|epcc|

THE UNIVERSITY OF EDINBURGH

# Scientific Computing
# Cellular Automaton Exercise

## Aim

The aim of this exercise is to design serial and parallel algorithms to run a simple cellular automaton which attempts to model traffic flow. You should write some code to implement the serial algorithm

## The Model

The simulation box is a line of $N$ cells numbered $1, 2, \ldots, N-1, N$ (the road) which can each only have two values: 1 if a car is present on that section of road; 0 if it is empty. We will denote the value of cell number $i$ at time $t$ by $R^t(i)$.

The *update rules* for each time step are very simple:

- if the space in front of a car is empty then it moves forward one cell;
- otherwise it stays where it is.

Remember that we proceed through a series of iterations where the values of the cells at time $t+1$ depend entirely on the values at time $t$. In other words you should be able to apply the update rules to the cells at time $t$ in any order and you should still get the same values at $t + 1$.

After each iteration we compute the *average velocity* of the cars. This is calculated as the number of cars that move in an iteration divided by the total number of cars, and is a value between 0 and 1.

We use *periodic boundary conditions*, i.e. when a car moves off the right-hand-side of the road it reappears on the left, and vice-versa (this is the same as saying we are simulating a roundabout rather than a straight section of road). To do this, we identify $R^t(0)$ (the cell to the left of cell number 1) with $R^t(N)$ and $R^t(N + 1)$ (the cell to the right of cell number $N$) with $R^t(1)$.

## 1 Rules for Cellular Automaton

It should be fairly clear that, when we update to a new time step, the new value $R^{t+1}(i)$ depends on its old value at time $t$, and also on the old values of the neighbours. In other words, $R^{t+1}(i)$ depends on the three values $R^t(i-1)$, $R^t(i)$ and $R^t(i+1)$.

**You should fill in Table 1 and Table 2** to get the explicit form of the update rules for this cellular automaton. For convenience, we have split the 8 possible cases into two tables, one for an initially empty cell $R^t(i) = 0$, and the other for a full cell $R^t(i) = 1$. Writing things out in this explicit form you can see that there are in fact 256 possible 1D cellular automata (each corresponding to different update rules), but very few of them are in fact of any interest!

| | $R^t(i-1)=0$ | $R^t(i-1)=1$ |
|---|---|---|
| $R^t(i+1)=0$ | | |
| $R^t(i+1)=1$ | | |

Table 1: Values of $R^{t+1}(i)$ if $R^t(i)=0$

| | $R^t(i-1)=0$ | $R^t(i-1)=1$ |
|---|---|---|
| $R^t(i+1)=0$ | | |
| $R^t(i+1)=1$ | | |

Table 2: Value of $R^{t+1}(i)$ if $R^t(i)=1$

# 2 Serial Program

Having written the update rules, **write some code (in any programming language)** that implements them for a road of length $N$. The easiest approach is to have two arrays, one corresponding to the $old$ values at time $t$ and the other to the $new$ values at $t+1$. Loop through the cells computing the values of the the $new$ array based on $old$. After each complete time step, copy the $new$ array back to $old$ in preparation for the next time step. You must consider how to implement the periodic boundary conditions.

We need to define some initial conditions, i.e. the state of the road at time $t = 0$. This is best done using a random number generator; such generators typically return a real value distributed evenly between 0.0 and 1.0. We need to generate an initial road with a specific density of cars, e.g. we might want a reasonably congested road with a density of 0.75 (where, on average, three out of four cells are occupied):

- loop over the whole road $i = 1, 2, \ldots, N-1, N$
- pick a random number $rand$ in the range 0.0 to 1.0.
- if $rand < density$
  - set $R^0(i)$ = 1
- else
  - set $R^0(i)$ = 0
- end loop over $i$

It is easy to see that this works for densities of 0.0 and 1.0 (random number generators normally have the range $0.0 \leq rand < 1.0$), and in fact works for any value of the density.

## 2.1 Experiments

The aim is to reproduce the graph of average velocity versus car density that was outlined in the lectures.

- pick a large value for $N$, a few thousand should suffice
- choose a target value for the car density
- initialise the road as above
- compute the actual density which will be close to (but not exactly the same as) your target
- loop over many time steps $t = 0, 1, 2, \ldots$
  - update the road from time $t$ to $t + 1$ by updating all the cells
  - compute the average car velocity for this time step
- end loop over time steps

**You should do this for a range of densities to map out how the average velocity depends on density, and plot a graph of the results.**

You can just use a single (large) value of $N$, although you could check that the results do not depend strongly on this value (e.g. run two simulations of size $N$ and $2N$ for the same density).

Here are a few hints to help you:

1. As the initial distribution of cars is entirely random, the simulation may take many time steps to settle down to a steady state. You should see the velocity varying with time step, but eventually settling down to a fairly steady value. What you are looking for is the $asymptotic$ velocity, i.e. the average velocity per step after many time steps.
2. The amount of time taken to reach this steady state depends on the density (and the road length $N$), so it may take some experimentation to find out how long to run the model.
3. To check your code is correct, try a few simple examples where you know the answer:
    - a solid road (density of 1.0) should have velocity 0.0
    - a solitary car should move forward with velocity 1.0
    - if you fill half of the road with cars in a line and leave the other half free, the average velocity should increase linearly with time step until it reaches a maximum of 1.0 after $N/2$ time steps (think of the cars at a set of traffic lights)
    - the number of cars should always remain the same

# 3 Parallelisation

Examine the code that you have just written and consider how you would split up the calculation among multiple processors. Points to consider include:

- What parts of the calculation could be done independently and what parts would require some form of cooperation (ag synchronisation or communication)?
- What is the best way to handle any communications in your parallel code?
- Is it possible to implement both the boundary conditions and the communications using the same basic approach?

Remember also that you not only have to update the cars at each generation, but you also have to compute the total number of cars that move in order to calculate the velocity.

# 4 Message Passing Implementation

How would you implement your parallel scheme using a message-passing programming model on a distributed-memory parallel machine? Points to consider include:

- How would you divide the data amongst the different processes?
- When is communications required?
- How would you calculate the velocity at each time step?

Write some pseudo-code to illustrate your solution, assuming that you have access to simple routines to send and receive data between processes. You should consider two cases:

1. sending data is done *asynchronously* like sending a letter;
2. sending data is done *synchronously* like making a phone call.

Be careful to avoid deadlock, particularly in the second case.

If you prefer to think of an analogy, imagine that you and a friend want to simulate the traffic model by hand on the blackboards in your offices. You decide to split the calculation between you, and luckily your offices are relatively nearby. Unfortunately, the sound insulation is very good and you don't have a phone, so the only way to communicate is to go and speak to each other. You each can also only work in your own office as the blackboards are only large enough to hold half of the road. How would you organise the calculation? Try and come up with a scheme that minimises the number of times you have to get up and leave your office. How would you generalise your algorithm for three, four or eight people?

# 5 Shared Variables Implementation

How would you implement your parallel scheme using a shared-variables programming model on a shared-memory parallel machine? Points to consider include:

- How would you divide the calculation amongst the different threads?
- What data is shared and what is private?
- When is synchronisation required?
- How would you calculate the velocity at each time step?

Write some pseudo-code to illustrate your solution assuming that you have access to simple routines to create and manage multiple threads.

If you prefer an analogy, imagine that you and a friend want to simulate the traffic model by hand on a single large blackboard. The blackboard can hold all of the road and you and your friend can both work in the same office. You both also have private notebooks, but unfortunately you are now not allowed to speak to each other as it would make too much noise. How would you organise the calculation when you can only communicate by writing to and reading from the blackboard? Try and come up with a scheme that maximises the amount of time you both spend doing useful work. What variables should be shared (i.e. a single copy on the blackboard) or private (i.e. separate copies in each of your notebooks)? How would you generalise your algorithm for three, four or eight people?

## 5.1 Extra exercise

Can you guarantee that your method will give *exactly identical* answers if you repeat the calculation more than once using the same number of threads (or people)? What about using a different number of threads?