***LAB 8-PARALLEL AND DISTRIBUTED COMPUTING***

***GROUP MEMBERS***

AISHWARYA.S 15BCB0055

GOUTHAMI REDDY 15BCB0061

Conway’s Game of Life is played on a rectangular grid of cells that may or may not contain an organism. The state of the cells is updated each time step by applying the following set of rules:

1. Every organism with two or three neighbours survives.
2. Every organism with four or more neighbours dies from overpopulation.
3. Every organism with zero or one neighbours dies from isolation.
4. Every empty cell adjacent to three organisms gives birth to a new one.

Create an MPI program that evolves a board of arbitrary size (dimensions could be specified at the command line) over several iterations. The board could be randomly generated or real from a file. Try applying the geometric decomposition pattern to partition the work among your process.

CODE:

#include <stdio.h> #include <strings.h> #include <stdlib.h> #include <time.h> #include <unistd.h> #include "mpi.h"

#define NUMBERROWS 28

#define esc 27

#define cls() printf("%c[2J",esc)

#define pos(row,col) printf("%c[%d;%dH",esc,row,col)

char \*DISH0[ NUMBERROWS ]; char \*DISH1[ NUMBERROWS ];

char \*PATTERN[NUMBERROWS] = {

" ",

" # ",

" # # ### ",

" ## ",

" ",

" # ",

" # # ",

" ## ",

" ",

" ",

" ",

" ",

" # ",

" # # ",

" ## ",

" ",

" ",

" ",

" ",

" ",

" ",

" ",

" ",

" ",

" ",

" ",

" ",

" "

};

int ROWSIZE = strlen( " ") + 1;

//------------------------------- prototypes --------------------------------

void life( char\*\*, char\*\*, int ); void initDishes( int );

void print( char \*\*, int );

// --------------------------------------------------------------------------

// initDishes

// inits the dishes (current and future) void initDishes( int rank ) {

int i;

//--- initialize other dish with spaces. Make it same dimension as DISH0. --- for (i = 0; i< NUMBERROWS; i++ ) {

DISH0[i] = (char \*) malloc( ( strlen( PATTERN[0] ) + 1 ) \* sizeof( char ) ); strcpy( DISH0[i], PATTERN[i] );

DISH1[i] = (char \*) malloc( (strlen( DISH0[0] )+1) \* sizeof( char ) ); strcpy( DISH1[i], PATTERN[i] );

}

}

// --------------------------------------------------------------------------

// initDishes2

// inits the dishes (current and future)

// (Buggy: attempts to declare only 1 half of the array, plus boundary

// rows, depending on rank. Needs a bit more debugging...) void initDishes2( int rank ) {

int i;

// init to null all entries. This way we'll be

// able to tell if a row belongs to us or not. for ( i=0; i<NUMBERROWS; i++ ) { DISH0[i] = NULL;

DISH1[i] = NULL;

}

//--- Init RANK 0 rows --- if ( rank == 0 ) {

//--- initialize dishes with lower half of pattern --- for (i = 0; i< NUMBERROWS/2+1; i++ ) {

DISH0[i] = (char \*) malloc( (strlen( PATTERN[0] ) + 1 ) \* sizeof( char ) ); strcpy( DISH0[i], PATTERN[i] );

DISH1[i] = (char \*) malloc( (strlen( DISH0[0] )+1) \* sizeof( char ) ); strcpy( DISH1[i], DISH0[i] );

}

//--- initialize top row of dishes, as they are neighbors of Row 0 ---

DISH0[NUMBERROWS-1] = (char \*) malloc( (strlen( PATTERN[0] ) + 1 ) \* sizeof( char ) ); strcpy( DISH0[NUMBERROWS-1], PATTERN[NUMBERROWS-1] );

DISH1[NUMBERROWS-1] = (char \*) malloc( (strlen( PATTERN[0] ) + 1 ) \* sizeof( char ) ); strcpy( DISH1[NUMBERROWS-1], PATTERN[NUMBERROWS-1] );

}

//--- Init RANK 1 rows --- if ( rank == 1 ) {

//--- initialize dishes with upper half of pattern ---

for (i = NUMBERROWS/2-1; i< NUMBERROWS; i++ ) {

DISH0[i] = (char \*) malloc( (strlen( PATTERN[0] ) + 1 ) \* sizeof( char ) ); strcpy( DISH0[i], PATTERN[i] );

DISH1[i] = (char \*) malloc( (strlen( DISH0[0] )+1) \* sizeof( char ) ); strcpy( DISH1[i], DISH0[i] );

}

//--- initialize bottom row of dishes, as they are neighbors of top row --- DISH0[0] = (char \*) malloc( (strlen( PATTERN[0] ) + 1 ) \* sizeof( char ) ); strcpy( DISH0[0], PATTERN[0] );

DISH1[0] = (char \*) malloc( (strlen( PATTERN[0] ) + 1 ) \* sizeof( char ) ); strcpy( DISH1[0], PATTERN[01] );

}

}

// --------------------------------------------------------------------------

// print

void print( char\* dish[], int rank ) { int i;

if ( rank == 0 ) {

//--- display lower half only ---

for (i=0; i<NUMBERROWS/2; i++ ) {

if ( dish[i] == NULL ) continue; pos( i, 0 );

printf( "%s\n", dish[i] );

}

}

if ( rank == 1 ) {

//--- display upper half only ---

for (i=NUMBERROWS/2; i<NUMBERROWS; i++ ) {

if ( dish[i] == NULL ) continue; pos( i, 0 );

printf( "%s\n", dish[i] );

}

}

}

// --------------------------------------------------------------------------

void check( char\*\* dish, char\*\* future ) { int i, j, k, l;

l = sizeof( dish )/sizeof( dish[0] ); printf( "length of dish = %d\n", l );

for ( i=0; i<l; i++ ) { k = strlen( dish[i] );

printf( "%d %s\n", k, dish[i] );

}

printf( "\n\n" );

l = sizeof( future )/sizeof( future[0] ); printf( "length of future = %d\n", l );

for ( i=0; i<l; i++ ) {

k = strlen( future[i] );

printf( "%d %s\n", k, future[i] );

}

printf( "\n\n" );

}

// --------------------------------------------------------------------------

void life( char\*\* dish, char\*\* newGen, int rank ) {

/\*

* Given an array of string representing the current population of cells
* in a petri dish, computes the new generation of cells according to
* the rules of the game. A new array of strings is returned.

\*/

int i, j, row;

int rowLength = strlen( dish[0] );

int dishLength = NUMBERROWS; int lowerRow, upperRow;

//--- slice the array into two halves. Rank 0 is lower half, ---

//-- Rank 1 is upper half. --- if ( rank == 0 ) {

lowerRow = 0;

upperRow = NUMBERROWS/2;

}

if ( rank == 1 ) {

lowerRow = NUMBERROWS/2; upperRow = NUMBERROWS;

}

for (row = lowerRow; row < upperRow; row++) {// each row if ( dish[row] == NULL )

continue;

for ( i = 0; i < rowLength; i++) { // each char in the

// row

int r, j, neighbors = 0;

char current = dish[row][i];

// loop in a block that is 3x3 around the current cell

// and count the number of '#' cells. for ( r = row - 1; r <= row + 1; r++) {

// make sure we wrap around from bottom to top int realr = r;

if (r == -1)

realr = dishLength - 1; if (r == dishLength) realr = 0;

for (int j = i - 1; j <= i + 1; j++) {

// make sure we wrap around from left to right int realj = j;

if (j == -1)

realj = rowLength - 1; if (j == rowLength) realj = 0;

if (r == row && j == i)

continue; // current cell is not its

// neighbor

if (dish[realr][realj] == '#') neighbors++;

}

}

if (current == '#') {

if (neighbors < 2 || neighbors > 3) newGen[row][i] = ' ';

else

newGen[row][i] = '#';

}

if (current == ' ') { if (neighbors == 3)

newGen[row][i] = '#'; else

newGen[row][i] = ' ';

}

}

}

}

// --------------------------------------------------------------------------

int main( int argc, char\* argv[] ) {

int gens = 3000; // # of generations int i;

char \*\*dish, \*\*future, \*\*temp;

//--- MPI Variables --- int noTasks = 0;

int rank = 0;

MPI\_Status status; // required variable for receive routines

//--- initialize MPI --- MPI\_Init( &argc, &argv );

//--- get number of tasks, and make sure it's 2 --- MPI\_Comm\_size( MPI\_COMM\_WORLD, &noTasks ); if ( noTasks != 2 ) {

printf( "Number of Processes/Tasks must be 2. Number = %d\n\n", noTasks ); MPI\_Finalize();

return 1;

}

//--- get rank ---

MPI\_Comm\_rank( MPI\_COMM\_WORLD, &rank );

//--- init the dishes as half of the original problem --- initDishes( rank );

dish = DISH0; future = DISH1;

//check( dish, future );

//--- clear screen --- cls();

print( dish, rank ); // # first generation, in petri dish

// iterate over all generations for ( i = 0; i < gens; i++) {

pos( 33+rank, 0 );

printf( "Rank %d: Generation %d\n", rank, i );

// apply the rules of life to the current population and

// generate the next generation. life( dish, future, rank );

// display the new generation

//print( dish, rank );

// add a bit of a delay to better see the visualization

// remove this part to get full timing.

//if (rank == 0 ) sleep( 1 );

if (rank==0 ) {

// buffer #items item-size src/dest tag world

MPI\_Send( future[ 0 ], ROWSIZE, MPI\_CHAR, 1, 0, MPI\_COMM\_WORLD ); MPI\_Send( future[NUMBERROWS/2-1], ROWSIZE, MPI\_CHAR, 1, 0,

MPI\_COMM\_WORLD );

MPI\_Recv( future[NUMBERROWS-1], ROWSIZE, MPI\_CHAR, 1, 0,

MPI\_COMM\_WORLD, &status );

MPI\_Recv( future[NUMBERROWS/2], ROWSIZE, MPI\_CHAR, 1, 0,

MPI\_COMM\_WORLD, &status );

}

if (rank==1 ) {

MPI\_Recv( future[ 0 ], ROWSIZE, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD,

&status );

MPI\_Recv( future[NUMBERROWS/2-1], ROWSIZE, MPI\_CHAR, 0, 0,

MPI\_COMM\_WORLD, &status );

MPI\_Send( future[NUMBERROWS-1], ROWSIZE, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD );

MPI\_Send( future[NUMBERROWS/2], ROWSIZE, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD );

}

// copy future to dish temp = dish;

dish = future; future = temp;

}

//--- display the last generation --- print(dish, rank);

//--- close MPI --- pos( 30+rank, 0 );

printf( "Process %d done. Exiting\n\n", rank ); MPI\_Finalize();

return 0;

}

COMPILATION:



OUTPUT:





