MPI assignment Parallel Computing 2020: Game of Life

Emil Løvbak and Sahar Chehrazad

November 23, 2020

1 Background

In this assignment we will produce a parallel implementation of Conway's Game of Life[1] and run in on the VSC supercomputer *Thinking*. This simple program, simulates a 2D-grid of cells which are either alive or dead during a sequence of discrete time steps. The program runs as follows:

- Each cell in the grid is (randomly) initialized to be either alive or dead.
- At each time step each cell is updated based on a series of rules, which depend on how many of its eight neighboring cells are alive:
  - A cell that is alive, and has fewer than two live neighbors, dies from loneliness.
  - A cell that is alive, and has more than three live neighbors, dies from overpopulation.
  - A cell that is alive, and has two or three live neighbors, remains alive.
  - A cell that is dead, and has exactly three live neighbors, becomes alive, otherwise, it remains dead.

This program is interesting as it able to exhibit very complex behavior, despite implementing only very simple rules. It has been shown to be Turing complete and, as a consequence, it is in general impossible to decide whether a given initial condition will cause the program converge to a steady state or maintain dynamic behavior for ever.

2 Assignment

The main focus of this assignment will be to develop a parallel implementation of a small program with MPI. As such, you have already been provided with working serial code, to serve as a starting point. This implementation consists of two parts:

- **A C++ code that implements the Game of Life**: `gol.cpp`. This can be compiled using the command `g++ -o gol -O3 -std=c++1z gol.cpp`.

[1] `https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`

Once compiled the software can be run with the command `./gol nb_rows nb_cols iter_step iter_end`. Here, the first two parameters are the number of rows and columns in the grid, `iter_step` is the number of iterations between subsequent writes of the grid to a file and `iter_end` is the total number of iterations on the grid. For example, `./gol 50 50 2 6` will generate files representing the solution on a 50×50 grid at iterations 0, 2, 4 and 6. It will also generate a main output file containing some meta-data.

- **A python code for visualizing results**: `gol_visualization.py` This script should be invoked with the command `python3 gol_visualization.py main_file`, where `main_file` is the name of the main file generated by the C++ code.

You will be expected to modify the C++ code to make it run in parallel using MPI. A good way to go about this will be to have each process store both its own part of the board, and a copy of any data it needs from neighboring processes. To achieve this, you can modify the indices and sizes used when declaring and accessing the board.

You will need to modify the following existing routines in the C++ code:

- `main`: Adding MPI routines and communication.
- `updateBoard`: Make this routine work with the modified data structure.
- `writeBoardToFile` Make this routine work with the modified data structure.

To make a well structured (and easy to debug) implementation, you will likely want to write routines to take care of synchronizing data between processes and to convert indices in the local segment of the board to the global board. It should not be necessary to modify the provided Python script.

To make the implementation easy, you can choose to limit your program to boards which can be easily and equally partitioned on the chosen number of cores, i.e., you are free to choose which input parameters for which your program works, but you should think about what you would have to do in the more general case (see later). It is not necessary to do extensive testing of your code. If, however, you want to verify that your implementation is correct, a good test case can be to set the initial condition of one processor to completely alive and and set the others to be completely dead.

To compile your parallel implementation on the cluster you should use the above compile command for the single core version, substituting `g++` with `mpiicpc`[2]. You can then submit a job using a `pbs` file, using the command `qsub -A lp_edu_alg_parallel_comp_2021 gol.pbs`.

3 Practical details

We will not ask you to submit your code or a report, however we do expect you to complete the assignment. One of the questions on the exam will namely be a discussion of your experiences while implementing this assignment. As

[2]To have access to this command you will have to load it using `module load intel`.

preparation for this question, we expect that you have created a basic working parallel version of the code and that you have thought about the following questions:

- How did you partition the data? Why?
- What did you assume to reduce the complexity of your implementation? What would change if you did not have these assumptions?
- How did you synchronize the data between the processes? What did you synchronize?
- Is MPI a logical choice in this case? Would OpenMP be better? Why?
- What difficulties did you experience when making your parallel implementation?

You are free to write out some notes for your answers to these questions and bring them with you to the exam if you so wish.

During the exercise sessions, you will have reserved computing time on the *Thinking* cluster and the TA's will be available for asking questions. You can also test your code on the cluster outside of these sessions, but you may be placed in a queue. If wanted, we will provide further opportunities to work on your own laptop with TA's present during the lecture slots, once the lectures are done. If you feel that you have a need for this, or just want to ask a quick question, you can contact us via email at `emil.loevbak@kuleuven.be` and `sahar.chehrazad@kuleuven.be`. We, of course, expect you to also be present in the exercise sessions if you request extra moments to work on the project.