December 01, 2022

Project 2: Game of life

MPI, 2D topology, ghost layers

Aim

You are given a serial program that implements a modified version of the “Game of Life” cellular automation rule by J. H. Conway,

`https://en.wikipedia.org/wiki/Conway's_Game_of_Life`

You must parallelize it to show that you know how to

- use the MPI library;
- partition 2D grids;
- create a toroidal topology;
- communicate sub-grids via ghost layers;
- perform simple parallel I/O via the root process;
- do simple parallel reductions.

Your parallel program must produce exactly the same output as the serial program does.

Serial main program

The main program has the following core structure:

```
read *, height, width, max_gen
allocate(old_world(0:height + 1, 0:width + 1))
allocate(new_world(0:height + 1, 0:width + 1))
call read_map( old_world, ... )
call update_borders( old_world, ... )
do gen = 1, max_gen
    print "(a, i0)", "Generation ", gen
    call print_map( old_world, ... )
    call next_gen( old_world, new_world, ... )
    call update_borders( new_world, ... )
    if (world_is_still( old_world, new_world )) exit
    tmp_world => old_world;  old_world => new_world;  new_world => tmp_world
end do
if (associated( old_world )) deallocate(old_world)
if (associated( new_world )) deallocate(new_world)
```

Input file

```
7  11  50  2  2  ! height  width  max_gen  n_rows  n_cols
...........
..X..X.....
......X....
..X...X....
...XXXX....
...........
...........
```

The input file has five numbers in the first row: the height and width of a 2D world, the maximum number of generations, the number of block rows and columns in a 2D topology. The serial program reads the first three of them. The following two will be read by the parallel program. What follows is a 2D grid of dots and crosses that represents the initial state of the world. The input file provides only the internal range of cells in the world without boundaries. Boundaries are created and made periodic later in the program.

World states

The world state is stored in 2D array targets, manipulated using two pointers called `old_world` and `new_world`:

```
logical, dimension(:, :), pointer :: old_world, new_world, tmp_world
```

One more pointer, `tmp_world`, is used to swap them at each generation step:

```
do gen = 1, max_gen
    ...
    tmp_world => old_world;  old_world => new_world;  new_world => tmp_world
end do
```

Both targets are created with two extra rows (`i = 0` and `i = height + 1`) and two extra columns (`j = 0` and `j = width + 1`) to have periodic boundary conditions in 2D:

```
allocate(old_world(0:height + 1, 0:width + 1))
allocate(new_world(0:height + 1, 0:width + 1))
```

World states at I/O

The initial state is read line-by-line into `old_world` by `read_map()`. Each cell in the world can be either dead (`.false.`) or live (`.true.`). At I/O these states are represented by characters `.` and `X` for better legibility. The grid cells are read in only at internal ranges `i = 1..height` and `j = 1..width` as provided by the input file:

```
do i = 1, height
    read *, line
    do j = 1, width
        select case (line(j:j))
        case ('X')
            old_world(i, j) = .true.
        case ('.')
            old_world(i, j) = .false.
        case default
            stop "read_map: wrong input character `" // line(j:j) // "`"
        end select
    end do
end do
```

World states at I/O

At each generation, before the next state of the world is calculated, the current state in `old_world` is printed by `print_map()`. Only cells at internal ranges `i = 1..height` and `j = 1..width` are printed without boundary cells. Cell values are converted back from logical to character represenation using the intrinsic function `merge()`:

```
do i = 1, height
    do j = 1, width
        line(j:j) = merge( 'X', '.', old_world(i, j) )
    end do
    print "(a)", line
end do
```

Boundary conditions

Only one boundary layer is needed owing to the shape of the cellular rule stencil (see later). All four boundaries are made periodic in both dimensions by calling `update_borders()`:

```
! Inner rows
map(         0, 1:width) = map(height, 1:width)
map(height + 1, 1:width) = map(     1, 1:width)
! Full columns
map(0:height + 1,         0) = map(0:height + 1, width)
map(0:height + 1, width + 1) = map(0:height + 1,     1)
```

This is done for the `old_world` after reading the initial state into it,

```
call read_map( old_world, ... )
call update_borders( old_world, ... )
```

and for the `new_world` after calculating each next generation,

```
call next_gen( old_world, new_world, ... )
call update_borders( new_world, ... )
```

Cellular automation rule

At each generation the `new_world` state is calculated from the `old_world` state by calling `next_gen()`:

```
do j = 1, width
    do i = 1, height
        c = count( old_map(i - 1:i + 1, j - 1:j + 1) )
        if (old_map(i, j)) then ! cell is live
            new_map(i, j) = merge( .true., .false., 3 <= c .and. c <= 4 )
        else ! cell is dead
            new_map(i, j) = merge( .true., .false., c == 3 )
        end if
    end do
end do
```

Cellular automation rule

The rule is applied to each cell at `(i, j)` in the inner range `i = 1..height` and `j = 1..width` using a 3×3 stencil in Fig. 1 by calculating the total number of live neighbors, this cell including, using the intrinsic function `count()`.

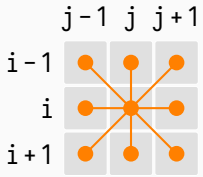

Figure 1: Cellular automation rule stencil at the (i, j) cell.

If a live cell is surrounded by 2–3 other live cells (3–4 in total), then it lives on to the next generation. If a dead cell is surrounded by 3 live cells, then it becomes live in the next generation. In all other cases this cell either dies or remains dead.

Convergence test

The world in the Game of Life either evolves into a collection of still states, what includes the empty state too, or into a collection of oscillating or moving states, which don't evolve anymore. For this reason, iterations stop either when the maximum generation `max_gen` has been reached (which is specified in the input file), or when the `world_is_still()` function returns `.true.`:

```
do gen = 1, max_gen
    ...
    if (world_is_still( old_world, new_world )) exit
    ...
end do
```

This function checks if both states are the same,

```
world_is_still = all( old_map .eqv. new_map )
```

What the parallel version must do?

The parallel version must read from the input file all five parameters including the numbers of blocks, `n_rows` and `n_cols`. They define how many block rows and colmuns must be in a 2D toroidal topology. Each input file must be run with `-np n_ranks`, where the number of processes `n_ranks = n_rows × n_cols`. These numbers will be changed when testing your code. The parallel version must run with all possible number of processes, for example, 1×1, 32×1, or 8×8.

The input numbers must be communicated to all processes. The program must check if the topology dimensions correspond to the number of processes, otherwise it cannot run.

2D toroidal topology

You must create a toroidal topology out of all processes by arranging their ranks on a 2D grid with both dimensions being periodic. For each rank you must calculate the row and column in the topology and ranks of all four neighbors to communicate with.

Figure 2: A possible arrangement of 12 process ranks (black) in a 2D toroidal topology with `n_rows` = 3 and `n_cols` = 4. Process ranks, placed out of ranges (orange), must be corresponding neighbors owing to periodic boundaries.

2D grid partitioning and synchronization

The world's grid must be partitioned between all processes as evenly as possible. Sub-grids must be created at each rank with through indexing. That is, the sub-grid array ranges must be contiguous across corresponding topology rows or columns with respect to the global grid ranges. Each sub-grid must have one outer layer of ghost cells to store values from neighbors: one top row, one bottom row, one left column, one right column.

Each time the inner cells are updated, the ghost cells must be synchronized between the neighbor processes by `update_borders()`. You need MPI data types to communicate rows and columns. These types can be reused for parallel I/O as well. You must correctly declare, resize (if necessary), use, and release them.

Figures 3 and 4 illustrate such partitioning for the initial grid state from the `spaceship-1.in` file.

2D grid partitioning

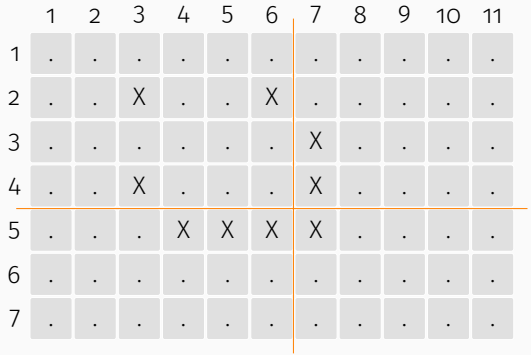

Figure 3: An input grid from `spaceship-1.in`. It is partitioned between 4 processes with `n_rows = 2` and `n_cols = 2` blocks as shown by orange cuts. Crosses are `.true.` values of live cells, dots are `.false.` values of dead cells.

2D grid partitioning

Figure 4: Sub-grids from `spaceship-1.in` as they must be partitioned between 4 processes with through indexing of cells and properly synchronized ghost cells at boundaries.

Parallel I/O

You must make subroutines `read_world()` and `print_world()` parallel. The I/O must only go via the root process (rank 0). On reading, the root process must read the world's grid line-by-line from the input file, convert each line to logical values, and distribute parts of it to corresponding processes. On printing, each process must send its sub-grid line-by-line to the root, which collects them into full lines, converts to characters, and prints them line-by-line. You should not allocate any 2D auxiliary arrays at the root otherwise you may exceed the memory limit if the grid is big enough.

Only cells from the inner ranges must be read and printed, boundary ghost cells must not be used in the I/O. The input file provides only the inner range of the grid, while the boundary cells will be updated later when you synchronize them between neighbors.

You must use point-to-point calls because gathering and scattering won't work in a 2D topology as you don't know how to split the communicator.

Next generation in parallel

Do not change the core part of the `next_gen()` subroutine. Only the local ranges of indices need to be changed. If everything is correctly coded, then the core part of the algorithm needs no parallelization as it works similarly both in the serial and parallel versions.

At each iteration step you must do a correct parallel reduction to check for convergence.

Evaluation

This project mark contributes 1/3 to the total mark of the course. To pass, you need to get at least a 50% mark. You get a 100% mark if your code is correctly parallelized, if for different input files it produces outputs, identical to those, produced by the serial program, and if all dynamic variables and MPI data types are correctly released to avoid memory leaks[a].

This project is individual. You can discuss general ideas and technical details between each other, but the code you submit you must write yourself. When in doubt, I'll ask you to explain your solution.

The Game of Life is a famous problem, so there are many parallel MPI realizations of it on-line. Most of them are either oversimplified or overengineered. For your own benefit (not to get confused and lost), it's better if you don't look into them and try to solve this problem by yourself.

[a] There might be leaks in the MPI library, but that's not your fault.

Breakdown of marks

5%: for reading in and distributing the input numbers to all processes;

5%: for making sure that the topology dimensions are compatible with the number of processes;

15%: for creating a 2D toroidal topology by arranging ranks on a 2D grid (7.5%) and for finding ranks of all four neighbor processes (7.5%);

10%: for partitioning the world's grid between processes as evenly as possible;

15%: for correctly declaring and using MPI datatypes to synchronize ghost cells and to do parallel I/O through the root rank (10%), good, if you commit no more than two datatypes (5%);

10%: for correctly synchronizing ghost cells between the neighbors with no deadlocks;

30%: for parallel I/O via the root rank, by emulating root-to-all in `read_map()` (15%) and all-to-root in `print_map()` (15%) using point-to-point calls;

10%: for using the correct parallel reduction to check for convergence.