Exercise 1 - Parallel Programming

Foundations of HPC @ UniTS-DSSC 2022

Assignment for the _Foundations of High Performance Computing course at "Data Science and Scientific Computing", University of Trieste, 2022-2023

Stefano Cozzini `stefano.cozzini at areasciencepark.it`
Luca Tornatore `luca.tornatore at inaf.it`

Due time: 1 week before taking the oral exam
v1.2
26th Jan, 2023

History of changes

v1.2 : added a section about the expected structure of the report; added a detail about how to run hybrid code when calling `mpirun`.
v1.1 : minor adjustments & clarifications
v1.0 : first release

Game of Life

You'll be prompted to implement a parallel version of a variant of the famous Conway's "*game of life*" (references: 1, 2), that is cellular automaton which plays autonomously on a infinite 2D grid.
The game itself has several exciting properties and pertains to a very interesting field, but we leave to the readers going deeper into it at will.

Playground

As said, the game evolves on a 2D discrete world, where the tiniest position is a single cell; actually you can imagine it as a point on a system of integer coordinates.
The neighbours of a cell are the 8 most adjacent cells, i.e. those that on a grid representation share an edge with the considered cell, as depicted in the Fig. 1 below.

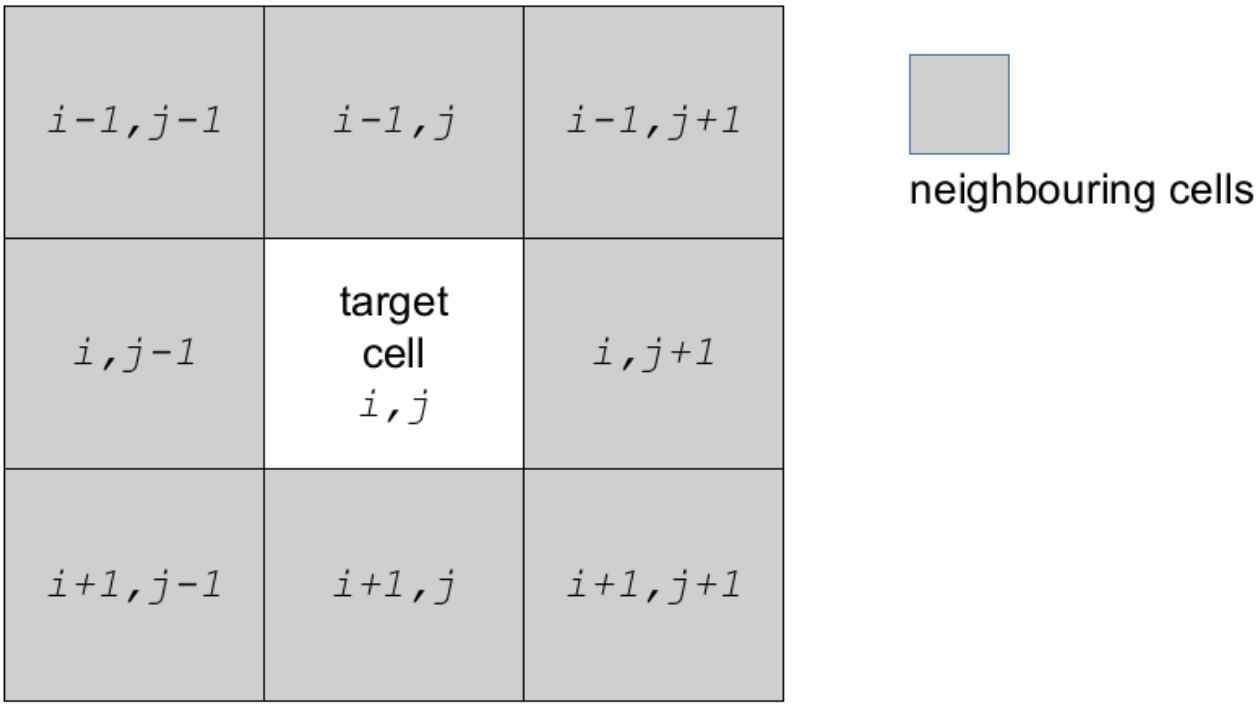

Figure 1: *The cell's neightbours are the 8 immediately adjacent cells*

The playground, that will be a grid of size $k \times k$, has periodic boundary conditions at the edges. It- means that cells at an edge have to be considered neighbours of the cells at the opposite edge along the same axis. For instance, cell $(k-1, j)$ will have cells $(0, j-1), (0, j)$ and $(0, j+1)$ as neighbours.

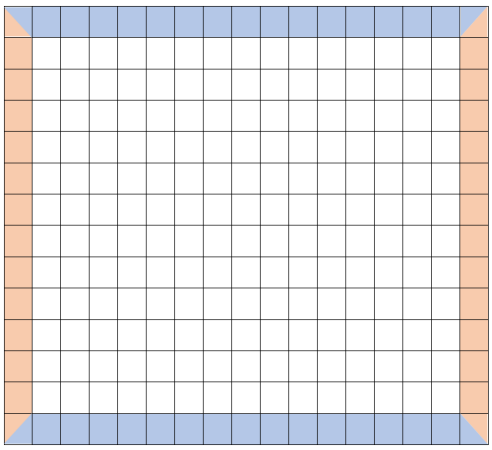

Figure 2: *Periodic boundary conditions: the cells at the edges with the same colour are adjacent* .

Rules of the game

Each cell can be either "alive" or "dead" depending on the conditions of the neighboring cells:

- a cell becomes, or remains, **alive** if 2 to 3 cells in its neighborhood are alive;

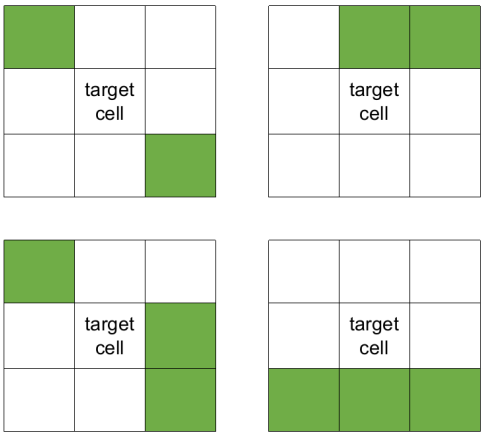

Figure 3: *examples for a cell to become or to remain alive*

- a cell dies, or do not generate new life, if either less than 2 cells or more than 3 cells in its neighborhood are alive (*under*-population or *over*-population conditions, respectively).

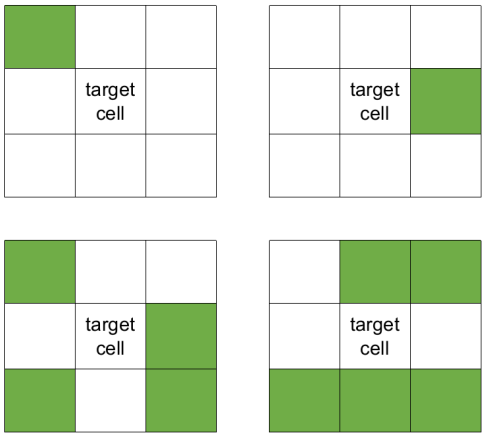

Figure 4: *examples for a cell going to die*

In this way, the evolution of the system is completely determined by the initial conditions and by the rules adopted to decide the update of the cells' status.

Upgrading cells status and evolving the universe

Classically, the cells are upgraded in row-major order, like in the code:

```
for ( int i = 0; i < k_i; i++ )
    for ( int j = 0; j < k_j; i++ )
        upgrade_cell(i,j);   // calculate the status of neighb. cells
and update
```

Since, obviously, changing a cell's status impacts on the fate of adjacent cells, that inserts a "spurious" signal in the system evolution. Let's call that "**ordered evolution**". Note that this evolution is intrinsically serial because of the inherent dependency that descends from its definition; in that case, the only issue related o the parallelism is the correct propagation among the OpenMP threads.
A second option is to disentangle the status evaluation and status update of each cell, in that the status of all the cells (i.e., the computation of how many alive adjacent each of them has) should be evaluated at first, freezing the system, and updating the status of the cells only afterwards. Let's call this "**static evolution**".
A third, among many others, option is that the ordered evolution does not always start from the same $(0, 0)$ point but from a random position and propagate in all directions as a square wave, as illustrated in Fig. 5.
An additional simple option is to evolve the cells in a "**white-black**" order, i.e. considering the playground as it was a chessboard and evolving the white positions at first and the black ones afterwards.

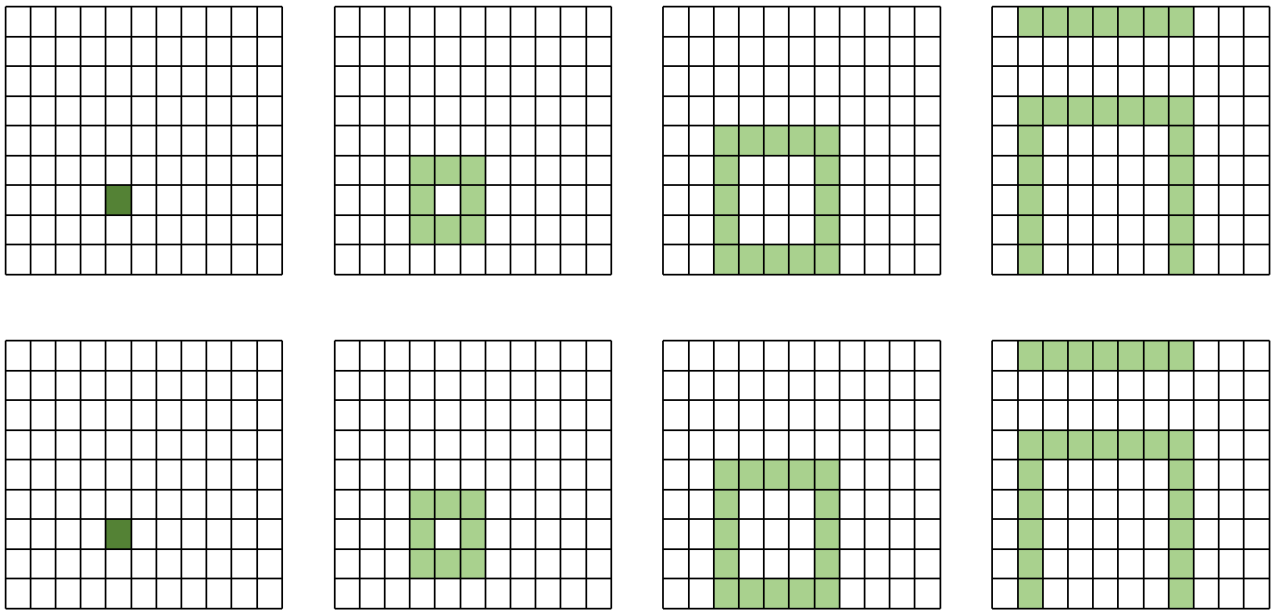

Figure 5: *cells upgrade starting from a random point, propagating as a square wave*.

Requirements

1. You have to implement an hybrid MPI + OpenMP code, using a domain decomposition to distribute the workload among the MPI tasks. Each MPI task will then further parallelize its own work using OpenMP threadization (see the appendix for some comments on hybrid codes).
2. The playground must be a general $k \times k$ checker (you are free to generalize to any rectangular $k_x \times k_y$) with $k \geq 100$ and unlimited upper value.
3. Initialize a playground and write it to a file, in a binary format (see the section "*File formats*" below).

4. Load a playground from a file and run it for a given number of steps.
5. Along a run, a dump of the system has to be saved with a given frequency.
6. You are required to provide a scalability study:
    a. OpenMP scalability: fix the number of MPI tasks to 1 per socket, and report the behaviour of the code when you increase the number of threads per task from 1 up to the number of cores present on the socket;
    b. Strong MPI scalability: given a fixed size (you may opt for several increasing sizes) show the run-time behaviour when you increase the number of MPI tasks (use as many nodes as possible, depending on the machine you run on).
7. [OPTIONAL] To spice-up the game, let's introduce a further rule to the two original and classical ones, let's call it "Finger of God": due to a random interaction with a, say, cosmic ray a living cell may die or generate like probabilities p_D and p_L respectively (these are two parameters whose values have to be provided at run-time).
   If a cell is set alive by chance, and it has less than 2 alive neighbours, then it spreads life on neighbour cells so that at least 2 of them are set alive. It's easy to understand that, given the previous rules, whether this new life will survive depends on its initial shape (you will quickly discover that a surprising number of life forms can even move).
   You are free to define whatever set of initial live patterns to be used in this case, and to experiment with it.
   An obvious advice is to use very small values for p_D and p_L to keep the Finger of God a perturbation instead of the dominant force.
8. [OPTIONAL] implement the evolution with a square-wave signal from a grid point randomly chosen at every time-step.
9. [OPTIONAL] implement the white-black evolution.

Your code should accept and handle the following command-line arguments:

ARGUMENT	MEANING
-i	initialize a playground
-r	run a playground
-k *num. value*	playground size
-e *[0|1]*	evolution type; 0 means "ordered", 1 meas "static"
-f *string*	the name of the file to be either read or written
-n *num. value*	number of steps to be calculated

ARGUMENT	MEANING
-s *num. value*	every how many steps a dump of the system is saved on a file (*0* meaning only at the end)

A couple of examples to clarify the usage of the command-line options:

- to create initial conditions:
  `$executable -i -k 10000 -f $my_initial_condition`

  this produces a file named `$my_initial_condition` that contains a playground of size `10000x10000`
- to run a play ground:
  `$executable -r -f $initial_conditions -n 10000 -e 1 -s 1`

  this evolves the initial status of the file `$initial_conditions` for 10000 steps with the static evolution mode, producing a snapshot at every time-step.

CLARIFICATION: following a question from a student, let be clear that `$my_initial_condition` and `$initial_conditions` are just placeholders that stands for any name the files may have. So, the `-f` option is intended to give to the program the name of the file that must be created (when used with the `-i` option) or read (with the `-r` option).

File format

The adopted file format is the `pgm` binary format for both the initial conditions and the evolutions snapshots, that allows to use any snapshot as a new initial conditions file for a new experiment.

The snapshot file name should be:

`snapshot_nnnnn`

where `nnnnn` (with 5 digits, padded with zeros) is the time-step it refers to.

Appendix I - Reading/Writing a `PGM` image

The `PGM` image format, companion of the `PBM` and `PPM` formats, is a quite simple and portable one.
It consists in a small header, written in ASCII, and in the pixels that compose the image, written all one after the others as integer values.
Each pixel may occupy either 1 or 2 bytes, and its value in `PGM` corresponds to the grey

level of the pixel expressed in the range [0..255] or [0..65535]; since in our case the only possibility is "dead/alive" then we adopt the single byte representation and the only two values that are meaningful are 0 and 255.
Even if also the pixels can be written in ASCII format, we require the usage of a binary format.

The header is a string that can be formatted like the following:

```
printf( "%2s %d %d\n%d\n", magic, width, height, maximum_value );
```

where `magic` is a magic number that for `PGM` is "" `P4`", `width` and `heigth` are the dimensions of the image in pixels, and `maximum_value` is either `<256` or `<65536`.
If `maximum_value < 256`, then 1 byte is sufficient to represent all the possible values and each pixel will be stored as 1 byte. instead, if `256 <= maximum_value < 65535`, 2 bytes are needed to represent each pixel (that in the current case would be a waste of bytes).

In the sample file `read_write_pgm_image.c` that you find the folder, there are the functions `write_pgm_image()` and `read_pgm_image()` that you can use to respectively write and read such a file.

In the same file, there is a sample code that generates a square image and write it using the `read_write_pgm_image()` function.
It generates a vertical gradient of $N_x \ times N_y$ pixels, where N_x and N_y are parameters. Whether the image is made by single-byte or 2-bytes pixels is decided by the maximum colour, which is also a parameter.
The usage of the code is as follows :

```
cc -o read_write_pgm_image read_write_pgm_image.c
./read_write_pgm_image [ max_val] [ width height]
```

as output you will find the image `image.pgm` which should be easily rendered by any decent visualizer .
NOTE: the `pbm` file format is conceptually similar to the proposed `pgm`; every pixel requires one bit only of information because it is a black&white encoding. As such - and that is the slightly trickier part - every byte in the file corresponds to 8 pixels that are mapped onto the single bits. In the `pgm` format, instead, a pixel corresponds to a byte in the file, which is easier to code. Optionally, you may want to implement also the `pbm` format, which is perfectly adequate to our case because every cell (that is a pixel in the image) may just be dead (0) or alive (1).

Appendix II - A note about hybrid MPI+OpenMP

As we mentioned in the class, a simple hybridization of MPI with OpenMP is quite straightforward. As you have seen, it is obviously not a requirement but just an opportunity for those among you that like to be challenged.

As long as you use OpenMP regions in a MPI process for computation *only* and *not* to execute MPI calls, everything is basically safe and you can proceed as usual with both MPI and OpenMP calls and constructs.

At a more advanced level, the same thread that initializes an OpenMP region (i.e. the thread 0), and only that one, can make the MPI calls from within an OpenMP region ("funneled" mode).
Possibly, every thread could call MPI routines but only one at one time ("serialized" mode).
Eventually, multiple threads can make MPI calls at the same time, which is to be handled carefully ("multiple" mode).

Initialize the MPI library with a call slightly different than `MPI_Init()`:

```
int mpi_provided_threaD_level;

MPI_Init_thread( &argc, &argv, MPI_THREAD_FUNNELED,
&mpi_provided_thread_level);

if ( mpi_provided_thread_level < MPI_THREAD_FUNNELED ) {
     printf("a problem arise when asking for MPI_THREAD_FUNNELED
level\n");
   MPI_Finalize();
   exit( 1 );
}

...;  // your code

MPI_Finalize();
return 0;
```

Running an hybrid code

On several platforms you may notice that running something like the following

```
export OMP_NUM_THREADS=$MY_NUM_THREADS
export OMP_PLACES=$MY_PLACE_CHOICE
export OMP_PROC_BIN=$MY_BIND_CHOICE
export OMP_DISPLAY_ENV=TRUE
mpirun -np $MPI_NTASKS $MY_EXEC ...
```

results in all the threads spawned by an MPI task run on the same physical core.
That is due to the default mapping of MPI tasks to physical resource. You know that it is possible to request a given mapping of the MPI tasks pool onto the hardware by, for instance, `-map-by`.
According to the standard,

```
Supported options include slot, hwthread, core, L1cache, L2cache,
L3cache, socket, numa, board, node, sequential, distance, and ppr
[ from the mpirun man page ]
```

So, the set of resources visible to a given MPI task will include only those included at the required mapping level. For instance, if you require `--map-by core`, then the resources visible to a given MPI task will include the logical threads of the core it will be assigned to.
If you require `--map-by L3cache` the resource set will consist of all the cores that share the L3 cache with the core that hosts the task itself. And so on.
Hence, we have this effect:

`--MAP-BY`	VISIBLE RESOURCE	IMPACT ON THREADS
`hwthread`	only the logical core on which the task runs	whatever is the `PLACES` and `BIND` options, all of them will run on the same logical core
`core`	the physical core on which the task runs	they will distribute on the visible logical cores accordingly to your policy choice
`L?cache`	the phisical cores that share the ? level of cache with the core the task run onto	*same as above*
`socket`	the phisical cores on the same socket the task run onto	*same as above*
`numa`	the numa region to which the core that hosts the task belongs to	*same as above*

`--map-by`	VISIBLE RESOURCE	IMPACT ON THREADS
`node`	all the node onto the task run	*same as above*

Hence, we suggest to run with `--map-by socket` at least.

Appendix III - Structure of the Report

The Report that you must submit is the major source of documentation for your work. As such, please, take care about its overall quality.
In the following we suggest a structure that is somehow typical for any report/paper of this kind.

SECTION	*DESCRIPTION*
Introduction	Brief overview and description of the problem tackled by your work. Be concise and focused, leaving a comprehensive discussion to some well-motivated reference.
Methodology	A discussion at *abstract level* of your approach and algorithms. Describe the possible options, if any, and describe and motivate your choices.
Implementation	Present and discuss the significant technical details of your implementation. For instance, for a given task you may have opted for a technical option among several possible: present, motivate and discuss your choice, also quoting a code snippet if needed. Example: how do you allocate the memory? Did you opt for either domain- or functional-decomposition among MPI tasks? Why? and how did you design it? How is your code threaded? etc.
Results & Discussion	Present and discuss your results, which at least are the scaling relations listed above in the text. Add any other result and relative figures from optional topics you you did select any.
Conclusions	Here you put your final remarks. How is the quality that you achieved with your code? did you understand the pitfalls? and above all: how would you improve it ?