Message-Passing Thought Exercise

Traffic Modelling

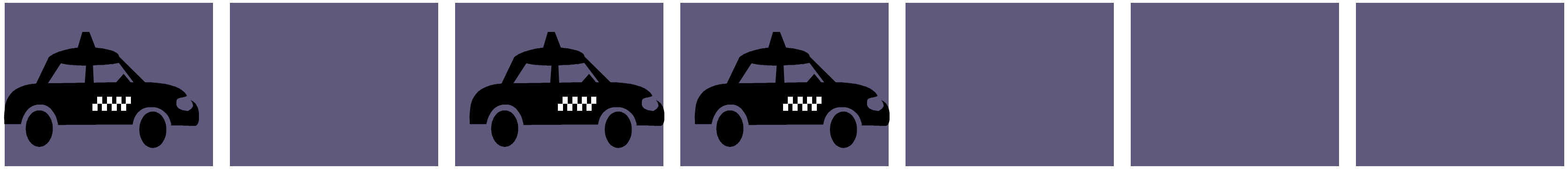
David Henty
EPCC
d.henty@epcc.ed.ac.uk
+44 131 650 5960

traffic flow

- we want to predict traffic flow
  - – to look for effects such as congestion
- build a computer model

simple traffic model

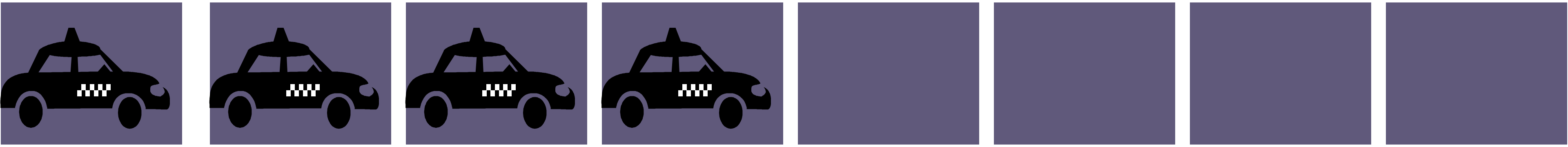
- divide road into a series of cells
  - either occupied or unoccupied
- perform a number of steps
  - each step, cars move forward if space ahead is empty



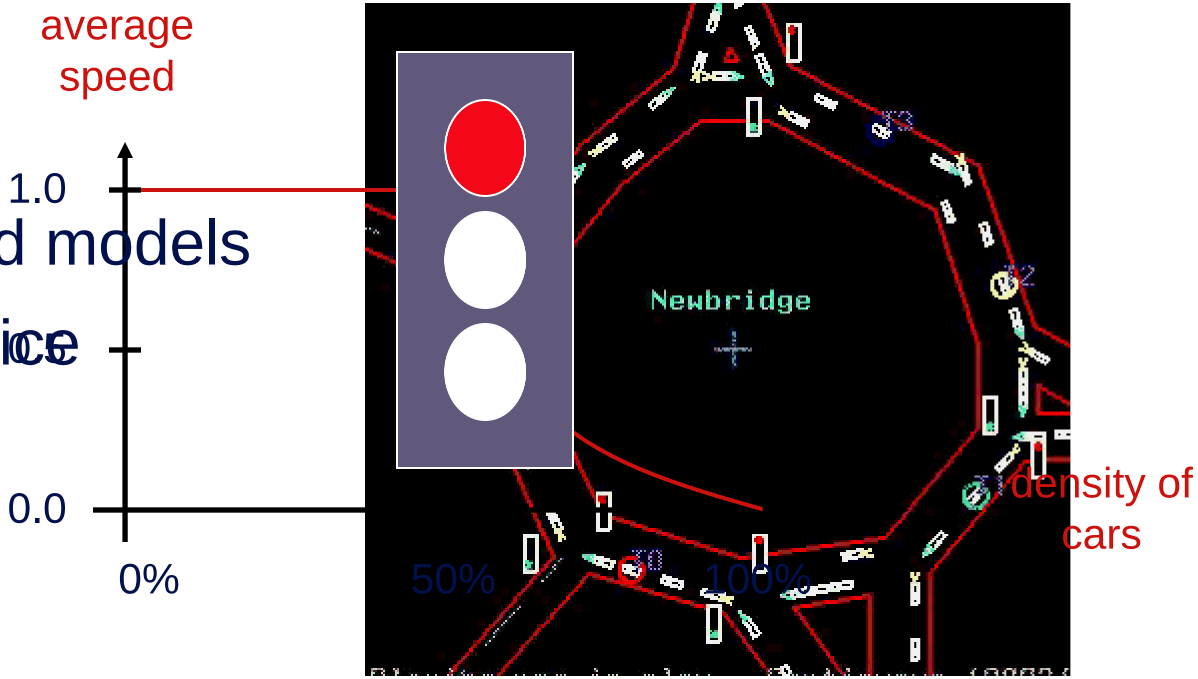
could do this by moving pawns on a chess board

traffic behaviour

- model predicts a number of interesting features
- traffic lights



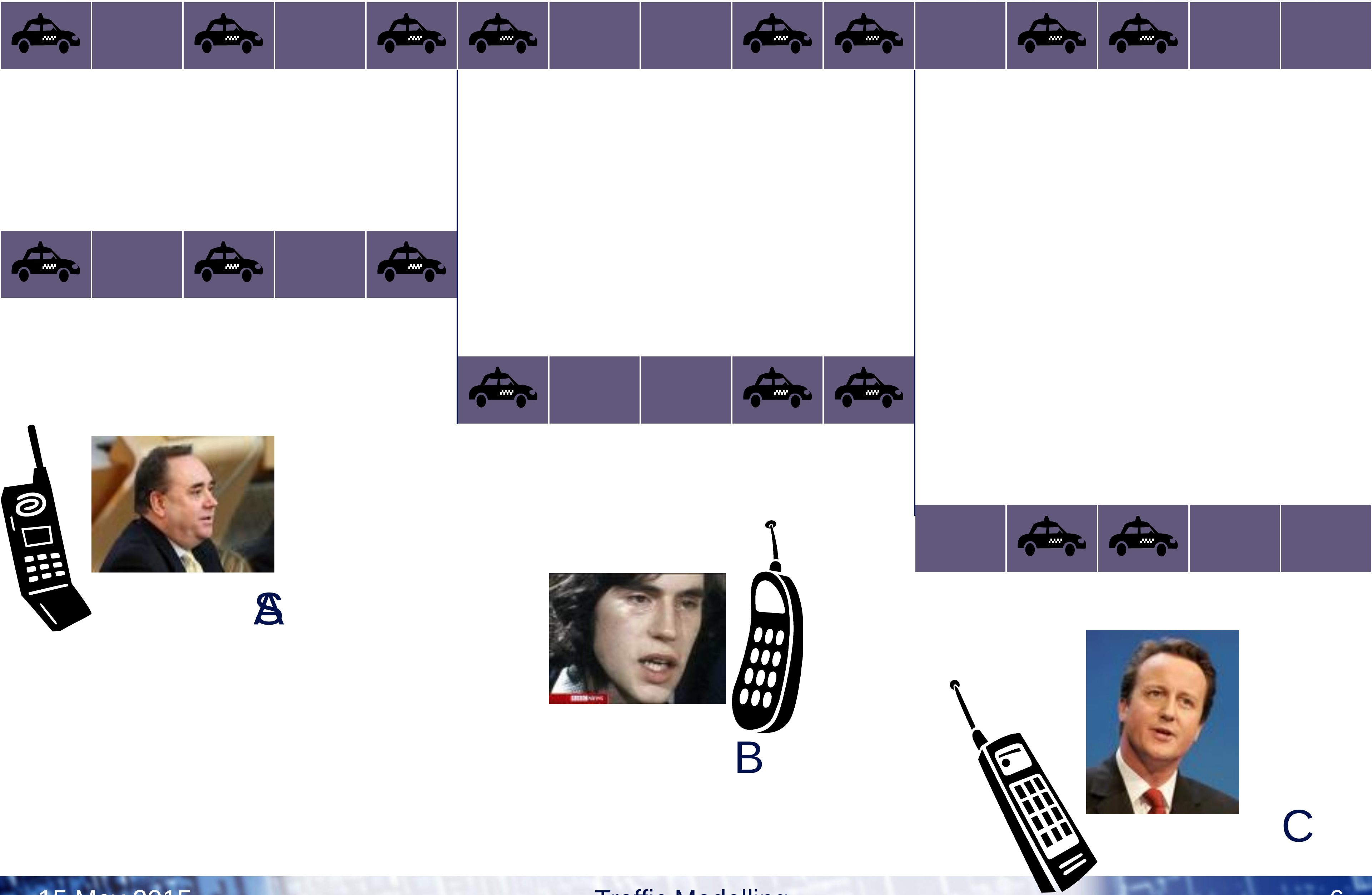
- congestion
- more complicated models are used in practice



how fast can we run the model?

- measure speed in Car Operations Per second
  - how many COPs?
- around 2 COPs
- but what about three people?
  - can they do six COPs?

a parallel traffic model



State Table

- If $R^t(i) = 0$, then $R^{t+1}(i)$ is given by:

	$R^t(i\text{-}1) = 0$	$R^t(i\ \text{-}1) = 1$
$R^t(i+1) = 0$	0	1
$R^t(i+1) = 1$	0	1

- If $R^t(i) = 1$, then $R^{t+1}(i)$ is given by:

	$R^t(i\text{-}1) = 0$	$R^t(i\ \text{-}1) = 1$
$R^t(i+1) = 0$	0	0
$R^t(i+1) = 1$	1	1

Pseudo Code (serial)

```
declare arrays old(i) and new(i), i = 0,1,...,N,N+1
initialise old(i) for i = 1,2,...,N-1,N (eg randomly)
loop over iterations
  set old(0) = old(N) and set old(N+1) = old(1)
  loop over i = 1,...,N
    if old(i) = 1
      if old(i+1) = 1 then new(i) = 1 else new(i) = 0
    if old(i) = 0
      if old(i-1) = 1 then new(i) = 1 else new(i) = 0
  end loop over i
  set old(i) = new(i) for i = 1,2,...,N-1,N
end loop over iterations
```

Pseudo Code (serial with subroutines)

```
   declare arrays old(i) and new(i), i = 0,1,...,N,N+1
   initialise old(i) for i = 1,2,...,N-1,N (eg randomly)
   loop over iterations
!   Implement boundary conditions
      set old(0) = old(N) and set old(N+1) = old(1)
!   Update road
      call newroad(new, old, N)
!   Prepare for next iteration
      set old(i) = new(i) for i = 1,2,...,N-1,N
   end loop over iterations
```

Pseudo Code (distributed memory)

```
!  assume we are running on P processes
   declare arrays old(i) and new(i), i = 0,1,...,N/P,N/P+1
   initialise old(i) for i = 1,2,...,N/P-1,N/P (eg randomly)
   loop over iterations
!   Implement boundary conditions (processes arranged as a ring)
      set old(0) on this process to old(N/P) from previous process
      set old(N/P+1) on this process to old(1) from next process
!   Update road
      call newroad(new, old, N/P)
!   Prepare for next iteration
      set old(i) = new(i) for i = 1,2,...,N/P-1,N/P
   end loop over iterations
```

Halo swapping

```
! Implement boundary conditions
    set old(0) on this process to old(N/P) from previous process
    set old(N/P+1) on this process to old(1) from next process
```

- Implement this using blocking receives (e.g. MPI_Recv) and:
  - synchronous send (routine blocks until message is received)
  - e.g. MPI_Ssend
- or
  - asynchronous send (message copied into buffer, returns straight away)
  - e.g. MPI_Bsend
- or
  - non-blocking synchronous send (no buffering but immediate return)
  - e.g. MPI_Issend / MPI_Wait

Synchronous sends

```
! Implement boundary conditions
   Ssend(old(N/P),   up)
   Recv (old(1),   down)
   Ssend(old(1),   down)
   Recv (old(N/P+1), up)
```

- Guaranteed to deadlock

Asynchronous (buffered) sends

```
! Implement boundary conditions
    Bsend(old(N/P),   up)
    Recv (old(1),   down)
    Bsend(old(1),   down)
    Recv (old(N/P+1), up)
```

- Where do synchronisation issues become important?

  `call newroad(new, old, N/P) ?`

  - OK because we are writing new but only reading old
  - **`set old(i) = new(i) ?`**
  - only OK because **`Bsend`** has copied **`old(1)`** and **`old(N/P)`**

- We **don’t** really care if/when the message is received
  - we **do** really care if/when we can safely reuse the local send buffers

Non-blocking (immediate) sends

```
! Implement boundary conditions
   Issend(old(N/P),   up)
   Recv  (old(1),   down)
   Issend(old(1),   down)
   Recv  (old(N/P+1), up)
   call newroad(new, old, N/P)
   set old(i) = new(i) for i = 1,2,...,N/P-1,N/P)
```

Non-blocking (immediate) sends

```
! Implement boundary conditions
   Issend(old(N/P),   up)
   Recv  (old(1),   down)
   Issend(old(1),   down)
   Recv  (old(N/P+1), up)
   call newroad(new, old, N/P)
   set old(i) = new(i) for i = 1,2,...,N/P-1,N/P)
! Wait for communications to complete before next iteration
   wait(up)
   wait(down)
```

Non-blocking (immediate) sends

```
! Implement boundary conditions
   Issend(old(N/P),   up)
   Recv  (old(1),   down)
   Issend(old(1),   down)
   Recv  (old(N/P+1), up)
   call newroad(new, old, N/P)
   set old(i) = new(i) for i = 1,2,...,N/P-1,N/P)
! Wait for communications to complete before next iteration
   wait(up)
   wait(down)
```

- Incorrect!
  - overwriting old is the key issue
  - need to know boundary values of old are sent before overwriting

Non-blocking sends: correct

```
! Implement boundary conditions
    Issend(old(N/P),   up)
    Recv  (old(1),   down)
    Issend(old(1),   down)
    Recv  (old(N/P+1), up)
   call newroad(new, old, N/P)
    wait(up)
    wait(down)
    set old(i) = new(i) for i = 1,2,...,N/P-1,N/P)
```

Delaying the waits

```
! Implement boundary conditions
   Issend(old(N/P),   up)
   Recv  (old(1),    down)
   Issend(old(1),    down)
   Recv  (old(N/P+1), up)
   call newroad(new, old, N/P)
   set old(i) = new(i) for i = 2,3,...,N/P-1)
   wait(up)
   old(N/P = new(M/P)
   wait(down)
   old(1) = new(1)
```

RMA synchronisation

- Similar synchronisation issues to non-blocking message passing
  - but worse!

Remote Memory Access

- Imagine we can do halo swaps **directly** with read or write
  - where do synchronisation issues become important?
  - what assumptions are you making about remote reads and writes?
- Consider remote read first

```
old(0)     = old(N/P) from previous process
old(N/P+1) = old(1)   from next process


call newroad(new, old, N/P)


set old(i) = new(i) for i = 1,2,...,N/P-1,N/P
```

Remote Memory Access

- Imagine we can do halo swaps **directly** with read or write
  - where do synchronisation issues become important?
  - what assumptions are you making about remote reads and writes?
- Consider remote read first

```
    old(0)     = old(N/P) from previous process
    old(N/P+1) = old(1)   from next process

    call newroad(new, old, N/P)
!  synchronise to ensure my old values have all been read
    set old(i) = new(i) for i = 1,2,...,N/P-1,N/P
!  synchronise to ensure neighbours' old values have been
!  updated before I read them on the next iteration
```

assuming reads are blocking like Recv

Remote Memory Access

- Imagine we can do halo swaps **directly** with read or write
  - where do synchronisation issues become important?
  - what assumptions are you making about remote reads and writes?
- Consider remote writes

```
set old(0)     on next process     = old(N/P)
set old(N/P+1) on previous process = old(1)


call newroad(new, old, N/P)


set old(i) = new(i) for i = 1,2,...,N/P-1,N/P
```

Remote Memory Access

- Imagine we can do halo swaps **directly** with read or write
  - where do synchronisation issues become important?
  - what assumptions are you making about remote reads and writes?
- Consider remote writes

```
     set old(0)     on next process     = old(N/P)
     set old(N/P+1) on previous process = old(1)
! synchronise to ensure my halos on old have been updated
     call newroad(new, old, N/P)

     set old(i) = new(i) for i = 1,2,...,N/P-1,N/P
```

Remote Memory Access

- Imagine we can do halo swaps **directly** with read or write
  - where do synchronisation issues become important?
  - what assumptions are you making about remote reads and writes?
- Consider remote writes

```
     set old(0)     on next process     = old(N/P)
     set old(N/P+1) on previous process = old(1)
! synchronise to ensure my halos on old have been updated
     call newroad(new, old, N/P)

     set old(i) = new(i) for i = 1,2,...,N/P-1,N/P
! synchronise to ensure my neighbours have finished with their
! old arrays (in “newroad”) before overwriting them
```

assuming writes behave like a Bsend

Summary

- Synchronisation in PGAS approaches is not simple
  - easy to write programs with subtle synchronisation errors

- Think first, code later!