Graph Front End Lab Manual

The task is to build a python program that uses the Graph Front End (GFE).

Summary

This python script should use the GFE to instantiate a working example of the Conway's Game of Life[1]. Conway's Game of Life consist of a 2D fabric of cells, each of which has 2 states. These states are either Alive or Dead, and switching between these two states is decided upon the states of their 8 neighbouring cells.

The rules which dictate the changing of state are as follows:

1. Any live cell with fewer than two live neighbours dies, as if caused by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if caused by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The application needs to be able to handle different initial states of the cells within the 2d fabric, but the default states for tasks 1 to 8 should look like Figure 1 and for task 9 to 13 should look like Figure 2.

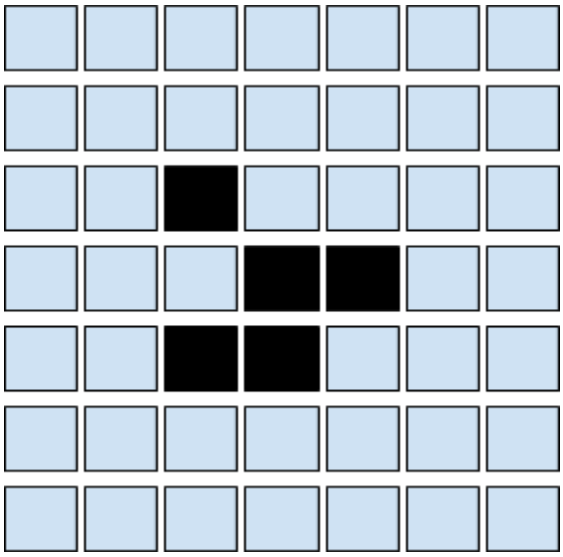

Figure 1: Basic initial state (7 by 7 grid)

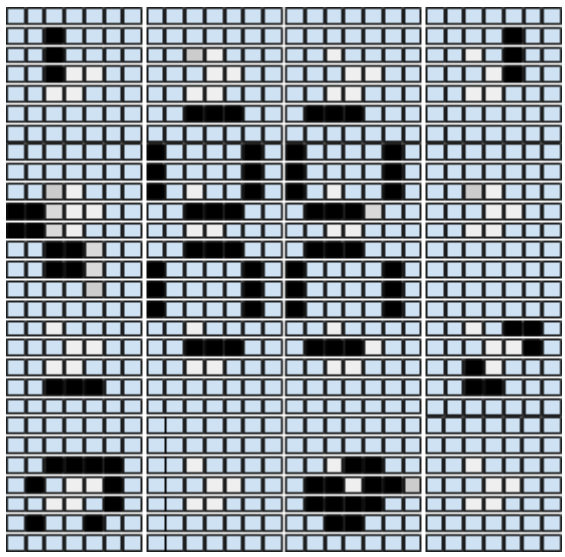

Figure 2: Advanced State (28 by 28 grid)

[1] https://en.wikipedia.org/wiki/Conway's_Game_of_Life

Step 1 (Easy)

Create a python class which will represent a Conway Cell.

Step 2 (Easy)

Build a python script which builds a collection of Conway cells (as vertices) in the GFE machine graph to form a 7x7 grid (hint: use the add_machine_vertex_instance() functionality supplied with the GFE __init__.py interface to add vertex instances). Add edges between the cells.

Step 3 (Medium)

Build the c code that represents the functionality that will run on the SpiNNaker machine. It'll be easier to use the interfaces provided in simulation.h and data_specification.h.

At this point running the script should not produce errors, but you won't be able to tell what's happening inside.

Step 4 (Medium)

Add a data region for storing the state per timer tick iteration in SDRAM into both the c and python class. Add code to store the data in C and retrieve it from the machine in python.

Step 5 (Easy)

Build a simple text-based visualiser to replay a simulation run using the stored state.

At this point you should be able to run the simulation and get a textual display of the state of the simulation per timer tick.

Step 6 (Hard)

Stream the state of the simulation to the host PC during the simulation run. Display the output as text as the simulation runs.

Step 7 (Medium)

Use the **tubogrid** perl application supplied within spinnaker_tools to create a live visualisation of the simulation, or build your own.

Step 8 (Easy)

Try building a 28x28 grid of cells and see what happens. Can you explain why it doesn't work? What ways could you go about making it work (hint: there's at least 3 ways of doing this)?

Step 9 (Fiendishly Hard)

Convert your application so that instead of using machine vertices, you use an application vertex to represent the entire grid of cells. Note that this will require you to receive and interpret several ids from each base routing key (hint: sPyNNaker does this using something called a population table to map between a base key and a block of connections).

Step 10 (Easy)

Run your new simulation with both sets of initial parameters. Try scaling up so that you hit the limits of the system to simulate the application, and therefore make the CPU calculation correct so that the partitioner will stop you going over the limits.

Step 11 (Medium)

Upgrade **turbogrid** or your own visualisation to use the database so that it auto configures itself for the shape of the application space.

Step 12 (Epically Hard)

Build your own application for SpiNNaker using the GFE front end.