|epcc|

MPI Casestudy: Parallel Cellular Automaton

David Henty

Contents

List of Figures

1 Introduction

The aim of this exercise is to write a complete MPI parallel program for a simple 2D cellular automaton. We will start with a serial code that performs the required calculation but is also designed to make subsequent parallelisation as straightforward as possible. The grid of cells is stored in a large two-dimensional array, so the natural parallel approach is to use regular domain decomposition. As the update algorithm involves nearest-neighbour interactions between grid points, this will require boundary swapping between neighbouring processes and adding halos to the arrays. The exercise also utilises parallel scatter and gather operations. For simplicity, we will use a one-dimensional process grid (i.e. decompose the problem into slices).

The solution can easily be coded using either C or Fortran. The only subtlety is that the natural direction for the parallel decomposition of the arrays, whether to slice up the 2D grid over the first or second dimension, is different for the two languages. A sample serial solution, including a function to write the result to an image file, can be found in `MPP-automaton-ser.tar` on the MPP course web pages.

2 The Cellular Automaton

The update rules for a cell are based on the values of the cell and its four nearest neighbours (left, right, up and down).

First, we initialise the grid:

- declare an integer array of size $L \times L$;
- each of the cells are set to 1 ("alive") or 0 ("dead") with probability ρ using a random number generator - for example, if $\rho = 0.60$ then roughly 60% of the cells will be alive.

We then apply the update rules many times:

- Loop over many steps
  - Compute the number of neighbours of each cell including itself (a number between 0 and 5)
  - If the nmber of neighnours is 2, 4 or 5: new cell is alive
  - Otherwise: new cell is dead
- End loop over steps

Our solution involves looking for the four nearest neighbours of each cell, but what about cells at the edges of the grid which have no neighbours? We do not want to write extra code to handle these as a special case, so a common solution to this problem is to define a "halo" around the grid: we surround our grid with a fixed layer of dead cells.

For an $L \times L$ grid we therefore use an $(L + 2) \times (L + 2)$ array and set the halos values to be zero.

Download the serial code `MPP-automaton-ser.tar` from the MPP course web pages. It should compile and run as supplied. The default parameters are:

- system size $L = 768$
- initial cell density $\rho = 0.49$
- maximum number of steps = $10 \times L$

It takes a single command-line parameter – the random number seed. To reproduce the results shown here use $seed = 1234$.

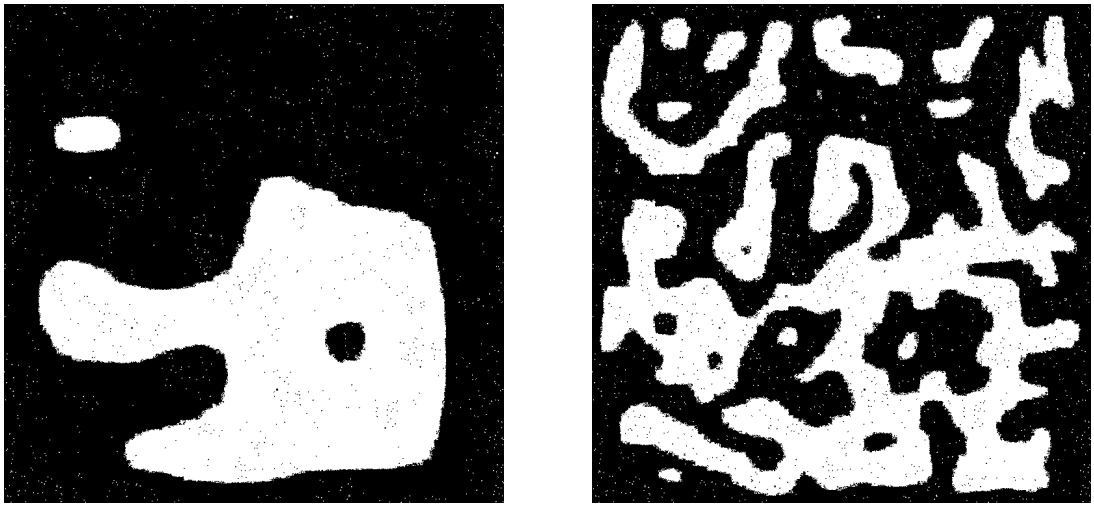

Figure 1: Output with default parameters (left), and maximum steps = L instead of $10 \times L$ (right)

```
[user@cirrus]$ tar xvf MPP-automaton-ser.tar
automaton/C-SER/
automaton/C-SER/arralloc.h
automaton/C-SER/automaton.c
...
automaton/C-SER/Makefile
automaton/C-SER/unirand.c
[user@cirrus$] cd automaton/C-SER/
[user@cirrus$] module load intel-compilers-19
[user@cirrus]$ make
icc -O3 -Wall -c automaton.c
icc -O3 -Wall -c cellio.c
icc -O3 -Wall -c unirand.c
icc -O3 -Wall -o automaton automaton.o cellio.o unirand.o
[user@cirrus]$ ./automaton 1234
automaton: L = 768, rho = 0.490000, seed = 1234, maxstep = 7680
automaton: rho = 0.490000, live cels = 289513, actual density = 0.490846
automaton: number of live cells on step 500 is 256177
automaton: number of live cells on step 1000 is 242046
automaton: number of live cells on step 1500 is 232972
...
automaton: number of live cells on step 7000 is 186961
automaton: number of live cells on step 7500 is 185340
cellwrite: opening file <cell.pbm>
cellwrite: writing data ...
cellwrite: ... done
cellwrite: file closed
[user@cirrus]$ module load ImageMagick
[user@cirrus]$ display cell.pbm
```

You should see output identical to the left-hand image in Figure 1. The right-hand image is to help illustrate how the simulation progresses: small clusters of cells gradually merge into a few large clusters.

2.1 Method

We need to have halos on the main arrays. It is simplest if the indices range from 0 to $N+1$, where the grid itself is represented by indices $1, 2, 3, \ldots, N$ and the halos are indices 0 and $N+1$. We can do this in C and Fortran by using:

```
int     cell[LX+2][LY+2];      // C
integer cell(0:LX+1, 0:LY+1)   !  Fortran
```

Here, LX and LY are the local dimensions of the cell array which in general will depend on the number of processes P.

For initialisation and IO we also use an $L \times L$ array called $allcell$ which *does not have any halos*.

In the serial code, although we have separate constants LX and LY for the two dimensions of the arrays, these are both set to L. However, in the parallel algorithm, we will have to alter these definitions as the array sizes on each process will be smaller by a factor of the number of processes P.

We will see that, for C programs, $LX = \frac{L}{P}$ and $LY = L$; for Fortran, $LX = L$ and $LY = \frac{L}{P}$. This is why we distinguish between L, LX and LY in the serial algorithm even though these values are all the same - it is to make it easier to parallelise.

2.2 Algorithm in pseudocode

1. declare $LX \times LY$ integer arrays *cell* and *neigh* with halos, and an $L \times L$ array *allcell* without halos
2. seed the random number generator from the command-line option and initialise *allcell*
3. loop over $i = 1, LX$; $j = 1, LY$
   - $cell_{i,j} = allcell_{i-1,j-1}$ (in C)
   - $cell_{i,j} = allcell_{i,j}$ (in Fortran)
4. zero the bottom, top, left and right halos of $cell$.
5. begin loop over steps
   - loop over $i = 1, LX$; $j = 1, LY$
     - set $neigh_{i,j} = cell_{i,j} + cell_{i-1,j} + cell_{i+1,j} + cell_{i,j-1} + cell_{i,j+1}$
   - loop over $i = 1, LX$; $j = 1, LY$
     - if $neigh_{i,j}$ = 2, 4 or 5
       - $cell_{i,j}$ = 1
       - increment number of live cells
     - else
       - $cell_{i,j}$ = 0
   - report the number of changes every 500 steps
6. end loop over steps
7. loop over $i = 1, LX$; $j = 1, LY$
   - $allcell_{i-1,j-1} = cell_{i,j}$ (in C)
   - $allcell_{i,j} = cell_{i,j}$ (in Fortran)
8. write out the result by passing $allcell$ to *cellwrite*

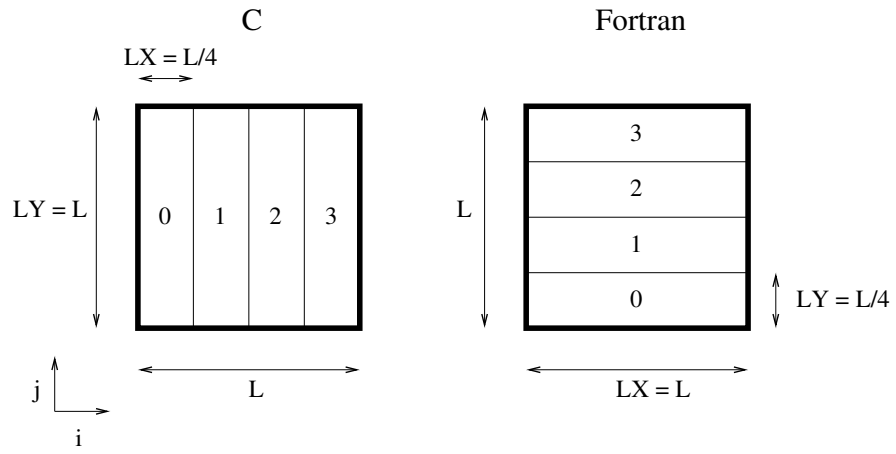

Figure 2: Decomposition strategies for 4 processes

3 Initial Parallelisation

For the initial parallelisation we will simply use trivial parallelism, i.e. each process will work on different sections of the image but with no communication between them. Although this will not be entirely correct, since we are not performing the required halo swaps, it serves as a good intermediate step to a full parallel code. Most importantly it will have exactly the same data decomposition and parallel IO approaches as a fully working parallel code.

It is essential that you complete this initial parallelisation before continuing any further

The entire parallelisation process is made much simpler if we ensure that the slices of the grids operated on by each process are contiguous pieces of the whole image (in terms of the layout in memory). This means dividing up the image between processes over the first dimension i for a C array `cell[i][j]`, and the second dimension j for a Fortran array `cell(i,j)`. Figure 2 illustrates how this would work on 4 processes where the slices are numbered according to the rank of the process that owns them.

Again, for simplicity, we will always assume that L is exactly divisible by the number of processes P. It is easiest to program the exercise if you make P a compile time constant (a `#define` in C or a `parameter` in Fortran).

- for C: $LX = L/P$ and $LY = L$
- for Fortran: $LX = L$ and $LY = L/P$

The simplest approach for initialisation and output is to do it all on one controller process. The $allcell$ array is scattered to processes after initialisation, and gathered back before file IO. Note this is not particularly efficient in terms of memory usage as we need enough space to store the whole map on a single process.

3.1 The parallel program

The steps to creating the first parallel program are as follows.

a) set LX and LY appropriately for the parallel program

b) create a new $LX \times LY$ array called $smallcell$ without any halos

c) initialise MPI, compute the $size$ and $rank$, and check that $size = P$

d) initialise $allcell$ on the controller process only

e) scatter the map from the controller to all other processes using `MPI_Scatter` with $sendbuf = allcell$ and $recvbuf = smallcell$

f) in steps 3 and 7, replace $allcell$ with $smallcell$

g) follow steps 4 through 6 of the original serial code exactly as before.

h) gather the map back to the controller from all other processes using `MPI_Gather` with $sendbuf = smallcell$ and $recvbuf = allcell$

i) write out the result from $allcell$ on the controller process only

Since we do the initialisation and file output in serial (i.e. on the controller process only), these parts do not require to be changed.

3.2 Testing the partial parallel program

For testing this initial (partial) paralellisation, the results are much clearer if you use a higher density such as $\rho = 0.61$.

Note that, as you have not yet implemented halo swapping, the parallel program will not be correct. However, the output on a single process should be identical to the serial program; on multiple processes you should see multiple small but independent cellular automata. Importantly, you can check that you have not broken the algorithm completely (run on a single process) and that you have implemented the scatter and gather correctly (senisible output on multiple processes).

How does your output compare to the correct (serial) answer on 1, 2 or 4 processes? Is the number of live cells the same? Do you understand what is happening?

As you increase the number of processes, does the total execution time decreasing as you expect? In terms of Amdahl's law, what are the inherently serial and potentially parallel parts of your (incomplete) MPI program?

4 Full Parallel Code

The only change now required to complete the full parallel code is to add halo swaps to the $cell$ array (which is the only array for which there are non-local array references of the form $cell_{i-1,j}$, $cell_{i,j+1}$ etc). This should be done once every step, immediately after the start of the loop over steps but before any other computation has taken place.

To do this, each process must know the $rank$ of its neighbouring processes. Referring to the C decomposition as shown in Figure 2, these are given by $rank - 1$ and $rank + 1$. However, since we have fixed boundary conditions on the edges, $rank$ 0 does not send any data to (or receive from) the left and $rank$ 3 need not send any data to (or receive from) the right. This is best achieved by defining a 1D Cartesian topology with non-periodic boundary conditions - you should already have code to do this from the previous "message round a ring" exercise. You also need to ensure that the processes do not deadlock by all trying to do synchronous sends at the same time. Again, you should re-use the code from the previous exercise.

The communications involves sending and receiving entire vertical or horizontal lines of data (depending on whether you are using C or Fortran). The process is as follows - it may be helpful to look at the appropriate decomposition in Figure 2.

Each of the two send-receive pairs is basically the same as a step of the ring exercise except that data is being sent in different directions (first clockwise then anti-clockwise). Remember that you can send and receive entire halos as a single message due to the way we have chosen to split the data amongst processes: you **should not** issue LY separate calls, one for every element.

For C:

- send the `LY` array elements (`cell[LX][j]; j = 1,LY`) to $rank + 1$
- receive `LY` array elements from $rank - 1$ into (`cell[0][j]; j = 1,LY`)
- send the `LY` array elements (`cell[1][j]; j = 1,LY`) to $rank - 1$
- receive `LY` array elements from $rank + 1$ into (`cell[LX+1][j]; j = 1,LY`)

For Fortran:

- send the `LX` array elements (`cell(i,LY); i = 1,LX`) to $rank + 1$
- receive `LX` array elements from $rank - 1$ into (`cell(i,0); i = 1,LX`)
- send the `LX` array elements (`cell(i,1); i = 1,LX`) to $rank - 1$
- receive `LX` array elements from $rank + 1$ into (`cell(i,LY+1); i = 1,LX`)

4.1 Testing the complete code

Again, run your program and compare the output images to the serial code. Are they exactly the same? Is the number of live cells the same at each step?

How do the execution times compare to the serial code and the previous (incomplete) parallel code? How does the time scale with P?

Plot parallel scaling curves for a range of problem sizes. You may want to insert explicit timing calls into the code so you can exclude the initialisation and IO overheads.