Aaron Holt
HPSC Conway's Part 1

High Level Program Structure Overview:

A high level overview of the code is shown below:

Parse Arguments
Initialize MPI
Read in PGM file
Dynamically allocate space

For every iteration:
    Count buggies
    Generate PGM for animation
    Communicate with MPI
    Update cells

Finalize MPI

Program Structure: Input and Data Structure

The provided C file pgm.c was used to read in the required pgm file. Every process received a full copy of the initial matrix and then picked out their own part. Matrices were dynamically allocated using malloc to account for different input file sizes.

One change from the given code was the deletion of an allocated 'ghost row' around the individual process' matrix. Instead rows, columns, and corners were allocated to account for the 'ghost rows'. They were named 'top', 'topright', 'right', etc. This made reasoning about communication easier later as you could MPI_Send(right…) and MPI_Recv(left…). This also alleviated the need for additional derived datatypes. Only a row datatype was created as both rows and columns were allocated as a 1d array.

Communication:

Communication was done by alternating which processes were receiving and which ones were sending. In this way deadlocking was avoided. The general pattern is seen below in pseudocode:

```
//Send top, receive bot
calculate_where_to_send()
calculate_who_to_receive_from()
if odd process:
    send(top_row)
    receive(bot_row)
else if even process:
    receive(bot_row)
    send(top_row)
```

The amount of communication varied by run type. For the serial program, no communication was needed. For the blocked implementation, only two rounds of communication were needed: one for sending the top row, and one for sending the bot row. For the checkered communication eight rounds of communication were needed: one for each side and one for each corner.

Updating Cells:

Cells were updated AFTER communication had completed. In this way all the points gathered from other matrices were available. A function called ‘count_neighbors’ looped over every point in the local matrix and counted how many neighboring cells were alive. This data was saved in a matrix called ‘neighbors’. Using the updated neighbors matrix, each cell in the local matrix was updated according to the rules.

Generating an Animation:

At the beginning of each iteration, MPI_Reduce was used to collect all the matrices onto process 0. Process 0 then called ‘write_matrix_to_pgm’ which wrote the current matrix to a file in an animation directory. Then, ImageMagick’s ‘convert’ function was used to join all the images into a GIF.

```
aaron@AaronPC:~/Documents/MPI/CONWAY$ make
mpicc -g -Wall -lm -c -o pgm.o pgm.c
mpicc -g -Wall -lm -c -o pprintf.o pprintf.c
mpicc -g -Wall -lm -c -o hw5.1-holtat.o hw5.1-holtat.c
mpicc -o hw5.1-holtat hw5.1-holtat.o pgm.o pprintf.o -lm
aaron@AaronPC:~/Documents/MPI/CONWAY$ mpiexec -np 1 ./hw5.1-holtat -v 0 1001 100
Verbose=3, RunType=0, Iterations=1001, CountWhen=100
[ -1:000]               : life.pgm: P5 900 900 255
[ -1:001]               : 25301 total buggies
rsize,csize,NP = 900, 900, 1
Iteration=    0,Count= 25301
Iteration=  100,Count= 23284
Iteration=  200,Count= 22557
Iteration=  300,Count= 22241
Iteration=  400,Count= 21197
Iteration=  500,Count= 20932
Iteration=  600,Count= 20341
Iteration=  700,Count= 19792
Iteration=  800,Count= 19402
Iteration=  900,Count= 19261
Iteration= 1000,Count= 18340
aaron@AaronPC:~/Documents/MPI/CONWAY$ mpiexec -np 1 ./hw5.1-holtat -v 0 10001 1000
Verbose=3, RunType=0, Iterations=10001, CountWhen=1000
[ -1:000]               : life.pgm: P5 900 900 255
[ -1:001]               : 25301 total buggies
rsize,csize,NP = 900, 900, 1
Iteration=    0,Count= 25301
Iteration= 1000,Count= 18340
Iteration= 2000,Count= 16512
Iteration= 3000,Count= 16001
Iteration= 4000,Count= 15449
Iteration= 5000,Count= 14953
Iteration= 6000,Count= 14953
Iteration= 7000,Count= 14953
Iteration= 8000,Count= 14953
Iteration= 9000,Count= 14953
Iteration=10000,Count= 14953
aaron@AaronPC:~/Documents/MPI/CONWAY$
```

Figure 1: Output for NP=1 for iterations=1,001 and 10,001. The extra iteration made it so the previous one printed.

```
aaron@AaronPC:~/Documents/MPI/CONWAY$ mpiexec -np 9 ./hw5.1-holtat -v 1 1001 200
Verbose=3, RunType=1, Iterations=1001, CountWhen=200
[ -1:000]                : 25301 total buggies
rsize,csize,NP = 900, 100, 9
Iteration=    0,Count= 25301
Iteration=  200,Count= 22557
Iteration=  400,Count= 21197
Iteration=  600,Count= 20341
Iteration=  800,Count= 19402
Iteration= 1000,Count= 18340
aaron@AaronPC:~/Documents/MPI/CONWAY$ mpiexec -np 25 ./hw5.1-holtat -v 1 1001 200
Verbose=3, RunType=1, Iterations=1001, CountWhen=200
[ -1:000]                : 25301 total buggies
rsize,csize,NP = 900, 36, 25
Iteration=    0,Count= 25301
Iteration=  200,Count= 22557
Iteration=  400,Count= 21197
Iteration=  600,Count= 20341
Iteration=  800,Count= 19402
Iteration= 1000,Count= 18340
aaron@AaronPC:~/Documents/MPI/CONWAY$ mpiexec -np 36 ./hw5.1-holtat -v 1 1001 200
Verbose=3, RunType=1, Iterations=1001, CountWhen=200
[ -1:000]                : 25301 total buggies
rsize,csize,NP = 900, 25, 36
Iteration=    0,Count= 25301
Iteration=  200,Count= 22557
Iteration=  400,Count= 21197
Iteration=  600,Count= 20341
Iteration=  800,Count= 19402
Iteration= 1000,Count= 18340
aaron@AaronPC:~/Documents/MPI/CONWAY$ mpiexec -np 4 ./hw5.1-holtat -v 1 1001 200
Verbose=3, RunType=1, Iterations=1001, CountWhen=200
[ -1:000]                : 25301 total buggies
rsize,csize,NP = 900, 225, 4
Iteration=    0,Count= 25301
Iteration=  200,Count= 22557
Iteration=  400,Count= 21197
Iteration=  600,Count= 20341
Iteration=  800,Count= 19402
Iteration= 1000,Count= 18340
aaron@AaronPC:~/Documents/MPI/CONWAY$
```

Figure 2: Outputs for NP={4,9,25,36} for the blocked version up to 1,000 iterations. The buggie count matches for each processor count.

```
aaron@AaronPC:~/Documents/MPI/CONWAY$ mpiexec -np 4 ./hw5.1-holtat -v 1 10001 1000
Verbose=3, RunType=1, Iterations=10001, CountWhen=1000
[ -1:000]                : 25301 total buggies
rsize,csize,NP = 900, 225, 4
Iteration=    0,Count= 25301
Iteration= 1000,Count= 18340
Iteration= 2000,Count= 16512
Iteration= 3000,Count= 16001
Iteration= 4000,Count= 15449
Iteration= 5000,Count= 14953
Iteration= 6000,Count= 14953
Iteration= 7000,Count= 14953
Iteration= 8000,Count= 14953
Iteration= 9000,Count= 14953
Iteration=10000,Count= 14953
aaron@AaronPC:~/Documents/MPI/CONWAY$ mpiexec -np 36 ./hw5.1-holtat -v 1 10001 1000
Verbose=3, RunType=1, Iterations=10001, CountWhen=1000
[ -1:000]                : 25301 total buggies
rsize,csize,NP = 900, 25, 36
Iteration=    0,Count= 25301
Iteration= 1000,Count= 18340
Iteration= 2000,Count= 16512
Iteration= 3000,Count= 16001
Iteration= 4000,Count= 15449
Iteration= 5000,Count= 14953
Iteration= 6000,Count= 14953
Iteration= 7000,Count= 14953
Iteration= 8000,Count= 14953
Iteration= 9000,Count= 14953
Iteration=10000,Count= 14953
aaron@AaronPC:~/Documents/MPI/CONWAY$ mpiexec -np 9 ./hw5.1-holtat -v 1 10001 2000
Verbose=3, RunType=1, Iterations=10001, CountWhen=2000
[ -1:000]                : 25301 total buggies
rsize,csize,NP = 900, 100, 9
Iteration=    0,Count= 25301
Iteration= 2000,Count= 16512
Iteration= 4000,Count= 15449
Iteration= 6000,Count= 14953
Iteration= 8000,Count= 14953
Iteration=10000,Count= 14953
aaron@AaronPC:~/Documents/MPI/CONWAY$
```

Figure 3: Output for NP={4,36,9} for 10,000 iterations for blocked version.

```
aaron@AaronPC:~/Documents/MPI/CONWAY$ mpiexec -np 36 ./hw5.1-holtat -v 2 1001 100
Verbose=1, RunType=2, Iterations=1001, CountWhen=100, Animation=32591
[ -1:000]                : 25301 total buggies
rsize,csize,NP = 150, 150, 36
Iteration=    0,  Count= 25301
Iteration=  100,  Count= 23284
Iteration=  200,  Count= 22557
Iteration=  300,  Count= 22241
Iteration=  400,  Count= 21197
Iteration=  500,  Count= 20932
Iteration=  600,  Count= 20341
Iteration=  700,  Count= 19792
Iteration=  800,  Count= 19402
Iteration=  900,  Count= 19261
Iteration= 1000,  Count= 18340
aaron@AaronPC:~/Documents/MPI/CONWAY$ mpiexec -np 25 ./hw5.1-holtat -v 2 1001 100
Verbose=1, RunType=2, Iterations=1001, CountWhen=100, Animation=32678
[ -1:000]                : 25301 total buggies
rsize,csize,NP = 180, 180, 25
Iteration=    0,  Count= 25301
Iteration=  100,  Count= 23284
Iteration=  200,  Count= 22557
Iteration=  300,  Count= 22241
Iteration=  400,  Count= 21197
Iteration=  500,  Count= 20932
Iteration=  600,  Count= 20341
Iteration=  700,  Count= 19792
Iteration=  800,  Count= 19402
Iteration=  900,  Count= 19261
Iteration= 1000,  Count= 18340
aaron@AaronPC:~/Documents/MPI/CONWAY$ mpiexec -np 16 ./hw5.1-holtat -v 2 1001 200
Verbose=1, RunType=2, Iterations=1001, CountWhen=200, Animation=32573
[ -1:000]                : 25301 total buggies
rsize,csize,NP = 225, 225, 16
Iteration=    0,  Count= 25301
Iteration=  200,  Count= 22557
Iteration=  400,  Count= 21197
Iteration=  600,  Count= 20341
Iteration=  800,  Count= 19402
Iteration= 1000,  Count= 18340
aaron@AaronPC:~/Documents/MPI/CONWAY$
```

Figure 4: Output for NP={36,25,16} for checkered version with 1,000. Ouptuts match with different processor counts.

```
aaron@AaronPC:~/Documents/MPI/CONWAY$ mpiexec -np 9 ./hw5.1-holtat -v 2 10001 1000
Verbose=1, RunType=2, Iterations=10001, CountWhen=1000, Animation=32543
[ -1:000]                : 25301 total buggies
rsize,csize,NP = 300, 300, 9
Iteration=    0,  Count= 25301
Iteration= 1000,  Count= 18340
Iteration= 2000,  Count= 16512
Iteration= 3000,  Count= 16001
Iteration= 4000,  Count= 15449
Iteration= 5000,  Count= 14953
Iteration= 6000,  Count= 14953
Iteration= 7000,  Count= 14953
Iteration= 8000,  Count= 14953
Iteration= 9000,  Count= 14953
Iteration=10000,  Count= 14953
aaron@AaronPC:~/Documents/MPI/CONWAY$ mpiexec -np 4 ./hw5.1-holtat -v 2 10001 1000
Verbose=1, RunType=2, Iterations=10001, CountWhen=1000, Animation=32699
[ -1:000]                : 25301 total buggies
rsize,csize,NP = 450, 450, 4
Iteration=    0,  Count= 25301
Iteration= 1000,  Count= 18340
Iteration= 2000,  Count= 16512
Iteration= 3000,  Count= 16001
Iteration= 4000,  Count= 15449
Iteration= 5000,  Count= 14953
Iteration= 6000,  Count= 14953
Iteration= 7000,  Count= 14953
Iteration= 8000,  Count= 14953
Iteration= 9000,  Count= 14953
Iteration=10000,  Count= 14953
aaron@AaronPC:~/Documents/MPI/CONWAY$
```

Figure 5: Output for NP={4,9} for checkered version with 10,000 iterations.

```
#include "stdlib.h"
#include "argp.h"
#include "mpi.h"
#include "stdio.h"
#include "math.h"
#include "string.h"
#include "unistd.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 1
// 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
printwhen = print buggie count at printwhen interval
*/



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  CountOnMultipleOfN";

/* 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[3];
    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 >= 4)
            /* 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,
                 int* 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[9], int* section,
            int* top, int* bot, int* left, int* 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 = info[5];
    topright = info[6];
    botleft = info[7];
    botright = info[8];

    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, int* 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, int* 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) {}

//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;
}

//counters
int i,j,k,x,y;

// 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("life.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 );
}


//Communicators used in checkered
//row communicator
MPI_Comm my_row_comm;
MPI_Comm_split(MPI_COMM_WORLD, my_row, rank, &my_row_comm);

//column communicator
MPI_Comm my_col_comm;
MPI_Comm_split(MPI_COMM_WORLD, my_col, rank, &my_col_comm);

// 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);
}


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(int), MPI_INT, &column);
MPI_Type_commit(&column);

//Create new row derived datatype
MPI_Datatype row;
//count, blocklength, stride, oldtype, *newtype
MPI_Type_hvector(rsize, 1, sizeof(int), MPI_INT, &row);
MPI_Type_commit(&row);
```

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

//corners
int topleft,topright,botleft,botright; //used in calculations
int 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
int* full_matrix;
int* full_matrix_buffer;
if (animation == 1)
{
    int msize1 = rsize*ncols*csize*nrows;
    full_matrix = (int*)malloc(msize1*sizeof(int));
    full_matrix_buffer = (int*)malloc(msize1*sizeof(int));
    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_heig
ht,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_INT, MPI_SUM, MPI_COMM_WORLD );
    // if (rank == 0)
    // {
    //     printf("COUNT 4 = %d\n", total);
    // }

}


// /////// CUSTOM INPUITS /////////////////
// if (world_size == 0 && run_type == 0)
// {
//     //blinker
//     int offset = 0;
//     section[rsize+offset] = 0;
//     section[rsize+1+offset] = 0;
//     section[rsize+2+offset] = 0;

//     //block
//     int offset2 = 5;
//     section[offset2] = 0;
//     section[offset2+1] = 0;
//     section[rsize+offset2] = 0;
//     section[rsize+offset2+1] = 0;
// }
// else if (world_size>0 && run_type == 1)
```

```
// {
//     //blinker
//     int offset = 0;
//     if (rank == 2)
//     {
//         section[offset] = 0;
//         section[offset+1] = 0;
//         section[offset+2] = 0;
//     }

//     //block
//     int offset2 = 5;
//     if (rank == 1)
//     {
//         section[rsize*(csize-1)+offset2] = 0;
//         section[rsize*(csize-1)+offset2+1] = 0;
//     }
//     if (rank == 2)
//     {
//         section[offset2] = 0;
//         section[offset2+1] = 0;
//     }

// }
// else if (world_size>0 && run_type == 2)
// {
//     //corners for np=4
//     if (rank == 0)
//     {
//         section[rsize*csize-1] = 0;
//     }
//     else if (rank == 1)
//     {
//         section[rsize*(csize-1)] = 0;
//     }
//     else if (rank == 2)
//     {
//         section[rsize-1] = 0;
//     }
//     else if (rank == 3)
//     {
//         section[0] = 0;
//     }


//     //left/right debug
//     if (rank == 2)
//     {
```

```
//          section[rsize*3+rsize-1] = 0;
//          section[rsize*4+rsize-1] = 0;
//      }
//      else if (rank == 3)
//      {
//          section[0+rsize*3] = 0;
//          section[0+rsize*4] = 0;
//      }
// }



int send_to;
int receive_from;
int info[9];
info[2] = rank;
info[3] = rsize;
info[4] = csize;
info[5] = topleft;
info[6] = topright;
info[7] = botleft;
info[8] = 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_COMM_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, topright, botleft, botright);
        }
    }
```

```
//If animation is requested
if (animation == 1)
{
    //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_INT,
        MPI_SUM, 0, MPI_COMM_WORLD);

    //Write to file
    if (rank == 0)
    {
        write_matrix_to_pgm(k, rsize*ncols, csize*nrows, full_matrix);
    }
}


// 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/nrows==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/nrows==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/nrows==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/nrows==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_INT, send_to, 0, MPI_COMM_WORLD);
    }
    if (receive_from<world_size && receive_from>=0 && receive_from/nrows==my_row+1)
    {
        MPI_Recv(&botleft, 1, MPI_INT, receive_from, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    }
}
else if (rank%2==1)
{
    if (receive_from<world_size && receive_from>=0 && receive_from/nrows==my_row+1)
    {
        MPI_Recv(&botleft, 1, MPI_INT, 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(&ttopright, 1, MPI_INT, 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_INT, send_to, 0, MPI_COMM_WORLD);
    }
    if (receive_from<world_size && receive_from>=0 && receive_from/nrows==my_row+1)
    {
        MPI_Recv(&botright, 1, MPI_INT, receive_from, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    }
}
else if (rank%2==1)
{
    if (receive_from<world_size && receive_from>=0 && receive_from/nrows==my_row+1)
    {
        MPI_Recv(&botright, 1, MPI_INT, 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_INT, 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_INT, send_to, 0, MPI_COMM_WORLD);
    }
    if (receive_from<world_size && receive_from>=0 && receive_from/nrows==my_row-1)
    {
        MPI_Recv(&topright, 1, MPI_INT, receive_from, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    }
}
else if (rank%2==1)
{
    if (receive_from<world_size && receive_from>=0 && receive_from/nrows==my_row-1)
```

```
            {
                MPI_Recv(&topright, 1, MPI_INT, 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_INT, 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_INT, send_to, 0, MPI_COMM_WORLD);
            }
            if (receive_from<world_size && receive_from>=0 && receive_from/nrows==my_row-1)
            {
                MPI_Recv(&topleft, 1, MPI_INT, receive_from, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
            }
        }
        else if (rank%2==1)
        {
            if (receive_from<world_size && receive_from>=0 && receive_from/nrows==my_row-1)
            {
                MPI_Recv(&topleft, 1, MPI_INT, 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(&tbotright, 1, MPI_INT, send_to, 0, MPI_COMM_WORLD);
            }
        }


        info[5] = topleft;
        info[6] = topright;
        info[7] = botleft;
        info[8] = 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+rsize);



////////// 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, 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);
}
```

```
// System includes
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

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

typedef enum { false, true } bool; // Provide C++ style 'bool' type in C

bool readpgm( char *filename )
{
  // Read a PGM file into the local task
  //
  // Input: char *filename, name of file to read
  // Returns: True if file read successfully, False otherwise
  //
  // Preconditions:
  //  * global variables nrows, ncols, my_row, my_col must be set
  //
  // Side effects:
  //  * sets global variables local_width, local_height to local game size
  //  * sets global variables field_width, field_height to local field size
  //  * allocates global variables field_a and field_b

  // pp_set_banner( "pgm:readpgm" );

  // Open the file
  // if( rank==0 )
  //   pprintf( "Opening file %s\n", filename );
  FILE *fp = fopen( filename, "r" );
  if( !fp )
  {
    pprintf( "Error: The file '%s' could not be opened.\n", filename );
    return false;
  }

  // Read the PGM header, which looks like this:
  //  |P5        magic version number
  //  |900 900       width height
  //  |255         depth
  char header[10];
  int width, height, depth;
  int rv = fscanf( fp, "%6s\n%i %i\n%i\n", header, &width, &height, &depth );
  if( rv != 4 )
  {
```

```
  if(rank==0)
    pprintf( "Error: The file '%s' did not have a valid PGM header\n",
      filename );
  return false;
}
// if( rank==0 )
//   pprintf( "%s: %s %i %i %i\n", filename, header, width, height, depth );



// Make sure the header is valid
if( strcmp( header, "P5") )
{
  if(rank==0)
    pprintf( "Error: PGM file is not a valid P5 pixmap.\n" );
  return false;
}
if( depth != 255 )
{
  if(rank==0)
    pprintf( "Error: PGM file has depth=%i, require depth=255 \n",
      depth );
  return false;
}



// Make sure that the width and height are divisible by the number of
// processors in x and y directions
// printf("WR=%d,%d,%d,HERE1\n",rank,width,ncols);
if( width % ncols )
{
  if( rank==0 )
    pprintf( "Error: %i pixel width cannot be divided into %i cols\n",
        width, ncols );
  return false;
}

// printf("WR=%d,HERE3\n",rank);
if( height % nrows )
{
  if( rank==0 )
    pprintf( "Error: %i pixel height cannot be divided into %i rows\n",
        height, nrows );
  return false;
}
// printf("WR=%d,PAST\n",rank);
```

```
// Divide the total image among the local processors
local_width = width / ncols;
local_height = height / nrows;

// Find out where my starting range it
int start_x = local_width * my_col;
int start_y = local_height * my_row;

// pprintf( "Hosting data for x:%03i-%03i y:%03i-%03i\n",
//     start_x, start_x + local_width,
//     start_y, start_y + local_height );

// Create the array!
field_width = local_width + 2;
field_height = local_height + 2;
field_a = (int *)malloc( field_width * field_height * sizeof(int));
field_b = (int *)malloc( field_width * field_height * sizeof(int));

// Read the data from the file. Save the local data to the local array.
int b, ll, lx, ly, y, x;
for( y=0; y<height; y++ )
{
  for( x=0; x<width; x++ )
  {
    // Read the next character
    b = fgetc( fp );
    if( b == EOF )
    {
      pprintf( "Error: Encountered EOF at [%i,%i]\n", y,x );
      return false;
    }

    // From the PGM, black cells (b=0) are bugs, all other
    // cells are background
    if( b==0 )
    {
      b=1;
    }
    else
    {
      b=0;
    }


    // If the character is local, then save it!
    if( x >= start_x && x < start_x + local_width &&
      y >= start_y && y < start_y + local_height )
```

```
      {
        // Calculate the local pixels (+1 for ghost row,col)
        lx = x - start_x + 1;
        ly = y - start_y + 1;
        ll = (ly * field_width + lx );
        field_a[ ll ] = b;
        field_b[ ll ] = b;
      } // save local point

    } // for x
  } // for y

  fclose( fp );

  // pp_reset_banner();
  return true;
}
```

```
/* $Id: pprintf.c,v 1.5 2006/02/09 20:42:25 mccreary Exp $ */

/*
 * Copyright (c) 2006 Sean McCreary <mccreary@mcwest.org>. All rights
 * reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution.
 *
 * 3. The name of the author may not be used to endorse or promote products
 * derived from this software without specific prior written permission
 *
 * THIS SOFTWARE IS PROVIDED ''AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL
 * THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

/* Pretty printf() wrapper for MPI processes */

#include <stdio.h>
#include <stdarg.h>
#include <string.h>

#define PP_MAX_BANNER_LEN       14
#define PP_MAX_LINE_LEN         81
#define PP_PREFIX_LEN           27
#define PP_FORMAT               "[%3d:%03d] %-14s : "

static int pid = -1;
static int msgcount = 0;
static char banner[PP_MAX_BANNER_LEN] = "";
static char oldbanner[PP_MAX_BANNER_LEN] = "";
```

```
int init_pprintf(int);
int pp_set_banner(char *);
int pp_reset_banner();
int pprintf(char *, ...);

int init_pprintf( int my_rank )
{
    pp_set_banner("init_pprintf");
    pid = my_rank;
/*
    pprintf("PID is %d\n", pid);
*/
    return 0;
}

int pp_set_banner( char *newbanner )
{
    strncpy(oldbanner, banner, PP_MAX_BANNER_LEN);
    strncpy(banner, newbanner, PP_MAX_BANNER_LEN);
    return 0;
}

int pp_reset_banner()
{
    strncpy(banner, oldbanner, PP_MAX_BANNER_LEN);
    return 0;
}

int pprintf( char *format, ... )
{
    va_list ap;
    char output_line[PP_MAX_LINE_LEN];

    /* Construct prefix */
    snprintf(output_line, PP_PREFIX_LEN+1, PP_FORMAT, pid, msgcount, banner);

    va_start(ap, format);
    vsnprintf(output_line + PP_PREFIX_LEN,
                PP_MAX_LINE_LEN - PP_PREFIX_LEN, format, ap);
    va_end(ap);

    printf("%s", output_line);
    fflush(stdout);
    msgcount++;
    return 0;
}
```