Aaron Holt
Conway's Part 2

MPIO and Animation Generation:

In Conway's part 1, the 'game board' was collected onto one process and then written to a file with fprintf. This was rather inefficient as it involved global communication as well a section of code where only 1 process was working. Thus for Conway's part 2, MPIO was used to write the 'game board' to a file in parallel. The PGM files were only written on iterations specified by the command line, which included inputs such as 1,4,20-40,100-500.

Each process had its own part of the matrix (game board), and thus a derived datatype was used to described the location of this matrix chunk. The code for the blocked versions derived datatype is shown below:

```
MPI_datatype submatrix;
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;

//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_distribs,
      array_of_dargs, array_of_psize, MPI_ORDER_C, MPI_UNSIGNED_CHAR,
&submatrix);
MPI_Type_commit(&submatrix);
```

Figure 1: Code for creating a derived datatype for the blocked version. The code for creating the checkered datatype is identical to the code in Figure 1 except for that the array_of_gsizes[1] was changed to account for multiple columns.

MPI_Type_create_darray was used to create the derived datatype as it supports different processor configurations. In other words a matrix can be mapped onto processors in many different ways. Unsigned chars were used as they are the smallest type that can fit values between 0 and 255, which was all that was needed for the PGM file.

Next the matrix was written to a file using MPI_File_open, MPI_File_write, MPI_File_write_all, and MPI_FIle_set_view. This is seen in Figure 2.

```
//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);
```

Figure 2: Writing to a file with MPIO.

The filename was dynamically generated using a character buffer and snprintf. Next, the file was opened using MPI_File_open. The view was set to the first byte of the file with MPI_File_set_view such that the header could be written with MPI_File_write. Next the view was changed to the 15th byte of the file (just after the header) such that the matrix could be written to the file. The matrix was written in parallel to the file using MPI_File_write_all. Finally the view was changed again and the footer (a trailing newline as per PGM specs) was written to the file.

Input choices and Animation Generation:

A png containing many glider guns and other dynamic life forms was downloaded from google and converted to a PGM. It took hundreds of iterations for this input to reach steady state. A PGM was output on every iteration for 500 iterations and converted to a GIF using Image Magick's convert tool.

Profiling with Allinea:

Note to grader: I couldn't get MPE working, and had this entire report written using Allinea before you posted your solution. I didn't have time to re-write it.

Profiling was done using Allinea 5.0 on Janus with 16 nodes using a 512x512 PGM file. Profiling for was done for the checkered and blocked implementation with nothing printed to stdout. Both the checkered and blocked implementations were run with and without an animation being generated. Iterations used depended on the runtype.

Instead of profiling specific parts of the code, Allinea profiles everything and then generates charts, graphs, and reports later from which information about specific parts of the code can be derived.

Checkered with Animation Generation:

Allinea offers several different charts and metrics to help profile your code. The first is a set of three gantt charts which display the total thread activity. The first chart shows main thread activity, the second floating point operations, and the third memory usage. The horizontal axis on this chart represents time whereas the vertical axis represents the total system resources. These results are shown in the Figure 3:

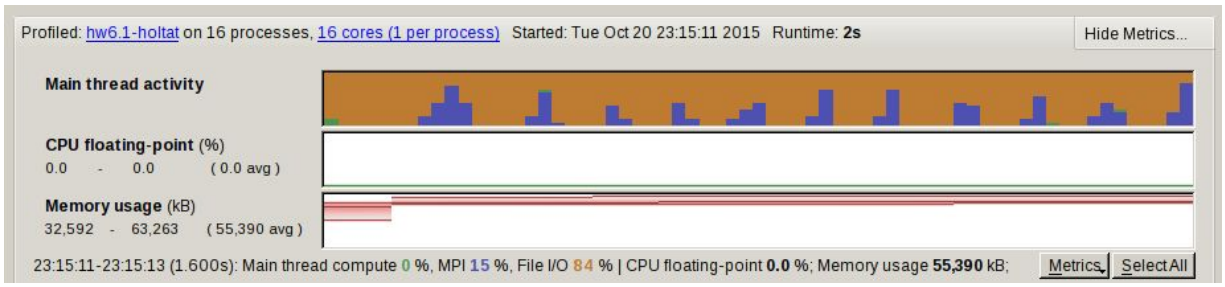

Figure 3: Charts and graphs for checkered run with animation generation over 11 iterations.

The total runtime for the checkered implementation was 1.600s. The first chart in Figure 3 has file I/O in orange, MPI calls in blue, and floating point in green. The x-axis is time and the y-axis is percent of the program used by resource. The file I/O dominated the runtime, taking 84% of the total time. The MPI calls took the next most time with 14%, leaving floating point at around 1%. Clearly the file I/O is a huge time sink and will negatively impact the scaling of the program. The last two charts show detailed charts of floating point and memory usage, which are minimal in this application.

The next view displays which specific lines of code are taking all the time. Figure 4 shows the file I/O lines of the code and how much time they take.

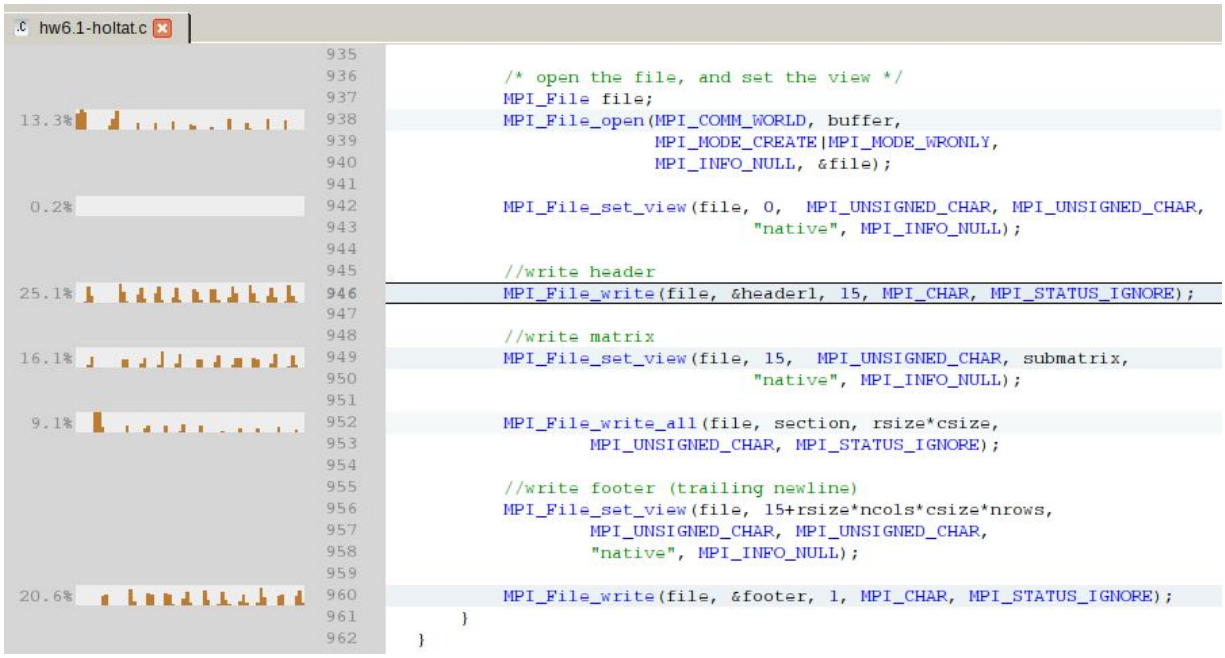

Figure 4: Code profiling for checkered with animation generation over 11 iterations.

Almost the entirety of the file I/O was spent in these lines of code (83% of 84%), with the other 1% going to the initial read and closing the files. Interestingly enough, writing the header and footer (16 bytes total) took almost 45% of the total time whereas writing the matrix to the file (262144 bytes) took only 15%. This clearly indicates that the header and footer calls are bottlenecks and should be optimized to improve performance. The rest of the file I/O was spent opening the file and setting views. One possible way to improve performance would be to use MPI_File_write_at to avoid setting views when writing the header and footer.

Figure 5 shows how the floating point operations and communication.

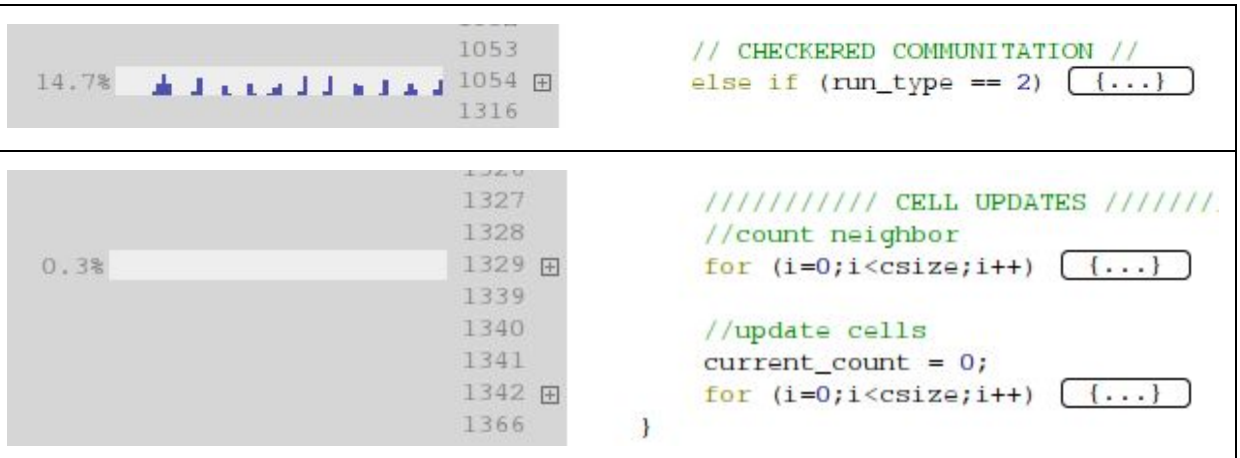

Figure 5: Floating point operations and communication breakdown for checkered with animation generation for 11 iterations.

In Figure 5 the code for the communication and computation areas have been compressed such that they can be summarized. Communication took ~15% of the total time and updating cells as well as counting neighbors took <1% of the total time.

Blocked with Animation Generation:

In theory, the blocked version should use less overall communication and the roughly the same amount of floating point operations, making it more efficient than the checkered version (The blocked version only has to send the top and bottom row instead of all four sides and the corners). In order to test this, the blocked version was profiled with Allinea as well. Figure 6 shows the chart view from this output.

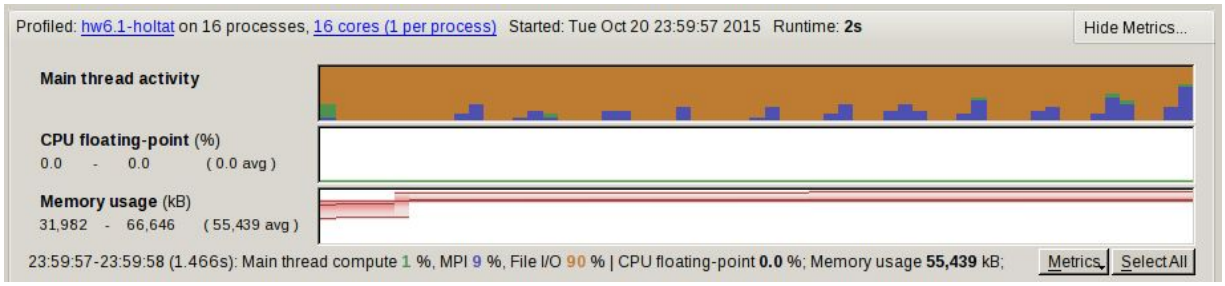

Figure 6: Charts for blocked version with animation generation for 11 iterations.

The total runtime for the blocked configuration was 1.466s, about 10% faster than the checkered configuration. While this result was expected, it should be noted that this code was run only once with 11 iterations. While 11 iterations gives some confidence about the results, it isn't enough to be fully confident. Code profiled with more iterations is discussed in the next section.

Overall Speed Performance Comparison:

In order to get a more accurate result of which version performed better, both implementations were run for more iterations. The first time they ran for 500 iterations and generated an PGM file on every iteration. The second time they ran for 10,000 iterations and no PGM's were generated. The results are shown in Table 1:

Table 1: Blocked Vs Checkered Overall Efficiency

	Blocked (s)	Checkered (s)
Animation (500 iterations)	306	331
No animation (10,000 iterations)	7.142	7.475

Table 1 confirms the expected result that the blocked implementation runs slightly faster than the checkered implementation. If a larger node count was used the blocked version might do even better. This is because the checkered version would have even more communication as more interior blocks would be needed. The interior blocks have to communicate with 8 other nodes, unlike the exterior ones which communicate with 3-6 other nodes. In the blocked version each node communicates with at most 2 other nodes.

Checkered without Animation:

In the runs which generated PGM files, almost the entirety of the program was dedicated to file I/O and MPI calls relating to image generation. In order to get an idea of how Conway's would scale without animation generation, it was profiled again. 50 iterations were used as there wasn't enough data for accurate results with 10 iterations (according to Allinea).

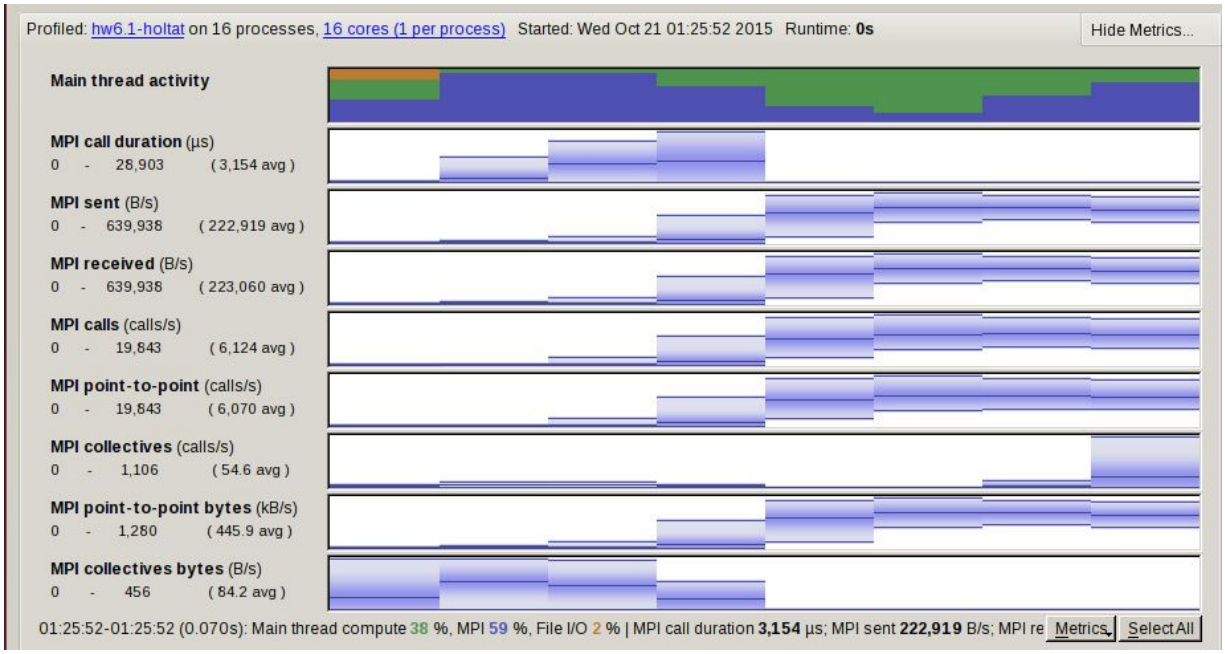

Figure 7: Charts for checkered with no image generation.

When image generation was removed, the program speed increased significantly. The 50 iteration run took 0.7 seconds, which means that each of the 8 blocks of time in Figure 7 represents ~0.09 seconds of runtime. Unlike the runs with image generation, almost no time is spent on file I/O, a significant amount of time is spent on computation (38%), and a majority of the time is spent on communication (59%).

The bottom 8 charts in Figure 7 show a more detailed view of all the MPI data collected during the run. For these charts, the horizontal axis represents time and the vertical axis represents variance between nodes. Thus if there is a block full with color, there was a large difference between the longest time it took a node to complete this task and the shortest time. An example of this is seen in Figure 8.

Figure 8: Call duration for checkered with no image generation.

The fourth block in Figure 8 has high variance as it is all the way filled in. The shortest call duration was 0us on rank 12 (too short to be measured by Allinea) and the longest was 28,903us on rank3. High variance typically means that one or more nodes are spending significantly more time on a task than the others. It's likely that one process is having to wait to significantly longer receive its data, but it is difficult to say. Going through with the Allinea debugger may lead to hints as to why there is such variance.

Figure 9 below zooms in on the initialization period of the program.

Figure 9: Initialization, partitioning, and data ingest for checkered with no image generation. Orange represents file I/O, green is floating point, and blue is MPI calls.

The initialization, partitioning, and data ingest took so little time that it all fit into one block. Even then, the MPI calls in blue are actually mostly from the main loop of the program. This shows one of the shortcomings of the Allinea profiler: it has a relatively infrequent sampling rate which limits the detail of the profiling (although apparently you're supposed to use their debugger for this sort of work).

There was only one global communication in the entire program (for parallel versions not generating images). It was an MPI_All_reduce on an integer that took so little time that it registered as taking 0% of the total time. The majority of the communication time was taken by MPI_Recv calls as seen in Figure 10.

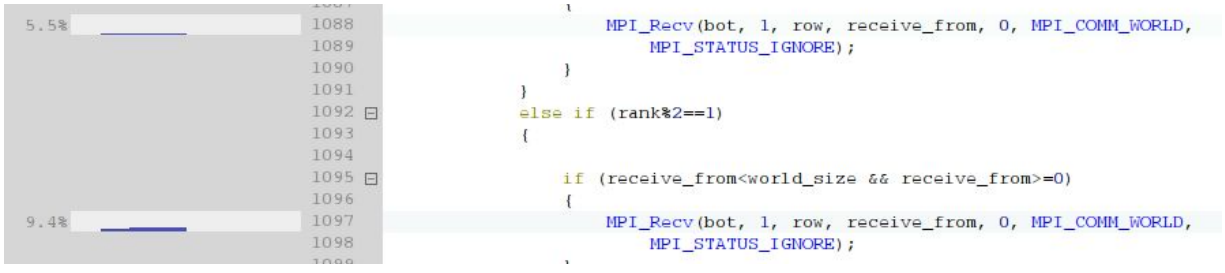

Figure 10: Example MPI_Recv calls and the percentage of the time it took to complete them. (Checkered no animation generation).

There were 16 total MPI_Recv calls which averaged about 3% of the total runtime per call. There was a high variance between these with some taking as much as 9.4% and some taking as little as 0.8% of the total time. The MPI_Send calls and the MPI_Finalize call effectively made up the rest of the communication time.

Blocked without Animation:

Figure 11 shows an overview of the blocked run. Overall the code spent 2% of the time in file I/O, 41% of the time on floating point, and 57% of the time on MPI calls. These results are nearly identical to the checkered version.

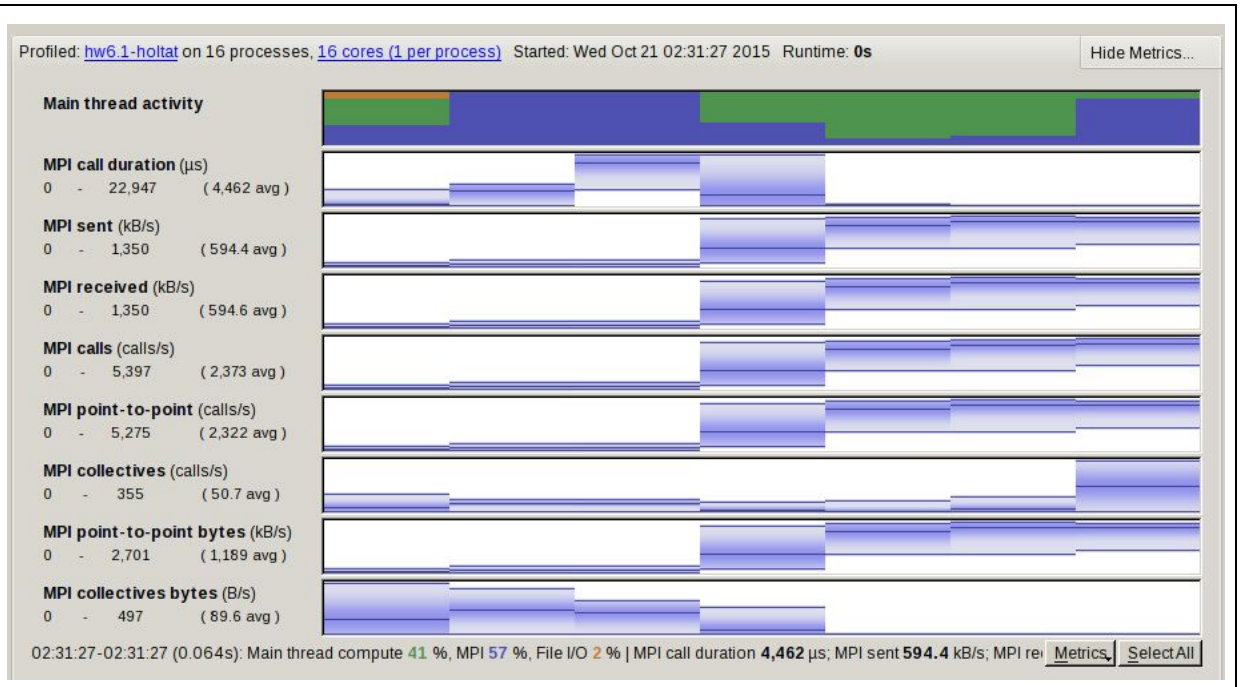

Figure 11: Charts for blocked version with no image generation.

The MPI data collected in Figure 11 is also similar to the the checkered MPI data from Figure 7. The main difference is that the MPI send and received rate is double what it was for the checkered version. In the blocked version larger arrays are sent using fewer communications than in the checkered version. Aside from this, the communication and computation patterns are similar.

There were 4 receive calls in the Blocked version of the code. Over the course of the whole run they took 8.7% of the total time on average. There was significant variance in the time each call took, which is detailed in Figures 12 and 13.

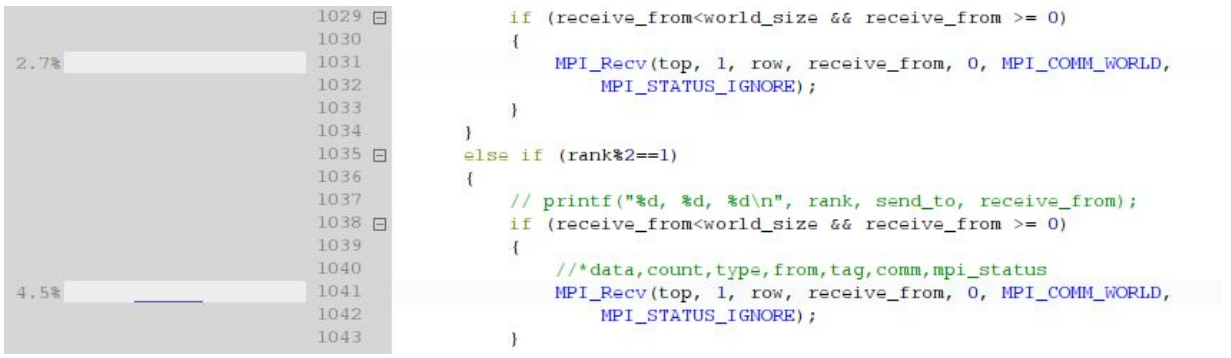

Figure 12: First two MPI_Recv calls for blocked communication with no animation generation.

The two MPI_Recv calls in Figure 12 took 2.7% and 4.5% of the total time of the program run.

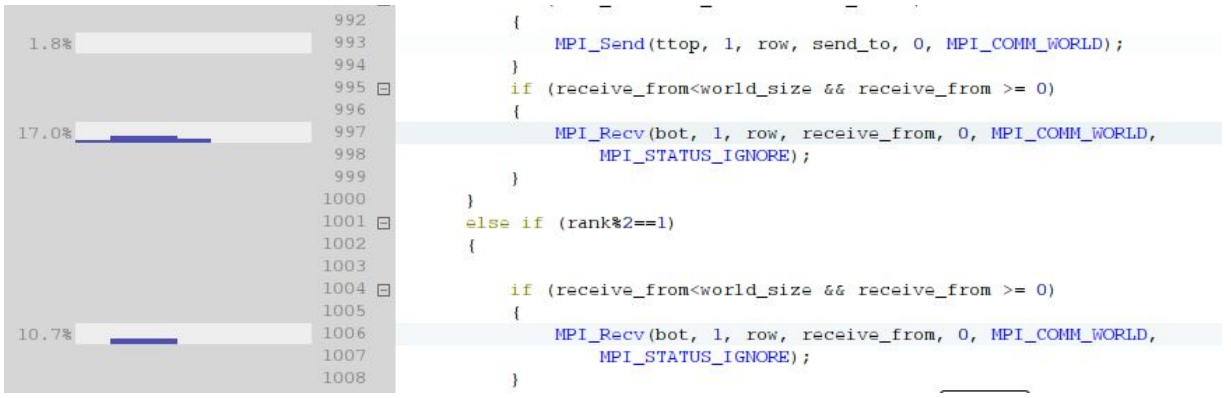

Figure 13: Second set of MPI_Recv calls for blocked communication with no animation generation.

The two MPI_Recv calls in Figure 13 took 17.0% and 10.7% of the total time respectively. All of the MPI_Recv calls in the blocked version are receiving the same datatype of the same size. The MPI_Send calls took very little time which suggests that the processes are out of sync and some processes are waiting a significant amount of time to receive data. This theory is supported by Figure 11 there is high variance in the MPI call duration on the third and fourth blocks. Reordering the MPI_Send's and MPI_Recv's could improve performance.

Command Line Output:

```
[holtat@login02 CONWAY_2]$ python checkered_submit.py
sbatch SCRIPTS/checkered.sh
mpirun -np 16 ./hw6.1-holtat -v -a 2 51 50 1-50
[holtat@login02 CONWAY_2]$ cat RESULTS/checkered.txt
Verbose=1, RunType=2, Iterations=51, CountWhen=50, Animation=1
[ -1:000]                : 7788 total buggies
rsize,csize,NP = 128, 128, 16
Iteration=    0,  Count=  7788
Iteration=   50,  Count=  6379
[holtat@login02 CONWAY_2]$
[holtat@login02 CONWAY_2]$ python checkered_submit.py
sbatch SCRIPTS/checkered.sh
mpirun -np 16 ./hw6.1-holtat -v -a 2 1001 1000 1,2,3,4-6
[holtat@login02 CONWAY_2]$ cat RESULTS/checkered.txt
Verbose=1, RunType=2, Iterations=1001, CountWhen=1000, Animation=1
[ -1:000]                : 7788 total buggies
rsize,csize,NP = 128, 128, 16
Iteration=    0,  Count=  7788
Iteration= 1000,  Count=  5526
[holtat@login02 CONWAY_2]$
[holtat@login02 CONWAY_2]$ python checkered_submit.py
sbatch SCRIPTS/checkered.sh
mpirun -np 16 ./hw6.1-holtat -v 2 1001 1000
[holtat@login02 CONWAY_2]$ cat RESULTS/checkered.txt
Verbose=1, RunType=2, Iterations=1001, CountWhen=1000, Animation=0
[ -1:000]                : 7788 total buggies
rsize,csize,NP = 128, 128, 16
Iteration=    0,  Count=  7788
Iteration= 1000,  Count=  5526
[holtat@login02 CONWAY_2]$
```

Figure 14: Command line output for checkered runs. The first run was for 51 iteration with an animation generated at every iteration. The second run was for 1000 iterations with frames generated at iterations 1,2,3,4-6. The third run had no animation generation.

```
[holtat@login02 CONWAY_2]$ python blocked_submit.py
sbatch SCRIPTS/blocked.sh
mpirun -np 16 ./hw6.1-holtat -v -a 1 51 50 1-50
[holtat@login02 CONWAY_2]$ cat RESULTS/blocked.txt
Verbose=1, RunType=1, Iterations=51, CountWhen=50, Animation=1
[ -1:000]                : 7788 total buggies
rsize,csize,NP = 512, 32, 16
Iteration=    0,  Count=  7788
Iteration=   50,  Count=  6379
[holtat@login02 CONWAY_2]$
[holtat@login02 CONWAY_2]$ python blocked_submit.py
sbatch SCRIPTS/blocked.sh
mpirun -np 16 ./hw6.1-holtat -v -a 1 1001 1000 1,2,3,4-6
[holtat@login02 CONWAY_2]$ cat RESULTS/blocked.txt
Verbose=1, RunType=1, Iterations=1001, CountWhen=1000, Animation=1
[ -1:000]                : 7788 total buggies
rsize,csize,NP = 512, 32, 16
Iteration=    0,  Count=  7788
Iteration= 1000,  Count=  5526
[holtat@login02 CONWAY_2]$
[holtat@login02 CONWAY_2]$ python blocked_submit.py
sbatch SCRIPTS/blocked.sh
mpirun -np 16 ./hw6.1-holtat -v 1 1001 1000
[holtat@login02 CONWAY_2]$ cat RESULTS/blocked.txt
Verbose=1, RunType=1, Iterations=1001, CountWhen=1000, Animation=0
[ -1:000]                : 7788 total buggies
rsize,csize,NP = 512, 32, 16
Iteration=    0,  Count=  7788
Iteration= 1000,  Count=  5526
[holtat@login02 CONWAY_2]$
```

Figure 15: Command line output for blocked runs. The first run was for 51 iteration with an animation generated at every iteration. The second run was for 1000 iterations with frames generated at iterations 1,2,3,4-6. The third run had no animation generation.

```c
#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);
}
```