/\*Amirhossein Jafarzadeh Ghazi

Student number: 100745718

HPC assignment:

This code is for zombie land problem

considering interactions between individuals and

partitioning the rectangular field in order to parallelization\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include<time.h>

#include <sys/time.h>

#define M 10000

#define N 10000

int n\_s=100, n\_z=100, n\_f\_s=100, n\_f\_z=100; /\*fixing the number of

zombies and susceptible individuals\*/

#define default 1000000

void initialization(int n\_s, int n\_z);

void assign\_dimensions(int index, int i);

void assign\_state(int n\_s, int n\_z);

void movement(int index, int i, int n\_s, int n\_z);

void print\_individuals(int n\_s, int n\_z);

void interaction(int n\_s, int n\_z);

void partition(int i, int n\_thread, int n\_s, int n\_z);

typedef struct PTnode {

int i\_dim;

int j\_dim;

char state;

int index;

int partition;

char position;

} PTnode;

PTnode P[default];

int main() {

srand(time(NULL));

struct timeval stop, start;

int n\_thread=4;/\*fixing the number of threads, but it can be easily changed it must be 1,2, or 4\*/

int i,j,r,k,t;

int T\_max=200;/\*Initialization for number of movements\*/

initialization(n\_s, n\_z);

for(i=0;i<n\_s+n\_z;i++)

{

partition(i, n\_thread, n\_s, n\_z);

}

for(i=0;i<n\_s+n\_z;i++)

{

assign\_dimensions(P[i].index,i);

}

assign\_state(n\_s,n\_z);

for(int q=0;q<20;q++){/\*repeating movement loop for different increasing number of maximum movements, 20 times\*/

gettimeofday(&start, NULL);

for(t=0;t<T\_max;t++) /\*We move individuals T times in order of partition, from partition one to four depending on number of threads\*/

{

for(i=0;i<n\_s+n\_z;i++)

{

/\*/////////Moving interior individuals/////////////\*/

/\*first if finds individuals in the first partition and moves them\*/

if(P[i].partition==1 && P[i].position=='I' && (n\_thread==1 || n\_thread==2 || n\_thread==4))

{

movement(P[i].index, i, n\_s, n\_z);

partition(i, n\_thread , n\_s, n\_z);

}

/\*second if finds individuals in the second partition and moves them\*/

if(P[i].partition==2 && P[i].position=='I' && (n\_thread==2 || n\_thread==4))

{

movement(P[i].index, i, n\_s, n\_z);

partition(i, n\_thread , n\_s, n\_z);

}

/\*third if finds individuals in the third partition and moves them\*/

if(P[i].partition==3 && P[i].position=='I' && n\_thread==4)

{

movement(P[i].index, i, n\_s, n\_z);

partition(i, n\_thread , n\_s, n\_z);

}

/\*fourth if finds individuals in the fourth partition and moves them\*/

if(P[i].partition==4 && P[i].position=='I' && n\_thread==4)

{

movement(P[i].index, i, n\_s, n\_z);

partition(i, n\_thread , n\_s, n\_z);

}

/\*///////////Moving top boundary individuals///////////\*/

/\*first if finds individuals in the first partition and moves them\*/

if(P[i].partition==1 && P[i].position=='T' && (n\_thread==1 || n\_thread==2 || n\_thread==4))

{

movement(P[i].index, i, n\_s, n\_z);

partition(i, n\_thread , n\_s, n\_z);

}

/\*second if finds individuals in the second partition and moves them\*/

if(P[i].partition==2 && P[i].position=='T' && (n\_thread==2 || n\_thread==4))

{

movement(P[i].index, i, n\_s, n\_z);

partition(i, n\_thread , n\_s, n\_z);

}

/\*third if finds individuals in the third partition and moves them\*/

if(P[i].partition==3 && P[i].position=='T' && n\_thread==4)

{

movement(P[i].index, i, n\_s, n\_z);

partition(i, n\_thread , n\_s, n\_z);

}

/\*fourth if finds individuals in the fourth partition and moves them\*/

if(P[i].partition==4 && P[i].position=='T' && n\_thread==4)

{

movement(P[i].index, i, n\_s, n\_z);

partition(i, n\_thread , n\_s, n\_z);

}

/\*///////Moving bottom boundary individuals/////////\*/

/\*first if finds individuals in the first partition and moves them\*/

if(P[i].partition==1 && P[i].position=='B' && (n\_thread==1 || n\_thread==2 || n\_thread==4))

{

movement(P[i].index, i, n\_s, n\_z);

partition(i, n\_thread , n\_s, n\_z);

}

/\*second if finds individuals in the second partition and moves them\*/

if(P[i].partition==2 && P[i].position=='B' && (n\_thread==2 || n\_thread==4))

{

movement(P[i].index, i, n\_s, n\_z);

partition(i, n\_thread , n\_s, n\_z);

}

/\*third if finds individuals in the third partition and moves them\*/

if(P[i].partition==3 && P[i].position=='B' && n\_thread==4)

{

movement(P[i].index, i, n\_s, n\_z);

partition(i, n\_thread , n\_s, n\_z);

}

/\*fourth if finds individuals in the fourth partition and moves them\*/

if(P[i].partition==4 && P[i].position=='B' && n\_thread==4)

{

movement(P[i].index, i, n\_s, n\_z);

partition(i, n\_thread , n\_s, n\_z);

}

}

interaction(n\_s, n\_z);/\*after each step we should consider interactions between individuals\*/

}

gettimeofday(&stop, NULL);

printf("\n%lu micro seconds for %d Movements\n", (stop.tv\_sec - start.tv\_sec) \* 1000000 + stop.tv\_usec - start.tv\_usec, T\_max);

T\_max+=200; /\*increasing number of movements for the next test\*/

}

for(i=0;i<n\_s+n\_z;i++)

{

assign\_dimensions(P[i].index,i);

}

return 0;

}

void initialization(int n\_s, int n\_z){

int i, j, k, r, l;

P[0].index=(rand()%(M\*N))+1;

for(i=1;i<n\_s+n\_z;i++)

{

int s=0;

r=(rand()%(M\*N-i))+1;

for(j=0;j<i;j++)

{

if(r<P[j].index)

{s=1;

break;}

else

r=r+1;

}

if(s==1)

{for(k=i;k>j;k--)

P[k].index=P[k-1].index;}

P[j].index=r;

}

}

void movement(int index, int i, int n\_s, int n\_z){

int m, s=0, L=1, R=1, D=1, U=1, v, L\_s=0, R\_s=0, U\_s=0, D\_s=0;

while(L\_s==0 || R\_s==0 || U\_s==0 || D\_s==0)/\*If there is a place to go for individual number i, it must move and cannot stay. This movement is randomly\*/

{

m=rand()%4; /\*choose one direction(left, right, up, or down) randomly\*/

if(m==0)/\*move to left\*/

{

L\_s=1;

if(index%N!=1)

{

for(v=0;v<n\_s+n\_z;v++)

{

if(P[v].index==index-1)/\*check if the left lattice site is empty\*/

{

L=0;

break;

}

}

if(L==1)

{

P[i].index=index-1;/\*move\*/

break;

}

}

}

if(m==1)/\*move to right\*/

{

R\_s=1;

if(index%N!=0)

{

for(v=0;v<n\_s+n\_z;v++)

{

if(P[v].index==index+1)/\*check if the right lattice site is empty\*/

{

R=0;

break;

}

}

if(R==1)

{

P[i].index=index+1;/\*move\*/

break;

}

}

}

if(m==2)/\*move upwards\*/

{

U\_s=1;

if(index/N!=0 && index!=N)

{

for(v=0;v<n\_s+n\_z;v++)

{

if(P[v].index==index-N)/\*check if the upper lattice site is empty\*/

{

U=0;

break;

}

}

if(U==1)

{

P[i].index=index-N;/\*move\*/

break;

}

}

}

if(m==3)/\*move downwards\*/

{

D\_s=1;

if(index/N!=M-1 && index!=M\*N)

{

for(v=0;v<n\_s+n\_z;v++)

{

if(P[v].index==index+N)/\*check if the lower lattice site is empty\*/

{

D=0;

break;

}

}

if(D==1)

{

P[i].index=index+N;/\*move\*/

break;

}

}

}

}

}

void assign\_dimensions(int index, int i){

if(index%N!=0)

{

P[i].i\_dim=1+(index/N);

P[i].j\_dim=index%N;

}

else

{

P[i].i\_dim=index/N;

P[i].j\_dim=N;

}

}

void assign\_state(int n\_s, int n\_z){

int i, k;

for(i=0;i<n\_s+n\_z;i++)

P[i].state='U';

for(i=0;i<n\_s;i++)

{

k=rand()%(n\_s+n\_z);

if(P[k].state=='U')

P[k].state='S';

else

i--;

}

for(i=0;i<n\_s+n\_z;i++)

{

if(P[i].state=='U')

P[i].state='Z';

}

}

void interaction(int n\_s, int n\_z){

int k ,v;

for(k=0;k<n\_s+n\_z;k++)

{

for(v=0;v<n\_s+n\_z;v++)

{

if(P[k].index%N!=0 && P[k].state=='S')

{

if(P[k].index+1==P[v].index)/\*check if the right lattice site is occupied\*/

{

if(P[v].state=='Z')/\*check if the right lattice site is zombie\*/

{

P[k].state='Z';

n\_f\_z++;

n\_f\_s--;

break;

}

}

}

if(P[k].index%N!=1 && P[k].state=='S')

{

if(P[k].index-1==P[v].index)/\*check if the left lattice site is occupied\*/

{

if(P[v].state=='Z')/\*check if the left lattice site is zombie\*/

{

P[k].state='Z';

n\_f\_z++;

n\_f\_s--;

break;

}

}

}

if(P[k].index/N!=M-1 && P[k].index!=M\*N && P[k].state=='S')

{

if(P[k].index+N==P[v].index)/\*check if the lower lattice site is occupied\*/

{

if(P[v].state=='Z')/\*check if the lower lattice site is zombie\*/

{

P[k].state='Z';

n\_f\_z++;

n\_f\_s--;

break;

}

}

}

if(P[k].index/N!=0 && P[k].index!=N && P[k].state=='S')

{

if(P[k].index-N==P[v].index)/\*check if the upper lattice site is occupied\*/

{

if(P[v].state=='Z')/\*check if the upper lattice site is zombie\*/

{

P[k].state='Z';

n\_f\_z++;

n\_f\_s--;

break;

}

}

}

}

}

}

void partition(int i, int threads, int n\_s, int n\_z){

int t;

P[i].position='I';

/\*partitioning for 1 number of threads\*/

if(threads==1)

P[i].partition=1;

/\*partitioning for 2 number of threads\*/

if(threads==2 && M%2==0)

{

if(M<2)

{

printf("Your rectangular field is too small to use this number of threads\n");

}

if(P[i].index>0 && P[i].index<=M\*N/2 && M>1)

P[i].partition=1;

if(P[i].index>M\*N/2-N && P[i].index<=M\*N/2 && M>1)

P[i].position='B';

if(P[i].index>M\*N/2 && P[i].index<=M\*N/2+N && M>1)

P[i].position='T';

if(P[i].index>M\*N/2 && P[i].index<=M\*N && M>1)

P[i].partition=2;

}

if(threads==2 && M%2!=0)

{

if(M<2)

{

printf("Your rectangular field is too small to use this number of threads\n");

}

if(P[i].index>0 && P[i].index<=(1+M/2)\*N && M>1)

P[i].partition=1;

if(P[i].index>((1+M/2)\*N)-N && P[i].index<=(1+M/2)\*N && M>1)

P[i].position='B';

if(P[i].index>(1+M/2)\*N && P[i].index<=((1+M/2)\*N)+N && M>1)

P[i].position='T';

if(P[i].index>(1+M/2)\*N && P[i].index<=M\*N && M>1)

P[i].partition=2;

}

/\*partitioning for 4 number of threads\*/

if(threads==4 && M%4==0)

{

if(M<4)

{

printf("Your rectangular field is too small to use this number of threads\n");

}

if(P[i].index>0 && P[i].index<=M\*N/4 && M>3)

P[i].partition=1;

if(P[i].index>(M\*N/4)-N && P[i].index<=M\*N/4 && M>1)

P[i].position='B';

if(P[i].index>M\*N/4 && P[i].index<=(M\*N/4)+N && M>1)

P[i].position='T';

if(P[i].index>M\*N/4 && P[i].index<=2\*(M\*N/4) && M>3)

P[i].partition=2;

if(P[i].index>(2\*(M\*N/4))-N && P[i].index<=2\*(M\*N/4) && M>1)

P[i].position='B';

if(P[i].index>2\*(M\*N/4) && P[i].index<=(2\*(M\*N/4))+N && M>1)

P[i].position='T';

if(P[i].index>2\*(M\*N/4) && P[i].index<=3\*(M\*N/4) && M>3)

P[i].partition=3;

if(P[i].index>(3\*(M\*N/4))-N && P[i].index<=3\*(M\*N/4) && M>1)

P[i].position='B';

if(P[i].index>3\*(M\*N/4) && P[i].index<=(3\*(M\*N/4))+N && M>1)

P[i].position='T';

if(P[i].index>3\*(M\*N/4) && P[i].index<=M\*N && M>3)

P[i].partition=4;

}

if(threads==4 && M%4==1)

{

if(M<4)

{

printf("Your rectangular field is too small to use this number of threads\n");

}

if(P[i].index>0 && P[i].index<=(1+M/4)\*N && M>3)

P[i].partition=1;

if(P[i].index>((1+M/4)\*N)-N && P[i].index<=(1+M/4)\*N && M>1)

P[i].position='B';

if(P[i].index>(1+M/4)\*N && P[i].index<=((1+M/4)\*N)+N && M>1)

P[i].position='T';

if(P[i].index>(1+M/4)\*N && P[i].index<=(1+M/4)\*N+(M/4)\*N && M>3)

P[i].partition=2;

if(P[i].index>((1+M/4)\*N+(M/4)\*N)-N && P[i].index<=(1+M/4)\*N+(M/4)\*N && M>1)

P[i].position='B';

if(P[i].index>(1+M/4)\*N+(M/4)\*N && P[i].index<=((1+M/4)\*N+(M/4)\*N)+N && M>1)

P[i].position='T';

if(P[i].index>(1+M/4)\*N+M/4\*N && P[i].index<=(1+M/4)\*N+(2\*M/4)\*N && M>3)

P[i].partition=3;

if(P[i].index>((1+M/4)\*N+(2\*M/4)\*N)-N && P[i].index<=(1+M/4)\*N+(2\*M/4)\*N && M>1)

P[i].position='B';

if(P[i].index>(1+M/4)\*N+(2\*M/4)\*N && P[i].index<=((1+M/4)\*N+(2\*M/4)\*N)+N && M>1)

P[i].position='T';

if(P[i].index>(1+M/4)\*N+(2\*M/4)\*N && P[i].index<=M\*N && M>3)

P[i].partition=4;

}

if(threads==4 && M%4==2)

{

if(M<4)

{

printf("Your rectangular field is too small to use this number of threads\n");

}

if(P[i].index>0 && P[i].index<=(1+M/4)\*N && M>3)

P[i].partition=1;

if(P[i].index>((1+M/4)\*N)-N && P[i].index<=(1+M/4)\*N && M>1)

P[i].position='B';

if(P[i].index>(1+M/4)\*N && P[i].index<=((1+M/4)\*N)+N && M>1)

P[i].position='T';

if(P[i].index>(1+M/4)\*N && P[i].index<=2\*(1+M/4)\*N && M>3)

P[i].partition=2;

if(P[i].index>(2\*(1+M/4)\*N)-N && P[i].index<=2\*(1+M/4)\*N && M>1)

P[i].position='B';

if(P[i].index>2\*(1+M/4)\*N && P[i].index<=(2\*(1+M/4)\*N)+N && M>1)

P[i].position='T';

if(P[i].index>2\*(1+M/4)\*N && P[i].index<=2\*(1+M/4)\*N+(M/4)\*N && M>3)

P[i].partition=3;

if(P[i].index>(2\*(1+M/4)\*N+(M/4)\*N)-N && P[i].index<=2\*(1+M/4)\*N+(M/4)\*N && M>1)

P[i].position='B';

if(P[i].index>2\*(1+M/4)\*N+(M/4)\*N && P[i].index<=(2\*(1+M/4)\*N+(M/4)\*N)+N && M>1)

P[i].position='T';

if(P[i].index>2\*(1+M/4)\*N+(M/4)\*N && P[i].index<=M\*N && M>3)

P[i].partition=4;

}

if(threads==4 && M%4==3)

{

if(M<4)

{

printf("Your rectangular field is too small to use this number of threads\n");

}

if(P[i].index>0 && P[i].index<=(1+M/4)\*N && M>3)

P[i].partition=1;

if(P[i].index>((1+M/4)\*N)-N && P[i].index<=(1+M/4)\*N && M>1)

P[i].position='B';

if(P[i].index>(1+M/4)\*N && P[i].index<=((1+M/4)\*N)+N && M>1)

P[i].position='T';

if(P[i].index>(1+M/4)\*N && P[i].index<=2\*(1+M/4)\*N && M>3)

P[i].partition=2;

if(P[i].index>(2\*(1+M/4)\*N)-N && P[i].index<=2\*(1+M/4)\*N && M>1)

P[i].position='B';

if(P[i].index>2\*(1+M/4)\*N && P[i].index<=(2\*(1+M/4)\*N)+N && M>1)

P[i].position='T';

if(P[i].index>2\*(1+M/4)\*N && P[i].index<=3\*(1+M/4)\*N && M>3)

P[i].partition=3;

if(P[i].index>(3\*(1+M/4)\*N)-N && P[i].index<=3\*(1+M/4)\*N && M>1)

P[i].position='B';

if(P[i].index>3\*(1+M/4)\*N && P[i].index<=(3\*(1+M/4)\*N)+N && M>1)

P[i].position='T';

if(P[i].index>3\*(1+M/4)\*N && P[i].index<=M\*N && M>3)

P[i].partition=4;

}

}

void print\_individuals(int n\_s, int n\_z){

for(int j=0;j<n\_s+n\_z;j++)

{

printf(" i=%d-" , P[j].i\_dim);

printf("j=%d-" , P[j].j\_dim);

printf("State=%c-" , P[j].state);

printf("Thread=%d\n" , P[j].partition);

}

}