


# Conway's Game of Life using MPI

**By Yashraj Agarwal(18BCI0183)**

# Overview



The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves. It is Turing complete and can simulate a universal constructor or any other Turing machine.

# Rules

To write a Conway's game of life there are basically two types of cell state: alive and dead. There are various rules to this game. In Conway's game of life, each cell value in the next generation is computed based on how many neighbors the cell had in the previous generation, and whether or not the cell was alive in the previous generation.

If a cell is alive in the current generation, it will be alive in the next generation if and only if it has either 2 or 3 neighbors in the current generation.

If a cell is dead in the current generation, it will be alive in the next generation if and only if it has exactly 3 neighbors in the current generation.

1

# CODE

```

"
"
"
"
"
"
"
"
};
int ROWSIZE = strlen( "                                " ) + 1;

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

void initDishes( int rank ) {
    int i;

    for ( i = 0; i < ROWS; i++ ) {
        a0[i] = (char *) malloc( ( strlen( patt[0] ) + 1 ) * sizeof( char ) );
        strcpy( a0[i], patt[i] );

        a1[i] = (char *) malloc( ( strlen( a0[0] ) + 1 ) * sizeof( char ) );
        strcpy( a1[i], patt[i] );
    }
}

void initDishes2( int rank ) {
    int i;

    for ( i=0; i<ROWS; i++ ) {
        a0[i] = NULL;
        a1[i] = NULL;
    }
}
```

# CODE

```
if ( rank == 0 ) {

    for ( i = 0; i< ROWS/2+1; i++ ) {

        a0[i] = (char *) malloc( (strlen( patt[0] ) + 1 ) * sizeof( char ) );
        strcpy( a0[i], patt[i] );

        a1[i] = (char *) malloc( (strlen( a0[0] )+1) * sizeof( char ) );
        strcpy( a1[i], a0[i] );

    }

    a0[ROWS-1] = (char *) malloc( (strlen( patt[0] ) + 1 ) * sizeof( char ) );
    strcpy( a0[ROWS-1], patt[ROWS-1] );
    a1[ROWS-1] = (char *) malloc( (strlen( patt[0] ) + 1 ) * sizeof( char ) );
    strcpy( a1[ROWS-1], patt[ROWS-1] );

}

if ( rank == 1 ) {

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

        a0[i] = (char *) malloc( (strlen( patt[0] ) + 1 ) * sizeof( char ) );
        strcpy( a0[i], patt[i] );

        a1[i] = (char *) malloc( (strlen( a0[0] )+1) * sizeof( char ) );
        strcpy( a1[i], a0[i] );

    }

    a0[0] = (char *) malloc( (strlen( patt[0] ) + 1 ) * sizeof( char ) );
    strcpy( a0[0], patt[0] );
    a1[0] = (char *) malloc( (strlen( patt[0] ) + 1 ) * sizeof( char ) );
    strcpy( a1[0], patt[01] );

}

}
```

# CODE

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

    if ( rank == 0 ) {

        for (i=0; i<ROWS/2; i++ ) {
            if ( dish[i] == NULL ) continue;
            pos( i, 0 );
            printf( "%s\n", dish[i] );
        }
    }

    if ( rank == 1 ) {

        for (i=ROWS/2; i<ROWS; 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 );
}
```

# CODE

```
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 ) {

    int i, j, row;
    int rowLength = strlen( dish[0] );
    int dishLength = ROWS;

    int lr, ur;

    if ( rank == 0 ) {
        lr = 0;
        ur = ROWS/2;
    }
    if ( rank == 1 ) {
        lr = ROWS/2;
        ur = ROWS;
    }

    for (row = lr; row < ur; row++) {

        if ( dish[row] == NULL )
            continue;

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

            int r, j, neighbors = 0;
            char current = dish[row][i];
```



# CODE

```
for ( r = row - 1; r <= row + 1; r++) {

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

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

        int rj = j;
        if (j == -1)
            rj = rowLength - 1;
        if (j == rowLength)
            rj = 0;

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

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

if (current == '#') {
    if (neighbors < 2 || neighbors > 3)
        newGen[row][i] = ' ';
    else
        newGen[row][i] = '#';
}

if (current == ' ') {
    if (neighbors == 3)
        newGen[row][i] = '#';
    else
```

# CODE

```
        newGen[row][i] = ' ';
    }
}

}

int main( int argc, char* argv[] ) {
    int genes = 3000;
    int i;
    char **dish, **future, **temp;

    int noTasks = 0;
    int rank = 0;
    MPI_Status status;

    MPI_Init( &argc, &argv );

    MPI_Comm_size( MPI_COMM_WORLD, &noTasks );
    if ( noTasks != 2 ) {
        printf( "Number of Processes/Tasks must be 2.  Number = %d\n", noTasks );
        MPI_Finalize();
        return 1;
    }

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    initDishes( rank );

    dish = a0;
    future = a1;
```

# CODE

```
cls();

print( dish, rank );
for ( i = 0; i < genes; i++) {

    pos( 33+rank, 0 );
    printf( "Rank %d: Generation %d\n", rank, i );

    life( dish, future, rank );

    if (rank==0 ) {

        MPI_Send( future[ 0 ], ROWSIZE, MPI_CHAR, 1, 0, MPI_COMM_WORLD );
        MPI_Send( future[ROWS/2-1], ROWSIZE, MPI_CHAR, 1, 0, MPI_COMM_WORLD );
        MPI_Recv( future[ROWS-1], ROWSIZE, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &status );
        MPI_Recv( future[ROWS/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[ROWS/2-1], ROWSIZE, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &status );
        MPI_Send( future[ROWS-1], ROWSIZE, MPI_CHAR, 0, 0, MPI_COMM_WORLD );
        MPI_Send( future[ROWS/2], ROWSIZE, MPI_CHAR, 0, 0, MPI_COMM_WORLD );
    }

    temp = dish;
    dish = future;
    future = temp;
}
```

# CODE

```
print(dish, rank);

pos( 30+rank, 0 );
printf( "Process %d done.  Exiting\n\n", rank );
MPI_Finalize();

return 0;
}
```

1

[illegible]