****

**Project Title:** Simulating a Zombie Infection Using Cellular Automata in a Parallel Computing Environment

**Course:** Parallel Computing (COMP H3036)

**Submitted by:** [Your Full Name]

**Student ID:** [Your Student ID]

**Instructor:** Dr. Kevin Farrell

**Institution:** Technological University Dublin

**Submission Date:**

**Abstract:**  
This project models a Zombie Infection outbreak using Cellular Automata, implementing serial and parallel computing for the basic and latent infection scenarios. It simulates infection dynamics, analyzes performance speed-up, and evaluates population outcomes under varying conditions.

**Declaration of Academic Honesty:**  
I am aware of the University policy on plagiarism in assignments and examinations (3AS08). I understand that plagiarism, collusion, and copying are grave and serious offenses in the University, and I will accept the penalties that could be imposed if I engage in any such activity.  
This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. I declare that this material, which I now submit for assessment, is entirely of my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work.

**Flowchart for Serial Simulation**

1. **Start**
2. **Initialize the grid**: Populate the grid with all cells as SUSCEPTIBLE except one ZOMBIE in the center.
3. **Simulation Loop** (for each day):
   * For each cell in the grid:
     + If the cell is SUSCEPTIBLE:
       - Count the number of ZOMBIE neighbors.
       - Decide the state change based on probabilities (SUSCEPTIBLE → ZOMBIE or REMOVED).
     + Else: Copy the current state to the next grid.
   * Copy the nextGrid to currentGrid.
   * Every 10 days, save the grid state to a file.
4. **End of Simulation**: After completing the specified days, print completion and terminate.

### ****Flowchart for Parallel Simulation****

1. **Start**
2. **Initialize the grid**: Same as the serial version.
3. **Create threads**: Divide rows among threads.
4. **Thread Execution**:
   * For each thread:
     + Process assigned rows using the same decision logic as in the serial version.
     + Use a barrier to synchronize threads after processing rows for a day.
     + One thread updates the currentGrid from nextGrid.
5. **Repeat simulation until the last day.**
6. **Destroy threads and terminate.**

|  |  |
| --- | --- |
| **Pseudo-Code for Serial**  BEGIN  FUNCTION InitialiseWorld():  FOR i FROM 0 TO GRID\_SIZE-1:  FOR j FROM 0 TO GRID\_SIZE-1:  currentGrid[i][j] = SUSCEPTIBLE  currentGrid[GRID\_SIZE/2][GRID\_SIZE/2] = ZOMBIE  FUNCTION CountZombieNeighbours(x, y):  DEFINE dx = [-1, -1, -1, 0, 0, 1, 1, 1]  DEFINE dy = [-1, 0, 1, -1, 1, -1, 0, 1]  count = 0  FOR k FROM 0 TO 7:  nx = (x + dx[k] + GRID\_SIZE) % GRID\_SIZE  ny = (y + dy[k] + GRID\_SIZE) % GRID\_SIZE  IF currentGrid[nx][ny] == ZOMBIE:  count += 1  RETURN count  FUNCTION DecideState(x, y):  zombieNeighbours = CountZombieNeighbours(x, y)  randomValue = RANDOM(0, 1)  IF randomValue < P\_INFECT \* zombieNeighbours:  nextGrid[x][y] = ZOMBIE  ELSE IF randomValue < P\_DEATH:  nextGrid[x][y] = REMOVED  ELSE:  nextGrid[x][y] = SUSCEPTIBLE  FUNCTION CopyNextGridToCurrent():  FOR i FROM 0 TO GRID\_SIZE-1:  FOR j FROM 0 TO GRID\_SIZE-1:  currentGrid[i][j] = nextGrid[i][j]  FUNCTION OutputWorld(day):  OUTPUT currentGrid TO FILE "grid\_day\_" + day  MAIN:  InitialiseWorld()  FOR day FROM 0 TO MAX\_DAYS-1:  FOR i FROM 0 TO GRID\_SIZE-1:  FOR j FROM 0 TO GRID\_SIZE-1:  IF currentGrid[i][j] == SUSCEPTIBLE:  DecideState(i, j)  ELSE:  nextGrid[i][j] = currentGrid[i][j]  CopyNextGridToCurrent()  IF day MOD 10 == 0:  OutputWorld(day)  PRINT "Simulation completed"  END | **Pseudo-Code for Parallel (Pthreads)**  BEGIN  FUNCTION InitialiseWorld():  SAME AS SERIAL VERSION  FUNCTION CountZombieNeighbours(x, y):  SAME AS SERIAL VERSION  FUNCTION DecideState(x, y):  SAME AS SERIAL VERSION  FUNCTION OutputWorld(day):  SAME AS SERIAL VERSION  FUNCTION SimulateZombieInfectionParallel(data):  DEFINE startRow = data.startRow  DEFINE endRow = data.endRow  FOR day FROM 0 TO MAX\_DAYS-1:  FOR i FROM startRow TO endRow:  FOR j FROM 0 TO GRID\_SIZE-1:  IF currentGrid[i][j] == SUSCEPTIBLE:  DecideState(i, j)  ELSE:  nextGrid[i][j] = currentGrid[i][j]  WAIT FOR BARRIER  IF IS\_MAIN\_THREAD():  COPY nextGrid TO currentGrid  IF day MOD 10 == 0:  OutputWorld(day)  WAIT FOR BARRIER  MAIN:  InitialiseWorld()  DEFINE threads[THREADS]  DEFINE threadData[THREADS]  rowsPerThread = GRID\_SIZE / THREADS  FOR i FROM 0 TO THREADS-1:  threadData[i].startRow = i \* rowsPerThread  threadData[i].endRow = (i + 1) \* rowsPerThread - 1  CREATE THREAD threads[i] USING SimulateZombieInfectionParallel(threadData[i])  FOR i FROM 0 TO THREADS-1:  JOIN THREAD threads[i]  PRINT "Parallel simulation completed"  END |

**Source Code**

|  |  |
| --- | --- |
| **Serial Source Code**  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  #define GRID\_SIZE 1000  #define MAX\_DAYS 1000  #define SUSCEPTIBLE 0  #define ZOMBIE 1  #define REMOVED 2  #define P\_INFECT 0.3  #define P\_DEATH 0.1  int currentGrid[GRID\_SIZE][GRID\_SIZE];  int nextGrid[GRID\_SIZE][GRID\_SIZE];  // Initialize the grid with all cells as SUSCEPTIBLE and one ZOMBIE  void initialiseWorld() {  for (int i = 0; i < GRID\_SIZE; i++) {  for (int j = 0; j < GRID\_SIZE; j++) {  currentGrid[i][j] = SUSCEPTIBLE;  }  }  currentGrid[GRID\_SIZE / 2][GRID\_SIZE / 2] = ZOMBIE;  }  // Output the grid to a file for visualization  void outputWorld(int day) {  char filename[50];  sprintf(filename, "grid\_day\_%d.txt", day);  FILE \*file = fopen(filename, "w");  for (int i = 0; i < GRID\_SIZE; i++) {  for (int j = 0; j < GRID\_SIZE; j++) {  fprintf(file, "%d ", currentGrid[i][j]);  }  fprintf(file, "\n");  }  fclose(file);  }  // Count the number of neighboring ZOMBIE cells  int countZombieNeighbours(int x, int y) {  int dx[] = {-1, -1, -1, 0, 0, 1, 1, 1};  int dy[] = {-1, 0, 1, -1, 1, -1, 0, 1};  int count = 0;  for (int k = 0; k < 8; k++) {  int nx = (x + dx[k] + GRID\_SIZE) % GRID\_SIZE;  int ny = (y + dy[k] + GRID\_SIZE) % GRID\_SIZE;  if (currentGrid[nx][ny] == ZOMBIE) {  count++;  }  }  return count;  }  // Decide state change for SUSCEPTIBLE cells  void decide\_S\_to\_ZorR(int x, int y) {  int zombieNeighbors = countZombieNeighbours(x, y);  float randomValue = rand() / (float)RAND\_MAX;  if (randomValue < P\_INFECT \* zombieNeighbors) {  nextGrid[x][y] = ZOMBIE;  } else if (randomValue < P\_DEATH) {  nextGrid[x][y] = REMOVED;  } else {  nextGrid[x][y] = SUSCEPTIBLE;  }  }  // Copy the nextGrid into currentGrid  void copyNextGridToCurrent() {  for (int i = 0; i < GRID\_SIZE; i++) {  for (int j = 0; j < GRID\_SIZE; j++) {  currentGrid[i][j] = nextGrid[i][j];  }  }  }  // Simulate the zombie infection for multiple days  void simulateZombieInfection() {  for (int day = 0; day < MAX\_DAYS; day++) {  for (int i = 0; i < GRID\_SIZE; i++) {  for (int j = 0; j < GRID\_SIZE; j++) {  if (currentGrid[i][j] == SUSCEPTIBLE) {  decide\_S\_to\_ZorR(i, j);  } else {  nextGrid[i][j] = currentGrid[i][j];  }  }  }  copyNextGridToCurrent();  if (day % 10 == 0) {  outputWorld(day);  }  }  }  int main() {  srand(time(NULL));  initialiseWorld();  simulateZombieInfection();  printf("Simulation completed.\n");  return 0;  } | **Parallel Source Code (Pthreads)**  #include <stdio.h>  #include <stdlib.h>  #include <pthread.h>  #include <time.h>  #include <bits/pthreadtypes.h>  #define GRID\_SIZE 1000  #define MAX\_DAYS 1000  #define THREADS 8  #define SUSCEPTIBLE 0  #define ZOMBIE 1  #define REMOVED 2  #define P\_INFECT 0.3  #define P\_DEATH 0.1  int currentGrid[GRID\_SIZE][GRID\_SIZE];  int nextGrid[GRID\_SIZE][GRID\_SIZE];  pthread\_barrier\_t barrier;  typedef struct {  int startRow;  int endRow;  } ThreadData;  // Initialize the grid with all cells as SUSCEPTIBLE and one ZOMBIE  void initialiseWorld() {  for (int i = 0; i < GRID\_SIZE; i++) {  for (int j = 0; j < GRID\_SIZE; j++) {  currentGrid[i][j] = SUSCEPTIBLE;  }  }  currentGrid[GRID\_SIZE / 2][GRID\_SIZE / 2] = ZOMBIE;  }  // Output the grid to a file for visualization  void outputWorld(int day) {  char filename[50];  sprintf(filename, "grid\_day\_%d.txt", day);  FILE \*file = fopen(filename, "w");  for (int i = 0; i < GRID\_SIZE; i++) {  for (int j = 0; j < GRID\_SIZE; j++) {  fprintf(file, "%d ", currentGrid[i][j]);  }  fprintf(file, "\n");  }  fclose(file);  }  // Count the number of neighboring ZOMBIE cells  int countZombieNeighbours(int x, int y) {  int dx[] = {-1, -1, -1, 0, 0, 1, 1, 1};  int dy[] = {-1, 0, 1, -1, 1, -1, 0, 1};  int count = 0;  for (int k = 0; k < 8; k++) {  int nx = (x + dx[k] + GRID\_SIZE) % GRID\_SIZE;  int ny = (y + dy[k] + GRID\_SIZE) % GRID\_SIZE;  if (currentGrid[nx][ny] == ZOMBIE) {  count++;  }  }  return count;  }  // Decide state change for SUSCEPTIBLE cells  void decide\_S\_to\_ZorR(int x, int y) {  int zombieNeighbors = countZombieNeighbours(x, y);  float randomValue = rand() / (float)RAND\_MAX;  if (randomValue < P\_INFECT \* zombieNeighbors) {  nextGrid[x][y] = ZOMBIE;  } else if (randomValue < P\_DEATH) {  nextGrid[x][y] = REMOVED;  } else {  nextGrid[x][y] = SUSCEPTIBLE;  }  }  // Thread function to process rows for simulation  void \*simulateZombieInfectionParallel(void \*arg) {  ThreadData \*data = (ThreadData \*)arg;  for (int day = 0; day < MAX\_DAYS; day++) {  for (int i = data->startRow; i <= data->endRow; i++) {  for (int j = 0; j < GRID\_SIZE; j++) {  if (currentGrid[i][j] == SUSCEPTIBLE) {  decide\_S\_to\_ZorR(i, j);  } else {  nextGrid[i][j] = currentGrid[i][j];  }  }  }  pthread\_barrier\_wait(&barrier);  if (data->startRow == 0) { // Main thread updates the current grid  for (int i = 0; i < GRID\_SIZE; i++) {  for (int j = 0; j < GRID\_SIZE; j++) {  currentGrid[i][j] = nextGrid[i][j];  }  }  if (day % 10 == 0) {  outputWorld(day);  }  }  pthread\_barrier\_wait(&barrier);  }  return NULL;  }  int main() {  pthread\_t threads[THREADS];  ThreadData threadData[THREADS];  pthread\_barrier\_init(&barrier, NULL, THREADS);  initialiseWorld();  int rowsPerThread = GRID\_SIZE / THREADS;  for (int i = 0; i < THREADS; i++) {  threadData[i].startRow = i \* rowsPerThread;  threadData[i].endRow = (i == THREADS - 1) ? GRID\_SIZE - 1 : (i + 1) \* rowsPerThread - 1;  pthread\_create(&threads[i], NULL, simulateZombieInfectionParallel, &threadData[i]);  }  for (int i = 0; i < THREADS; i++) {  pthread\_join(threads[i], NULL);  }  pthread\_barrier\_destroy(&barrier);  printf("Parallel simulation completed.\n");  return 0;  } |

### ****Cellular Automata Model****

#### **States**

#### Each cell in the grid represents an individual, categorized into the following states:

1. **SUSCEPTIBLE (S):** Healthy individuals vulnerable to infection.
2. **ZOMBIE (Z):** Infected individuals actively spreading the infection.
3. **REMOVED (R):** Individuals who died naturally or were removed after infection.
4. **INFECTED (I) (Optional):** Transition state before becoming a zombie.

#### **Transition Rules**

The state of each cell evolves based on interactions with its neighbors according to these rules:

1. **SUSCEPTIBLE → ZOMBIE:**
   * Trigger: Neighboring ZOMBIE cells.
   * Probability: **P\_infect** (e.g., 30%).
2. **SUSCEPTIBLE → REMOVED:**
   * Trigger: Natural death.
   * Probability: **P\_death** (e.g., 10%).
3. **SUSCEPTIBLE → INFECTED → ZOMBIE (Optional Latent Model):**
   * Trigger: Infection followed by a delay (e.g., 3 steps).
   * Transition: **P\_infect** applies first, then a fixed time leads to ZOMBIE.
4. **ZOMBIE → REMOVED:**
   * Trigger: External factors or resource depletion.
   * Occurs after a fixed number of steps.

#### **Grid Representation**

* **Structure:** 2D grid (e.g., 1000x1000 cells).
* **Boundary Conditions:** Periodic (neighbors wrap around edges).
* **Time Steps:** Discrete iterations where all cells update synchronously.

#### **Model Justification**

* **States and Rules:** Capture realistic infection dynamics with flexibility for parameter tuning.
* **Probabilities:** Add variability for simulating different outbreak scenarios.
* **Grid Design:** Ensures scalability and avoids boundary artifacts.

**Speed-up Analysis:**

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | Parallel Real Time (s) | Serial Real Time (s) | Speed-up (Real Time) |
| 8 | 10.029 | 9.4 | 0.937281883 |
| 4 | 10.314 | 9.4 | 0.911382587 |
| 2 | 10.727 | 9.4 | 0.876293465 |

**Simulation Analysis for Serial**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Simulation Analysis for Parallel (Pthread)**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |