```c
#include "common.h"

void game_of_life(struct Options *opt, int *current_grid, int *next_grid, int n, int m){
    int neighbours;
    int n_i[8], n_j[8];
    //loop is broken up and devided amongst threads to complete work using a specified scheduling method
during runtime.
    #pragma omp parallel for \
    default(none) shared(next_grid,current_grid,n,m,opt) private(neighbours, n_i, n_j) schedule(runtime)
    for(int i = 0; i < n; i++){
        for(int j = 0; j < m; j++){
            neighbours = 0;
            //collect neighbouring cell's indices.
            n_i[0] = i - 1; n_j[0] = j - 1;
            n_i[1] = i - 1; n_j[1] = j;
            n_i[2] = i - 1; n_j[2] = j + 1;
            n_i[3] = i;     n_j[3] = j + 1;
            n_i[4] = i + 1; n_j[4] = j + 1;
            n_i[5] = i + 1; n_j[5] = j;
            n_i[6] = i + 1; n_j[6] = j - 1;
            n_i[7] = i;     n_j[7] = j - 1;


            // count the number of neighbours, clockwise around the current cell.
            if(n_i[0] >= 0 && n_j[0] >= 0 && current_grid[n_i[0] * m + n_j[0]] == ALIVE) neighbours++;
            if(n_i[1] >= 0 && current_grid[n_i[1] * m + n_j[1]] == ALIVE) neighbours++;
            if(n_i[2] >= 0 && n_j[2] < m && current_grid[n_i[2] * m + n_j[2]] == ALIVE) neighbours++;
            if(n_j[3] < m && current_grid[n_i[3] * m + n_j[3]] == ALIVE) neighbours++;
            if(n_i[4] < n && n_j[4] < m && current_grid[n_i[4] * m + n_j[4]] == ALIVE) neighbours++;
            if(n_i[5] < n && current_grid[n_i[5] * m + n_j[5]] == ALIVE) neighbours++;
            if(n_i[6] < n && n_j[6] >= 0 && current_grid[n_i[6] * m + n_j[6]] == ALIVE) neighbours++;
            if(n_j[7] >= 0 && current_grid[n_i[7] * m + n_j[7]] == ALIVE) neighbours++;

            //depending on the number of neighbours, decide if the current cell should live or die
            if(current_grid[i*m + j] == ALIVE && (neighbours == 2 || neighbours == 3)){
                next_grid[i*m + j] = ALIVE;
            } else if(current_grid[i*m + j] == DEAD && neighbours == 3){
                next_grid[i*m + j] = ALIVE;
            }else{
                next_grid[i*m + j] = DEAD;
            }
        }
    }
}

void game_of_life_stats(struct Options *opt, int step, int *current_grid){
    unsigned long long num_in_state[NUMSTATES];
    int m = opt->m, n = opt->n;
    for(int i = 0; i < NUMSTATES; i++) num_in_state[i] = 0;

    //each thread counts cells in each state for assigned part of the loop and does a reduction
    //into the master thread by doing a sum operation.
    #pragma omp parallel for reduction(+: num_in_state) schedule(runtime)
    for(int loop = 0; loop < n*m; loop++){
            num_in_state[current_grid[loop]]++;
    }

    double frac, ntot = opt->m*opt->n;
    FILE *fptr;
    if (step == 0) {
        fptr = fopen(opt->statsfile, "w");
    }
    else {
        fptr = fopen(opt->statsfile, "a");
    }
    fprintf(fptr, "step %d : ", step);
    for(int i = 0; i < NUMSTATES; i++) {
        frac = (double)num_in_state[i]/ntot;
        fprintf(fptr, "Frac in state %d = %f,\t", i, frac);
    }
    fprintf(fptr, " \n");
    fclose(fptr);
}



int main(int argc, char **argv)
{
    struct Options *opt = (struct Options *) malloc(sizeof(struct Options));
    getinput(argc, argv, opt);
    int n = opt->n, m = opt->m, nsteps = opt->nsteps;
    int *grid = (int *) malloc(sizeof(int) * n * m);
    int *updated_grid = (int *) malloc(sizeof(int) * n * m);
    if(!grid || !updated_grid){
        printf("Error while allocating memory.\n");
        return -1;
    }
    int current_step = 0;
    int *tmp = NULL;
    generate_IC(opt->iictype, grid, n, m);
    struct timeval start, steptime;

    start = init_time();
    while(current_step != nsteps){
        steptime = init_time();
        visualise(opt->ivisualisetype, current_step, grid, n, m);
        game_of_life_stats(opt, current_step, grid);
        game_of_life(opt, grid, updated_grid, n, m);
```

```
            // swap current and updated grid
            tmp = grid;
            grid = updated_grid;
            updated_grid = tmp;
            current_step++;
            get_elapsed_time(steptime);
        }
        printf("Finnished GOL\n");
        get_elapsed_time(start);
        free(grid);
        free(updated_grid);
        free(opt);
        return 0;
}
```