Steven Akamelu B00132063 14/12/23

Simulating the Spread of Wild Fires, and an Analysis of Potential Extinguishing Strategies using Cellular Automata in a Parallel Computing Environment:

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."

This is my .c files with code underneath with variables and functions and declarations

1. basicThread.c

```
// Steven Akamelu B00132063
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "vars_defs_functions.h"

// "Child" Thread Function
void *basicThread(void *param)
{



  long thread_no = (long) param;
  printf("in a thread %ld \n", thread_no); // prints the thread number
```

```
// Declares variables to represent the current day and grid
    int day=0;
    int row=0;
    int col=0;

//Calculates the start and end indices for the thread's segment of the grid based on
the total number of rows (ROWS) and the number of threads (NUMTHREADS).
    int segment_each_thread = ROWS / NUMTHREADS;
    int start = thread_no * segment_each_thread;
    int end = (thread_no + 1) * segment_each_thread;
    end = end-1;

//Prints the calculated start and end indices for debugging.
    printf("-------------\n");
    printf("start %i \n", start);
    printf("end %i \n", end);
    printf("-------------\n");



//Starts a loop representing each day in the simulation.
    for(day=1; day <= TOTAL_DAYS; day++){ // main day loop

            printf("looping new day...\n"); // Prints a message showing the start of a new
simulation day.



                // setting start and end positions in array
                for(row=start; row<end; row++){
                        for(col=0; col<COLS; col++){
                          // what is the current cell



                          switch(current[row][col].state){

                          case 'F':

                            //int num_Burning_Neighbours =
countBurningNeighbours_ClosedBoundaries(row, col);

                            decide_F_to_B(row, col, 4);
```

```
              break;


              case 'B':


              decide_B_to_X_VaryingDays(row, col);

              break;

             }//switch

     } //for
 } //for



     int i=0;
     int j=0;
     pthread_mutex_lock(&mutex); // Locks a mutex, updates the current grid with
the values from the future grid, and then unlocks the mutex to ensure thread safety.
      //Overwrite the current world with the future world
      printf("saving future to current..\n");
      for(i=start; i<end; i++){
       for(j=0; j<COLS; j++) {
        current[i][j] = future[i][j];
       }
      }


     pthread_mutex_unlock(&mutex);


} // end main day loop ---------------------
```

```
   return 0;
}
```

2. countBurningNeighbours_ClosedBoundaries.c

```
// Steven Akamelu B00132063
#include <stdio.h>
#include "vars_defs_functions.h"

// calculates the number of burning neighbors for a given cell in a grid-based
simulation.
// Defines a function that takes the row and column indices of a cell as parameters
and returns the count of burning neighbors.
int countBurningNeighbours_ClosedBoundaries(int row,
        int col){

  int i, j;

// If the debug level is greater than 1, it prints a message showing the position of the
cell.
  #if DEBUG_LEVEL > 1
  printf("Cell[%d][%d] is Fuel:\n", row, col);
  #endif

  // Declares and initializes a local variable to store the count of burning neighbors for
the given cell.
  int local_num_Burning_Neighbours = 0;

  //Check neighbours of the candidate cell
  // Checks if the candidate cell is an interior cell.
  //If true, it iterates over the neighboring cells and excludes the candidate cell itself.
  if(row > 0 && row < ROWS-1 && col > 0 && col < COLS-1){ // Candidate cell is an
interior cell

    for(i=row-1; i<=row+1; i++)
      for(j=col-1; j<=col+1; j++){

  //Don't include the candidate cell
  if(i != row || j != col){

    // Prints debug information about the neighboring cell positions.
```

```
      #if DEBUG_LEVEL > 1
  printf("  Neighbour[%d][%d]\n",(i+ROWS)%ROWS, (j+COLS)%COLS);
      #endif

  // If the neighboring cell is burning, it increments the count of burning neighbors.
  if(current[i][j].state == 'B')
    local_num_Burning_Neighbours++;

}
    }
}
else if(row > 0 && row < ROWS-1){
  // Candidate on Left OR Right edge, but not a corner
  // => Five nearest neighbours

  // TWO vertical Neighbours of the Left or Right edge:
  if(current[row-1][col].state == 'B')
    local_num_Burning_Neighbours++;

  if(current[row+1][col].state == 'B')
    local_num_Burning_Neighbours++;

  // Candidate is Left edge
  if(col == 0){
    // THREE diagonal neighbours of the Left edge:
    if(current[row-1][col+1].state == 'B')
local_num_Burning_Neighbours++;

    if(current[row+1][col+1].state == 'B')
local_num_Burning_Neighbours++;

    if(current[row][col+1].state == 'B')
local_num_Burning_Neighbours++;
  }
  else
    // Candidate is Right edge
    if(col == COLS-1){
// THREE diagonal neighbours of Right edge:
if(current[row-1][col-1].state == 'B')
  local_num_Burning_Neighbours++;

if(current[row+1][col-1].state == 'B')
  local_num_Burning_Neighbours++;
```

```
if(current[row][col-1].state == 'B')
 local_num_Burning_Neighbours++;
  }
}
else if(col > 0 && col < COLS-1 ){
 // Candidate on Top edge OR Bottom edge, but not a corner
 // => Five nearest neighbours
 if(current[row][col-1].state == 'B')
  local_num_Burning_Neighbours++;

 if(current[row][col+1].state == 'B')
  local_num_Burning_Neighbours++;

 // Candidate is Top edge
 if(row == 0){
  // THREE diagonal neighbours
  if(current[row+1][col-1].state == 'B')
local_num_Burning_Neighbours++;

  if(current[row+1][col+1].state == 'B')
local_num_Burning_Neighbours++;

  if(current[row+1][col].state == 'B')
local_num_Burning_Neighbours++;
 }
 else
  // Candidate is Bottom edge
  if(row == ROWS-1){
// THREE diagonal neighbours
if(current[row-1][col-1].state == 'B')
 local_num_Burning_Neighbours++;

if(current[row-1][col+1].state == 'B')
 local_num_Burning_Neighbours++;

if(current[row-1][col].state == 'B')
 local_num_Burning_Neighbours++;
  }
}
else if(row == 0){
 // Top Corner cell (we have accounted for Top edge cells already)
 // Vertical neighbour below:
```

```
 if(current[row+1][col].state == 'B')
  local_num_Burning_Neighbours++;

 if(col == 0){
  // Top-Left Corner: ONE diagonal neighbour & ONE right neighbour
  if(current[row+1][col+1].state == 'B')
local_num_Burning_Neighbours++;

  if(current[row][col+1].state == 'B')
local_num_Burning_Neighbours++;
 }
 else
  if(col == COLS-1){
//Top-Right Corner: ONE diagonal neighbour & ONE left neighbour
if(current[row+1][col-1].state == 'B')
 local_num_Burning_Neighbours++;

if(current[row][col-1].state == 'B')
 local_num_Burning_Neighbours++;
  }
}
else if(row == ROWS-1){
 // Bottom Corner cell (we have accounted for Bottom edge cells already)
 // Vertical neighbour below:
 if(current[row-1][col].state == 'B')
  local_num_Burning_Neighbours++;

 if(col == 0){
  // Bottom-Left Corner: ONE diagonal neighbour & ONE right neighbour
  if(current[row-1][col+1].state == 'B')
local_num_Burning_Neighbours++;

  if(current[row][col+1].state == 'B')
local_num_Burning_Neighbours++;
 }
 else
  if(col == COLS-1){
//Bottom-Right Corner: ONE diagonal neighbour & ONE left neighbour
if(current[row-11][col-1].state == 'B')
 local_num_Burning_Neighbours++;

if(current[row][col-1].state == 'B')
 local_num_Burning_Neighbours++;
```

```
      }
  }

  return local_num_Burning_Neighbours;

}
```

3. decide_B_to_X_VaryingDays.c

```
 // Steven Akamelu B00132063
// decide_B_to_X_VaryingDays.c


#include <stdio.h>
#include <stdlib.h>
#include "vars_defs_functions.h"

void decide_B_to_X_VaryingDays(int row,
          int col){

    if(current[row][col].counter_B_to_X == 0){

      //Change state to X (Burnt):
      future[row][col].state = 'X';

      numX++;
      numB--;

      #if DEBUG_LEVEL > 1
      printf("B -> X\n\n");
      #endif
    }
    else{
      //Decrement B->X counter
      (future[row][col].counter_B_to_X)--;

      #if DEBUG_LEVEL > 1
      printf("State stays as B\n\n");
      #endif
    }
}
```

4. decide_F_to_b.c

```
// Steven Akamelu B00132063
//decide_F_to_B.c

#include <stdio.h>
#include <stdlib.h>
#include "vars_defs_functions.h"

void decide_F_to_B(int row,
       int col,
       int num_Burning_Neighbours){

  float chance;

  switch(num_Burning_Neighbours){

  case 0:  // 0 Burning Neighbours
    #if DEBUG_LEVEL > 1
    printf("State stays as F\n\n");
    #endif

    break;

  case 1: // 1 Burning Neighbour
    // Generate a random "chance" between 0.0 and 1.0
    chance = (float)rand() / (float)RAND_MAX;

    #if DEBUG_LEVEL > 1
    printf("chance = %f\n", chance);
    printf("PROB_F_TO_B_1 = %f\n", PROB_F_TO_B_1);
    #endif

    if(chance <= PROB_F_TO_B_1){
      future[row][col].state = 'B'; // Future cell changes from Fuel to Burning

      #if DEBUG_LEVEL > 1
      printf("F -> B\n\n");
      #endif

      numB++;
      numF--;
    }
    else{
```

```
    #if DEBUG_LEVEL > 1
    printf("State stays as F\n\n");
    #endif
  }

  break;


case 2: // 2 Burning Neighbours
  // Generate a random "chance" between 0.0 and 1.0
  chance = (float)rand() / (float)RAND_MAX;

  #if DEBUG_LEVEL > 1
  printf("chance = %f\n", chance);
  printf("PROB_F_TO_B_2 = %f\n", PROB_F_TO_B_2);
  #endif

  if(chance <= PROB_F_TO_B_2){
    future[row][col].state = 'B';

    #if DEBUG_LEVEL > 1
    printf("F -> B\n\n");
    #endif

    numB++;
    numF--;
  }
  else{
    #if DEBUG_LEVEL > 1
    printf("State stays as F\n\n");
    #endif
  }

  break;


case 3: // 3 Burning Neighbours
  // Generate a random "chance" between 0.0 and 1.0
  chance = (float)rand() / (float)RAND_MAX;

  #if DEBUG_LEVEL > 1
  printf("chance = %f\n", chance);
  printf("PROB_F_TO_B_3 = %f\n", PROB_F_TO_B_3);
```

```
 #endif

 if(chance <= PROB_F_TO_B_3){
  future[row][col].state = 'B';

  #if DEBUG_LEVEL > 1
  printf("F -> B\n\n");
  #endif

  numB++;
  numF--;
 }
 else{
  #if DEBUG_LEVEL > 1
  printf("State stays as F\n\n");
  #endif
 }

 break;


case 4: // 4 Burning Neighbours
 // Generate a random "chance" between 0.0 and 1.0
 chance = (float)rand() / (float)RAND_MAX;

 #if DEBUG_LEVEL > 1
 printf("chance = %f\n", chance);
 printf("PROB_F_TO_B_4 = %f\n", PROB_F_TO_B_4);
 #endif

 if(chance <= PROB_F_TO_B_4){
  future[row][col].state = 'B';

  #if DEBUG_LEVEL > 1
  printf("F -> B\n\n");
  #endif

  numB++;
  numF--;
 }
 else{
  #if DEBUG_LEVEL > 1
  printf("State stays as F\n\n");
```

```
    #endif
  }

  break;


case 5: // 5 Burning Neighbours
  // Generate a random "chance" between 0.0 and 1.0
  chance = (float)rand() / (float)RAND_MAX;

  #if DEBUG_LEVEL > 1
  printf("chance = %f\n", chance);
  printf("PROB_F_TO_B_5 = %f\n", PROB_F_TO_B_5);
  #endif

  if(chance <= PROB_F_TO_B_5){
    future[row][col].state = 'B';

    #if DEBUG_LEVEL > 1
    printf("F -> B\n\n");
    #endif

    numB++;
    numF--;
  }
  else{
    #if DEBUG_LEVEL > 1
    printf("State stays as F\n\n");
    #endif
  }

  break;


case 6: // 6 Burning Neighbours
  // Generate a random "chance" between 0.0 and 1.0
  chance = (float)rand() / (float)RAND_MAX;

  #if DEBUG_LEVEL > 1
  printf("chance = %f\n", chance);
  printf("PROB_F_TO_B_6 = %f\n", PROB_F_TO_B_6);
  #endif
  if(chance <= PROB_F_TO_B_6){
    future[row][col].state = 'B';

    #if DEBUG_LEVEL > 1
    printf("F -> B\n\n");
    #endif

    numB++;
    numF--;
  }
  else{
    #if DEBUG_LEVEL > 1
    printf("State stays as F\n\n");
    #endif
  }

  break;


case 7: // 7 Burning Neighbours
  // Generate a random "chance" between 0.0 and 1.0
  chance = (float)rand() / (float)RAND_MAX;

  #if DEBUG_LEVEL > 1
  printf("chance = %f\n", chance);
  printf("PROB_F_TO_B_7 = %f\n", PROB_F_TO_B_7);
  #endif

  if(chance <= PROB_F_TO_B_7){
    future[row][col].state = 'B';

    #if DEBUG_LEVEL > 1
    printf("F -> B\n\n");
    #endif

    numB++;
    numF--;
  }
  else{
    #if DEBUG_LEVEL > 1
    printf("State stays as F\n\n");
    #endif
  }
```

```
    break;


  case 8: // 8 Burning Neighbours
    // Generate a random "chance" between 0.0 and 1.0
    chance = (float)rand() / (float)RAND_MAX;

    #if DEBUG_LEVEL > 1
    printf("chance = %f\n", chance);
    printf("PROB_F_TO_B_8 = %f\n", PROB_F_TO_B_8);
    #endif

    if(chance <= PROB_F_TO_B_8){
      future[row][col].state = 'B';

      #if DEBUG_LEVEL > 1
      printf("F -> B\n\n");
      #endif

      numB++;
      numF--;
    }
    else{
      #if DEBUG_LEVEL > 1
      printf("State stays as F\n\n");
      #endif
    }

    break;

  }


}
```

5. parallel.c

```
// Steven Akamelu B00132063
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
#include "vars_defs_functions.h"
#include "timer.h"

//Global Variables

//The quantity of F, B, O, X cz
unsigned long int numF; //unsigned means positive number
unsigned long int numB;
unsigned long int numO;
unsigned long int numX;



CELL **current; //shows its the current state
CELL **future; //shows its the future state


//used for thread synchronisation in a multithreaded environment
pthread_mutex_t mutex;
pthread_cond_t cond_var;

int main(){




  double start, finish, elapsed;

  GET_TIME(start); //records the start time

  // make some space for the current and future
  //also used to store memory in each or the rows and columns
  current = (CELL**)malloc(ROWS*sizeof(CELL*));
  future = (CELL**)malloc(ROWS*sizeof(CELL*));

  for(int i=0; i<ROWS; i++){
    current[i] = (CELL*)malloc(COLS*sizeof(CELL));
    future[i] = (CELL*)malloc(COLS*sizeof(CELL));
  }
```

```
 // Initial values for #F, #B, #O, #X have to be zero
 numF = 0; // f means fuel
 numB = 0; // b means burning
 numO = 0; // o means non-vegetation
 numX = 0; // x means burnt

 pthread_mutex_init(&mutex, NULL);
```

```
// init the world
int i,j =0;
 for(i=0; i<ROWS; i++){
     // printf("looping rows\n");
   for(j=0; j<COLS; j++){
       //  printf("looping col\n");
     current[i][j].state = 'F'; // assigns f to all cells then randoms values push it to
burning then burnt

     // assigns random values for the counter of burning to burnt and initializes the
future array with the same values as current
     current[i][j].counter_B_to_X = rand()%(MAX_DAYS_B_TO_X - MIN_DAYS_B_TO_X
+ 1) + MIN_DAYS_B_TO_X;

     future[i][j] = current[i][j];

     numF++; //
   }
 }
 // two random burning fires!
 current[0][0].state = 'B';
 current[9][9].state = 'B';

 future[0][0].state = 'B';
 future[9][9].state = 'B';



 for(int x=0; x< 10; x++){
 current[4][x].state = 'O';
 future[4][x].state = 'O';

 }

// Prints the total number of days the simulation will loop.
 printf("Total days to loop..%d \n", TOTAL_DAYS);
  // -------------- thread bit ----------------------------------
  long thread;
  pthread_t *thread_handler;
  thread_handler = malloc(NUMTHREADS*sizeof(pthread_t)); //handler space






  // ------------------------------------------------------------
  // Create child threads
  // creates child threads using pthread_create, and assigns them the basicThread
function to execute.
  // -----------------------------------------------------------
  for(thread = 0; thread < NUMTHREADS; thread++) {
   pthread_create(&thread_handler[thread], NULL, &basicThread, (void *) thread);
  }




  // ------------------------------------------------------------
  // Join the threads threads
  //Waits for all child threads to finish using pthread_join.
  // -----------------------------------------------------------
  for(thread = 0; thread < NUMTHREADS; thread++){
   pthread_join(thread_handler[thread], NULL);
  }
```

```
 // clean up
 // cleans up memory and destroys the mutex
 // prints the final state of the simulation grid.
 free(thread_handler);
 pthread_mutex_destroy(&mutex);
```

```
printf("Final output........\n");
for(int i=0; i<ROWS; i++){
    for(int j=0; j<COLS; j++){
      printf(" %c ", current[i][j].state);


    }

    printf("\n");

}
```

```
  // get the final time and subtract to get the total time
  GET_TIME(finish);
  elapsed = finish - start;
  printf("Total time = %lf\n\n", elapsed);
  return 0;
}
```

6. timer.h

```
// Steven Akamelu B00132063
/* File:     timer.h
 *
 * Purpose:  Define a macro that returns the number of seconds that
 *           have elapsed since some point in the past.  The timer
 *           should return times with microsecond accuracy.
 *
 * Note:     The argument passed to the GET_TIME macro should be
 *           a double, *not* a pointer to a double.
 *
 * Example:
 *    #include "timer.h"
 *    . . .
 *    double start, finish, elapsed;
 *    . . .
 *    GET_TIME(start);
 *    . . .
 *    Code to be timed
 *    . . .
 *    GET_TIME(finish);
 *    elapsed = finish - start;
 *    printf("The code to be timed took %e seconds\n", elapsed);
 *
 * IPP:  Section 3.6.1 (pp. 121 and ff.) and Section 6.1.2 (pp. 273 and ff.)
 */
#ifndef _TIMER_H_
#define _TIMER_H_

#include <sys/time.h>

/* The argument now should be a double (not a pointer to a double) */
#define GET_TIME(now) { \
   struct timeval t; \
   gettimeofday(&t, NULL); \
   now = t.tv_sec + t.tv_usec/1000000.0; \
}
```

```
#endif
```

7. vars_defs_functions.h

```
// Steven Akamelu B00132063
// Global Variables and Definitions
//  vars_and_defs.h

#include <pthread.h>
#ifndef VARS_AND_DEFS
#define VARS_AND_DEFS



// Defines constants for the number of rows and columns in the grid.
#define ROWS 10
#define COLS 10



//Cell Datatype-definition
typedef struct cell {

  char state; // state can be one of: F for Fuel, B for Burning, O for Not Fuel/Non Vegetation, X for Burnt

  int counter_B_to_X; // Counter used to determine when a Burning cell gets changed to Burnt

} CELL;



extern CELL **current;
extern CELL **future;

//Defines constants for debugging level, total simulation days, and the number of threads.
#define DEBUG_LEVEL 1
#define TOTAL_DAYS 20
#define NUMTHREADS 2

//Probabilities for transition
```

```
// From Fuel to Burning, with different probabilities for eight different cases.
#define PROB_F_TO_B_1 0.50
#define PROB_F_TO_B_2 0.6
#define PROB_F_TO_B_3 0.7
#define PROB_F_TO_B_4 0.75
#define PROB_F_TO_B_5 0.8
#define PROB_F_TO_B_6 0.85
#define PROB_F_TO_B_7 0.9
#define PROB_F_TO_B_8 1.0

// Defines constants for the minimum and maximum number of days it takes for a
Burning cell to transition to Burnt.
#define MIN_DAYS_B_TO_X 1
#define MAX_DAYS_B_TO_X 2





// for the locks
// Declares external variables for a mutex and a condition variable used for thread
synchronization.
extern pthread_mutex_t mutex;
extern pthread_cond_t cond_var;



//Global Variables.
// Stores counts of Fuel, Burning, Not Fuel, and Burnt cells.
extern unsigned long int numF;
extern unsigned long int numB;
extern unsigned long int numO;
extern unsigned long int numX;
```

```
//Function Prototypes
void *basicThread(void *param);
void initialiseWorld(); // initializing the simulation world or grid.

// takes three parameters: row, col, num_Burning_Neighbours
// represents the number of burning neighbors for that cell.
void decide_F_to_B(int row,
        int col,
        int num_Burning_Neighbours); //decide when fuel goes to burning



// takes two parameters: row and col representing the position of a cell in the grid.
void decide_B_to_X_VaryingDays(int row,
        int col); //decide when burning goes to nothing

int countBurningNeighbours_ClosedBoundaries(int row,
            int col); //count all the burning neighbours



         #endif
```

8. Makefile

```
# To make the code, run "make par" to make the parallel version
# Type "make clean" if you want to clean up before
# finally ./par to run it

#used to store functions and variables
#//to run this file i would wirte make par
#//to clean the file i would write make clean

CC = gcc
CFLAGS = -g -Wall

DEPS =  vars_defs_functions.h timer.h
```

```
OBJ =   parallel.o\
   basicThread.o\
   countBurningNeighbours_ClosedBoundaries.o\
   decide_F_to_B.o\
   decide_B_to_X_VaryingDays.o
```