Game Of Life : Parallel Computing : Deliverables

Owen Morehead

AM250

June 5, 2022

1 PCAM Analysis

1.1 Partitioning

Task creation. Decompose computation and data into small tasks. Recognise parallel opportunities. Ignore number of processors and other practical issues and finer details.

For this Game Of Life (GOL) we want to perform a **domain decomposition**. In thinking about the algorithmic process of updating each element in the entire $N \times N$ grid (which I assume), we are performing the same update to each element in the domain. To take advantage of concurrency in this algorithm, we can perform a data decomposition where we partition the grid by *column*, or *row*, which would be 1D domain decompositions. We could also decompose the data using 2D sub-grid portions of our total grid domain. Partitioning the data by column is an ideal 1D decomp when working in the column major Fortran language. We can apply the GOL algorithm on each column independently, as long as there is communication with the data in the other columns. The tasks will be of comparable size so we should not encounter any load balancing issues. Additionally, the number of tasks scales with the problem size, so that if we increase the size of the $N \times N$ grid, there will be more columns to partition and update.

1.2 Communication

(local/global, static/dynamic, structured/unstructured, synchronous/asynchronous): Channel creation. Set up communication structure required to co-ordinate task execution. Scalability.

Essentially we need each column task to communicate with the neighboring columns. This is more-so a local communication. Therefore, a *ring* communication structure is used such that each column communicates with the column to their left and right. This means that the column to the left of the first column will be the last column, and similarly, the column to the right of the last column will be the first column. This is implemented because we assume periodic boundary conditions. 'Ghost cells' for the boundary conditions are also implemented so that we can update all the elements on the edge of the grid. This is a static communication since the communication partners do not change over time (i.e. the channels are fixed).

1.3 Agglomeration

Agglomeration (reduce communication and therefore costs): Evaluate task-channel communication structures with regard to performance and implementation costs. Combine into more efficient (larger?) tasks.

To reduce the amount of time spend communicating, we can agglomerate the previously defined column based domain decomposition by grouping multiple columns (or rows) together. This will reduce the amount of communication per computation, since elements on the inside of the column-grouped (or row-grouped) domain of data will be able to update without any communication to data which is on other processors. Only the edge elements of the column (or row) agglomeration will need to communicate to other processors in order to update. This agglomeration will indeed remove some concurrency, however there is still sufficient concurrency left to efficiently perform the GOL algorithm.

1.4 Mapping

Mapping (load balancing, task scheduling, static or run-time): Assign tasks to processors. Satisfy competing goals of maximal processor utilisation but minimal communication.

We can utilize a static mapping scheme in which an equal number of columns (or rows) are distributed to each processor (N/p columns (or rows) per processor) where N is the total number of columns, and p is the total number of processors used. This will result in equal load balancing among all processors. However, if $\text{mod}(N, p) \neq 0$, we can distribute the columns (or rows) such that the remainder gets added to one or some of the processors. Ideally this will not create any load balancing issues. I ended up adding the remainder all to one processor for simplicity, although it is ideal if the number of processors evenly divides the number of columns (or rows). If this is not the case, only a handful of extra columns (or rows) should need to be added to one of the processors. To make things better, one could distribute the extra columns (or rows), one by one to all the processors until the the remainder runs out. This would be the most idea load-balanced situation.

2 Running The Programs

Included are three versions of the program: a 1D column-wise domain decomposition, a 1D row-wise domain decomposition, and a 2D domain decomposition. The program files for each are contained inside their respective directories (`columns`, `rows`, and `2d`). Thus, to run one of the programs, move inside the desired directory, and use the `makefile` command to compile, and then run (details included inside the `README.md` file).

The programs consist of a module containing all relevant subroutines, `gol_subs.f90`, and driver routines specific to each program version:

- `gol_mpi_columns.f90` executable is `gol_columns`
- `gol_mpi_rows.f90` executable is `gol_rows`
- `gol_mpi_2d_decomp.f90` executable is `gol_2d`

After the executable is made (they are also already attached in the homework submission), they can be run using the command:

```
mpirun \np num_procs exe
```

where `exe` is the name of the executable, and the number of processors used to run the program, `num_procs` can be decided by the user.

A couple things to note regarding certain parameters when running the programs. The user will be prompted to input the initial pattern configuration of the grid, and then the dimension, N, of the square grid. Specific details to each program are listed below:

- `gol_mpi_columns.f90`, works for any value of N which is equal to or greater than the total number of processors used. Thus, unequal load balancing was accomplished with this program version.
- `gol_mpi_rows.f90`, only works for equal load balancing between processors. I was not able to fully accomplish unequal load balancing cases for this program version. So for this version, the user must choose N such that $\text{mod}(N, \text{numprocs}) = 0$, ex, $\text{numprocs} = 4$, $N = 20$.
- `gol_mpi_2d_decomp.f90`, this version can not handle much generality (something I will continue to work on). As of now, it only works with 4 processors, and a value of N such that $\text{mod}(N, \text{numprocs}/2) = 0$. In other words, the dimension of the total grid, N, must be divisible by the 2 processors which span that dimension. Thus, the total grid will be split such that each of the 4 processors have an even load. Nevertheless, this program will work for the test case, i.e., 4 processors and a 20×20 grid, which will give each processor a 10×10 subgrid.

3 Performance Model

I first note the communication time constants (t_s and t_w) which have been calculated in previous work (latency program), and the computation time constant, t_c, which was calculated from the `ones.f90` program:

$$t_s = 7.94 \times 10^{-6}\text{s} \tag{1}$$
$$t_w = 1.70 \times 10^{-9}\text{s} \tag{2}$$
$$t_c = 1.55 \times 10^{-8}\text{s} \tag{3}$$

t_c was computed by taking the average of the computation times calculated from updating each element in the 2D grid. I took an average from a range of grid sizes from 100×100 to 1000×1000. Updating each element consists of calculating the sum of the nearest 8 neighbors, and then updating the value in the array based on the value of the sum (updates to 1 if sum equals 3, does not change if sum equals 2, updates to 0 if sum equals anything else).

We now derive a simple performance model for this Game of Life program.
We define the total execution time of the program, T_{GOL} as:

$$T_{\text{GOL}} = \frac{T_{\text{COMP}} + T_{\text{COMM}}}{P} \tag{4}$$

where we consider the case when P divides N exactly such that there is equal load-balancing and little to no no idle time.

We consider our grid to be square of dimension $N \times N$. With 1D column-wise domain decomposition, and assuming no duplicated computation, we can define the total computation time for the program as,

$$T_{\text{COMP}} = t_c N^2 \tag{5}$$

where t_c is the computation time on a single element of the grid. We also consider the total communication time, T_{COMM}. We can assume that each *task* must exchange data with 2 neighboring columns = N points with each neighbor column. Thus T_{COMM} can be defined as:

$$T_{\text{COMM}} = 2P(t_s + t_w(N)). \tag{6}$$

This leaves us with,

$$T_{\text{GOL}} = \frac{t_c N^2}{P} + 2(t_s + t_w(N)) \tag{7}$$

which we can compare to our program.

Additionally we can calculate the efficiency as the ratio of the execution time on one processor over the execution time on P processors:

$$E = \frac{t_c N^2}{t_c N^2 + 2P(t_s + t_w(N))}. \tag{8}$$

We can also consider the isoefficiency function, which describes how must the amount of computation performed scale with P to keep E constant:

$$t_c N^2 = E\big(t_c N^2 + 2P(t_s + t_w(N))\big). \tag{9}$$

For large N and P, scaling $N \sim P$ will keep E approximately constant. Thus the amount of computation to keep E constant $\sim P^2$. This suggests the isoefficiency function $\sim \mathcal{O}(P^2)$ which is somewhat poorly scalable, since the amount of computation needs to increase as P^2. If we change to a 2D domain decomposition, we would expect the program to scale better, as has been shown in a previous assignment.

To compare our results to the model, the column-wise domain decomposition program was timed, taking an average after running the program for 100 iterations. We also ran the program for a range of N values from $N = 20$ to $N = 1000$, and recorded ant average program time data point for each. This program time data was plotted against the model described by Equation 7 using the coefficients described by Equations 1, 2, and 3. This can be seen in the figure below. We see that although the program timer data is slightly larger than predicted by the model, the two align fairly well. The program time is approximately proportional to N^2 for large N.

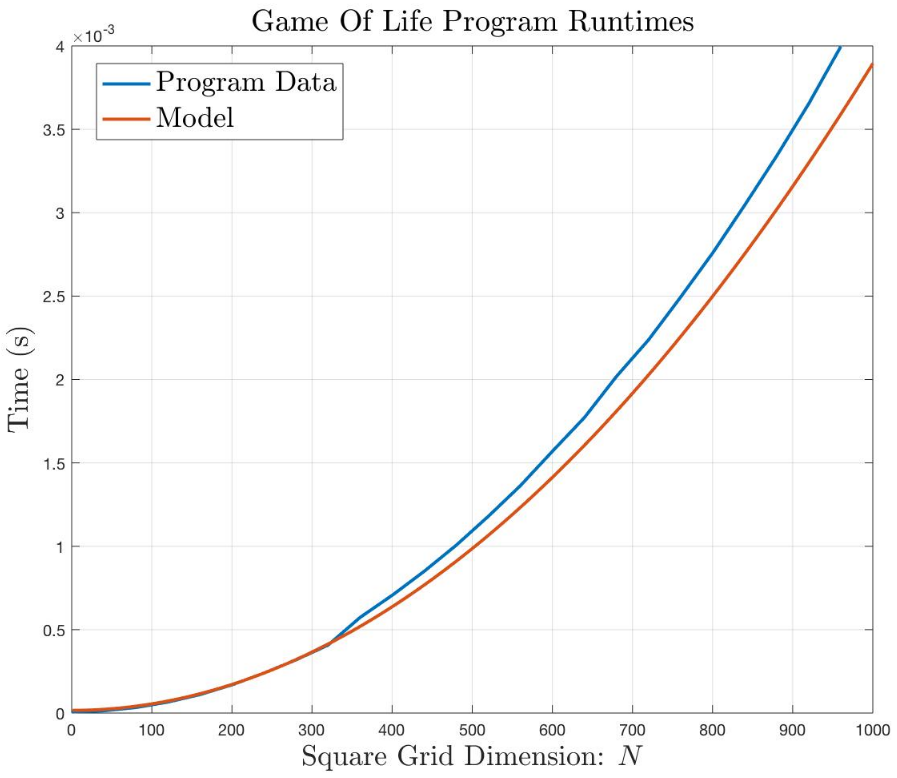
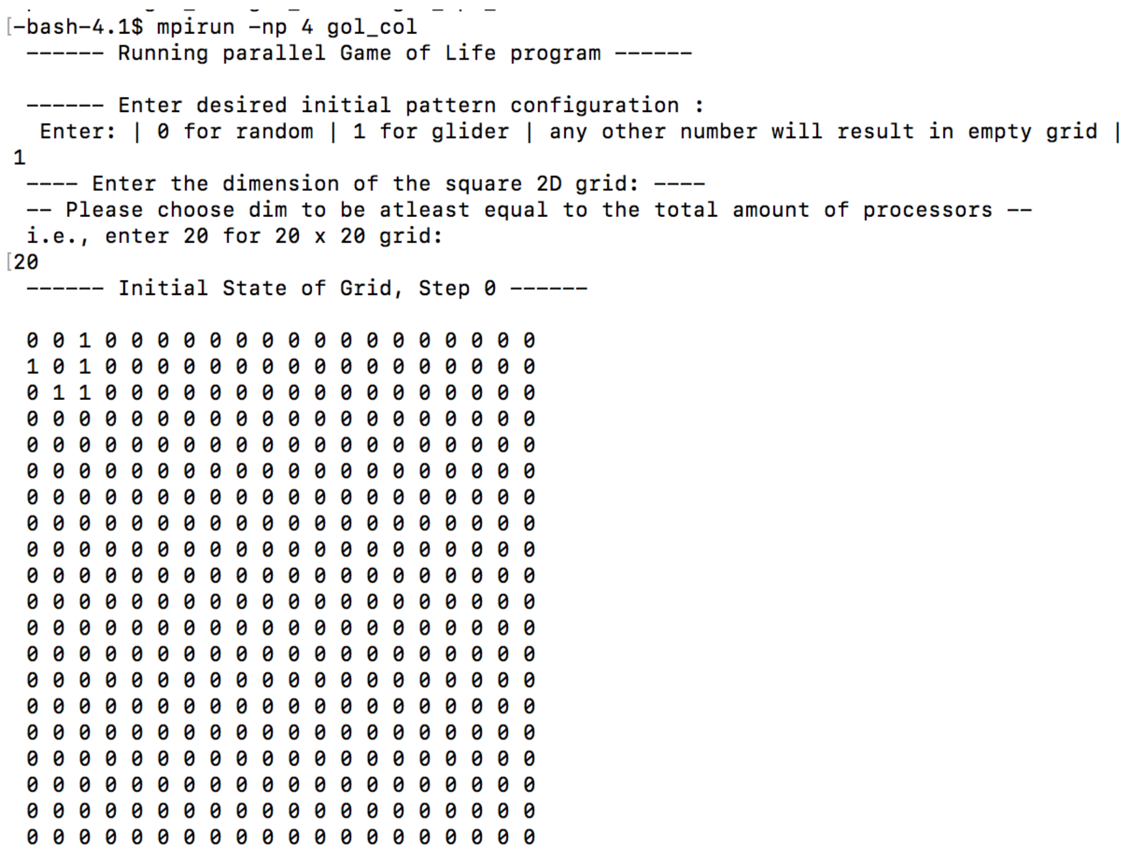


Figure 1: Plot of column-wise domain decomposition program timer data against the performance model defined by Equation 7.

4 Program Output

Below are screenshots of the terminal output from parallel Game of Life program using column-wise domain decomposition. 4 processors and a grid size of 20×20 is used. The glider configuration is initialized in the top left corner. The alive cells are denoted with a 1, and the dead cells with a 0. After 80 iterations, we see that the glider configuration makes its way back to exactly where it started, in the top left corner.

The output of the row-wise domain decomposition, as well as the 2D domain decomposition are exactly the same (not shown here, but can be seen if one runs the programs).

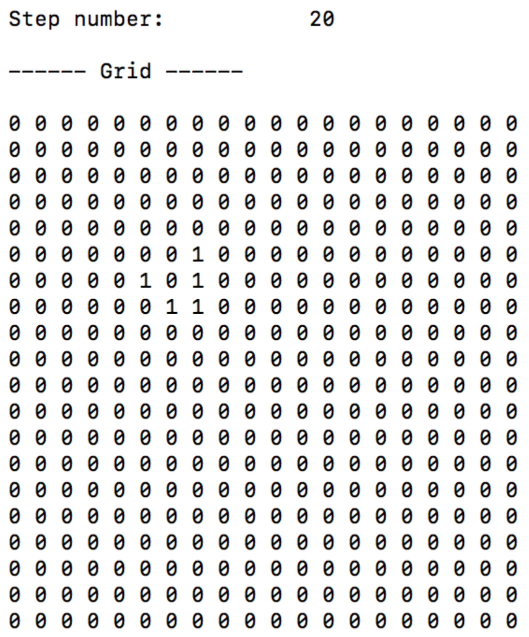


```
[-bash-4.1$ mpirun -np 4 gol_col
  ------ Running parallel Game of Life program ------

  ------ Enter desired initial pattern configuration :
   Enter: | 0 for random | 1 for glider | any other number will result in empty grid |
 1
  ---- Enter the dimension of the square 2D grid: ----
  -- Please choose dim to be atleast equal to the total amount of processors --
  i.e., enter 20 for 20 x 20 grid:
[20
  ------ Initial State of Grid, Step 0 ------

  0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 2: Screenshot of the terminal output from parallel Game of Life program using column-wise domain decomposition. 4 processors and a grid size of 20×20 is used. This is the initialized grid at step = 0. The glider configuration is initialized in the top left corner.

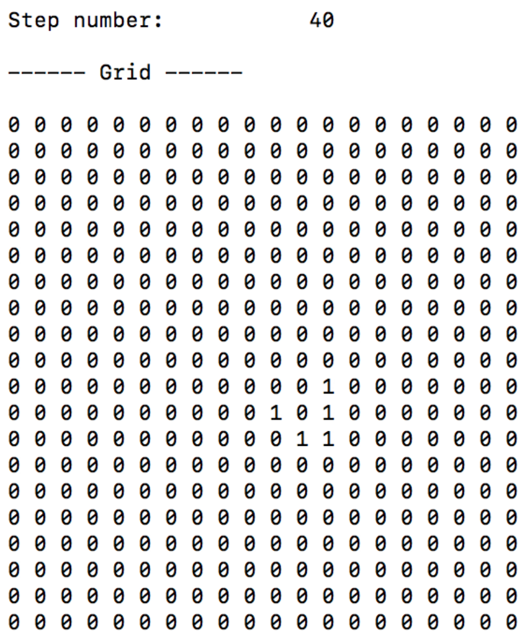


```
Step number:           20

------ Grid ------

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

(a) Grid at step = 20

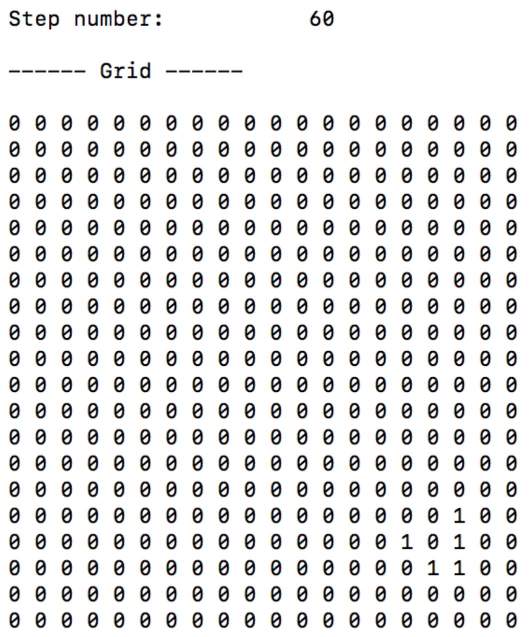


```
Step number:           40

------ Grid ------

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

(b) Grid at step = 40

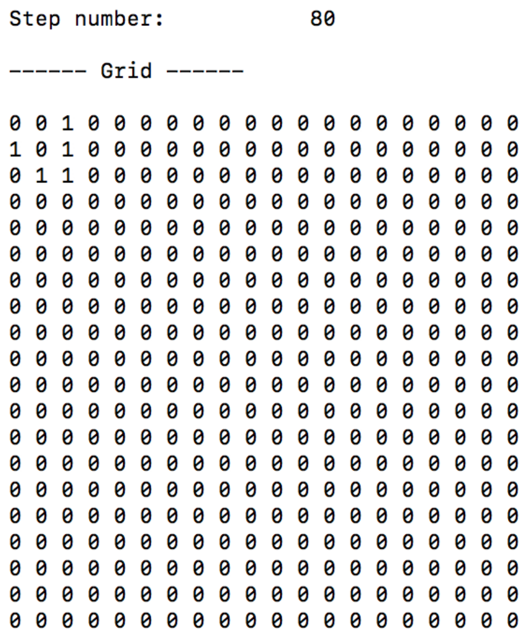


```
Step number:           60

------ Grid ------

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

(c) Grid at step = 60



```
Step number:           80

------ Grid ------

0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

(d) Grid at step = 80