```
#include "stdlib.h"
#include "argp.h"
#include "mpi.h"
#include "stdio.h"
#include "math.h"
#include "string.h"
#include "unistd.h"
#include "regex.h"

// Include global variables. Only this file needs the #define
#define __MAIN
#include "globals.h"
#undef __MAIN

// User includes
#include "pprintf.h"
#include "pgm.h"

//Aaron Holt
//HPSC
//Conways 2
// compile instructions: $ make
// run instructions:
/*
$ mpiexec -np NP ./hw5.1-holtat -v -a run_type iterations printwhen
-v for verbose (print out buggie counts etc)
-a to generate animation
runtype 0=serial, 1=blocked, 2=checkered
iterations = number of iterations desired
CountOnMultipleOfN = print buggie count at printwhen interval
PrintPgmWhen = create pgm file on iterations. specify with csv list like the follow
ing:
   1,4,5,6-9,20-50,100

example run
$mpiexec -np 1 ./hw6.1-holtat -v -a 0 11 10 1,2,4-8
*/



const char *argp_program_version =
    "argp-ex3 1.0";
const char *argp_program_bug_address =
    "<bug-gnu-utils@gnu.org>";

/* Program documentation. */
static char doc[] =
    "A program with options and arguments using argp";

/* A description of the arguments we accept. */
static char args_doc[] = "0=Serial,1=Block,2=Checker  Iterations  CountOnMultipleOf
N PrintPgmWhen";

/* The options we understand. */
static struct argp_option options[] = {
    {"verbose",  ’v’, 0,      0,  "Produce verbose output" },
    {"animation",  ’a’, 0,      0,  "Save an animation" },
    { 0 }
};

/* Used by main to communicate with parse_opt. */
struct arguments
{
    char *args[4];
    int verbose;
    int animation;
};
```

```

/* Parse a single option. */
static error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
    /* Get the input argument from argp_parse, which we
     know is a pointer to our arguments structure. */
    struct arguments *arguments = state->input;
    switch (key)
        {
        case ’v’:
            arguments->verbose = 1;
            break;
        case ’a’:
            arguments->animation = 1;
            break;

        case ARGP_KEY_ARG:
            if (state->arg_num >= 5)
            /* Too many arguments. */
            argp_usage (state);
            arguments->args[state->arg_num] = arg;
            break;

        case ARGP_KEY_END:
            if (state->arg_num < 2)
            /* Not enough arguments. */
            argp_usage (state);
            break;

        default:
            return ARGP_ERR_UNKNOWN;
        }
    return 0;
}

/* Our argp parser. */
static struct argp argp = { options, parse_opt, args_doc, doc };

//Takes in current frame number and matrix
void write_matrix_to_pgm(int frame, int rsize, int csize,
                 unsigned char* full_matrix)
{
    int i,j;

    // printf("rsize,csize = %d, %d\n ", rsize, csize);

    //dynamic filename with leading zeroes for easy conversion to gif
    char buffer[128];
    snprintf(buffer, sizeof(char)*128, "Animation/frame%04d.pgm", frame);

    //open
    FILE *fp;
    fp = fopen(buffer, "wb");

    //header
    fprintf(fp, "P2\n");
    fprintf(fp, "%4d %4d\n", rsize, csize);
    fprintf(fp, "255\n");

    //data
    for (i=0;i<csize;i++)
    {
        for (j=0;j<rsize;j++)
        {
            fprintf(fp, "%3d ", full_matrix[i*rsize+j]);
        }
        //newline after every row
        fprintf(fp, "\n");
    }
    //trailing newline
    fprintf(fp, "\n");

    //close file
    fclose(fp);
}

//Takes in current cell location, and all neighboring data
//outputs integer of alive neighbor cells
int count_neighbors(int info[5], unsigned char info2[4], unsigned char* section,
            unsigned char* top, unsigned char* bot,
            unsigned char* left, unsigned char* right)
            // int topleft, int topright, int botleft, int botright)
{
    int i,j,rsize,csize,topleft,topright,botright,botleft;
    i = info[0];
    j = info[1];
    // wr = info[2];
    rsize = info[3];
    csize = info[4];
    topleft = info2[0];
    topright = info2[1];
    botleft = info2[2];
    botright = info2[3];

    int total_around = 0;
    // printf("wr=%d, i=%d,j=%d\n",wr,i,j);
    // printf("wr=%d, top[j]=%d\n",wr,top[j]);

    //top center//
    //on top edge?
    if (i == 0)
    {
        //alive?
        if (top[j] == 0)
        {
            total_around += 1;
        }
        // printf("HERE@\n");
    }
    //in middle somewhere
    else if (section[(i-1)*rsize + j] == 0)
    {
        total_around += 1;
    }


    //bottom center//
    //on bot edge?
    if (i == (csize-1))
    {
        if (bot[j] == 0)
        {
            total_around += 1;
        }
    }
    else if (section[(i+1)*rsize + j] == 0)
    {
        total_around += 1;
    }

    //right//
    //on right edge?
    if(j == (rsize-1))
```

```
    {
        if(right[i] == 0)
        {
            total_around += 1;
        }
    }
    else if (section[i*rsize+j+1] == 0)
    {
        total_around += 1;
    }

    //left//
    //on left edge?
    if(j == 0)
    {
        if(left[i] == 0)
        {
            total_around += 1;
        }
    }
    else if (section[i*rsize+j-1] == 0)
    {
        total_around += 1;
    }

    //topleft//
    //on topleft corner?
    if (i==0 && j==0)
    {
        if (topleft == 0)
        {
            total_around += 1;
        }
    }
    //on top row?
    else if (i == 0)
    {
        if (top[j-1] == 0)
        {
            total_around += 1;
        }
    }
    //on left edge?
    else if (j == 0)
    {
        if (left[i-1] == 0)
        {
            total_around += 1;
        }
    }
    //in center?
    else if (section[(i-1)*rsize+j-1] == 0)
    {
        total_around += 1;
    }

    //topright//
    //topright corner?
    if (i==0 && j==rsize-1)
    {
        if (topright == 0)
        {
            total_around += 1;
        }
    }
    //on top row?
    else if (i == 0)
    {
        if (top[j+1] == 0)
        {
            total_around += 1;
        }
    }
    //on right edge?
    else if (j == rsize-1)
    {
        if (right[i-1] == 0)
        {
            total_around += 1;
        }
    }
    //in center?
    else if (section[(i-1)*rsize+j+1] == 0)
    {
        total_around += 1;
    }

    //botright//
    //botright corner?
    if (i==csize-1 && j==rsize-1)
    {
        if (botright == 0)
        {
            total_around += 1;
        }
    }
    //on bot row?
    else if (i == csize-1)
    {
        if (bot[j+1] == 0)
        {
            total_around += 1;
        }
    }
    //on right edge?
    else if (j == rsize-1)
    {
        if (right[i+1] == 0)
        {
            total_around += 1;
        }
    }
    //in center?
    else if (section[(i+1)*rsize+j+1] == 0)
    {
        total_around += 1;
    }

    //botleft//
    //botleft corner?
    if (i==csize-1 && j==0)
    {
        if (botleft == 0)
        {
            total_around += 1;
        }
    }
    //on bot row?
    else if (i == csize-1)
    {
        if (bot[j-1] == 0)
        {
            total_around += 1;
        }
```

```
    }
    //on left edge?
    else if (j == 0)
    {
        if (left[i+1] == 0)
        {
            total_around += 1;
        }
    }
    //in center?
    else if (section[(i+1)*rsize+j-1] == 0)
    {
        total_around += 1;
    }

    return total_around;
}

//counts number of buggies in a given matrix
int count_buggies(int rsize, int csize, unsigned char* matrix)
{
    int i,j,count;
    count = 0;
    for (i=0;i<csize;i++)
    {
        for (j=0;j<rsize;j++)
        {
            if (matrix[i*rsize+j]>0)
            {
                count += 1;
            }
        }
    }
    return count;
}

void print_matrix(int rsize, int csize, unsigned char* matrix)
{
        int i,j;
        for (i=0;i<csize;i++)
    {
        for (j=0;j<rsize;j++)
        {
                // printf("so %d\n", (int)sizeof(t_A));
                printf("%3d ", matrix[i*rsize+j]);
        }
        // printf("\nROW=%d\n",i);
        printf("\n");
        }
        // printf("\n");
}


int main (int argc, char **argv)
{
    struct arguments arguments;

    /* Parse our arguments; every option seen by parse_opt will
       be reflected in arguments. */
    argp_parse (&argp, argc, argv, 0, 0, &arguments);

    int run_type;
    run_type = 0; //default is serial
    if (sscanf (arguments.args[0], "%i", &run_type)!=1) {}

    int iterations;
    iterations = 0; //default is serial
    if (sscanf (arguments.args[1], "%i", &iterations)!=1) {}

    int count_when;
    count_when = 1000;
    if (sscanf (arguments.args[2], "%i", &count_when)!=1) {}

    char print_list[200]; //used for input list
    if (sscanf (arguments.args[3], "%s", &print_list)!=1) {}

    // printf("Print list = %s\n", print_list);

    //Extract animation list from arguments
    char char_array[20][12] = { NULL };   //seperated input list
    int animation_list[20][2] = { NULL }; //integer input list start,range
    char *tok = strtok(print_list, ",");

    //counters
    int i,j,k,x,y,ii,jj;
    ii = 0;
    jj = 0;

    //Loop over tokens parsing our commas
    int tok_len = 0;
    while (tok != NULL)
    {
        //first loop parses out commas
        tok_len = strlen(tok);
        for (jj=0;jj<tok_len;jj++)
        {
            char_array[ii][jj] = tok[jj];
        }

        // printf("Tok = %s\n", char_array[ii]);
        tok = strtok(NULL, ",");
        ii++;
    }

    //looking for a range input, convert to ints
    int stop;
    for (ii=0;ii<20;ii++)
    {
        //convert first number to int
        tok = strtok(char_array[ii], "-");
        if (tok != NULL)
        {
            animation_list[ii][0] = atoi(tok);
            tok = strtok(NULL, ",");
        }

        //look for second number, add to range
        if (tok != NULL)
        {
            stop = atoi(tok);
            animation_list[ii][1] = stop - animation_list[ii][0];
        }

        // if (rank == 0)
        // {
        //     printf("Animation_list = %i, %i\n",
        //         animation_list[ii][0], animation_list[ii][1]);

        // }
    }
```

```
    //should an animation be generated
    //prints a bunch of .pgm files, have to hand
    //make the gif...
    int animation;
    animation = arguments.animation;

    //verbose?
    int verbose;
    verbose = arguments.verbose;
    // printf("VERBOSE = %i",verbose);
    if (verbose>=0 && verbose<=10)
    {
        verbose = 1;
    }



    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    //Print run information, exit on bad command line input
    if (rank == 0)
    {
        printf("Verbose=%i, RunType=%i, Iterations=%i, CountWhen=%i, Animation=%i\n
",
            verbose,run_type,iterations,count_when, animation);
    }
    if (world_size>1 && run_type ==0)
    {
        printf("Runtype and processors count not consistant\n");
        MPI_Finalize();
        exit(0);
    }
    if (world_size==1 && run_type>0)
    {
        printf("Runtype and processors count not consistant\n");
        MPI_Finalize();
        exit(0);
    }
    if (count_when <= 0)
    {
        if (rank == 0)
        {
            printf("Invalid count interval, positive integers only\n");
        }
        MPI_Finalize();
        exit(0);
    }

     //serial
    if (world_size == 1 && run_type == 0)
    {

        ncols=1;
        nrows=1;
    }
    //Blocked
    else if (world_size>1 && run_type == 1)
    {
        ncols = 1;
        nrows = world_size;
        my_col = 0;
        my_row = rank;
    }
    //Checker
    else if (world_size>1 && run_type == 2)
    {
        ncols = (int)sqrt(world_size);
        nrows = (int)sqrt(world_size);

        my_row = rank/nrows;
        my_col = rank-my_row*nrows;

        if (ncols*nrows!=world_size)
        {
            if (rank == 0)
            {
                printf("Number of processors must be square, Exiting\n");
            }
            MPI_Finalize();
            exit(0);
        }
    }

    // if (verbose == 1)
    // {
    //     printf("WR,row,col=%i,%i,%i\n",rank,my_row,my_col);
    // }


    //////////////////////READ IN INITIAL PGM////////////////////////////////
    if(!readpgm("cool.pgm"))
    {
        // printf("WR=%d,HERE2\n",rank);
        if( rank==0 )
        {
            pprintf( "An error occured while reading the pgm file\n" );
        }
        MPI_Finalize();
        return 1;
    }

    // Count the life forms. Note that we count from [1,1] - [height+1,width+1];
    // we need to ignore the ghost row!
    i = 0;
    for(y=1; y<local_height+1; y++ )
    {
        for(x=1; x<local_width+1; x++ )
        {
            if( field_a[ y * field_width + x ] )
            {
                i++;
            }
        }
    }
    // pprintf( "%i local buggies\n", i );

    int total;
    MPI_Allreduce( &i, &total, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD );
    if( rank==0 )
    {
```

```
        pprintf( "%i total buggies\n", total );
    }



    // printf("WR=%d, Row=%d, Col=%d\n",rank,my_row,my_col);

    //Row and column size per processor
    int rsize, csize;
    rsize = local_width;
    csize = local_height;


    if (rank == 0 && verbose == 1)
    {
        printf("rsize,csize,NP = %d, %d, %d\n",rsize,csize,world_size);
    }

    //Create new derived datatype for writing to files
    MPI_Datatype submatrix;

    int array_of_gsizes[2];
    int array_of_distribs[2];
    int array_of_dargs[2];
    int array_of_psize[2];

    if (run_type == 1)
    {
        if (rank == 0)
        {
            printf("g0,g1 = %i,%i\n", local_height*ncols, local_width);
            printf("p0,p1 = %i,%i\n", nrows, ncols);
        }
        array_of_gsizes[0] = local_height*ncols;
        array_of_gsizes[1] = local_width;
        array_of_distribs[0] = MPI_DISTRIBUTE_BLOCK;
        array_of_distribs[1] = MPI_DISTRIBUTE_BLOCK;
        array_of_dargs[0] = MPI_DISTRIBUTE_DFLT_DARG;
        array_of_dargs[1] = MPI_DISTRIBUTE_DFLT_DARG;
        array_of_psize[0] = nrows;
        array_of_psize[1] = ncols;
        // int order = MPI_ORDER_C;

        //size,rank,ndims,array_gsizes,array_distribs,array_args,array_psizes
        //order,oldtype,*newtype
        MPI_Type_create_darray(world_size, rank, 2, array_of_gsizes, array_of_distr
ibs,
                array_of_dargs, array_of_psize, MPI_ORDER_C, MPI_UNSIGNED_CHAR, &su
bmatrix);
        MPI_Type_commit(&submatrix);
    }
    else if (run_type == 2)
    {
        if (rank == 0)
        {
            printf("g0,g1 = %i,%i\n", local_height*ncols, local_width*nrows);
            printf("p0,p1 = %i,%i\n", nrows, ncols);
        }
        array_of_gsizes[0] = local_height*ncols;
        array_of_gsizes[1] = local_width*nrows;
        array_of_distribs[0] = MPI_DISTRIBUTE_BLOCK;
        array_of_distribs[1] = MPI_DISTRIBUTE_BLOCK;
        array_of_dargs[0] = MPI_DISTRIBUTE_DFLT_DARG;
        array_of_dargs[1] = MPI_DISTRIBUTE_DFLT_DARG;
        array_of_psize[0] = nrows;
        array_of_psize[1] = ncols;
        // int order = MPI_ORDER_C;
        //size,rank,ndims,array_gsizes,array_distribs,array_args,array_psizes
        //order,oldtype,*newtype
        MPI_Type_create_darray(world_size, rank, 2, array_of_gsizes, array_of_distr
ibs,
                array_of_dargs, array_of_psize, MPI_ORDER_C, MPI_UNSIGNED_CHAR, &su
bmatrix);
        MPI_Type_commit(&submatrix);
    }



    MPI_Barrier(MPI_COMM_WORLD);

    /////////////////ALLOCATE ARRAYS, CREATE DATATYPES////////////////////

    //Create new column derived datatype
    MPI_Datatype column;
    //count, blocklength, stride, oldtype, *newtype
    MPI_Type_hvector(csize, 1, sizeof(unsigned char), MPI_UNSIGNED_CHAR, &column);
    MPI_Type_commit(&column);

    //Create new row derived datatype
    MPI_Datatype row;
    //count, blocklength, stride, oldtype, *newtype
    MPI_Type_hvector(rsize, 1, sizeof(unsigned char), MPI_UNSIGNED_CHAR, &row);
    MPI_Type_commit(&row);

    //allocate arrays and corner storage
    unsigned char *section;
    unsigned char *neighbors;
    //to use
    unsigned char *top;
    unsigned char *bot;
    unsigned char *left;
    unsigned char *right;
    //to send
    unsigned char *ttop;
    unsigned char *tbot;
    unsigned char *tleft;
    unsigned char *tright;
    //MALLOC!!
    section = (unsigned char*)malloc(rsize*csize*sizeof(unsigned char));
    neighbors = (unsigned char*)malloc(rsize*csize*sizeof(unsigned char));
    top = (unsigned char*)malloc(rsize*sizeof(unsigned char));
    bot = (unsigned char*)malloc(rsize*sizeof(unsigned char));
    left = (unsigned char*)malloc(csize*sizeof(unsigned char));
    right = (unsigned char*)malloc(csize*sizeof(unsigned char));
    ttop = (unsigned char*)malloc(rsize*sizeof(unsigned char));
    tbot = (unsigned char*)malloc(rsize*sizeof(unsigned char));
    tleft = (unsigned char*)malloc(csize*sizeof(unsigned char));
    tright = (unsigned char*)malloc(csize*sizeof(unsigned char));

    //corners
    unsigned char topleft,topright,botleft,botright; //used in calculations
    unsigned char ttopleft,ttopright,tbotleft,tbotright;
    topleft = 255;
    topright = 255;
    botleft = 255;
    botright = 255;

    //used for animation, each process will put there own result in and then
    //each will send to process 1 which will add them up
    unsigned char* full_matrix;
    unsigned char* full_matrix_buffer;
    if (animation == 1)
    {
```

```
        int msize1 = rsize*ncols*csize*nrows;
        full_matrix = (unsigned char*)malloc(msize1*sizeof(unsigned char));
        full_matrix_buffer = (unsigned char*)malloc(msize1*sizeof(unsigned char));
        for (i=0; i<msize1; i++)
        {
            full_matrix[i] = 0;
            full_matrix_buffer[i] = 0;
        }
    }


    // printf("Rsize,Lsize,Fsize=%i %i %i,Csize,Lsize,Fsize=%i %i %i\n",rsize,local
_width,field_width,csize,local_height,field_height);

    //Serial initialize vars
    int count = 0;
    if (world_size == 1 && run_type == 0)
    {
        for (i=0;i<csize;i++)
        {
            for (j=0;j<rsize;j++)
            {
                section[i*rsize + j] = 255;

                if (field_a[(i+1)*(2+rsize) + j + 1])
                {
                    section[i*rsize + j] = 0;
                    count += 1;
                }
                else
                {
                    section[i*rsize + j] = 255;
                }

                top[j] = 255;
                bot[j] = 255;
                ttop[j] = 255;
                tbot[j] = 255;
            }
            right[i] = 255;
            left[i] = 255;
            tright[i] = 255;
            tleft[i] = 255;
        }
        // printf("COUNT 4 = %d\n", count);
    }

    //Blocked/Checkered initializing variables
    else if (world_size > 1 && (run_type == 1 || run_type == 2))
    {
        //initialize
        for (i=0;i<csize;i++)
        {
            for (j=0;j<rsize;j++)
            {
                section[i*rsize + j] = 255;

                if (field_a[(i+1)*(2+rsize) + j + 1])
                {
                    section[i*rsize + j] = 0;
                    count += 1;
                }
                else
                {
                    section[i*rsize + j] = 255;
                }
                top[j] = 255;
                bot[j] = 255;
                ttop[j] = 255;
                tbot[j] = 255;
            }
            right[i] = 255;
            left[i] = 255;
            tright[i] = 255;
            tleft[i] = 255;
        }

        // MPI_Allreduce( &count, &total, 1, MPI_UNSIGNED_CHAR, MPI_SUM, MPI_COMM_W
ORLD );
        // if (rank == 0)
        // {
        //     printf("COUNT 4 = %d\n", total);
        // }

    }


    //header/footer for mpio writes
    char header1[15];
    header1[0] = 0x50;
    header1[1] = 0x35;
    header1[2] = 0x0a;
    header1[3] = 0x35;
    header1[4] = 0x31;
    header1[5] = 0x32;
    header1[6] = 0x20;
    header1[7] = 0x35;
    header1[8] = 0x31;
    header1[9] = 0x32;
    header1[10] = 0x0a;
    header1[11] = 0x32;
    header1[12] = 0x35;
    header1[13] = 0x35;
    header1[14] = 0x0a;

    char footer;
    footer = 0x0a;

    //make a frame or not?
    int create_frame = 0;

    //send to
    int send_to;
    int receive_from;
    int info[5];
    info[2] = rank;
    info[3] = rsize;
    info[4] = csize;
    unsigned char info2[4];
    info2[0] = topleft;
    info2[1] = topright;
    info2[2] = botleft;
    info2[3] = botright;

    int current_count;
    int location;

    //Gameplay
    for (k=0;k<iterations;k++)
    {
        //Count buggies
        if (k%count_when==0)
        {
```

```
            if (verbose == 1)
            {
                current_count = rsize*csize-count_buggies(rsize,csize,section);
                MPI_Allreduce( &current_count, &total, 1, MPI_INT, MPI_SUM, MPI_COM
M_WORLD );
                if (rank == 0)
                {
                    printf("Iteration=%5d,  Count=%6d\n", k,total);
                }
                ////corner debug
                // printf("WR,tl,tr,bl,br = %d %d %d %d %d\n", rank, topleft, topri
ght, botleft, botright);
            }
        }


        //Write to file serially for comparison
        //If animation is requested
        if (animation == 1 && run_type == 0)
        {
            //Put smaller matrix part into larger matrix
            for (i=0; i<csize; i++)
            {
                for (j=0; j<rsize; j++)
                {
                    location = (my_row*csize*rsize*ncols + my_col*rsize +
                                    i*rsize*ncols + j);

                    full_matrix_buffer[location] = section[i*rsize+j];
                }
                // if (rank == 0)
                // {
                //     printf("Location = %d\n", location);
                // }
            }

            //Gather matrix
            MPI_Reduce(full_matrix_buffer, full_matrix, rsize*ncols*csize*nrows,
                MPI_UNSIGNED_CHAR, MPI_SUM, 0, MPI_COMM_WORLD);


            if (rank == 0 && run_type == 0)
            {
                write_matrix_to_pgm(k, rsize*ncols, csize*nrows, full_matrix);
            }
        }
        //mpio write pgm
        else if (animation == 1 && (run_type == 1 || run_type == 2))
        {
            //default is no frame
            create_frame = 0;
            for (ii=0;ii<20;ii++)
            {
                for (jj=0;jj<animation_list[ii][1]+1;jj++)
                {
                    // if (rank == 0)
                    // {
                    //     printf("a,ii,j,k= %i,%i,%i,%i, Frame? = %i\n",
                    //         animation_list[ii][0],ii,jj,k,(animation_list[ii][0]
+jj-k)==0);
                    // }
                    if ((animation_list[ii][0] + jj - k) == 0)
                    {

                        create_frame = 1;
                        break;
                    }
            }
        }

        if (create_frame == 1)
        {
           //dynamic filename with leading zeroes for easy conversion to gif
            char buffer[128];
            snprintf(buffer, sizeof(char)*128, "Animation/frame%04d.pgm", k);

            /* open the file, and set the view */
            MPI_File file;
            MPI_File_open(MPI_COMM_WORLD, buffer,
                          MPI_MODE_CREATE|MPI_MODE_WRONLY,
                          MPI_INFO_NULL, &file);

            MPI_File_set_view(file, 0,  MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR,
                                   "native", MPI_INFO_NULL);

            //write header
            MPI_File_write(file, &header1, 15, MPI_CHAR, MPI_STATUS_IGNORE);

            //write matrix
            MPI_File_set_view(file, 15,  MPI_UNSIGNED_CHAR, submatrix,
                                   "native", MPI_INFO_NULL);

            MPI_File_write_all(file, section, rsize*csize,
                    MPI_UNSIGNED_CHAR, MPI_STATUS_IGNORE);

            //write footer (trailing newline)
            MPI_File_set_view(file, 15+rsize*ncols*csize*nrows,
                    MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR,
                    "native", MPI_INFO_NULL);

            MPI_File_write(file, &footer, 1, MPI_CHAR, MPI_STATUS_IGNORE);
        }
    }


    // BLOCKED COMMUNITATION //
    if (run_type == 1)
    {
        //change bot (send top) to account for middle area
        //alternate to avoid locking
        send_to = rank - 1;
        receive_from = rank + 1;

        //figure out what to send
        //top and bottom
        for (i=0;i<rsize;i++)
        {
            ttop[i] = section[i];
            tbot[i] = section[rsize*(csize-1)+i];
        }

        //left n right
        for (i=0;i<csize;i++)
        {
            tleft[i] = section[0 + rsize*i];
            tright[i] = section[rsize-1 + rsize*i];
        }

        //send top, receive bot
        if (rank%2==0)
        {
            if (send_to<world_size && send_to>=0)
            {
                MPI_Send(ttop, 1, row, send_to, 0, MPI_COMM_WORLD);
            }
            if (receive_from<world_size && receive_from >= 0)
            {
                MPI_Recv(bot, 1, row, receive_from, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
            }
        }
        else if (rank%2==1)
        {

            if (receive_from<world_size && receive_from >= 0)
            {
                MPI_Recv(bot, 1, row, receive_from, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
            }
            if (send_to<world_size && send_to>=0)
            {
                MPI_Send(ttop, 1, row, send_to, 0, MPI_COMM_WORLD);
            }
        }

        //change top to account for middle area
        //alternate to avoid locking
        send_to = rank + 1;
        receive_from = rank - 1;

        //send bot, receive top
        if (rank%2==0)
        {
            // printf("%d, %d, %d\n", rank, send_to, receive_from);
            if (send_to<world_size && send_to>=0)
            {
                MPI_Send(tbot, 1, row, send_to, 0, MPI_COMM_WORLD);
            }

            if (receive_from<world_size && receive_from >= 0)
            {
                MPI_Recv(top, 1, row, receive_from, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
            }
        }
        else if (rank%2==1)
        {
            // printf("%d, %d, %d\n", rank, send_to, receive_from);
            if (receive_from<world_size && receive_from >= 0)
            {
                //*data,count,type,from,tag,comm,mpi_status
                MPI_Recv(top, 1, row, receive_from, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
            }

            if (send_to<world_size && send_to>=0)
            {
                //*data,count,type,to,tag,comm
                MPI_Send(tbot, 1, row, send_to, 0, MPI_COMM_WORLD);
            }
        }
    }

    // CHECKERED COMMUNITATION //
    else if (run_type == 2)
    {
        //figure out what to send
        //top and bottom
        for (i=0;i<rsize;i++)
        {
            ttop[i] = section[i];
```

```
                tbot[i] = section[rsize*(csize-1)+i];
            }

            //left n right
            for (i=0;i<csize;i++)
            {
                tleft[i] = section[0 + rsize*i];
                tright[i] = section[rsize-1 + rsize*i];
            }

            //corners
            ttopleft = tleft[0];
            tbotleft = tleft[csize-1];
            ttopright = tright[0];
            tbotright = tright[csize-1];

            //Send top, receive bot
            send_to = rank - nrows;
            receive_from = rank + nrows;
            if (rank%2==0)
            {
                if (send_to<world_size && send_to>=0)
                {
                    MPI_Send(ttop, 1, row, send_to, 0, MPI_COMM_WORLD);
                }
                if (receive_from<world_size && receive_from>=0)
                {
                    MPI_Recv(bot, 1, row, receive_from, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
            }
            else if (rank%2==1)
            {

                if (receive_from<world_size && receive_from>=0)
                {
                    MPI_Recv(bot, 1, row, receive_from, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
                if (send_to<world_size && send_to>=0)
                {
                    MPI_Send(ttop, 1, row, send_to, 0, MPI_COMM_WORLD);
                }
            }

            //Send bot, receive top
            send_to = rank + nrows;
            receive_from = rank - nrows;
            if (rank%2==0)
            {
                if (send_to<world_size && send_to>=0)
                {
                    MPI_Send(tbot, 1, row, send_to, 0, MPI_COMM_WORLD);
                }
                if (receive_from<world_size && receive_from>=0)
                {
                    MPI_Recv(top, 1, row, receive_from, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
            }
            else if (rank%2==1)
            {

                if (receive_from<world_size && receive_from>=0)
                {
                    MPI_Recv(top, 1, row, receive_from, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
```

```
                }
                if (send_to<world_size && send_to>=0)
                {
                    MPI_Send(tbot, 1, row, send_to, 0, MPI_COMM_WORLD);
                }
            }

            //Send left, receive right
            send_to = rank - 1;
            receive_from = rank + 1;

            if (rank%2==0)
            {
                if (send_to<world_size && send_to>=0 && send_to/nrows==my_row)
                {
                    MPI_Send(tleft, 1, column, send_to, 0, MPI_COMM_WORLD);
                }
                if (receive_from<world_size && receive_from>=0 && receive_from/nrow
s==my_row)
                {
                    MPI_Recv(right, 1, column, receive_from, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
            }
            else if (rank%2==1)
            {
                if (receive_from<world_size && receive_from>=0 && receive_from/nrow
s==my_row)
                {
                    MPI_Recv(right, 1, column, receive_from, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
                if (send_to<world_size && send_to>=0 && send_to/nrows==my_row)
                {
                    MPI_Send(tleft, 1, column, send_to, 0, MPI_COMM_WORLD);
                }
            }

            //Send right, receive left
            send_to = rank + 1;
            receive_from = rank - 1;

            if (rank%2==0)
            {
                if (send_to<world_size && send_to>=0 && send_to/nrows==my_row)
                {
                    MPI_Send(tright, 1, row, send_to, 0, MPI_COMM_WORLD);
                }
                if (receive_from<world_size && receive_from>=0 && receive_from/nrow
s==my_row)
                {
                    MPI_Recv(left, 1, row, receive_from, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
            }
            else if (rank%2==1)
            {
                if (receive_from<world_size && receive_from>=0 && receive_from/nrow
s==my_row)
                {
                    MPI_Recv(left, 1, row, receive_from, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
                if (send_to<world_size && send_to>=0 && send_to/nrows==my_row)
                {
                    MPI_Send(tright, 1, row, send_to, 0, MPI_COMM_WORLD);
                }
            }

            //Send topright, receive botleft
            send_to = rank - ncols + 1;
            receive_from = rank + ncols - 1;

            if (rank%2==0)
            {
                if (send_to<world_size && send_to>=0 && send_to/nrows==my_row-1)
                {
                    MPI_Send(&ttopright, 1, MPI_UNSIGNED_CHAR, send_to, 0, MPI_COMM
_WORLD);
                }
                if (receive_from<world_size && receive_from>=0 && receive_from/nrow
s==my_row+1)
                {
                    MPI_Recv(&botleft, 1, MPI_UNSIGNED_CHAR, receive_from, 0, MPI_C
OMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
            }
            else if (rank%2==1)
            {
                if (receive_from<world_size && receive_from>=0 && receive_from/nrow
s==my_row+1)
                {
                    MPI_Recv(&botleft, 1, MPI_UNSIGNED_CHAR, receive_from, 0, MPI_C
OMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
                if (send_to<world_size && send_to>=0 && send_to/nrows==my_row-1)
                {
                    MPI_Send(&ttopright, 1, MPI_UNSIGNED_CHAR, send_to, 0, MPI_COMM
_WORLD);
                }
            }

            //Send topleft, receive botright
            send_to = rank - ncols - 1;
            receive_from = rank + ncols + 1;

            if (rank%2==0)
            {
                if (send_to<world_size && send_to>=0 && send_to/nrows==my_row-1)
                {
                    MPI_Send(&ttopleft, 1, MPI_UNSIGNED_CHAR, send_to, 0, MPI_COMM_
WORLD);
                }
                if (receive_from<world_size && receive_from>=0 && receive_from/nrow
s==my_row+1)
                {
                    MPI_Recv(&botright, 1, MPI_UNSIGNED_CHAR, receive_from, 0, MPI_
COMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
            }
            else if (rank%2==1)
            {
                if (receive_from<world_size && receive_from>=0 && receive_from/nrow
s==my_row+1)
                {
                    MPI_Recv(&botright, 1, MPI_UNSIGNED_CHAR, receive_from, 0, MPI_
COMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
                if (send_to<world_size && send_to>=0 && send_to/nrows==my_row-1)
                {
```

```
                    MPI_Send(&ttopleft, 1, MPI_UNSIGNED_CHAR, send_to, 0, MPI_COMM_
WORLD);
                }
            }

            //Send botleft, receive topright
            send_to = rank + ncols - 1;
            receive_from = rank - ncols + 1;

            if (rank%2==0)
            {
                if (send_to<world_size && send_to>=0 && send_to/nrows==my_row+1)
                {
                    MPI_Send(&tbotleft, 1, MPI_UNSIGNED_CHAR, send_to, 0, MPI_COMM_
WORLD);
                }
                if (receive_from<world_size && receive_from>=0 && receive_from/nrow
s==my_row-1)
                {
                    MPI_Recv(&topright, 1, MPI_UNSIGNED_CHAR, receive_from, 0, MPI_
COMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
            }
            else if (rank%2==1)
            {
                if (receive_from<world_size && receive_from>=0 && receive_from/nrow
s==my_row-1)
                {
                    MPI_Recv(&topright, 1, MPI_UNSIGNED_CHAR, receive_from, 0, MPI_
COMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
                if (send_to<world_size && send_to>=0 && send_to/nrows==my_row+1)
                {
                    MPI_Send(&tbotleft, 1, MPI_UNSIGNED_CHAR, send_to, 0, MPI_COMM_
WORLD);
                }
            }

            //Send botright, receive topleft
            send_to = rank + ncols + 1;
            receive_from = rank - ncols - 1;

            if (rank%2==0)
            {
                if (send_to<world_size && send_to>=0 && send_to/nrows==my_row+1)
                {
                    MPI_Send(&tbotright, 1, MPI_UNSIGNED_CHAR, send_to, 0, MPI_COMM
_WORLD);
                }
                if (receive_from<world_size && receive_from>=0 && receive_from/nrow
s==my_row-1)
                {
                    MPI_Recv(&topleft, 1, MPI_UNSIGNED_CHAR, receive_from, 0, MPI_C
OMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
            }
            else if (rank%2==1)
            {
                if (receive_from<world_size && receive_from>=0 && receive_from/nrow
s==my_row-1)
                {
                    MPI_Recv(&topleft, 1, MPI_UNSIGNED_CHAR, receive_from, 0, MPI_C
OMM_WORLD,
                        MPI_STATUS_IGNORE);
                }
                if (send_to<world_size && send_to>=0 && send_to/nrows==my_row+1)
                {
                    MPI_Send(&tbotright, 1, MPI_UNSIGNED_CHAR, send_to, 0, MPI_COMM
_WORLD);
                }
            }


            info2[0] = topleft;
            info2[1] = topright;
            info2[2] = botleft;
            info2[3] = botright;

        }

        // if (rank == 1){
        //     print_matrix(rsize, 1, top);
        //     print_matrix(rsize, csize, section);
        //     print_matrix(rsize, 1, bot);
        //     printf("\n");
        // }
        // printf("wr=%d,iteration=%d,maxval=%d, 11\n", rank, k,(csize-1)*rsize-1+r
size);



        /////////// CELL UPDATES /////////////////
        //count neighbor
        for (i=0;i<csize;i++)
        {
            for (j=0; j<rsize; j++)
            {
                info[0] = i;
                info[1] = j;
                neighbors[i*rsize+j] = count_neighbors(info, info2, section,
                                    top, bot, left, right);
            }
        }

        //update cells
        current_count = 0;
        for (i=0;i<csize;i++)
        {
            for (j=0; j<rsize; j++)
            {
                //cell currently alive
                if (section[i*rsize+j] == 0)
                {
                    //2 or 3 neighbors lives, else die
                    if (neighbors[i*rsize+j] < 2 ||
                        neighbors[i*rsize+j] > 3)
                    {
                        section[i*rsize+j] = 255;
                    }
                }
                else
                {
                    //Exactly 3 neighbors spawns new life
                    if (neighbors[i*rsize+j] == 3)
                    {
                        section[i*rsize+j] = 0;
                    }
                }
            }
        }
    }
```

```

    MPI_Barrier(MPI_COMM_WORLD);
    sleep(0.5);
    //free malloc stuff
    if( field_a != NULL ) free( field_a );
    if( field_b != NULL ) free( field_b );
    free(section);
    free(neighbors);
    free(top);
    free(bot);
    free(left);
    free(right);

    MPI_Finalize();
    exit (0);
}
```