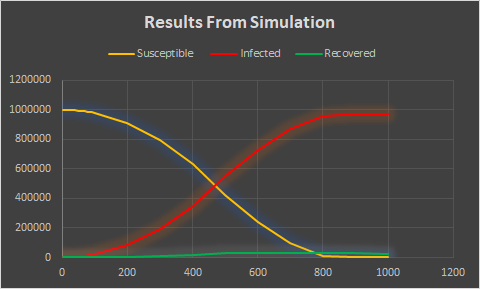
On the image above there are two coloured line a yellow and a red. Each of these lines represent a CPU core. As you can see in the image above one core is doing most of the work. Which is what we expected while the program was in execution.

**CPU Usage of Parallel Program Running**



On the image above there are two coloured line a yellow and a red. Each of these lines represent a CPU core. As you can see in the image above unlike the serial program the workload is shared more evenly between both CPU cores. Which is the results we expected while the program.

As we did not finish the visual simulation we had to come up with some other way to try visualise the data from the simulation. So we modified our program to write the total number of each state to file every iteration. From this we were able to create the following graph.



**Conclusion**

In our conclusion we are quite happy with the finished program. We feel we made a good attempt of the simulation using both a serial program and a parallel program. We were disappointed we didn’t finish the visual program to show a visualization of the simulation This was due to several factors time and the fact we spent a lot of time on getting the neighbours part of the serial program to function correctly. As we had a lot of issues with it no doing what we expected it to be doing. Once we had the neighbours issue fixed we were surprised we faced no real issue with the parallel program and the threads in particular. Which was new to us and something we plan on learning more about in the future. The one area of the program we feel we could have done better was on the rules we used. As these rules did not allow for infected cells to recover or die. Other than this we were very happy with the assignment and the learning outcomes we gained from completing the assignment such as memory management and thread synchronisation. These two areas we didn’t really understand before the assignment but now feel we understand them a lot better.

**Future Work**

If we had more time to do this project, we would make a third program, which would be used to simulate the running of the programs we created. This would work by the simulation program taking in the output from the serial and parallel programs and then outputting its own data. We would do this visually, using the graphics library of whatever programming language we decide to do the simulation in. This would be a good way to help users who are not really familiar with Cellular Automata to understand it easier and it always helps to understand things better when there is visual aid to help.

Another thing we would do is set up the program to produce different outcomes to be ready for simulation. We would do this by starting off with one cell infected, or a different number of cells infected. We would also change some of the other states in different cells to get different output from them. This in turn could help our program be used for more than it was first intended. This would also make our program more modular. This is very important when creating programs as it will be much easier to change some code in certain functions without having to alter the code much.

**Appendix <HEADING 1>**

**Serial Program Code <HEADING 2>**

/\*

============================================================================

Name : serialTestProject.c

Author : Derek McCarthy, Christopher Slattery, Joey Tierney, Jenny O'Connell

Version : 1.1

Copyright : Your copyright notice

Description : Cellular Automata model of the spread of the Ebola virus using the SIR model

============================================================================

\*/

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

/\*

--STATES--

0, EMPTY

1, SUSCEPTIBLE

2, INFECTED

3, RECOVERED

--RULES--

1. A cell in state 0 stays in state 0

2. A cell transitions from state 1 to state 2 if 2 or more neighbours are infected

3. A cell transitions from state 2 to state 3 if less than 2 neighbours are infected

4. A cell in state 3 stays in state 3

\*/

long ROWS = 1002;

long COLS = 1002;

long i=0, x=0, y=0, j=0;

int c=0;

const int INFECTED = 2;

const int RECOVERED = 3;

const int SUSCEPTIBLE = 1;

const int NUM\_NEIGHBOURS\_INFECTED = 2;

const int NUM\_ITERATIONS = 1000;

long int nextGenInf =0;

long int nextGenSus =0;

long int nextGenRec =0;

long int nextGenEmp =0;

long \*\*cell;

long \*\*nextGen;

long \*\*tempCell;

int counter = 1;

void checkNeighbours();

void initCells();

void checkNeighbours();

void outputToTextFile(int numDays);

void display();

void initCells() {

long count = 0;

long half = (ROWS/2);

cell = (long\*\*)malloc(ROWS \* sizeof(long\*));

nextGen = (long\*\*)malloc(ROWS \* sizeof(long\*));

tempCell = (long\*\*)malloc(ROWS \* sizeof(long\*));

if(!cell) {

fputs("Memory allocation failed. Run for your lives.\n", stderr);

exit(EXIT\_FAILURE);

}//end if

if(!nextGen) {

fputs("Memory allocation failed. Run for your lives.\n", stderr);

exit(EXIT\_FAILURE);

}//end if

if(!tempCell) {

fputs("Memory allocation failed. Run for your lives.\n", stderr);

exit(EXIT\_FAILURE);

}//end if

for (i = 0; i < ROWS; i++){

cell[i] = (long\*)malloc(COLS \* sizeof(long));

nextGen[i] = (long\*)malloc(COLS \* sizeof(long));

tempCell[i] = (long\*)malloc(COLS \* sizeof(long));

}//end for

for (i = 1;i < ROWS-1; i++){

for (x = 1; x < COLS-1; x++){

//random number between 0 and 3 (for states)

if (i >= half && x >= half-1 && count < 3){

cell[i][x] = 2;

count++;

}//end if

else{

cell[i][x] = 1;

}//end else

}//end inner for

}//end outer for

}//end init cells

void checkNeighbours(){

int random = 0;

for (i = 1;i < ROWS-1; i++){

for (x = 1; x < COLS-1; x++){

for(int ii = -1; ii <= 1; ii++) {

for(int j = -1; j <= 1; j++) {

if(cell[i+ii][x+j] == 1) {

nextGenSus++;

} else if(cell[i+ii][x+j] == 2) {

nextGenInf++;

} else if(cell[i+ii][x+j] == 3) {

nextGenRec++;

}// end else if

}// end inner

}

//--------------------Next Generation---------------------

if(cell[i][x]==SUSCEPTIBLE){

if(nextGenInf < INFECTED){

nextGen[i][x] = 1;

}//end if

else {

nextGen[i][x] = 2;

}//end else

}//end if

else if(cell[i][x]== INFECTED){

if(nextGenInf <= 2)nextGen[i][x] = 3;

else if(nextGenInf == 3){

random = rand() % 3;

if (random == 0)nextGen[i][x] = 3;

else nextGen[i][x] = 2;

}//end else if

else if(nextGenInf == 4){

random = rand() % 7;

if (random == 3)nextGen[i][x] = 3;

else nextGen[i][x] = 2;

}//end else if

else nextGen[i][x] = 2;

}//end else if

//else if(cell[i][x]== INFECTED && nextGenInf > NUM\_NEIGHBOURS\_INFECTED)nextGen[i][x] = 2;

else if(cell[i][x]== RECOVERED) nextGen[i][x] = 3;

else nextGen[i][x] = cell[i][x];

nextGenEmp=0;

nextGenSus=0;

nextGenInf=0;

nextGenRec=0;

}//end inner for

}//end outer for

}//end check neighbors

void outputToTextFile(int numDays){

FILE \*fp;

char buffer[64];

snprintf(buffer,sizeof(char) \* 64, "every10Iterations/Day%i.txt", numDays);

printf(buffer);

fp = fopen(buffer,"wb");

for (i = 1;i < ROWS; i++){

for (x = 1; x < COLS; x++){

//if statement so as not to print trailing zero

if(cell[i][x] != 0){

fprintf(fp,"%d",cell[i][x]);

}//end if

}//end for

}//end for

fclose(fp);

}//end outputToTextFile

void display() {

for (i = 1;i < COLS-1; i++){

for (x = 1; x < ROWS-1; x++){

printf("%ld ",cell[i][x]);

}//end inner for

printf("\n");

}//end outer for

}//end display

int main(){

srand(time(NULL));

initCells();

for(c=1;c<=NUM\_ITERATIONS;c++){

checkNeighbours();

if (c == 1){

outputToTextFile(c);

}

else if (c !=1 && counter == 10) {

outputToTextFile(c);

counter = 0;

}

printf("----------------------------");

printf("Cell state in round: %d",c);

printf("---------------------------\n");

counter++;

//display();

tempCell = cell;

cell = nextGen;

nextGen = tempCell;

}//end for

free(cell);

return 0;

}//end main

**Parallel Program <HEADING 2>**

/\*

============================================================================

Name : parallel.c

Author : Derek McCarthy, Christopher Slattery, Joseph Tierney, Jenny O'Connell

Version : 1.9

Copyright : Your copyright notice

Description : Cellular Automata model of the spread of the Ebola virus using the SIR model using pthreads

============================================================================

--STATES--

0, EMPTY

1, SUSCEPTIBLE

2, INFECTED

3, RECOVERED

--RULES--

1. A cell in state 0 stays in state 0

2. A cell transitions from state 1 to state 2 if 2 or more neighbours are infected

3. A cell transitions from state 2 to state 3 if less than 2 neighbours are infected

4. A cell in state 3 stays in state 3

\*/

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

long ROWS = 1002;

long COLS = 1002;

long i=0, x=0, y=0, j=0;

long \*\*cell;

long \*\*nextGen;

long \*\*tempCell;

int counter = 1;

/\*

\* pThreads Variables

\*/

//number of threads

unsigned int pThreads;

//array for thread ids

long \*thread\_ids;

//Barrier variable

pthread\_barrier\_t barr;

//prototypes

void initCells();

void outputToTextFile(int numDays);

void display();

void \*thread\_function(void \*t\_id);

/\*

\* Main function

\*/

int main(){

srand(time(NULL));

//number of threads

pThreads = 4;

initCells();

//create array of threads

pthread\_t thr[pThreads];

//allocate memory for the thread ids

thread\_ids = malloc(pThreads \* sizeof(pthread\_t));

if (thread\_ids == NULL){

printf("Problem in memory allocation of thread ID's array!\nProgram Exiting\n");

return -1;

}//end if

//init the barrier

if (pthread\_barrier\_init(&barr, NULL, pThreads)){

printf("Could not create barrier\n");

return -1;

}//end if

//create threads

for (i=0; i < pThreads; i++){

thread\_ids[i] = i;

if (pthread\_create(&thr[i], NULL, &thread\_function, (void \*)&thread\_ids[i])){

printf("Could not create thread");

return -1;

}//end if

}//end for

for (i = 0; i < pThreads; i++){

if (pthread\_join(thr[i], NULL)){

printf("Could not join thread %ld\n", i);

return -1;

}//end if

}//end for

free(cell);

//free(nextGen);

//tree(tempCell);

return 0;

}//end main

/\*

\* Function to populate 2D array

\*/

void initCells() {

//variables to get the center of the array

long count = 0;

long half = (ROWS/2);

//allocate memory for the 2D arrays

cell = (long\*\*)malloc(ROWS \* sizeof(long\*));

nextGen = (long\*\*)malloc(ROWS \* sizeof(long\*));

tempCell = (long\*\*)malloc(ROWS \* sizeof(long\*));

//if the allocation of memory fails display error message

if(!cell) {

fputs("Memory allocation failed. Run for your lives.\n", stderr);

exit(EXIT\_FAILURE);

}//end if

//if the allocation of memory fails display error message

if(!nextGen) {

fputs("Memory allocation failed. Run for your lives.\n", stderr);

exit(EXIT\_FAILURE);

}//end if

//if the allocation of memory fails display error message

if(!tempCell) {

fputs("Memory allocation failed. Run for your lives.\n", stderr);

exit(EXIT\_FAILURE);

}//end if

//loop through array to allocate memory

for (i = 0; i < ROWS; i++){

cell[i] = (long\*)malloc(COLS \* sizeof(long));

nextGen[i] = (long\*)malloc(COLS \* sizeof(long));

tempCell[i] = (long\*)malloc(COLS \* sizeof(long));

}//end for

//loop to populate array with states

for (i = 1;i < ROWS-1; i++){

for (x = 1; x < COLS-1; x++){

//random number between 0 and 3 (for states)

//get the center of the array

if (i >= half && x >= half-1 && count < 3){

cell[i][x] = 2;

count++;

}//end if

else{

cell[i][x] = 1;

}//end else

}//end inner for

}//end outer for

}//end function

/\*

\* Function to write to file

\*/

void outputToTextFile(int numDays){

FILE \*fp;

char buffer[64];

long count = 0;

snprintf(buffer,sizeof(char) \* 64, "testFiles/Day%i.txt", numDays);

printf(buffer);

fp = fopen(buffer,"wb");

for (i = 1;i < ROWS-1; i++){

for (x = 1; x < COLS-1; x++){

//if statement so as not to print trailing zero

if(cell[i][x] != 0){

fprintf(fp,"%ld",cell[i][x]);

}//end if

else{

break;

break;

break;

break;

}

count++;

}//end for

}//end for

fclose(fp);

//free(len);

}//end outputToTextFile

/\*

\* Function to display contents of array

\*/

void display() {

for (i = 1;i < COLS-1; i++){

for (x = 1; x < ROWS-1; x++){

printf("%ld ",cell[i][x]);

}//end inner for

printf("\n");

}//end outer for

}//end display

void \*thread\_function(void \*t\_id) {

int \*thread\_id=(int\*)t\_id;

//calculate the array bounds that each thread will process

long bound = ROWS / pThreads;

long myFirst = \*thread\_id \* bound;

long myLast = myFirst + bound;

long int nextGenInf =0;

long int nextGenSus =0;

long int nextGenRec =0;

const int INFECTED = 2;

const int RECOVERED = 3;

const int SUSCEPTIBLE = 1;

const long NUM\_ITERATIONS = 1000;

long i, x;

long long bn;

int q =0;

int random = 0;

// exclude externnal cells

if(\*thread\_id==0) myFirst++;

if(\*thread\_id==pThreads-1) myLast=ROWS-1;

//iteration loop

for (q=1; q<=NUM\_ITERATIONS; q++) {

//loop to traverse the array

for (i = myFirst;i < myLast; i++){

for (x = 1; x < COLS-1; x++){

//reset the next generation variables

nextGenSus=0;

nextGenInf=0;

nextGenRec=0;

/\*

\* Loop to check 3x3 grid of neighbors

\*/

for(int ii = -1; ii <= 1; ii++) {

for(int j = -1; j <= 1; j++) {

if(cell[i+ii][x+j] == 1) {

nextGenSus++;

} else if(cell[i+ii][x+j] == 2) {

nextGenInf++;

} else if(cell[i+ii][x+j] == 3) {

nextGenRec++;

}// end else if

}// end inner

}//end for

/\*

\* Apply the rules to the array

\*/

if(cell[i][x]==SUSCEPTIBLE){

if(nextGenInf < INFECTED){

nextGen[i][x] = 1;

}//end if

else {

nextGen[i][x] = 2;

}//end else

}//end if

else if(cell[i][x]== INFECTED){

if(nextGenInf <= 2)nextGen[i][x] = 3;

else if(nextGenInf == 3){

random = rand() % 3;

if (random == 0)nextGen[i][x] = 3;

else nextGen[i][x] = 2;

}//end else if

else if(nextGenInf == 4){

random = rand() % 7;

if (random == 3)nextGen[i][x] = 3;

else nextGen[i][x] = 2;

}//end else if

else nextGen[i][x] = 2;

}//end else if

//else if(cell[i][x]== INFECTED && nextGenInf > NUM\_NEIGHBOURS\_INFECTED)nextGen[i][x] = 2;

else if(cell[i][x]== RECOVERED) nextGen[i][x] = 3;

else nextGen[i][x] = cell[i][x];

//reset the next generation variables

nextGenSus=0;

nextGenInf=0;

nextGenRec=0;

}//end inner for

}//end outer

bn = pthread\_barrier\_wait(&barr);

if(bn != 0 && bn != PTHREAD\_BARRIER\_SERIAL\_THREAD){

printf("Could not wait on barrier\n");

exit(-1);

}//end if

//thread 0 is responsible to swap the pointers and write to file

if(bn == PTHREAD\_BARRIER\_SERIAL\_THREAD) {

printf("----------------------------");

printf("Cell state in round: %d",q);

printf("---------------------------\n");

//if statements to write the first iteration and every 10th after to file

if (q == 1){

outputToTextFile(q);

}//end if

else if (q !=1 && counter == 10) {

outputToTextFile(q);

counter = 0;

}//end else if

//display();

counter++;

//Swap pointers

tempCell = cell;

cell = nextGen;

nextGen = tempCell;

}//end if

//final barrier to make sure the array has been written to file

//and the pointers have been swapped

bn = pthread\_barrier\_wait(&barr);

if(bn != 0 && bn != PTHREAD\_BARRIER\_SERIAL\_THREAD){

printf("Could not wait on barrier\n");

exit(-1);

}//end if

}// End of 100 play rounds for loop

return 0;

}//Eend thread function

**References <HEADING 1>**

[1] <https://en.wikipedia.org/wiki/Moore_neighborhood>

[2] <https://www.onlinegdb.com/>

[3] <http://www.who.int/mediacentre/factsheets/fs103/en/>