PPAR **Master 1 CS**

The Game of Life in MPI

We will work on Conway's Game of Life, and we will develop a parallel version of the Game with MPI, where different parallelization strategies will be implemented. This work will take 2 time slots of 2 hours (4 hours in total). The codes that you're going to develop will have to be submitted at the address

`antonio.mucherino@irisa.fr`

after the end of the second time slot (the precise deadline is to be decided with your teacher). Please include "PPAR MPI" in the email subject. The codes (sequential version, parallel version, etc) should be organized in folders and included in a tarred and zipped archive.

The Game of Life

The Game of Life is a *cellular automaton* devised by the British mathematician John Horton Conway in 1970. In the Game of Life, the *world* is represented by a two-dimensional torus formed by $N \times N$ cells, where each cell can be in one of three possible states:

- occupied by individuals of type "x",
- occupied by individuals of type "o",
- empty.

Cells occupied by individuals are also said "live" cells. Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, and diagonally adjacent. At each step in time, the following transitions occur:

- live cells with fewer than two live neighbours die, as if caused by under-population, and free their cell;
- live cells with two or three live neighbours live on to the next generation;
- live cells with more than three live neighbours die, as if by over-population, and free their cell;
- empty cells with exactly three live neighbours become alive, as if by reproduction: the new born individual will have the character (either "x" or "o") imposed by the majority of its neighbouring cells.

The Game in sequential

Download the code from

`/share/m1info/PPAR/TP/TP5`

and fill the empty parts for performing the Game of Life in sequential in C. To this purpose, pay particular attention to the different functions that are already implemented, and complete the code for the function `newgeneration` by exploiting the existing functions. Please *do not* modify the existing code: the representation of the world and the existing functions will be strictly necessary in the following. You can verify that your code works properly in sequential by comparing the results that you'll obtain to the ones provided online on the website:

`http://www.bitstorm.org/gameoflife/`

The Game in MPI

Our aim is to write a parallel code performing the Game of Life in parallel with MPI. While working on this parallel implementation, we will explore different communication strategies in MPI. Consider that it's not necessary to modify the functions of the sequential program: only the main function needs to be adapted for the parallel execution.

Let us suppose that our world has size $N \times N$; let p be the number of processes involved in the parallel computation. The main idea is to assign, to every process, a region of the world formed by $\frac{N}{p}$ contiguous rows. In practice, process 0 can work on the first $\frac{N}{p}$ rows of the torus, while process 1 can work on the second group of $\frac{N}{p}$ rows, and so on.

Your parallel program will perform the following tasks:

- all processes verify that N is divisible by p: if not, the execution is aborted;
- process 0 generates the initial world;
- process 0 sends to all other processes the generated initial world (communication type: one-to-all);
- process 0 prints the initial world on the screen;
- every process computes the first and last row index of its world region;
- in the main `while` loop
  - every process invokes `newgeneration` with its first and last row index;
  - the pointers of `world1` and `world2` are inverted, as in the sequential version;
  - the processes exchange the neighbouring rows, necessary for computing the next generation (communication type: one-to-one);
- process 0 collects the results obtained by the other processes (communication type: all-to-one): consider that the partial results are stored in different regions of different torus representations, and that memory for representing the entire torus was allocated by all processes;
- process 0 prints the final result.

In this parallel implementation, it is not necessary to print the world at every generation: we can just print its initial and final status. This will make your implementation more efficient. Moreover, it is advisable to remove the stopping criteria based on the world status: to make it work, we'd need to perform additional communications. Therefore, to make things easier, we will consider only the stopping criteria based on the number of generations.

Bonus: shortening the messages

If you didn't find this parallel implementation extremely complicated, you might be interested in trying to make it more efficient. In the main `while` loop, the processes need to communicate and share messages of length N (corresponding to one row of the world). Every cell is represented by an `unsigned int`, to which we can assign three different states. However, only 2 bits out of `sizeof(unsigned int)` are actually exploited for representing these states, while the other bits are in practice useless.

Our messages can be therefore compacted, by using an array of `unsigned int` where every 2 consecutive bits represent the status of one single cell. This compact representation can reduce the length of our messages from N to $2N/$`sizeof(unsigned int)`. However, the corresponding implementation requires the use of bit-to-bit operations for accessing the information regarding the torus cells.