COMP 605: Parallel Batch Jobs
Homework 3: Conways Game of Life

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Due: 03/18/16
Posted: 03/02/16
Updated: 03/04/16
Table of Contents

1  HW 3, Conways Game of Life
   • Game of Life Programming Example: using basic arrays
   • Game of Life Programming Example: using structs and objects
   • Game of Life Using MPI Halo Exchange
   • HW #3: Instructions
     • Game of Life: Notes on the Rabbit Pattern
     • What to Report/Turn in
HW 3, Conways Game of Life

- Game of Life is a cellular automaton (CA).
- How to do it:
  - http://www.wikihow.com/Make-the-Conway’s-Game-of-Life-Cellular-Automaton
- Relevant links:
  - LifeLab: http://trevorrow.com/lifelab/index.html
- Parallel hints:
  - http://web.csc.fi/english/csc/courses/archive/material/prace-summer-school-material/MPI-tutorial
Life is played on an arbitrary-sized grid of square cells.
Each cell has two states: "dead" or "alive".
The state of every cell changes from one "generation" to the next according to the states of its 8 nearest neighbors:
- a dead cell becomes alive (a "birth") if it has exactly 3 live neighbors;
- a live cell dies out if it has less than 2 or more than 3 live neighbors.
The "game" of Life simply involves starting off with a pattern of live cells and watching it evolve.
Even though the rules for Life are completely deterministic, it is impossible to predict whether an arbitrary starting pattern will die out, or start oscillating, or fill the grid.
Life and other CAs provide a powerful demonstration of how a very simple system can generate extremely complicated behavior.

Source: http://trevorrow.com/lifelab/index.html
Conways Game of Life: using basic arrays

Conways Game of Life: Dreamincode src: Basic

```
#include <stdio.h>
#define HEIGHT 25
#define WIDTH 25
#define LIFE_YES 1
#define LIFE_NO 0

// make the rest of the function calls easier to read
typedef int TableType[HEIGHT][WIDTH];

void printTable(TableType table) {
    int height, width;

    for (height = 0; height < HEIGHT; height++) {
        for (width = 0; width < WIDTH; width++) {
            if (table[height][width] == LIFE_YES) {
                printf("X");
            } else {
                printf("-");
            }
        }
        printf("\n");
    }
    printf("\n");
}

// you already have a printTable, no need for this clear was a better name
void clearTable(TableType table) {
    int height, width;
    for (height = 0; height < HEIGHT; height++) {
        for (width = 0; width < WIDTH; width++)
            table[height][width] = LIFE_NO;
    }
}
```

Conways Game of Life: Dreamincode src: Basic

```c
int getNeighborValue(TableType table, int row, int col) {
    if (row < 0 || row >= HEIGHT
        || col < 0 || col >= WIDTH
        || table[row][col] != LIFE_YES )
    {
        return 0;
    } else {
        return 1;
    }
}

int getNeighborCount(TableType table, int row, int col) {
    int neighbor = 0;

    neighbor += getNeighborValue(table, row - 1, col - 1);
    neighbor += getNeighborValue(table, row - 1, col);
    neighbor += getNeighborValue(table, row - 1, col + 1);
    neighbor += getNeighborValue(table, row, col - 1);
    neighbor += getNeighborValue(table, row, col + 1);
    neighbor += getNeighborValue(table, row + 1, col - 1);
    neighbor += getNeighborValue(table, row + 1, col);
    neighbor += getNeighborValue(table, row + 1, col + 1);

    return neighbor;
}
```

Conways Game of Life: Dreamincode src: Basic

```
// code to load test data
void loadTestData(TableType table) {
    // toggle
    table[3][4] = LIFE_YES;
    table[3][5] = LIFE_YES;
    table[3][6] = LIFE_YES;

    // glider
    table[10][4] = LIFE_YES;
    table[10][5] = LIFE_YES;
    table[10][6] = LIFE_YES;
    table[11][6] = LIFE_YES;
    table[12][5] = LIFE_YES;
}

// Load Rabbits pattern:: (stabilizes at time 17331)
// A 9-cell {methuselah} found by Andrew Trevorrow in 1986.
// *...***
// ***..*.
// .*....
// place in center
void loadRabbitData(TableType table) {
    int r,c;
    r=HEIGHT/2;
    c=WIDTH/2;
    // rabbits
    table[r][c] = LIFE_YES;
    table[r][c+4] = LIFE_YES;
    table[r][c+5] = LIFE_YES;
    table[r][c+6] = LIFE_YES;
    table[r+1][c] = LIFE_YES;
    table[r+1][c+1] = LIFE_YES;
    table[r+1][c+2] = LIFE_YES;
    table[r+1][c+5] = LIFE_YES;
    table[r+2][c+1] = LIFE_YES;
}
```

void calculate(TableType tableA) {
    TableType tableB;
    int neighbor, height, width;

    for (height = 0; height < HEIGHT; height++) {
        for (width = 0; width < WIDTH; width++) {
            neighbor = getNeighborCount(tableA, height, width);
            // change this around to remove the ? : notation
            if (neighbor == 3) {
                tableB[height][width] = LIFE_YES;
            } else if (neighbor == 2 && tableA[height][width] == LIFE_YES) {
                tableB[height][width] = LIFE_YES;
            } else {
                tableB[height][width] = LIFE_NO;
            }
        }
    }

    // used to be swap
    for (height = 0; height < HEIGHT; height++) {
        for (width = 0; width < WIDTH; width++) {
            tableA[height][width] = tableB[height][width];
        }
    }
}
main(int argc, char *argv[]) {
    TableType table;
    char end;
    int stab, pop;
    int generation = 0;

    clearTable(table);
    //askUser(table);
    //loadTestData(table);
    loadGliderData(table);
    //loadRabbitData(table);
    //loadRpentominoData(table);
    printTable(table);

    do {
        calculate(table);
        printTable(table);
        printf("Generation %d\n", ++generation);
        printf("Press q to quit or 1 to continue: ");
        scanf(" %c", &end);
    } while (end != 'q');
    return 0;
}
Conways Game of Life: Dreamincode Output

Conways Game of Life: using structs and objects

#include <stdio.h>
#include <stdlib.h>

#define ROWS 20
#define COLS 20

#define GETCOL(c) (c%COLS)
#define GETROW(c) (c/COLS)

#define D_LEFT(c) ((GETCOL(c) == 0) ? (COLS-1) : -1)
#define D_RIGHT(c) ((GETCOL(c) == COLS-1) ? (-COLS+1) : 1)
#define D_TOP(c) ((GETROW(c) == 0) ? ((ROWS-1) * COLS) : -COLS)
#define D_BOTTOM(c) ((GETROW(c) == ROWS-1) ? (-(ROWS-1) * COLS) : COLS)

typedef struct _cell
{
    struct _cell *neighbour[8];
    char curr_state;
    char next_state;
} cell;

typedef struct
{
    int rows;
    int cols;
    cell* cells;
} world;

void evolve_cell(cell *c)
{
    int count=0, i;
    for (i=0; i<8; i++)
    {
        if (c->neighbour[i]->curr_state) count++;
    }
    if (count == 3 || (c->curr_state && count == 2)) c->next_state = 1;
    else c->next_state = 0;
}

void update_world(world *w)
{
    int nrcells = w->rows*w->cols, i;
    for (i=0; i<nrcells; i++)
    {
        evolve_cell(w->cells+i);
    }
    for (i=0; i<nrcells; i++)
    {
        w->cells[i].curr_state = w->cells[i].next_state;
        if (!(i%COLS)) printf("\n");
        printf("%c",w->cells[i].curr_state ? '#' : ' ');
    }
    printf("\n");
}

world* init_world()
{
    world* result = (world*) malloc(sizeof(world));
    result->rows = ROWS;
    result->cols = COLS;
    result->cells = (cell*) malloc(sizeof(cell) * COLS * ROWS);
    int nrcells = result->rows * result->cols, i;
    for (i = 0; i < nrcells; i++)
    {
        cell* c = result->cells + i;
        c->neighbour[0] = c + D_LEFT(i);
        c->neighbour[1] = c + D_RIGHT(i);
        c->neighbour[2] = c + D_TOP(i);
        c->neighbour[3] = c + D_BOTTOM(i);
        c->neighbour[4] = c + D_LEFT(i) + D_TOP(i);
        c->neighbour[5] = c + D_LEFT(i) + D_BOTTOM(i);
        c->neighbour[6] = c + D_RIGHT(i) + D_TOP(i);
        c->neighbour[7] = c + D_RIGHT(i) + D_BOTTOM(i);
        c->curr_state = rand() % 2;
    }
    return result;
}
int main()
{
    srand(3);
    world* w = init_world();

    while (1)
    {
        printf("\n=======================================\n");
        //system("cls");
        update_world(w);
        getchar();
    }
}
Conways Game of Life: using structs and objects

Source: http://cboard.cprogramming.com/c-programming/138126-recomposing-conway
GOL Programming Example: using structs and objects

Source: http://cboard.cprogramming.com/c-programming/138126-recomposing-conway
HW #3: Game of Life Using MPI Halo Exchange

Stencil computation in geometrically decomposed grids.

Source: F. B. Kjolstad and M. Snir, Ghost Cell Pattern, in Proceedings of the 2010 Workshop on Parallel Programming Patterns
Ghost Cell Layout: Each chunk receives a vector of ghost cells from neighboring chunks.

Source: F. B. Kjolstad and M. Snir, Ghost Cell Pattern, in Proceedings of the 2010 Workshop on Parallel Programming Patterns
HW #3: Deadlock-free Halo Exchange

```c
void exchange ? horizontal borders () {
    if (x coord % 2 == 0) {
        exchange east border ();
        exchange west border ();
    }
    else {
        exchange west border ();
        exchange east border ();
    }
}
```

*Figure 5: Deadlock-free border exchanges*

Source: F. B. Kjolstad and M. Snir, Ghost Cell Pattern, in Proceedings of the 2010 Workshop on Parallel Programming Patterns
Homework #3, Game of Life Instructions & Guidelines

- Write or choose a basic 2D game of life code in C.
- Choose one that can be parallelized.
- Define your stencil carefully
- Check that all cells are properly exchanged (e.g. corners)
  - Use block-block data distribution
  - Use halo/ghost exchange pattern
- We will run the "rabbit" test case described in the LifeLab website
  http://trevorrow.com/lifelab/index.html
Homework #3, LifeLab Rabbit Test Case

The initial pattern has 9 live cells. The male rabbit is on the left. :)
[scale = 8 pixels per cell]

After 1000 generations the pattern has expanded into a number of active regions.
The escaping glider on the right is about to be deleted.
[scale = 1 pixel per cell]

lifelab rabbit.

Source: http://trevorrow.com/lifelab/index.html
For the basic model:

- All key variables should be read via command line args (e.g. row & col size)
- You can use point-to-point or collective communications.
- Run jobs using the batch queue.
- Vary the number of PEs: \( N_p = [1, 4, 9, 16] \)
- Time the model: \( T_{\text{wall}} \) and any relevant critical blocks.
A few GOL Notes and observations

- General GOL Rules:
  - Any live cell with fewer than two live neighbors dies, as if caused by under-population.
  - Any live cell with two or three live neighbors lives on to the next generation.
  - Any live cell with more than three live neighbors dies, as if by over-population.
  - Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.
Game of Life: Patterns

- GOL has many different "patterns"
- You build patterns by setting cells on/off on the grid.
- Lexicon (list) of patterns:
  https://www.mathworks.com/moler/exm/exm/lexicon.txt
- some good examples:
Game of Life: Rabbit Pattern

- The Rabbit pattern is a Methuselah pattern
  - A methuselah is any 'small' pattern which takes a 'long time' to 'finish' or stabilize
  - http://www.conwaylife.com/wiki/Methuselah
  - stable: A pattern is said to be stable if it is a parent of itself.

- List of long lived Methuselahs:
Game of Life: Rabbit Pattern

- **Rabbits**: Initial pattern contains a "male" and a "female" rabbit, using total of 9 active cells.
- The initial pattern is the first generation.
- **Rabbits pattern**: A 9-cell Methuselah found by Andrew Trevorrow in 1986.

```
*...***
***..*.
.*.....
```

- **Predecessor (Trevorrow in October 1995)** has the same number of cells and lasts two generations longer.

```
..*.*
**
.*....
```

- **The rabbit pattern stabilizes** at 17331 generations, and a live population of 1744.
Game of Life: Hints on the rabbit pattern

- Run the model until the generations "stabilize": for rabbit pattern, live population = 1744, after 17331 generations.
- Observe live population # as a function of generation #: Hint: this will be affected by the board dimensions.
- Grid/Board: Game of Life based on infinite space
  - For rabbit pattern, you need to have enough generations to reach a live population of 1744. Hint: you only need to run at most 18,000 generations.
  - You need a grid of cells large enough to achieve stability for the Methuselah patterns.
- "Basic" game of life example – function below kills all life on the edges

```java
int getNeighborValue(TableType table, int row, int col) {
    if (row < 0 || row >= HEIGHT || col < 0 || col >= WIDTH ||
        table[row][col] != LIFE_YES )
    {
        return 0;
    } else {
        return 1;
    }
}
```
Game of Life: Small Test Case

The next slide contain snippets of the test case run below. You can find the full file on tuckoo in /COMP/605

```bash
# test case data for a 50x50 grid -- just as an example
# codes are named for grid size
# args passed include:
# - name of the pattern
# - max number of generations

[gidget:ga1meoflife/ser/basic] mthomas% ./gameoflife.v2.50x50
  Error: Not enough arguments, argv= 1
  Usage: gameoflife <pattern test number> <max #generations>
    Where lextyp='0'(rabbit);'1'(test);'2'(glider) '3'(Rpentimono);

[gidget:gameoflife] mthomas% ./gameoflife.50x50 0 50
```
### Instructions

#### Conway's Game of Life

- The 'dots' are dead/null.

#### Visual Representation

```
[ dead cell rows ]
```

```
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
```

- [ dead cell rows ]

```
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
| .................................................. |
```

- [ dead cell rows ]
# Game of Life: Small Test Case

Pattern: Rabbit, Board[50][50], MaxGenerations= 50

Rabbit Gens= 0: livepop=759003527, locations= 156520352,
Rabbit Gens= 2: livepop=13, locations= 6
Rabbit Gens= 3: livepop=15, locations= 14
Rabbit Gens= 4: livepop=20, locations= 11

[ dead cell rows ]

Rabbit Gens= 40: livepop=40, locations= 42
Rabbit Gens= 41: livepop=41, locations= 39
Rabbit Gens= 42: livepop=47, locations= 34
Rabbit Gens= 43: livepop=43, locations= 38
Rabbit Gens= 44: livepop=40, locations= 37
Rabbit Gens= 45: livepop=35, locations= 33
Rabbit Gens= 46: livepop=37, locations= 38
Rabbit Gens= 47: livepop=30, locations= 27
Rabbit Gens= 48: livepop=27, locations= 23
Rabbit Gens= 49: livepop=36, locations= 23
Rabbit Gens= 50: livepop=28, locations= 32
Game of Life: Small Test Case

# final board for 50 generations.

Rabbit Gens= 50: livepop=28, locations= 32

[ dead cell rows ]

Generation run completed for lexicon: Rabbit

Final State:
Rabbit Gens= 50: livepop=28, locations= 32,
Telapsed in seconds: 2.113e-03 seconds
Game of Life: Hints on the rabbit pattern

- Correct Result: 1744 live cells after 17331 generations
- Grid dimensions must be large enough to reach 1744 live cells. Hint: a 50x50 grid is too small, and 10000x100000 is too large.
- Choose enough generations to reach a live population of 1744.
- Rabbit locations: Try starting in the center of the grid.
- Print out a range of generations (eg 17320 to 17350) to see if you get close
- Time your code.
- Write your code so that you can load different patterns such as the glider, and the space ship. See: http://www.math.cornell.edu/~lipa/mec/lesson6.html
- You can turn in these test cases which demonstrate portions of your working code.
What to Report/Turn in:

- put all work into the directory named (in lower case): `/hw/hw#3`
- Short lab report with comments and figure and table labels. Put this into your HW #3 directory
- Explain results.
- Relevant tables of your test data
- Evidence you ran your jobs using the batch queue (short/small job)
- Print of screen shots.
- A copy of your code (single spaced, two sided, two column format is OK).
- Reference key sources of information in your report and code where applicable (Pacheco, lectures, Web, ).