Message Passing Programming

Modes, Tags and Communicators

- Lecture will cover
  - explanation of MPI modes (**`Ssend`**, **`Bsend`** and **`Send`**)
  - meaning and use of message tags
  - rationale for MPI communicators

- These are all commonly misunderstood
  - essential for all programmers to understand modes
  - often useful to use tags
  - certain cases benefit from exploiting different communicators

Modes

- **`MPI_Ssend`** (Synchronous Send)
  - guaranteed to be synchronous
  - routine will not return until message has been delivered
- **`MPI_Bsend`** (Buffered Send)
  - guaranteed to be asynchronous
  - routine returns before the message is delivered
  - system copies data into a buffer and sends it later on
- **`MPI_Send`** (standard Send)
  - may be implemented as synchronous or asynchronous send
  - this causes a lot of confusion (see later)

MPI_Ssend

Process A

Process B

Process A

Process B

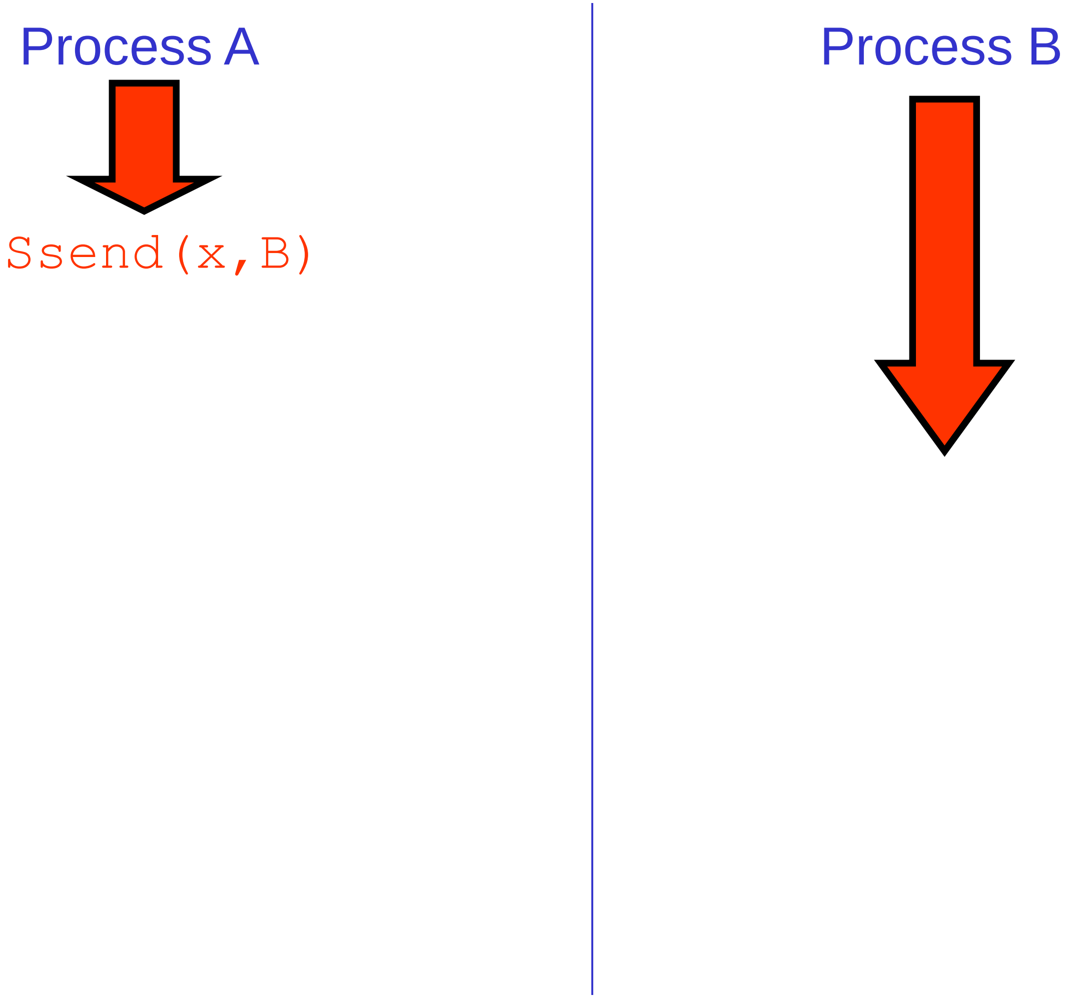
MPI_Ssend

Process A

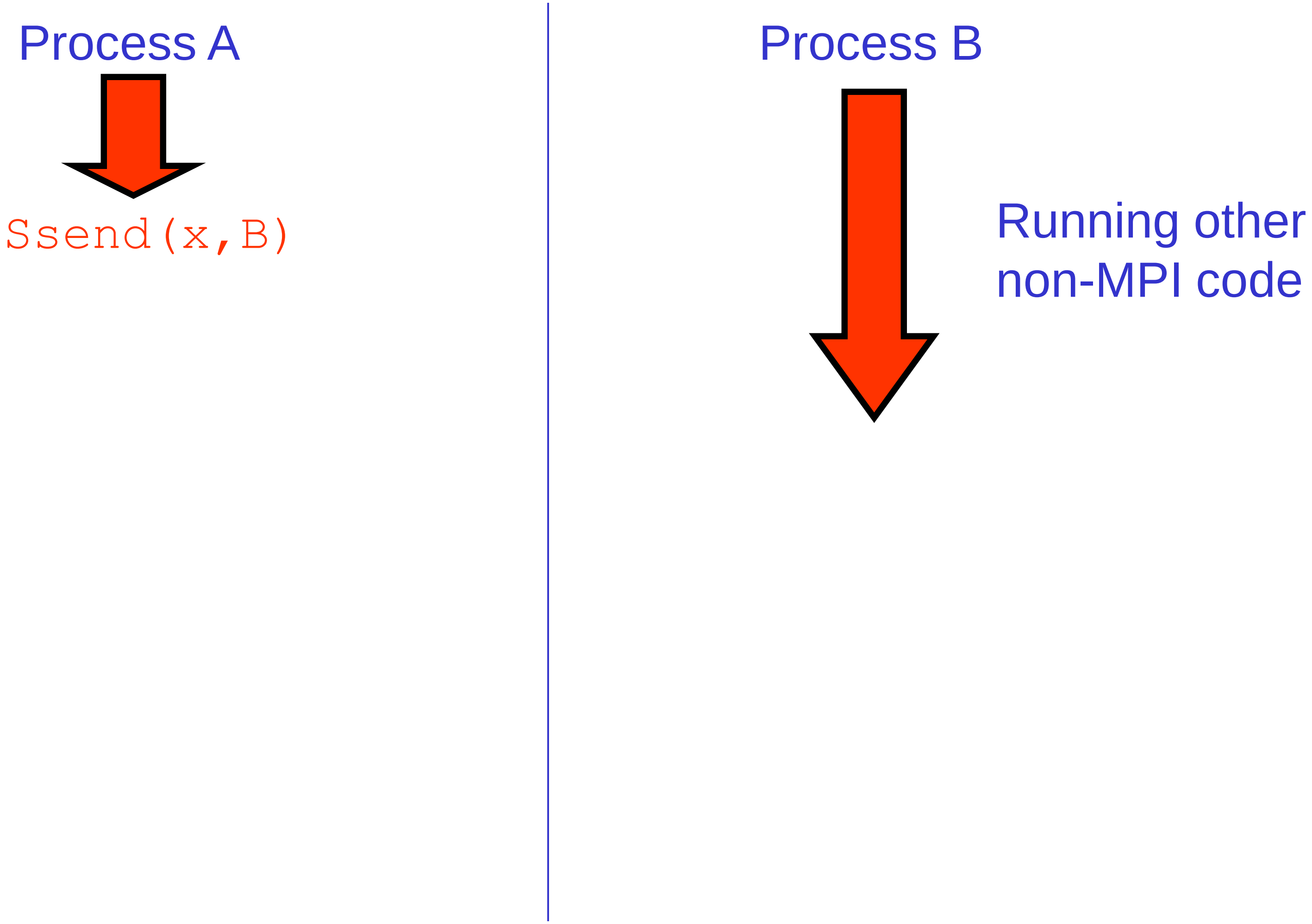
Process B

`Ssend(x,B)`

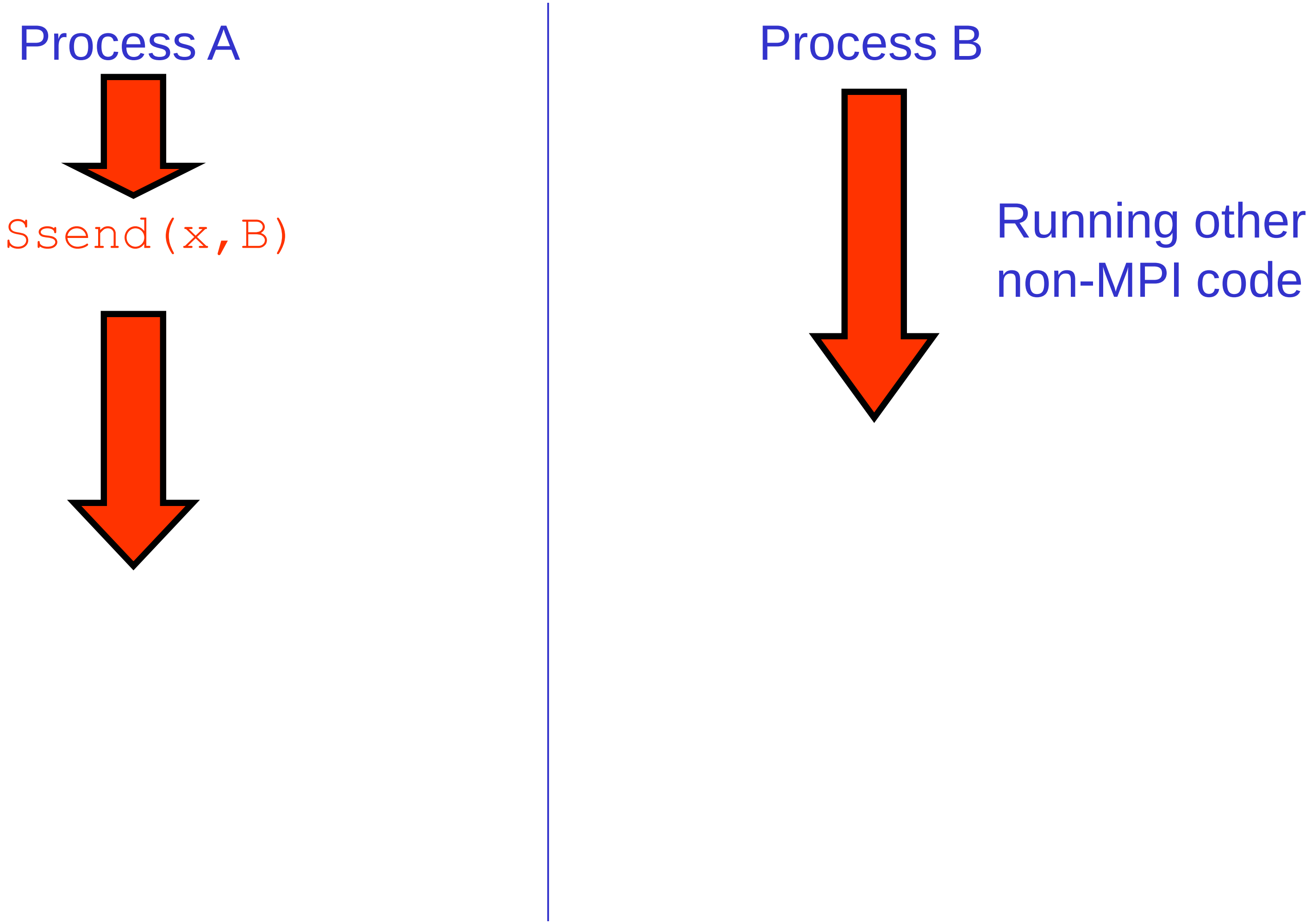
MPI_Ssend



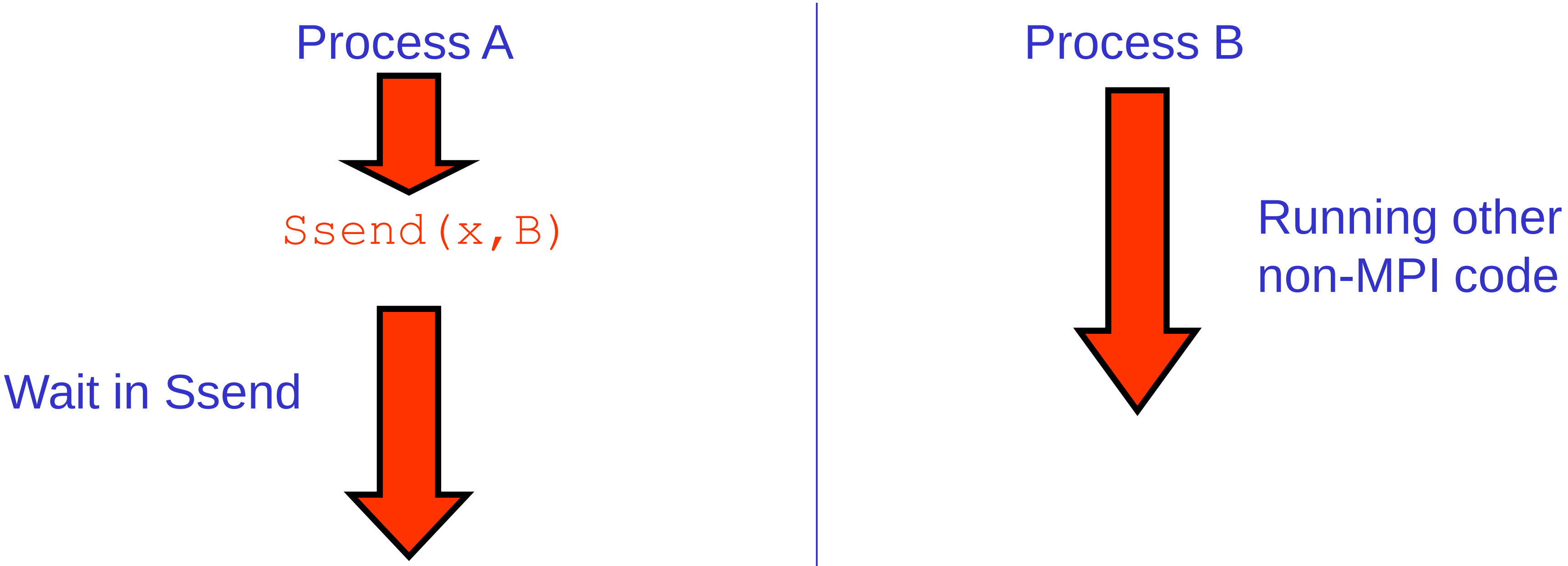
MPI_Ssend



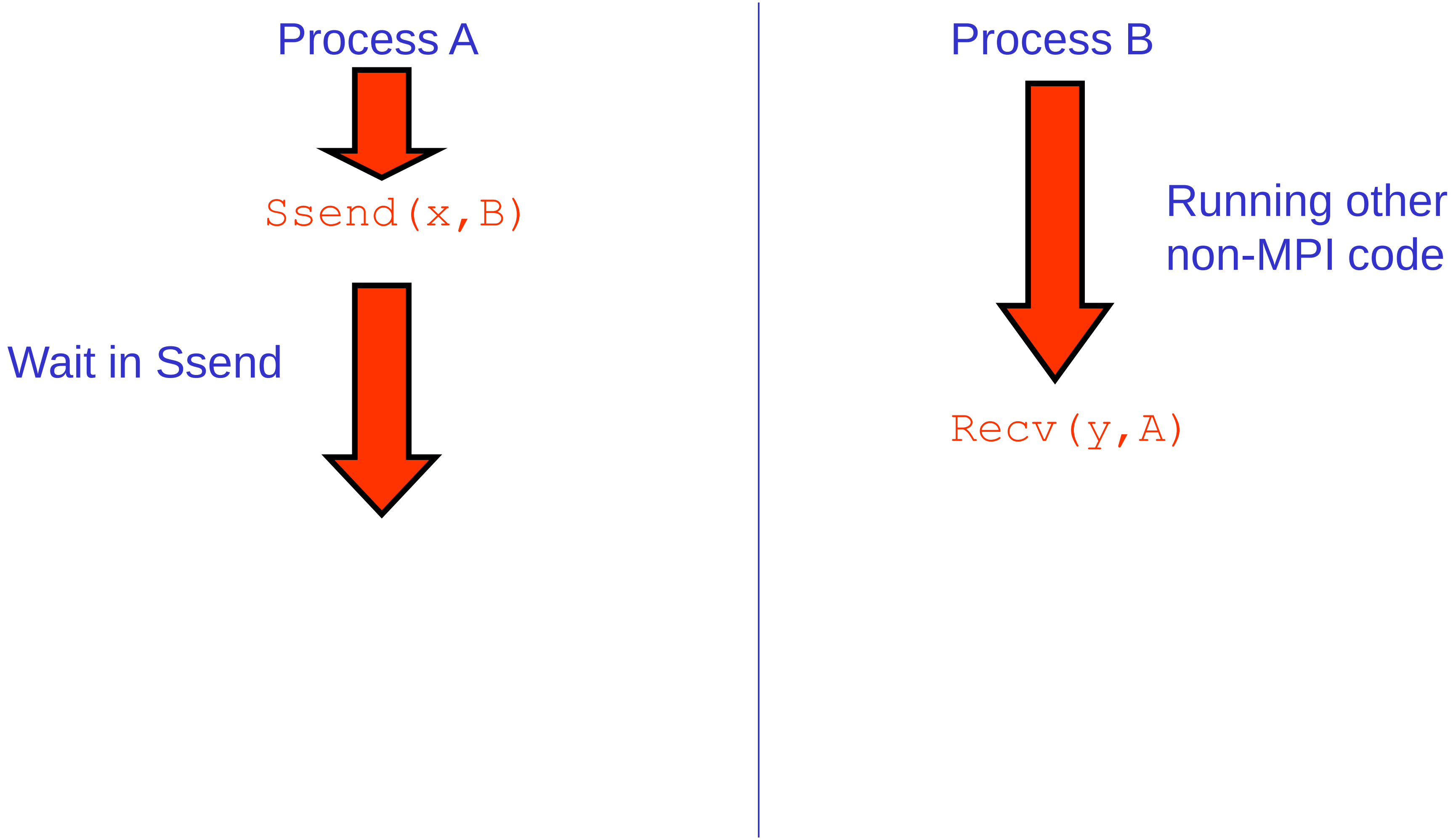
MPI_Ssend



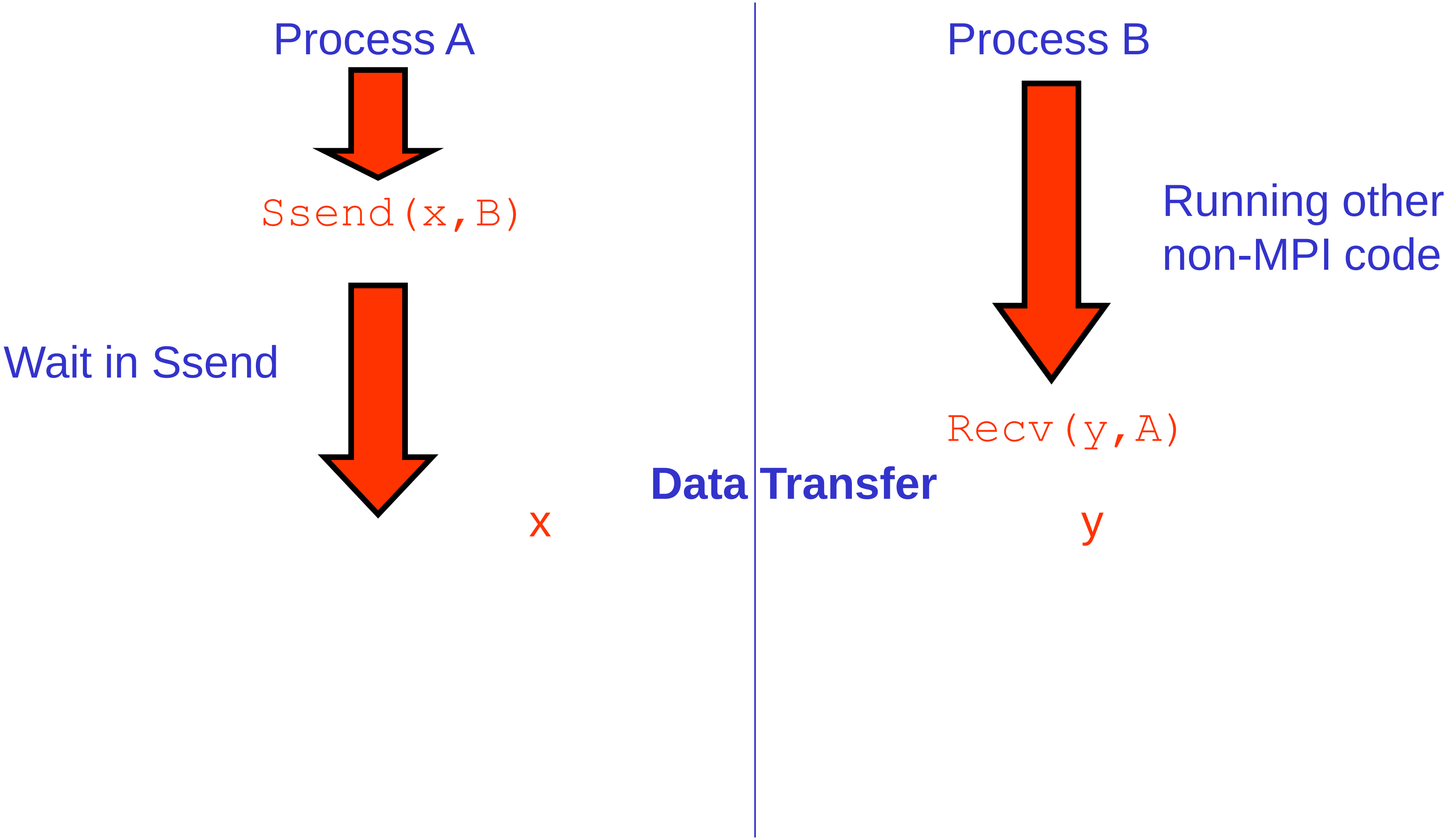
MPI_Ssend


Process A
Ssend(x,B)
Wait in Ssend
Process B
Running other
non-MPI code

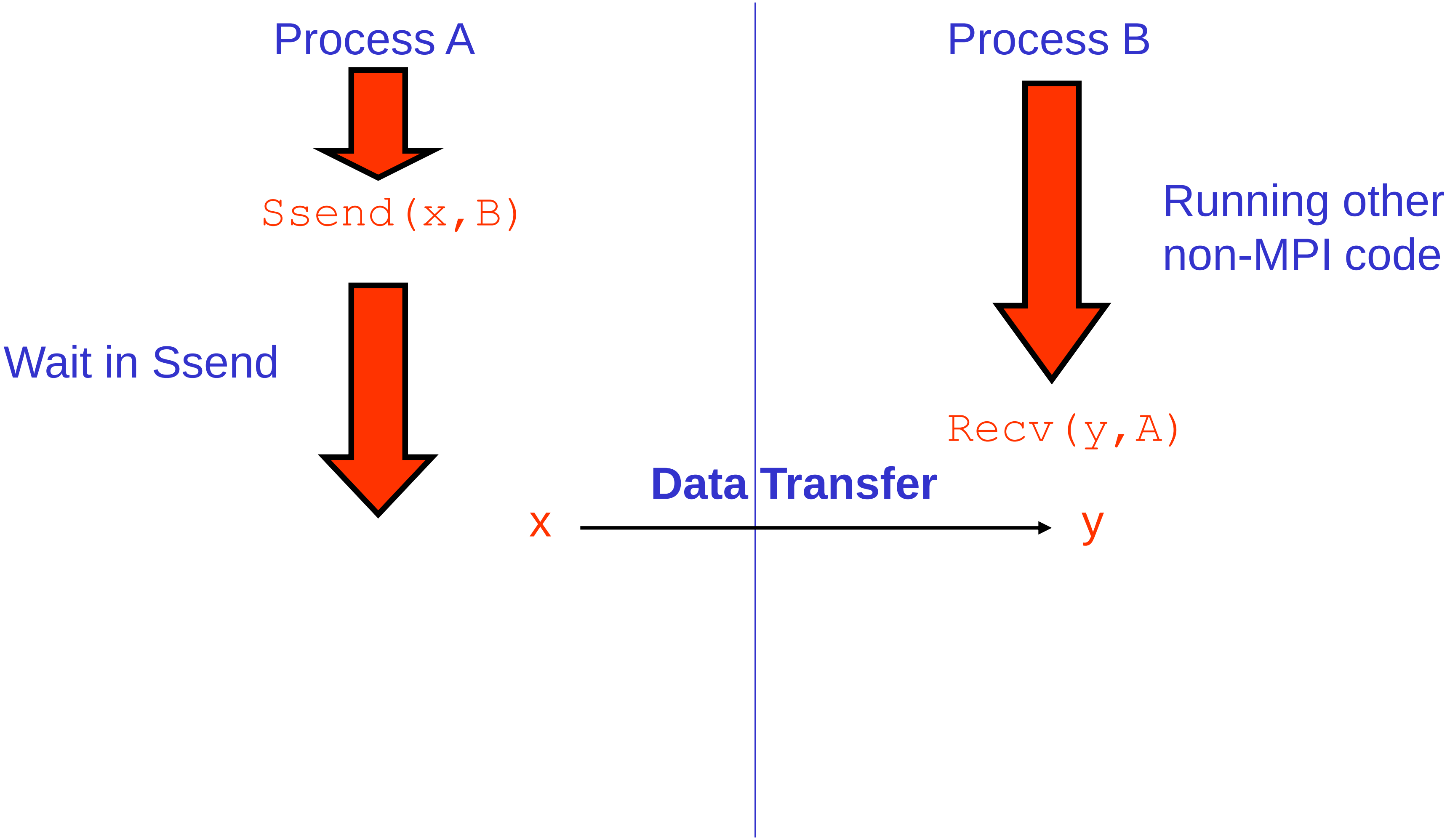
MPI_Ssend


Process A
Ssend(x,B)
Wait in Ssend
Process B
Running other
non-MPI code
Recv(y,A)

MPI_Ssend

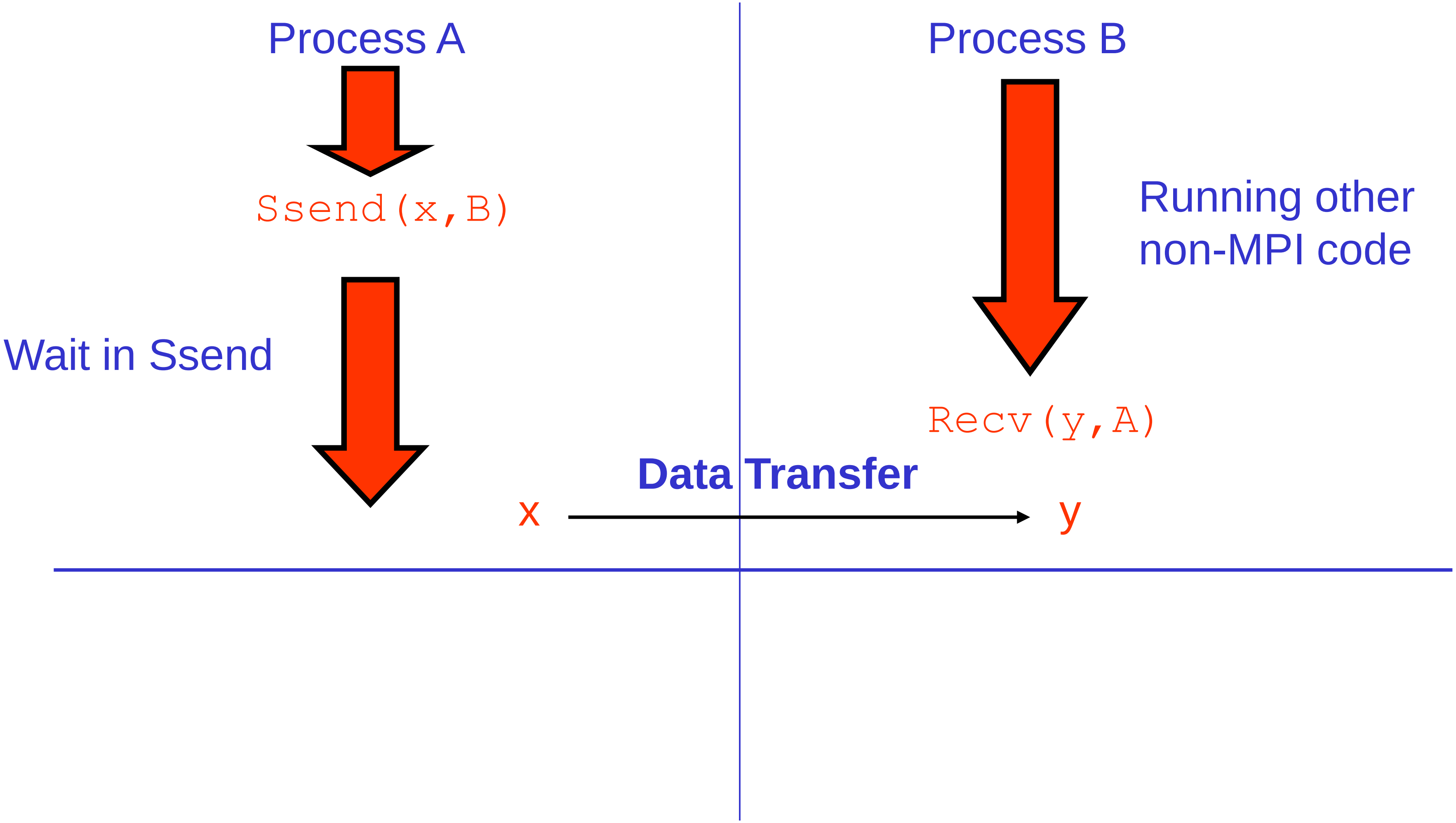

Process A
Ssend(x,B)
Wait in Ssend
x
Data Transfer
Process B
Running other non-MPI code
Recv(y,A)
y

MPI_Ssend

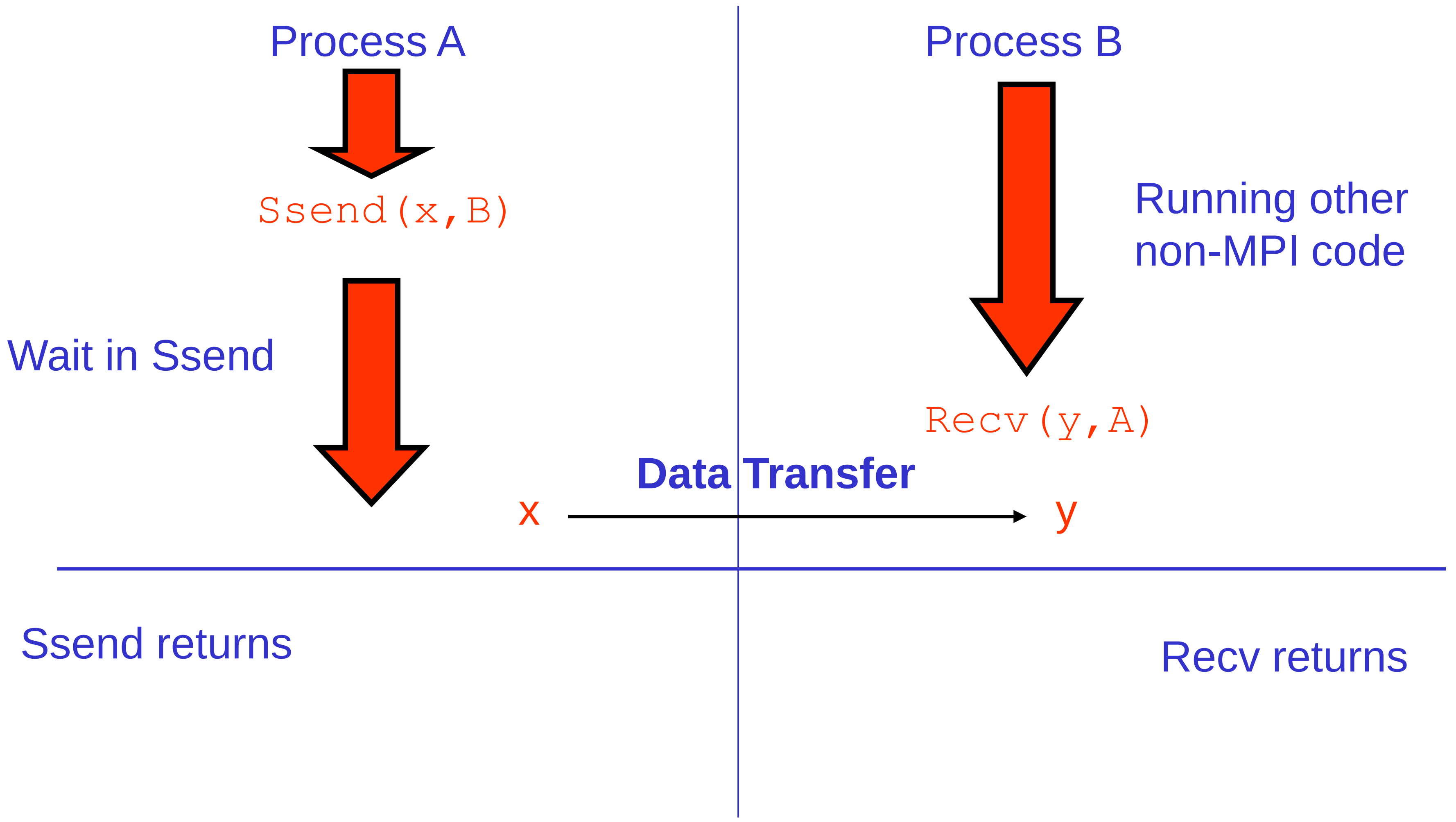


Process A
Process B
Ssend(x,B)
Running other non-MPI code
Wait in Ssend
Recv(y,A)
Data Transfer
x
y

MPI_Ssend

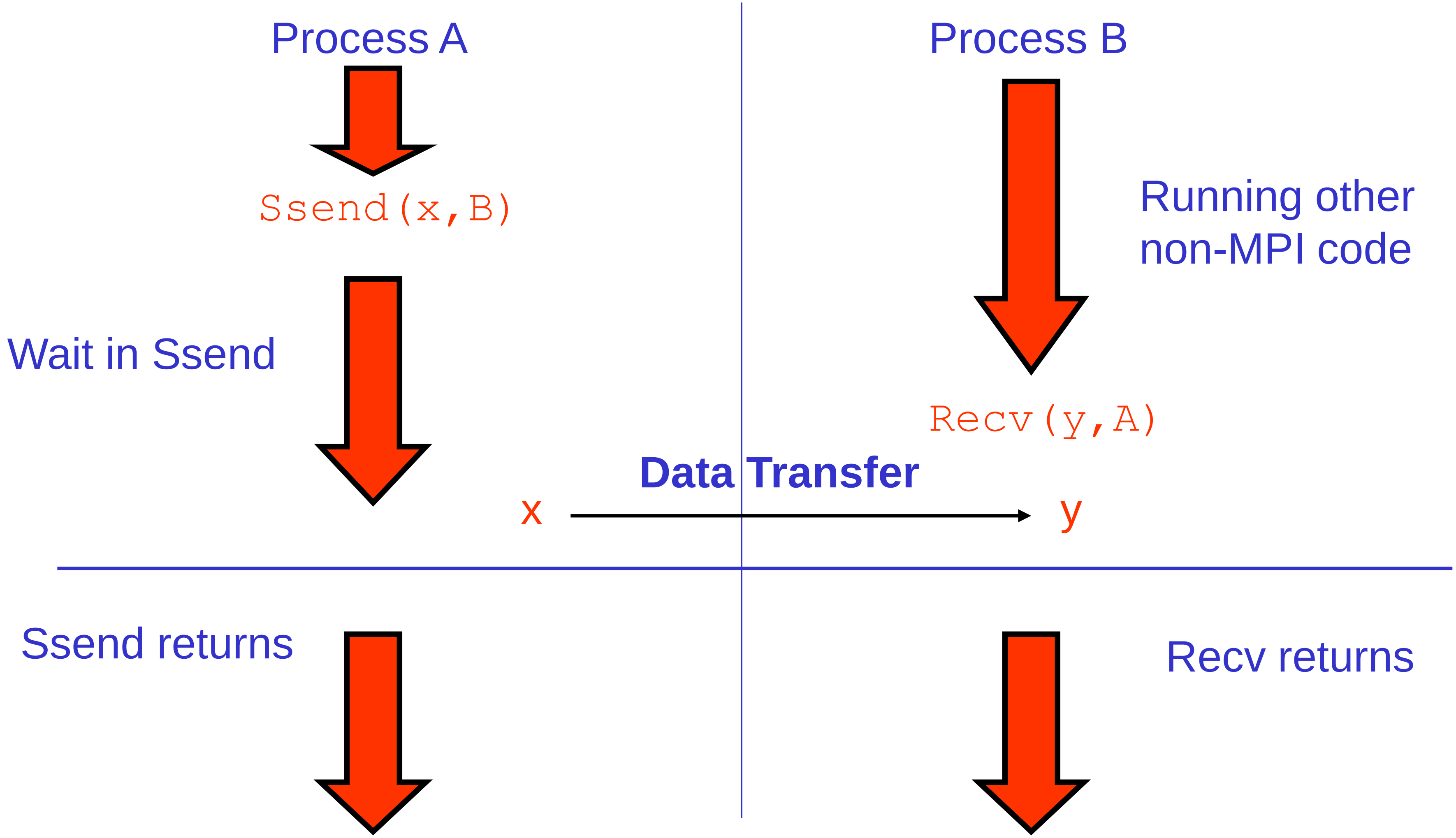

Process A
Process B
Ssend(x,B)
Running other non-MPI code
Wait in Ssend
Recv(y,A)
Data Transfer
x
y

MPI_Ssend



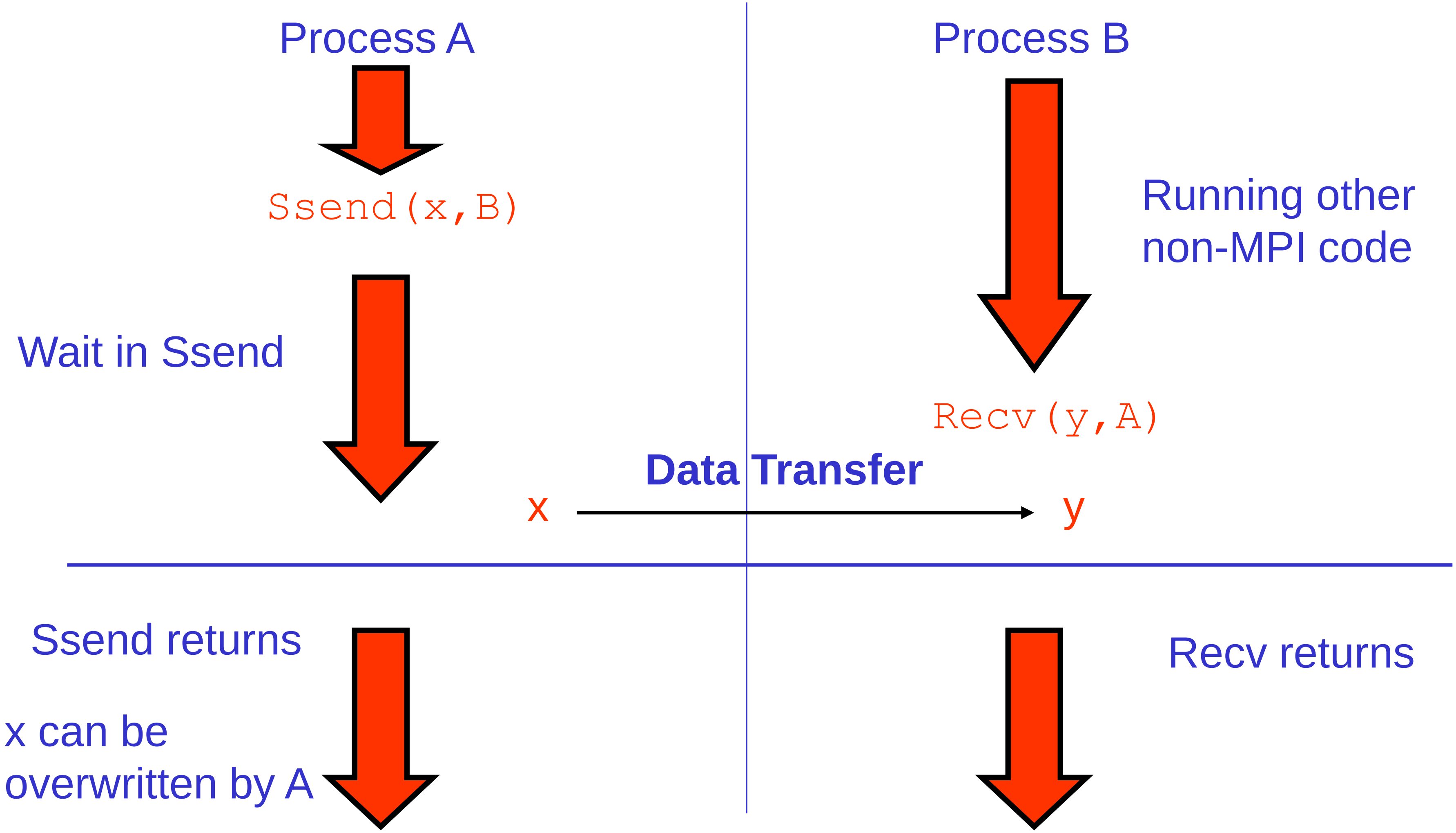
Process A
Ssend(x,B)
Wait in Ssend
Process B
Running other non-MPI code
Recv(y,A)
Data Transfer
x
y
Ssend returns
Recv returns

MPI_Ssend

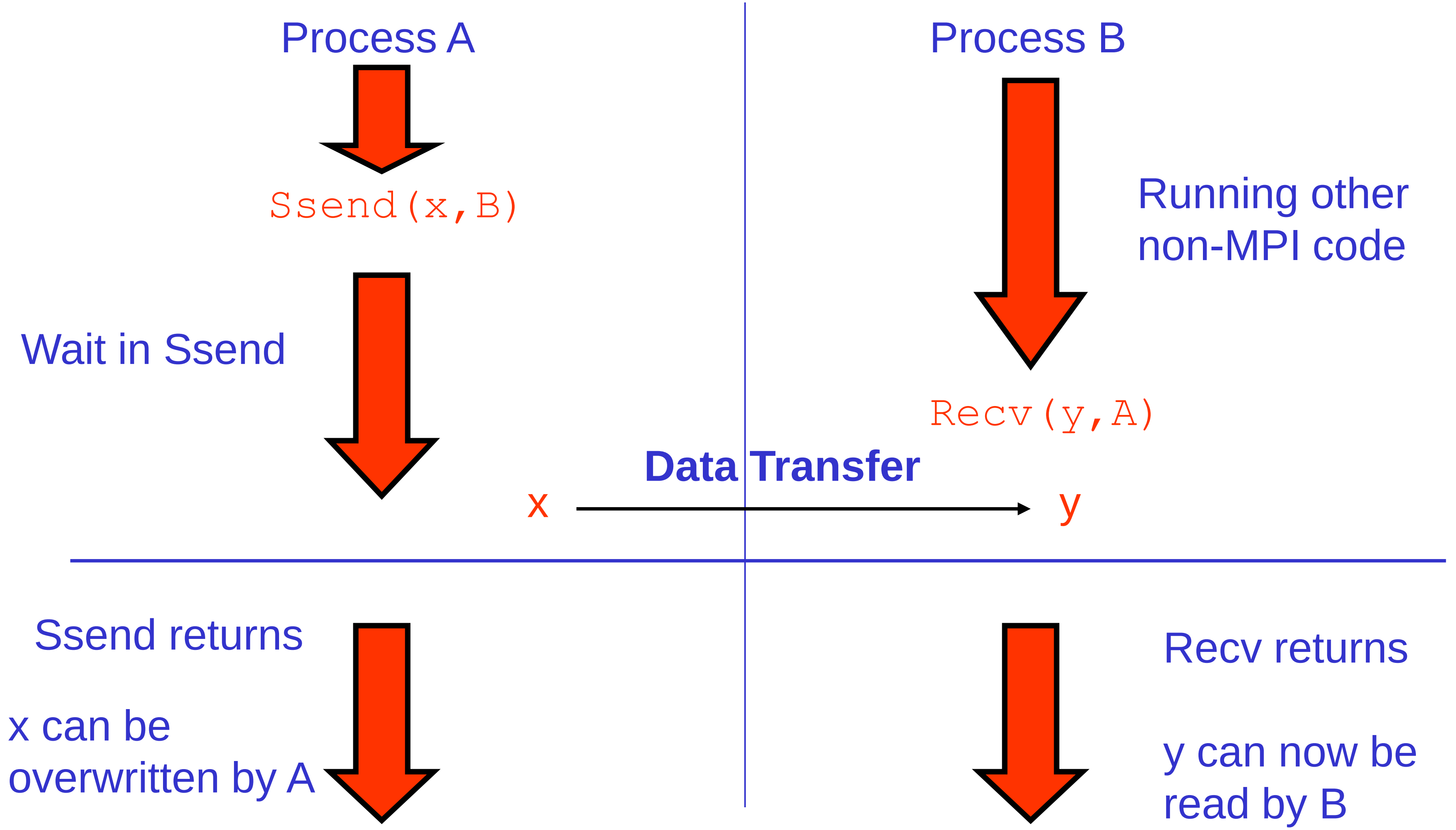


Process A
Process B
Ssend(x,B)
Running other non-MPI code
Wait in Ssend
Recv(y,A)
Data Transfer
x
y
Ssend returns
Recv returns

MPI_Ssend


Process A
Process B
Ssend(x,B)
Running other
non-MPI code
Wait in Ssend
Recv(y,A)
Data Transfer
x
y
Ssend returns
x can be
overwritten by A
Recv returns

MPI_Ssend


Process A
Ssend(x,B)
Wait in Ssend
Process B
Running other non-MPI code
Recv(y,A)
Data Transfer
x
y
Ssend returns
x can be overwritten by A
Recv returns
y can now be read by B

MPI_Bsend

Process A

Process B

MPI_Bsend

Process A

Process B

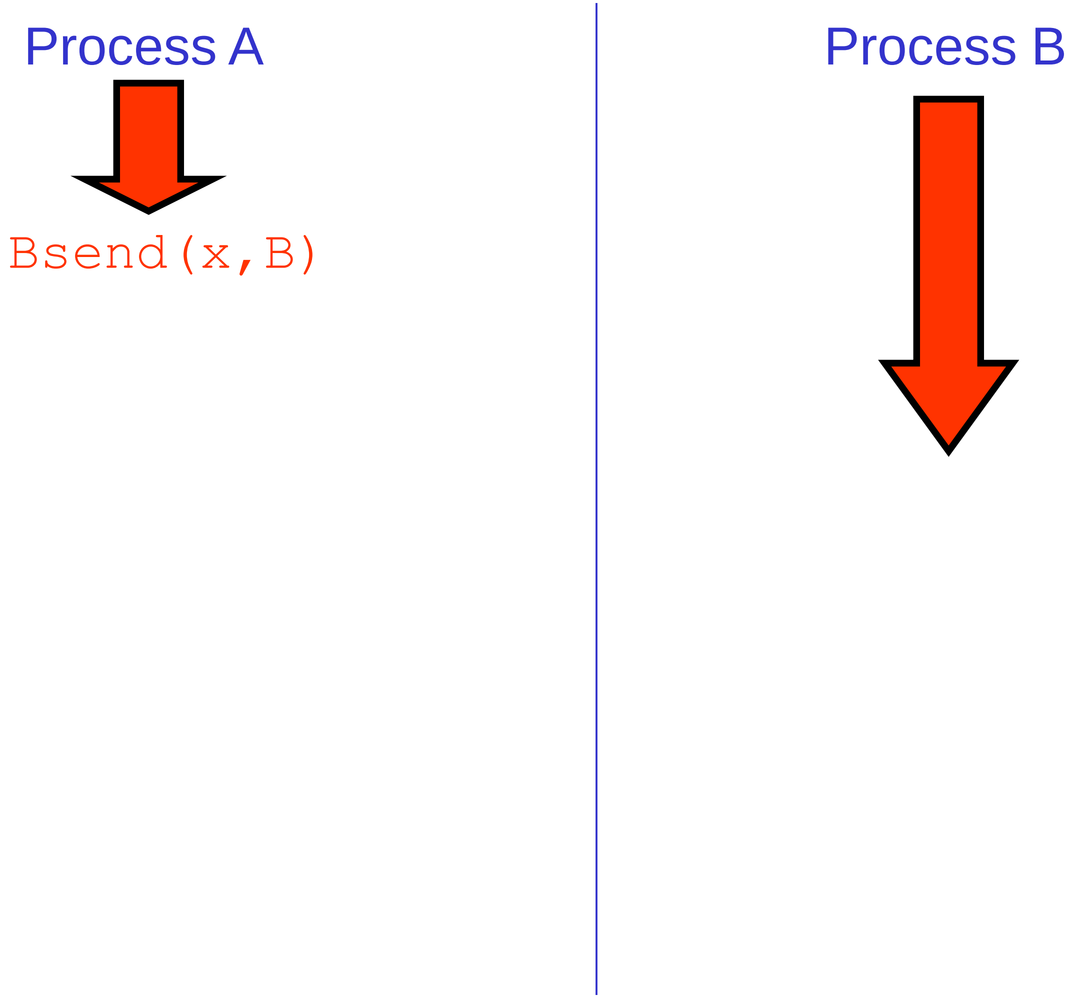
MPI_Bsend

Process A

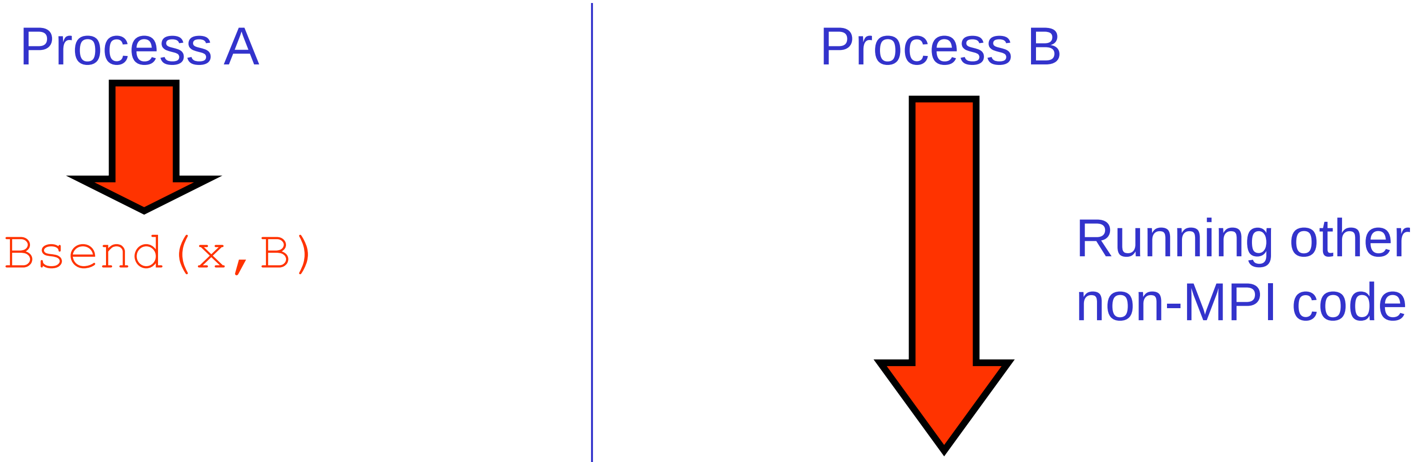
`Bsend(x,B)`

Process B

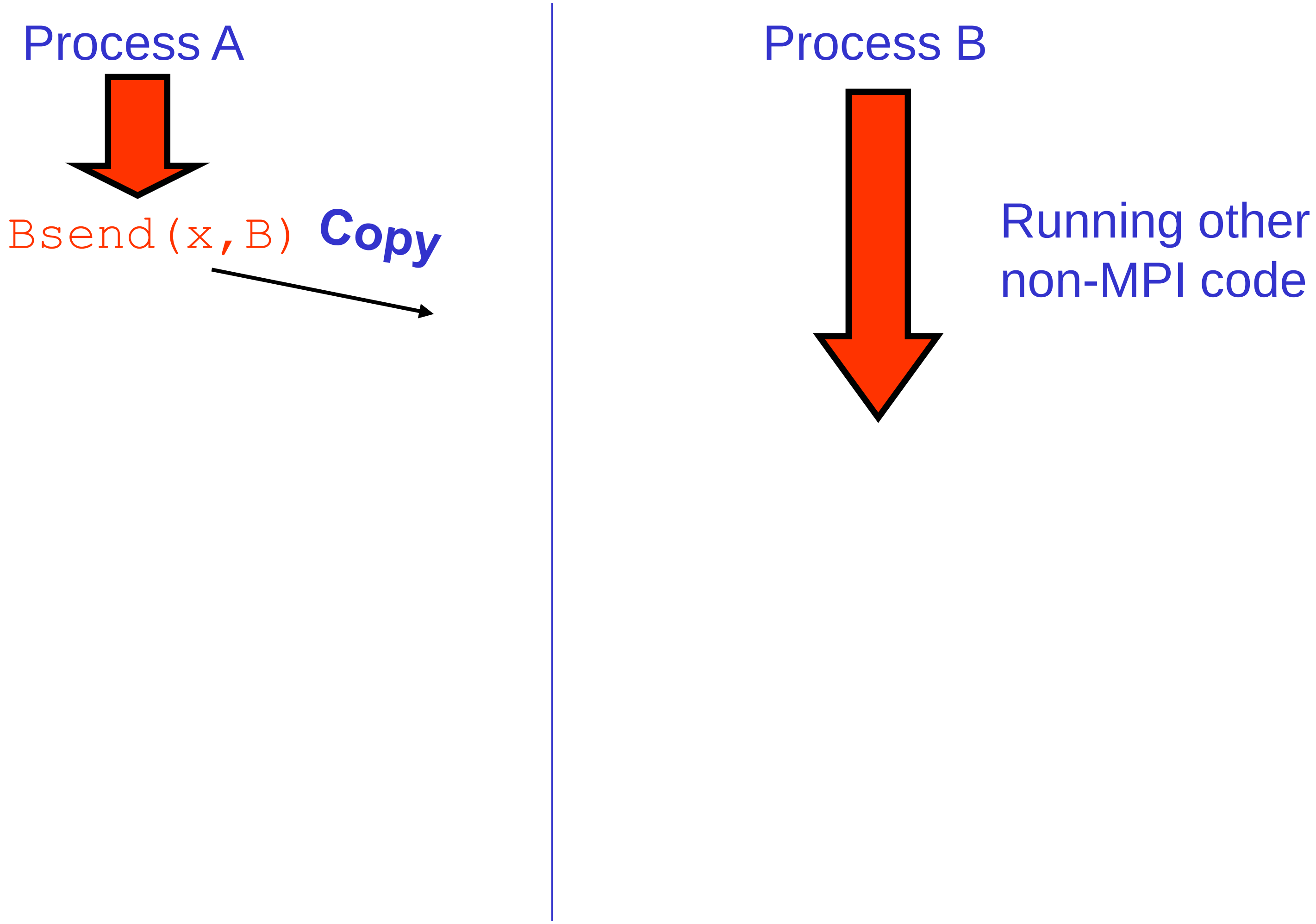
MPI_Bsend



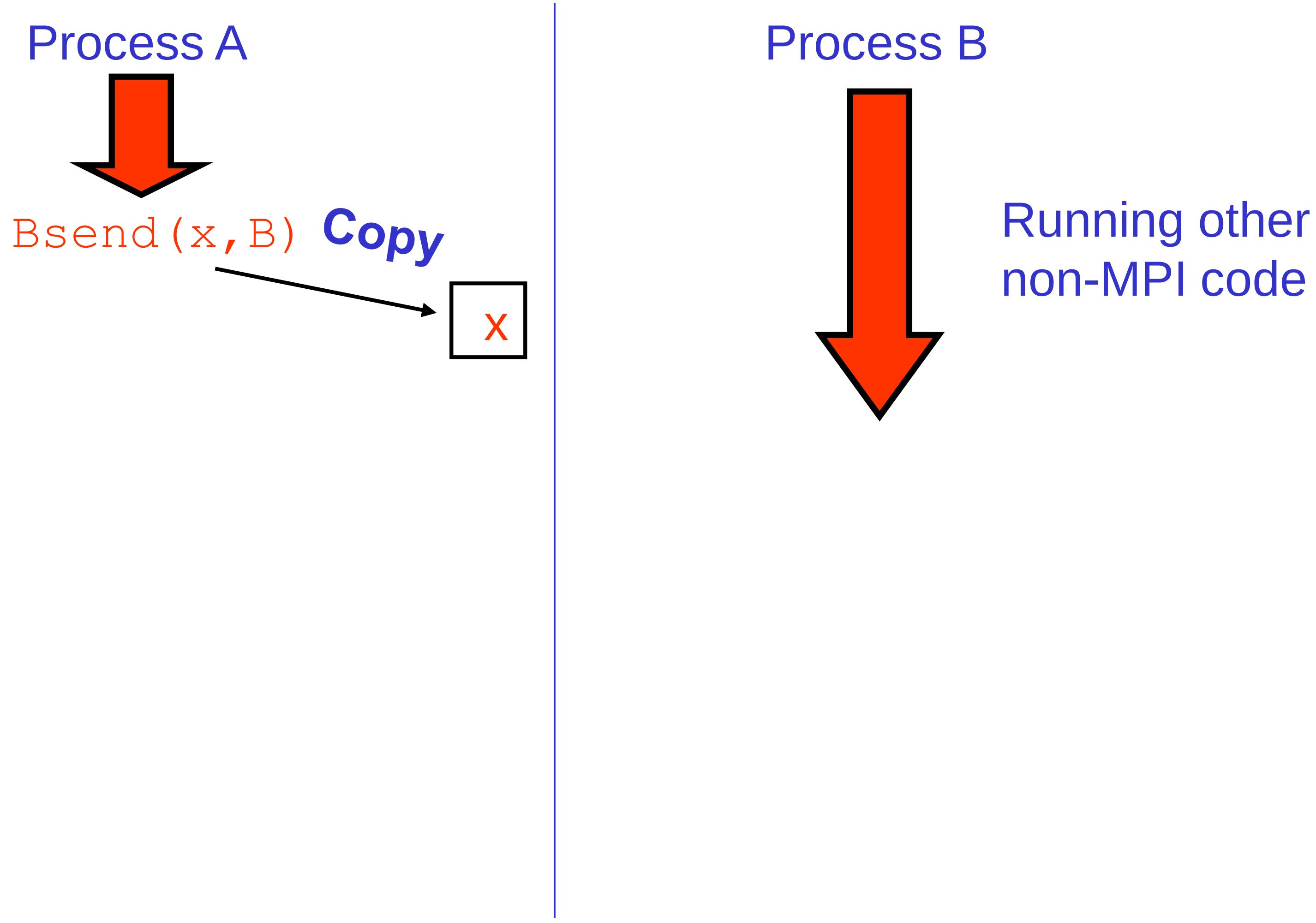
MPI_Bsend

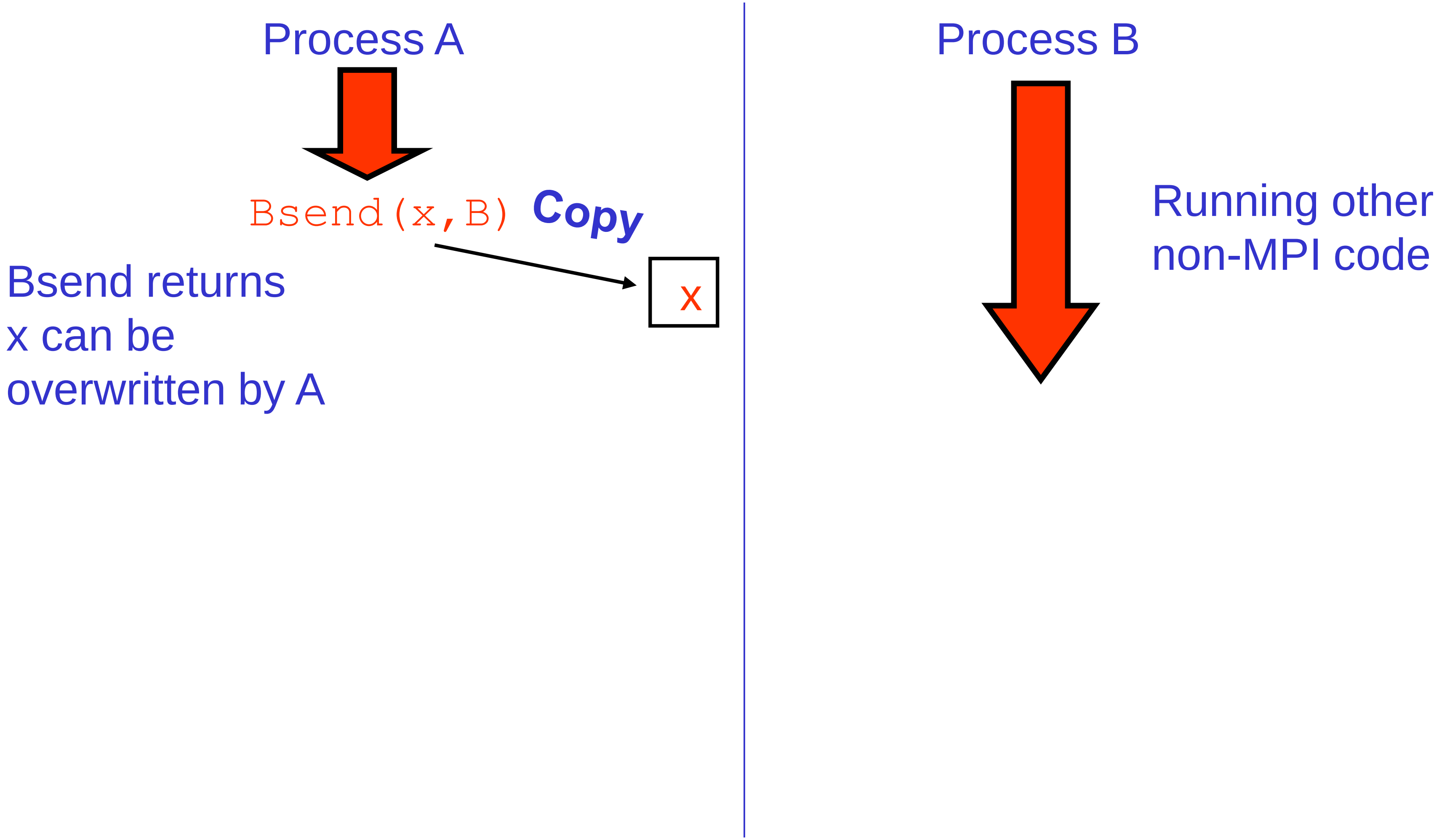


MPI_Bsend

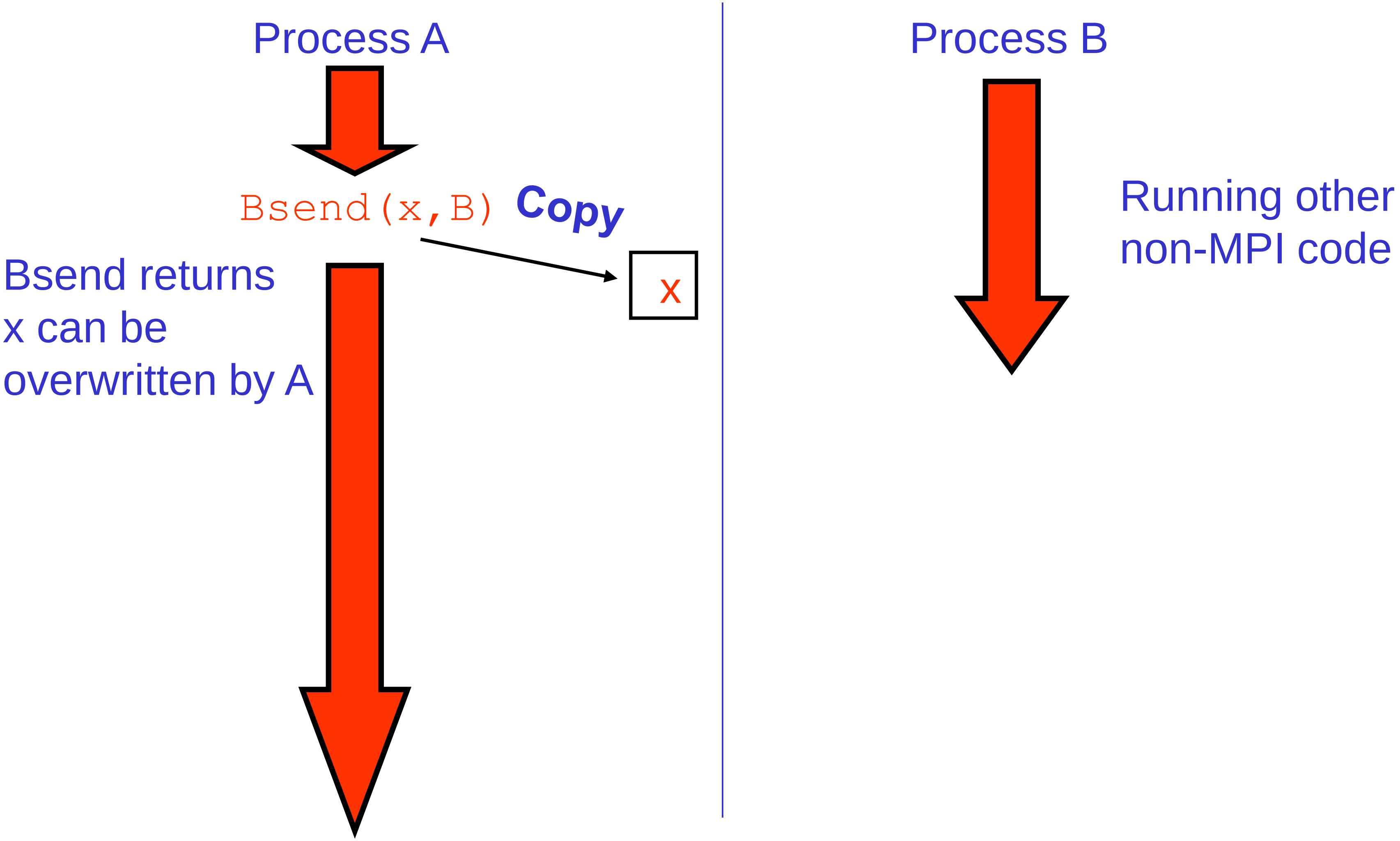


MPI_Bsend

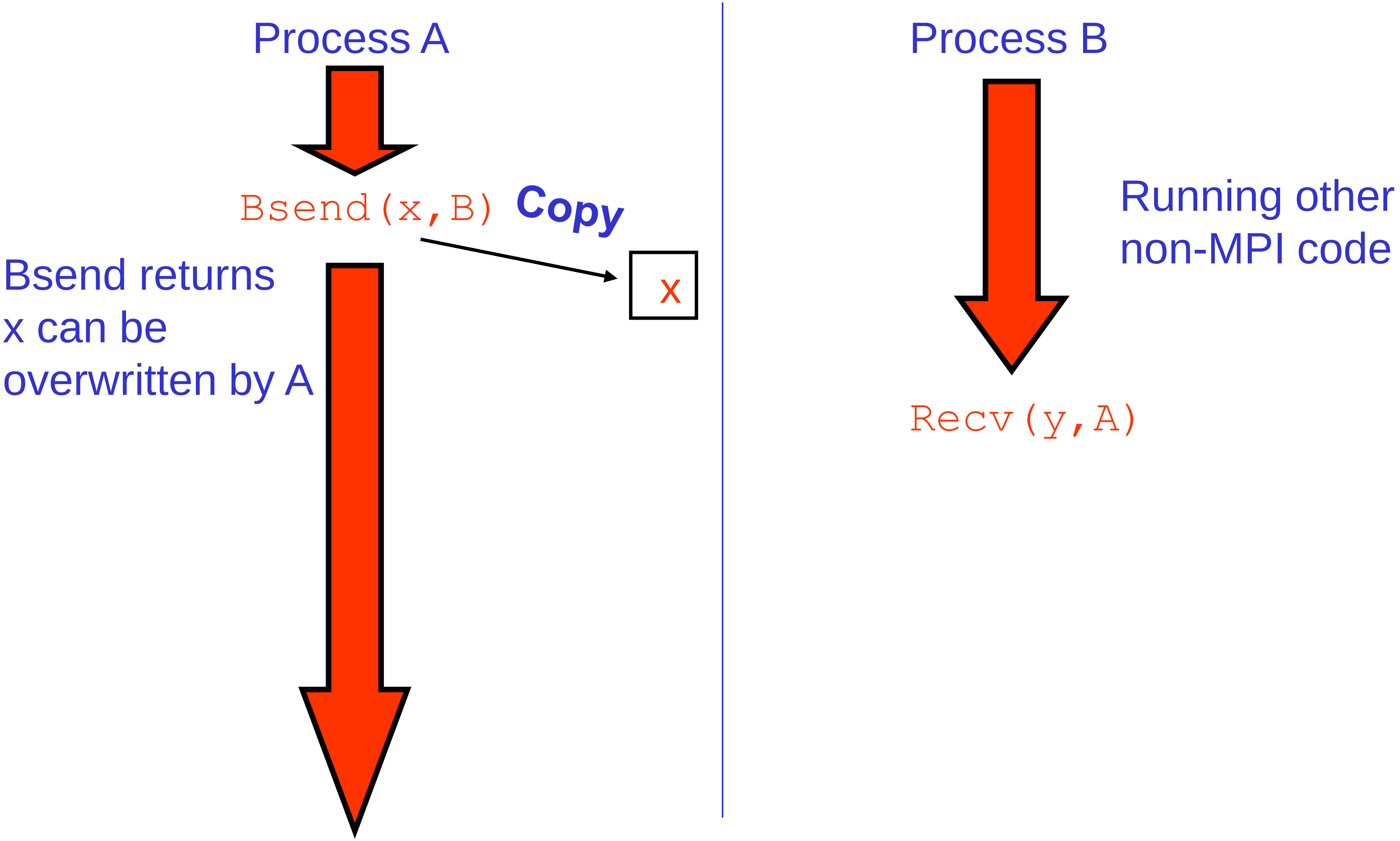

Process A
Bsend(x,B)
Copy
x
Process B
Running other non-MPI code


Process A
Bsend(x,B)
Copy
x
Bsend returns
x can be
overwritten by A
Process B
Running other
non-MPI code

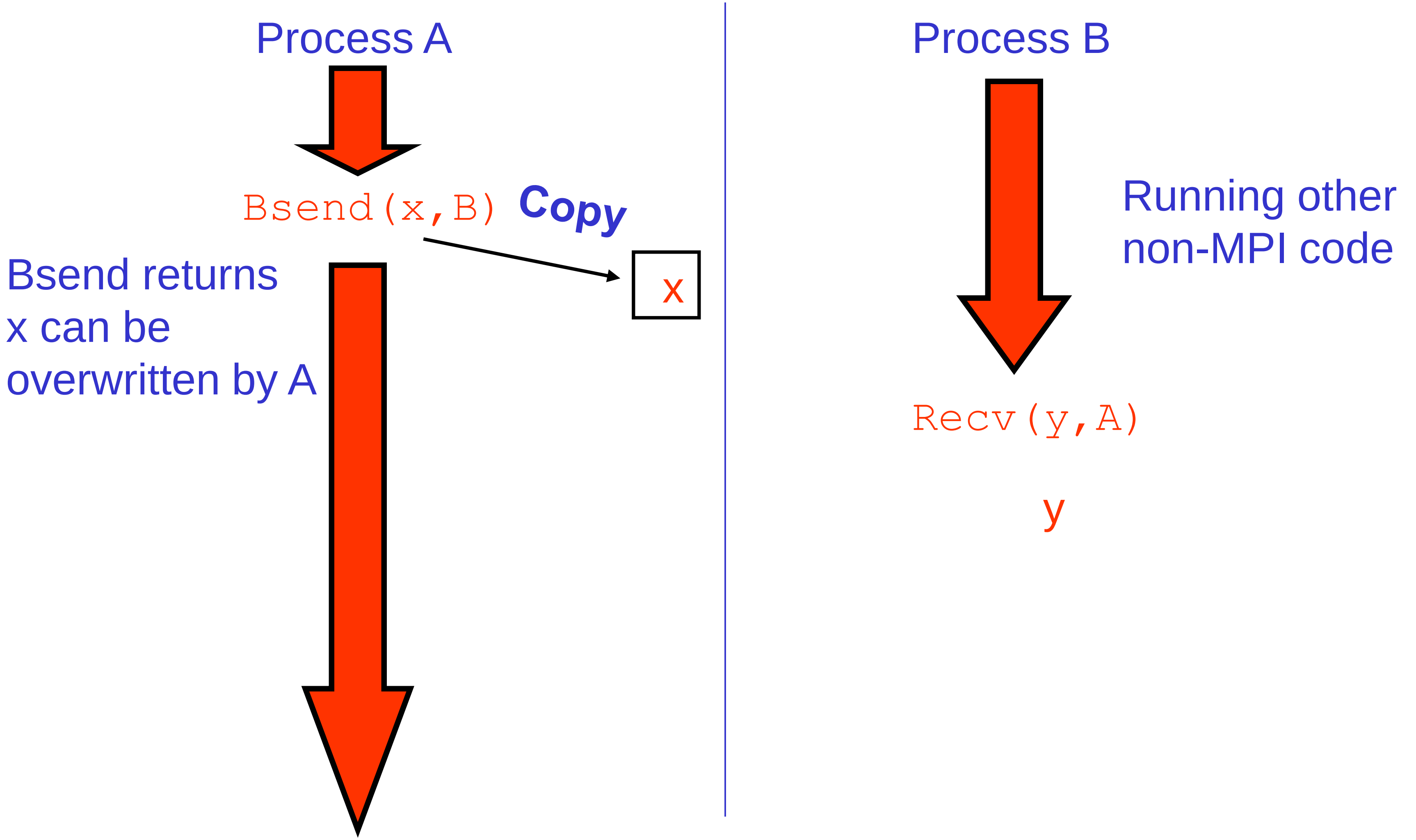
MPI_Bsend


Process A
Process B
Bsend(x,B)
Copy
x
Bsend returns
x can be
overwritten by A
Running other
non-MPI code

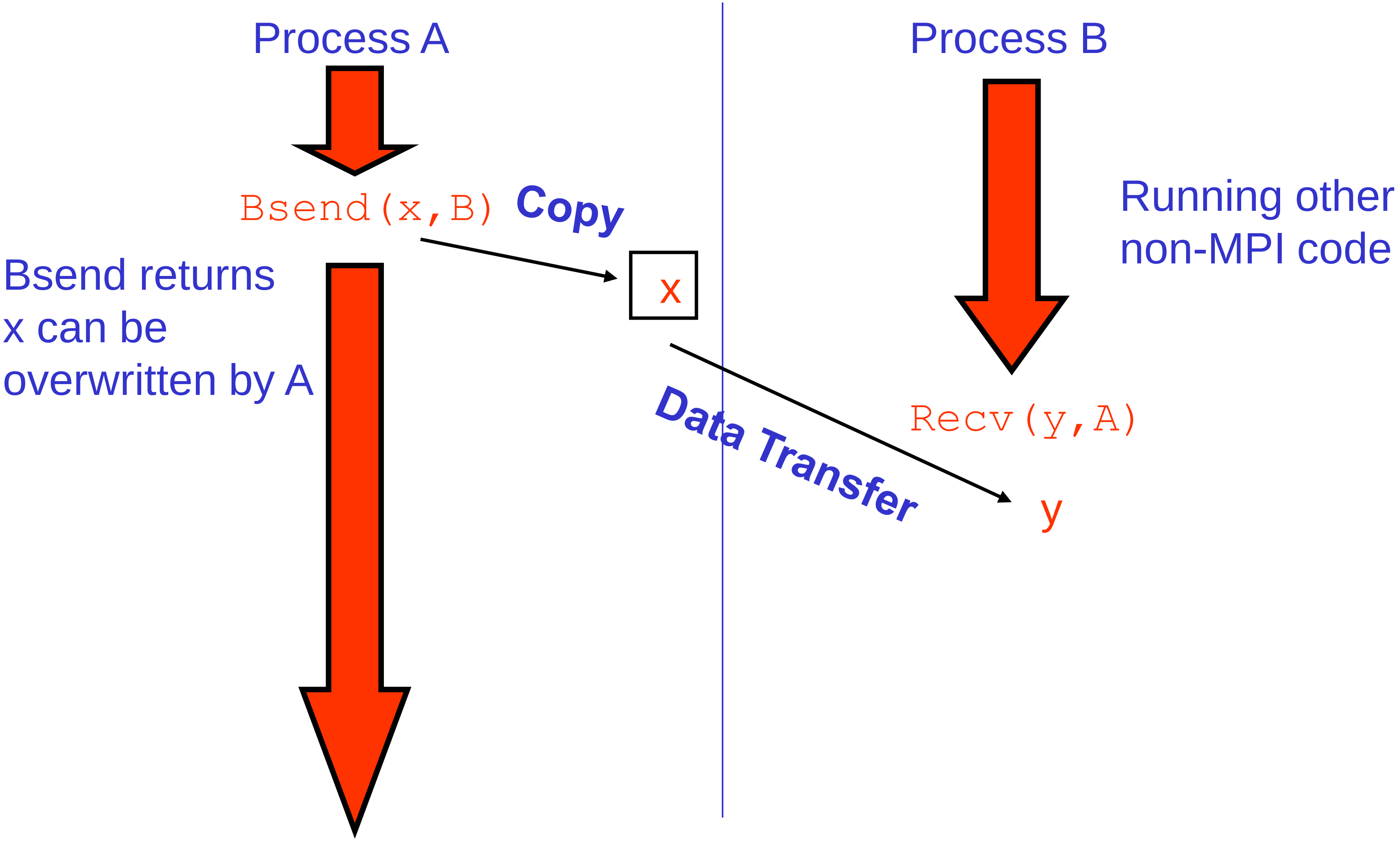
MPI_Bsend


Process A
Process B
Bsend(x,B)
Copy
x
Bsend returns
x can be
overwritten by A
Running other
non-MPI code
Recv(y,A)

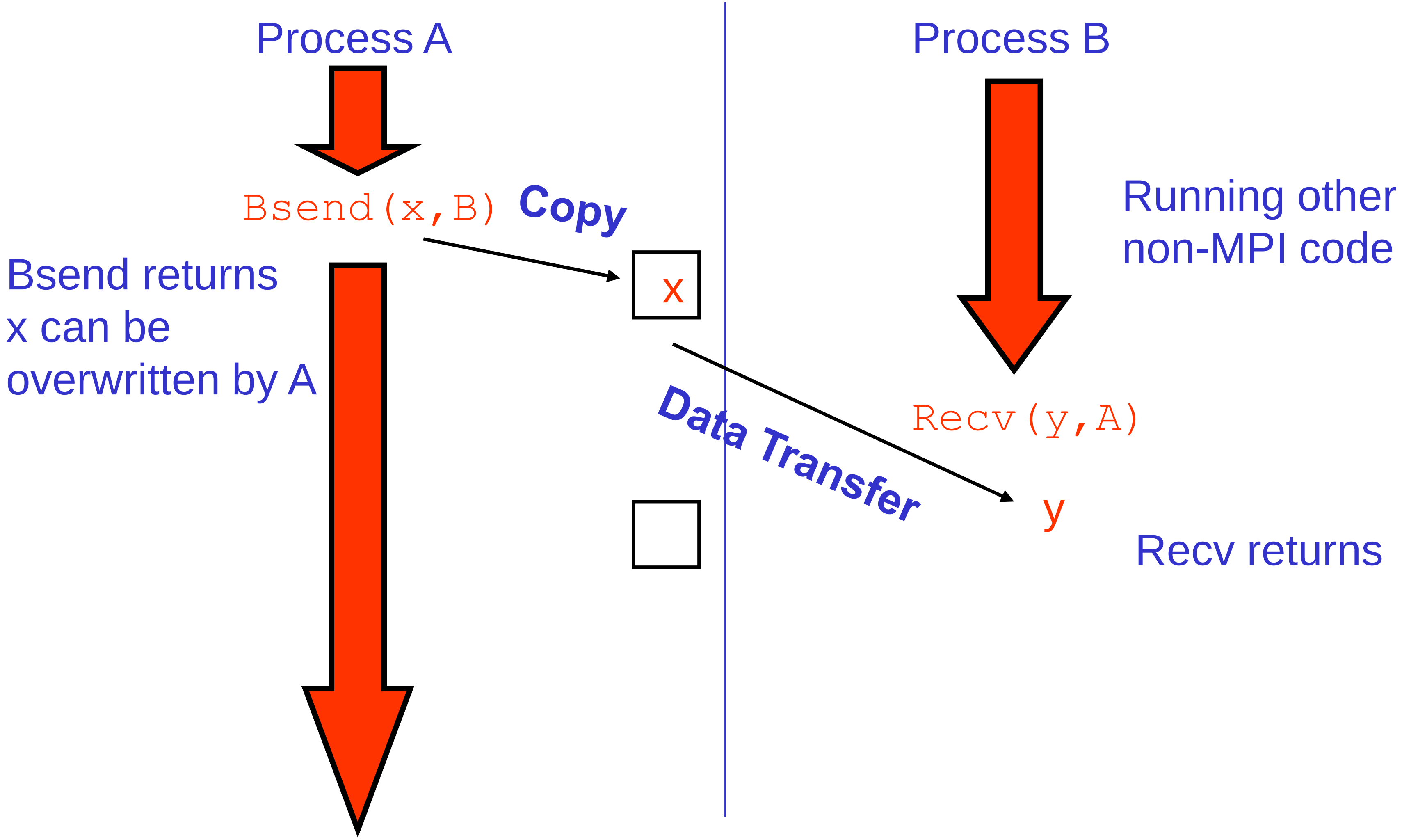
MPI_Bsend


Process A
Process B
Bsend(x,B)
Copy
x
Bsend returns
x can be
overwritten by A
Running other
non-MPI code
Recv(y,A)
y

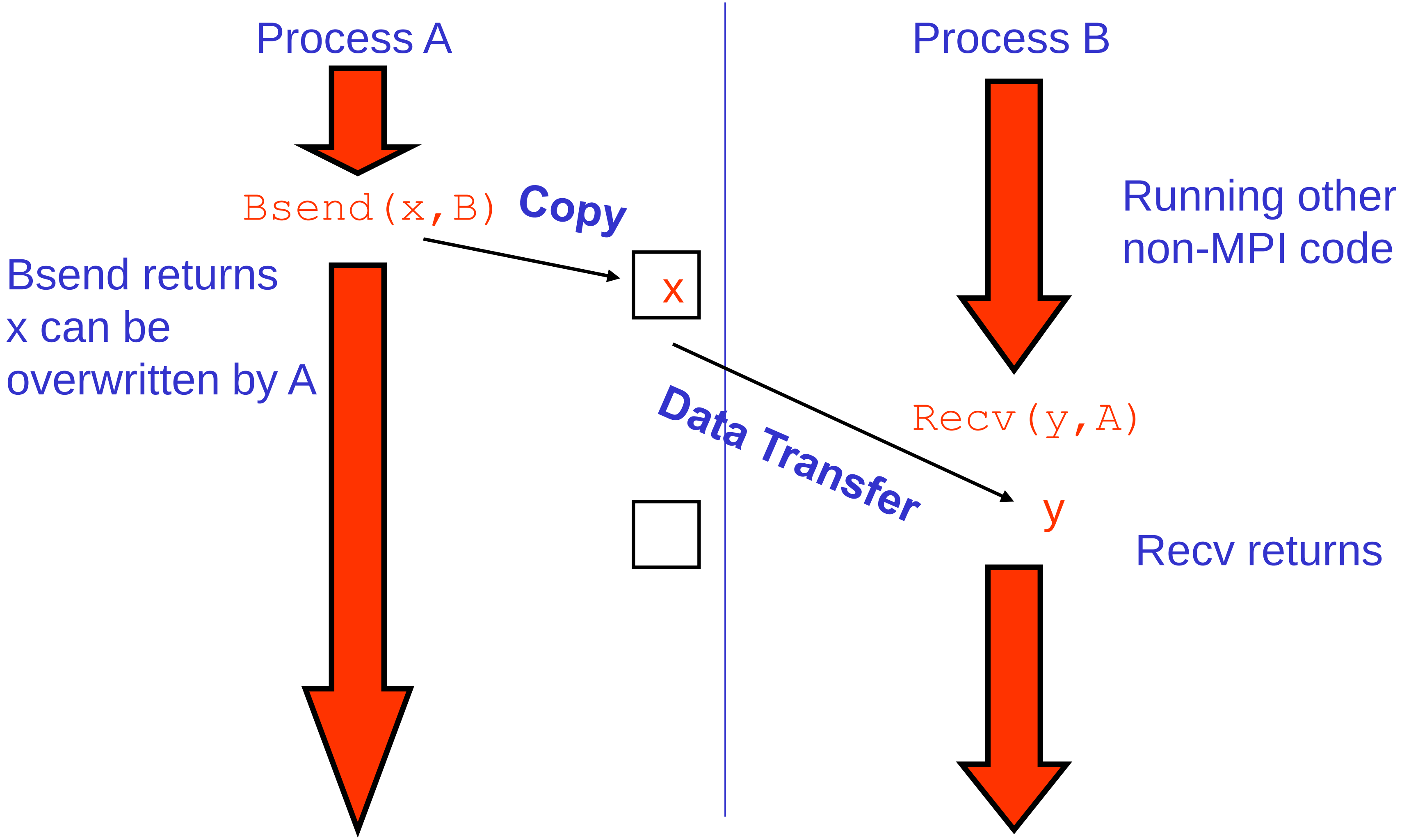
MPI_Bsend


Process A
Process B
Bsend(x,B)
Copy
x
Running other
non-MPI code
Bsend returns
x can be
overwritten by A
Data Transfer
Recv(y,A)
y

MPI_Bsend

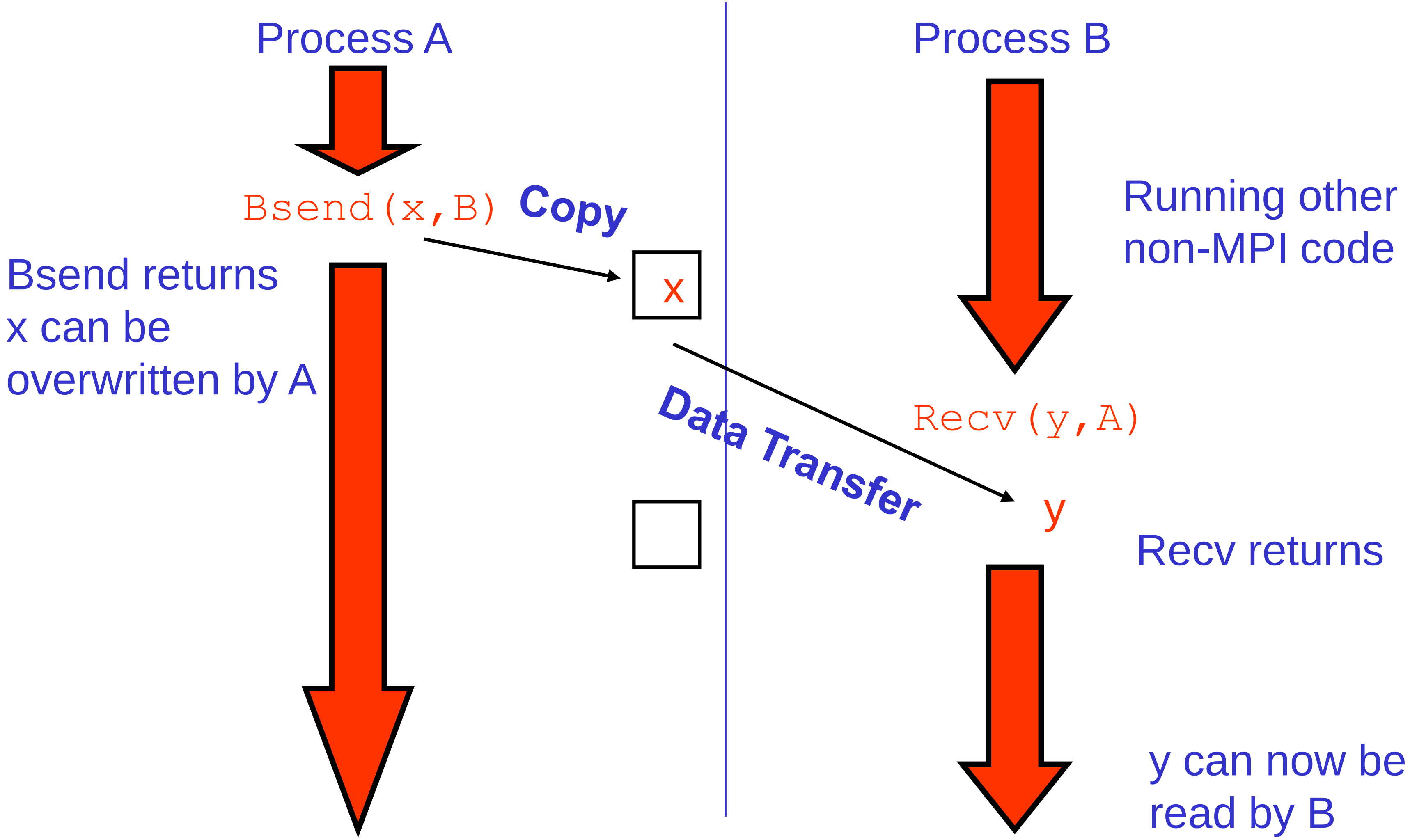

Process A
Process B
Bsend(x,B)
Copy
Running other
non-MPI code
Bsend returns
x can be
overwritten by A
x
Data Transfer
Recv(y,A)
y
Recv returns

MPI_Bsend



Process A
Process B
Bsend(x,B)
Copy
Bsend returns
x can be
overwritten by A
x
Running other
non-MPI code
Data Transfer
Recv(y,A)
y
Recv returns

MPI_Bsend



Process A
Process B
Bsend(x,B)
Copy
x
Bsend returns
x can be
overwritten by A
Running other
non-MPI code
Data Transfer
Recv(y,A)
y
Recv returns
y can now be
read by B

- **`Recv`** is always synchronous
  - if process B issued **`Recv`** before the **`Bsend`** from process A, then B would wait in the **`Recv`** until **`Bsend`** was issued

- Where does the buffer space come from?
  - for **`Bsend`**, the user provides a single large block of memory
  - make this available to MPI using **`MPI_Buffer_attach`**

- If A issues another **`Bsend`** before the **`Recv`**
  - system tries to store message in free space in the buffer
  - if there is not enough space then **`Bsend`** will FAIL!

- Problems
  - **`Ssend`** runs the risk of deadlock
  - **`Bsend`** less likely to deadlock, and your code may run faster, but
    - the user must supply the buffer space
    - the routine will FAIL if this buffering is exhausted
- **`MPI_Send`** tries to solve these problems
  - buffer space is provided by the system
  - **`Send`** will normally be asynchronous (like **`Bsend`**)
  - if buffer is full, **`Send`** becomes synchronous (like **`Ssend`**)
- **`MPI_Send`** routine is unlikely to fail
  - but could cause your program to deadlock if buffering runs out

MPI_Send

Process A	Process B
`Send(x,B)`	`Send(y,A)`
`Recv(x,B)`	`Recv(y,A)`

- This code is NOT guaranteed to work
  - will deadlock if **`Send`** is synchronous
  - is guaranteed to deadlock if you used **`Ssend`**!

Solutions

- To avoid deadlock
  - either match sends and receives explicitly
  - eg for ping-pong
    - process A sends then receives
    - process B receives then sends
- For a more general solution use non-blocking communications (see later)
- For this course you should program with **`Ssend`**
  - more likely to pick up bugs such as deadlock than **`Send`**

Checking for Messages

- MPI allows you to check if any messages have arrived
  - you can “probe” for matching messages
  - same syntax as receive except no receive buffer specified
- e.g. in C:

```
int MPI_Probe(int source, int tag,
              MPI_Comm comm, MPI_Status *status)
```

- Status is set as if the receive took place
  - e.g. you can find out the size of the message and allocate space prior to receive
- Be careful with wildcards
  - you can use, e.g., MPI_ANY_SOURCE in call to probe
  - but must use **specific** source in receive to guarantee matching same message
  - e.g. `MPI_Recv(buff, count, datatype, status.MPI_SOURCE, ...)`

Tags

- Every message can have a tag
  - this is a non-negative integer value
  - maximum value can be queried using MPI_TAG_UB attribute
  - MPI guarantees to support tags of at least 32767
  - not everyone uses them; many MPI programs set all tags to zero
- Tags can be useful in some situations
  - can choose to receive messages only of a given tag
- Most commonly used with **`MPI_ANY_TAG`**
  - receives the most recent message regardless of the tag
  - user then finds out the actual value by looking at the **`status`**

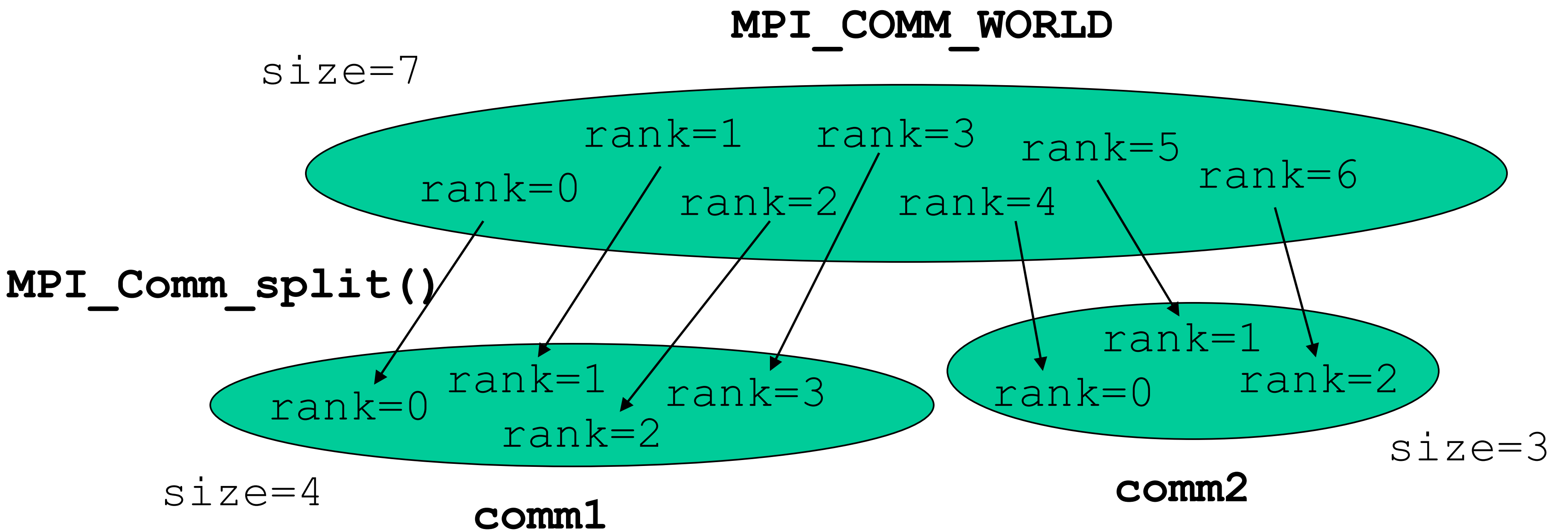
Communicators

- All MPI communications take place within a communicator
  - a communicator is fundamentally a group of processes
  - there is a pre-defined communicator: **`MPI_COMM_WORLD`** which contains ALL the processes
    - also **`MPI_COMM_SELF`** which contains only one process

- A message can ONLY be received within the same communicator from which it was sent
  - unlike tags, it is not possible to wildcard on **`comm`**

Uses of Communicators (i)

- Can split **`MPI_COMM_WORLD`** into pieces
  - each process has a new rank within each sub-communicator
  - guarantees messages from the different pieces do not interact
    - can attempt to do this using tags but there are no guarantees



Uses of Communicators (ii)

- Can make a copy of **`MPI_COMM_WORLD`**
  - e.g. call the **`MPI_Comm_dup`** routine
  - containing all the same processes but in a new communicator

- Enables processes to communicate with each other safely within a piece of code
  - guaranteed that messages cannot be received by other code
  - this is **essential** for people writing parallel libraries (eg a Fast Fourier Transform) to stop library messages becoming mixed up with user messages
    - user cannot intercept the the library messages if the library keeps the identity of the new communicator a secret
    - not safe to simply try and reserve tag values due to wildcarding

- Question: Why bother with all these send modes?
- Answer
  - it is a little complicated, but you should make sure you understand
  - **`Ssend`** and **`Bsend`** are clear
    - map directly onto synchronous and asynchronous sends
  - **`Send`** can be either synchronous or asynchronous
    - MPI is trying to be helpful here, giving you the benefits of **`Bsend`** if there is sufficient system memory available, but not failing completely if buffer space runs out
    - in practice this leads to endless confusion!
- The amount of system buffer space is variable
  - programs that run on one machine may deadlock on another
  - you should **NEVER** assume that **`Send`** is asynchronous!

Summary (ii)

- Question: What are the tags for?
- Answer
  - if you don’t need them don’t use them!
    - perfectly acceptable to set all tags to zero
  - can be useful for debugging
    - eg always tag messages with the rank of the sender

Summary (iii)

- Question: Can I just use **`MPI_COMM_WORLD`**?
- Answer
  - yes: many people never need to create new communicators in their MPI programs
  - however, it is probably bad practice to specify **`MPI_COMM_WORLD`** explicitly in your routines
    - using a variable will allow for greater flexibility later on, eg:

```
MPI_Comm comm;          /* or INTEGER for Fortran */
comm = MPI_COMM_WORLD;
...
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);
....
```

Parallel Programming

Thought exercise: traffic modelling

Traffic Flow

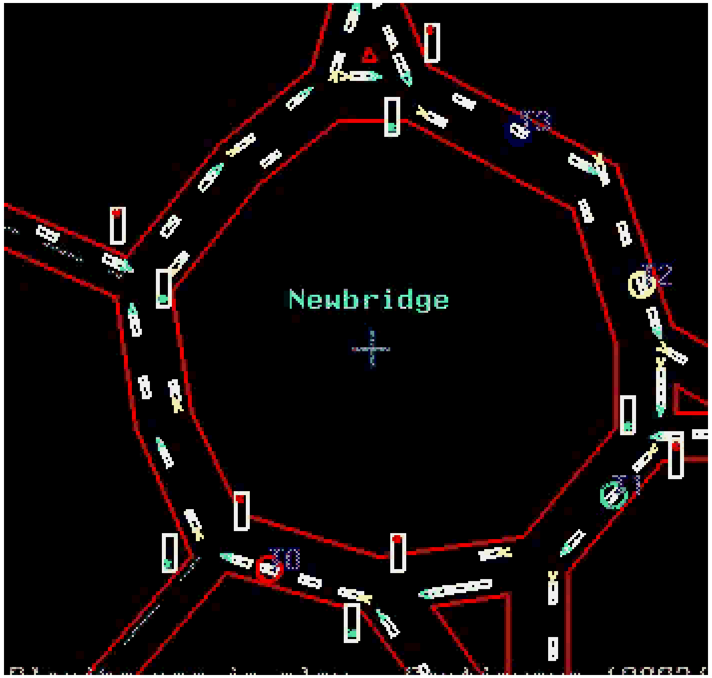
- we want to predict traffic flow

Traffic Flow

- we want to predict traffic flow
  - to look for effects such as congestion

Traffic Flow

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

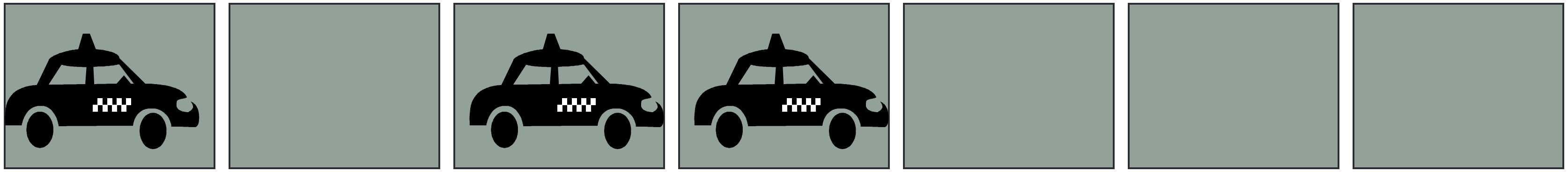


Simple Traffic Model

- divide road into a series of cells

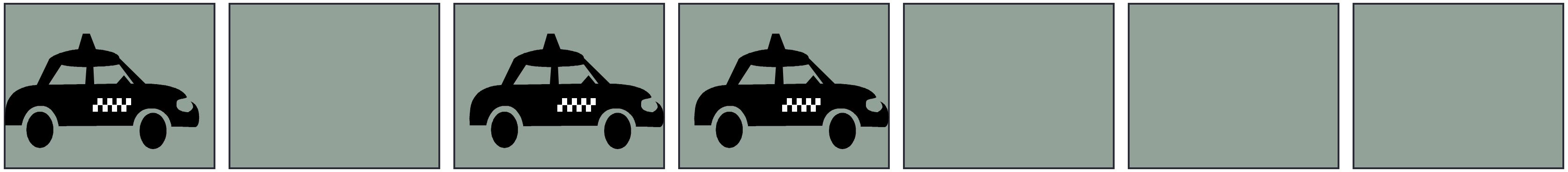
Simple Traffic Model

- divide road into a series of cells
  - either occupied or unoccupied



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

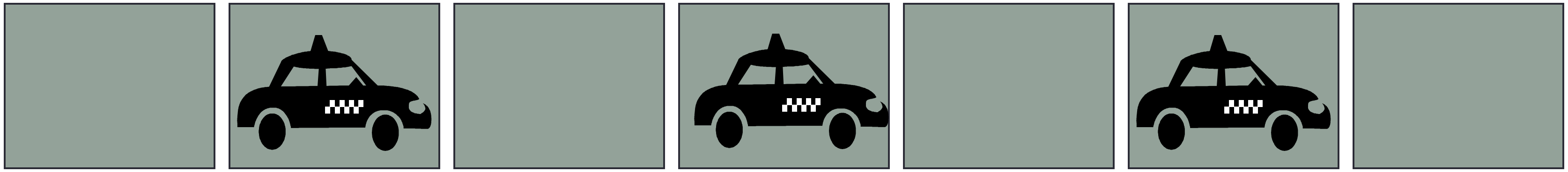


Simple Traffic Model

- 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

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



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

epcc

THE UNIVERSITY OF EDINBURGH

traffic behaviour

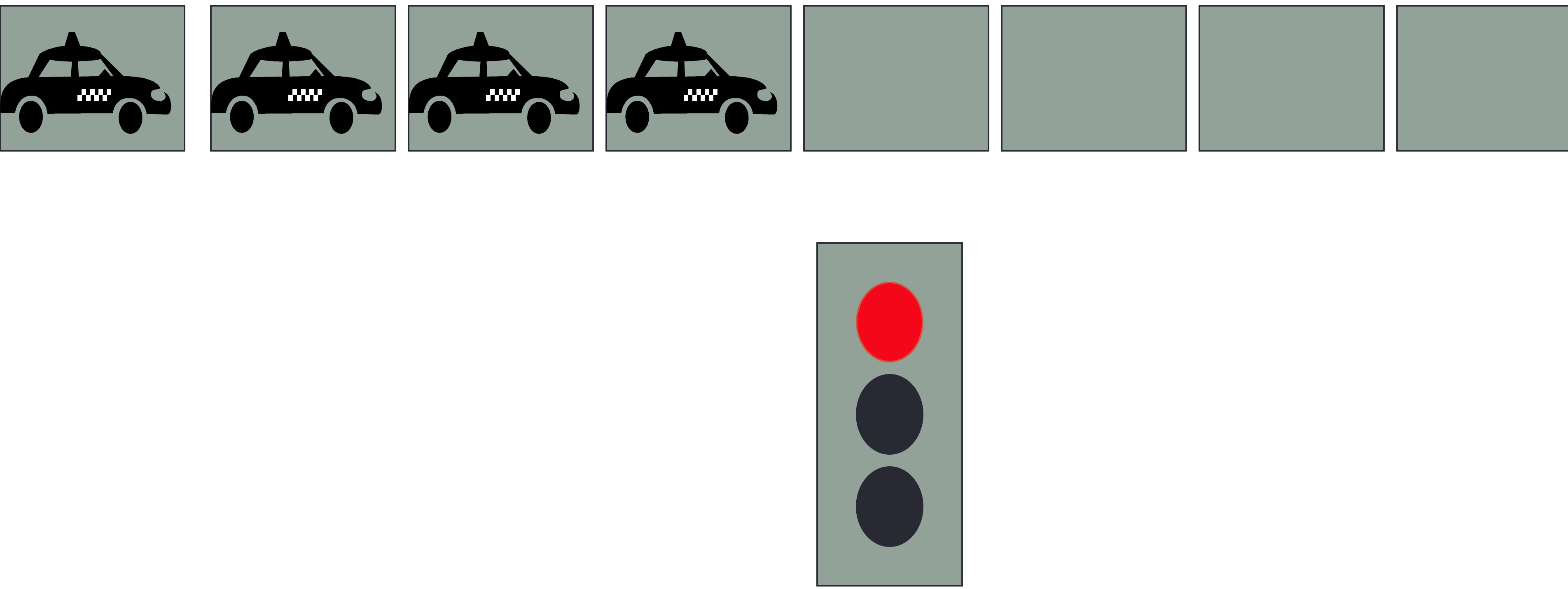
- model predicts a number of interesting features

traffic behaviour

- model predicts a number of interesting features
- traffic lights

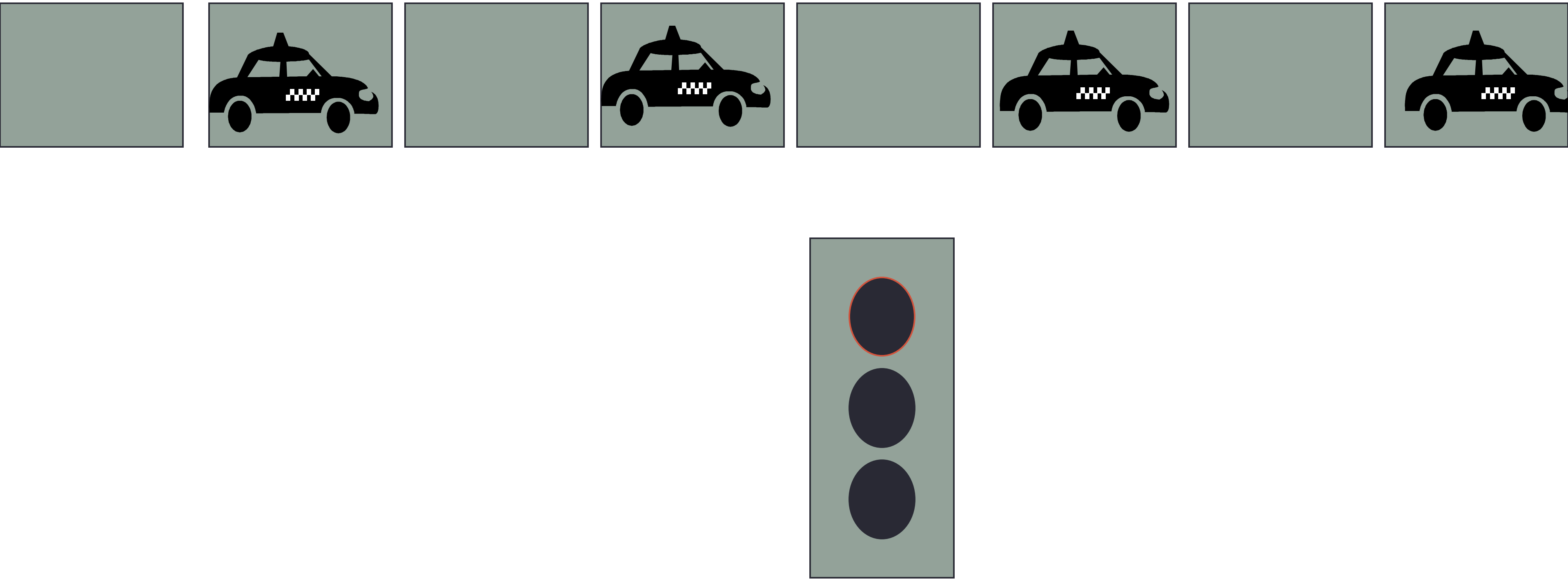
traffic behaviour

- model predicts a number of interesting features
- traffic lights



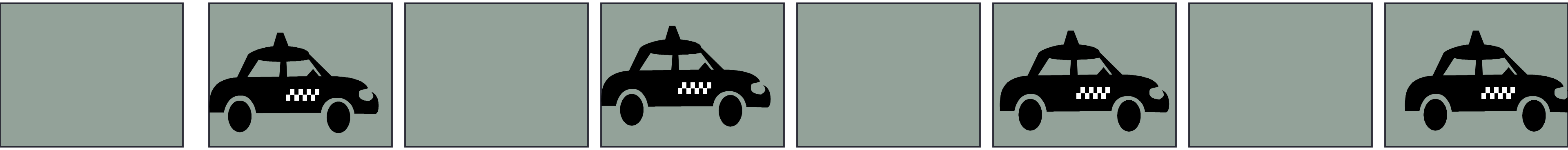
traffic behaviour

- model predicts a number of interesting features
  - traffic lights

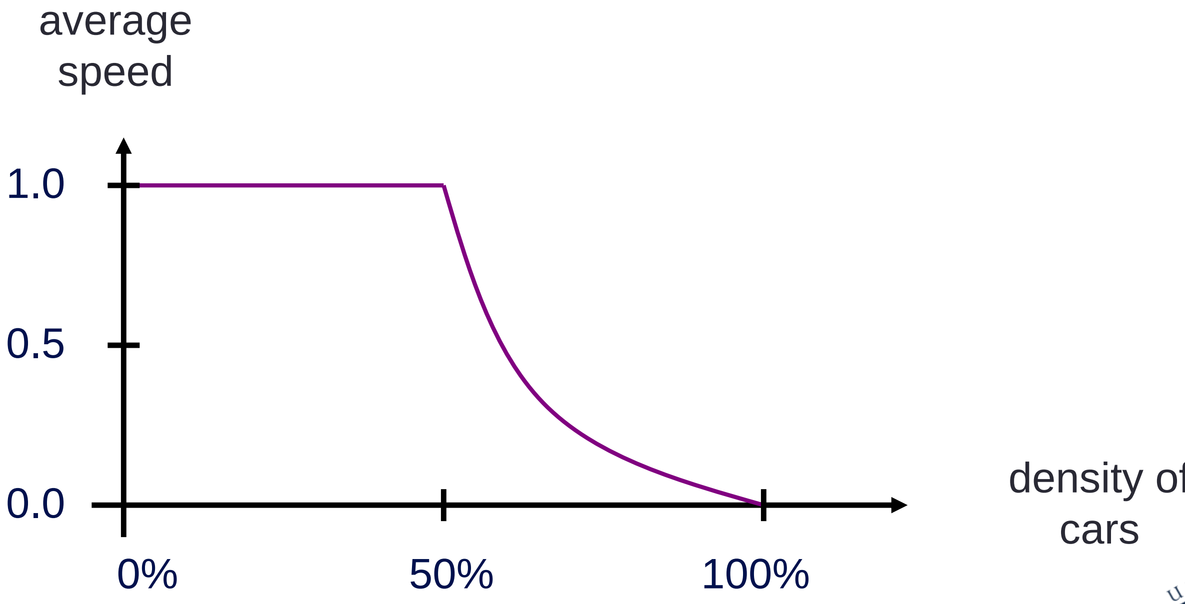


traffic behaviour

- model predicts a number of interesting features
- traffic lights

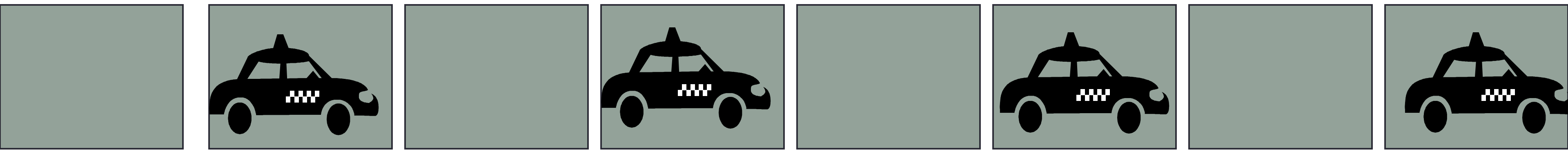


- congestion

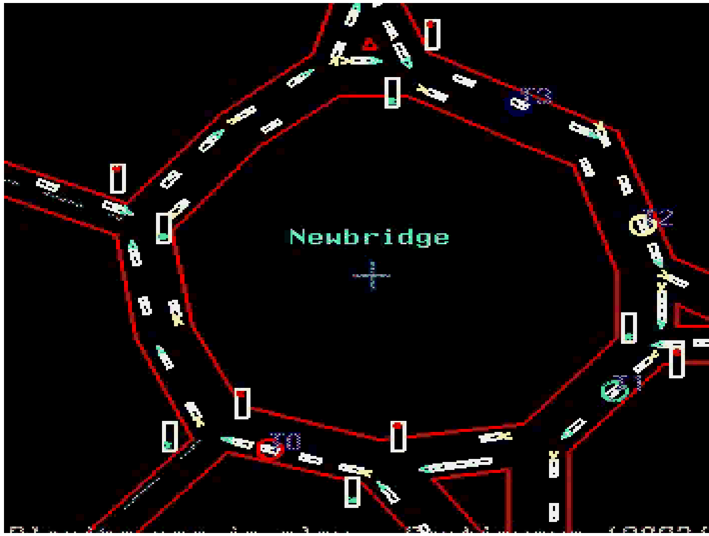


traffic behaviour

- model predicts a number of interesting features
- traffic lights

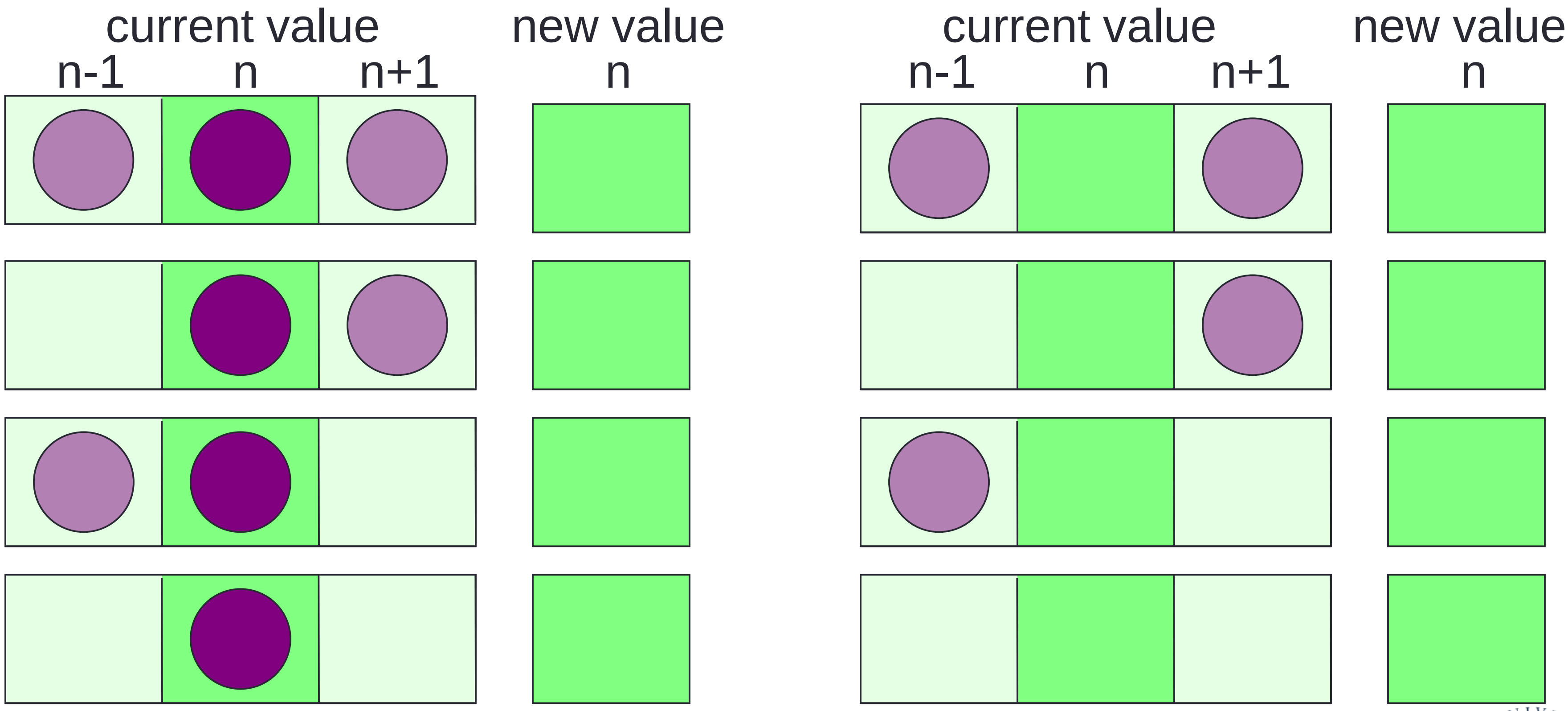


- more complicated models are used in practice



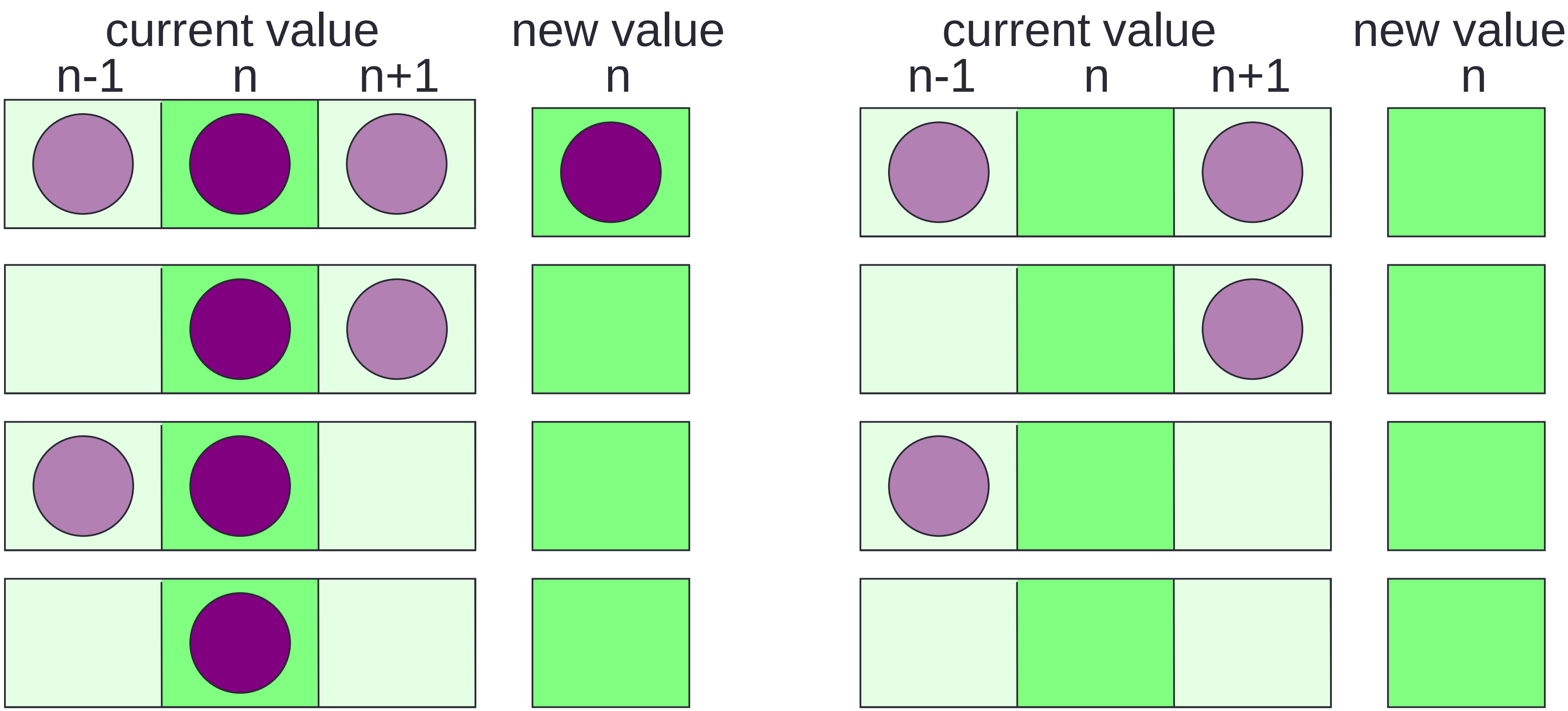
Traffic simulation

- Update rules depend on:
  - state of cell
  - state of nearest neighbours in both directions



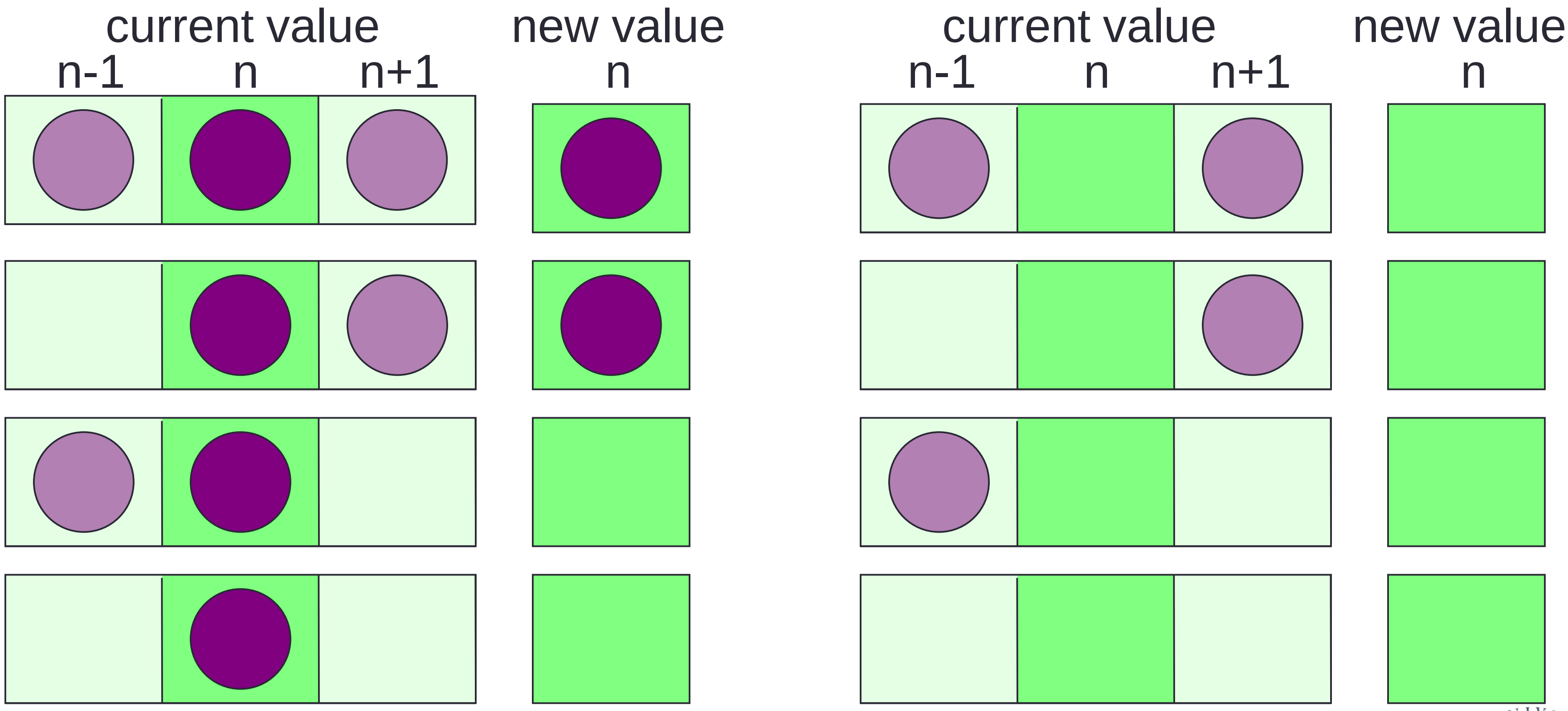
Traffic simulation

- Update rules depend on:
  - state of cell
  - state of nearest neighbours in both directions



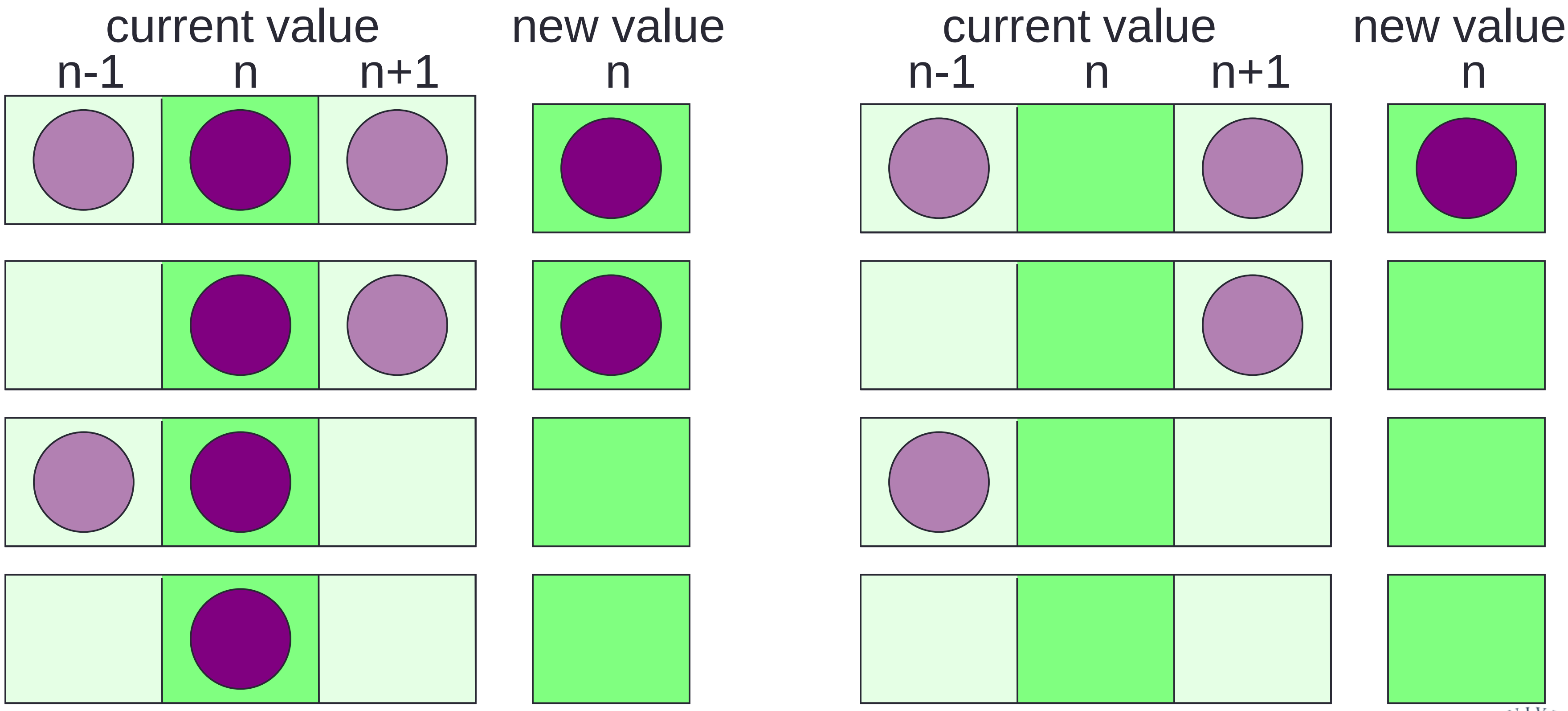
Traffic simulation

- Update rules depend on:
  - state of cell
  - state of nearest neighbours in both directions



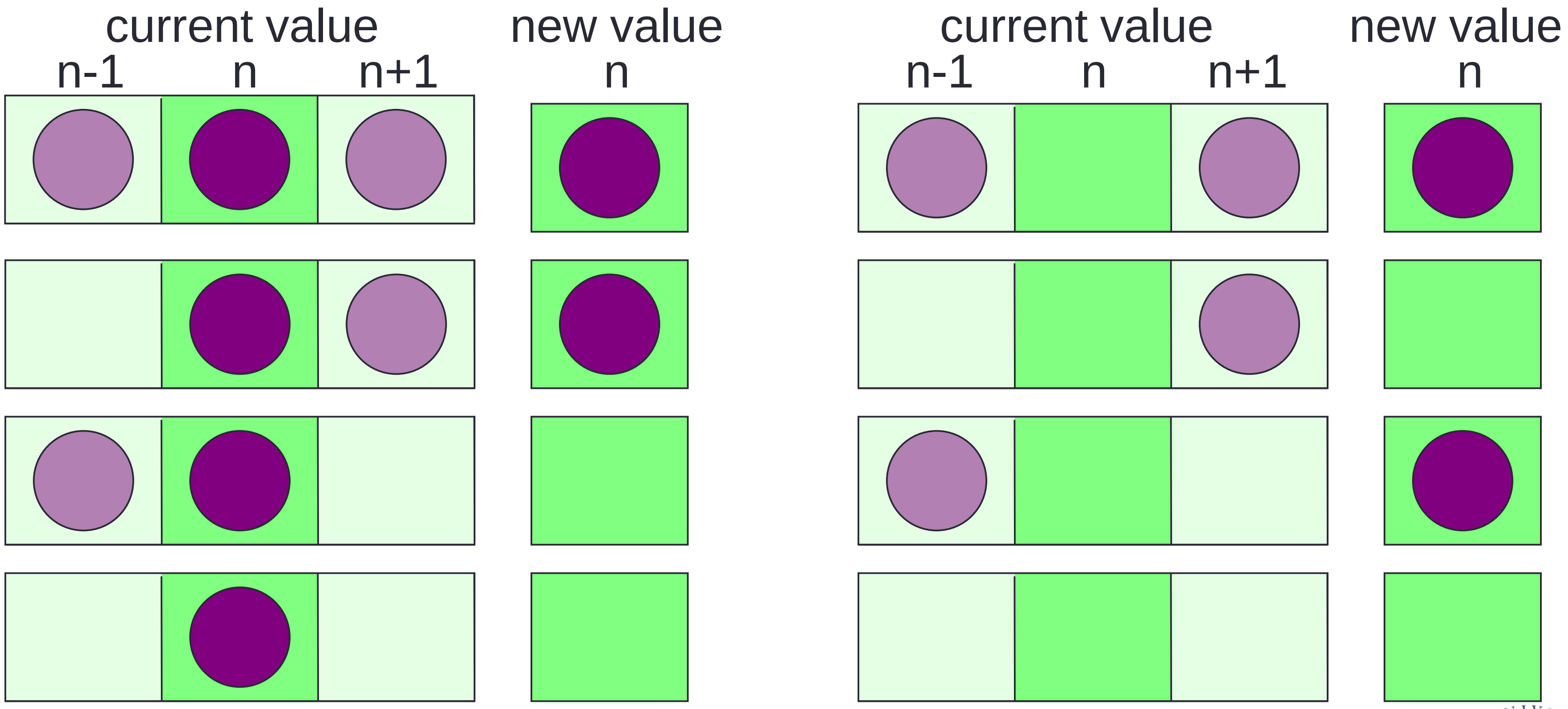
Traffic simulation

- Update rules depend on:
  - state of cell
  - state of nearest neighbours in both directions



Traffic simulation

- Update rules depend on:
  - state of cell
  - state of nearest neighbours in both directions



State Table

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

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

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

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

Pseudo Code

```
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
```

how fast can we run the model?

- measure speed in Car Operations Per second

how fast can we run the model?

- measure speed in Car Operations Per second
  - how many COPs?

how fast can we run the model?

- measure speed in Car Operations Per second
  - how many COPs?

how fast can we run the model?

- measure speed in Car Operations Per second
  - how many COPs?

how fast can we run the model?

- measure speed in Car Operations Per second
  - how many COPs?

how fast can we run the model?

- measure speed in Car Operations Per second
  - how many COPs?

how fast can we run the model?

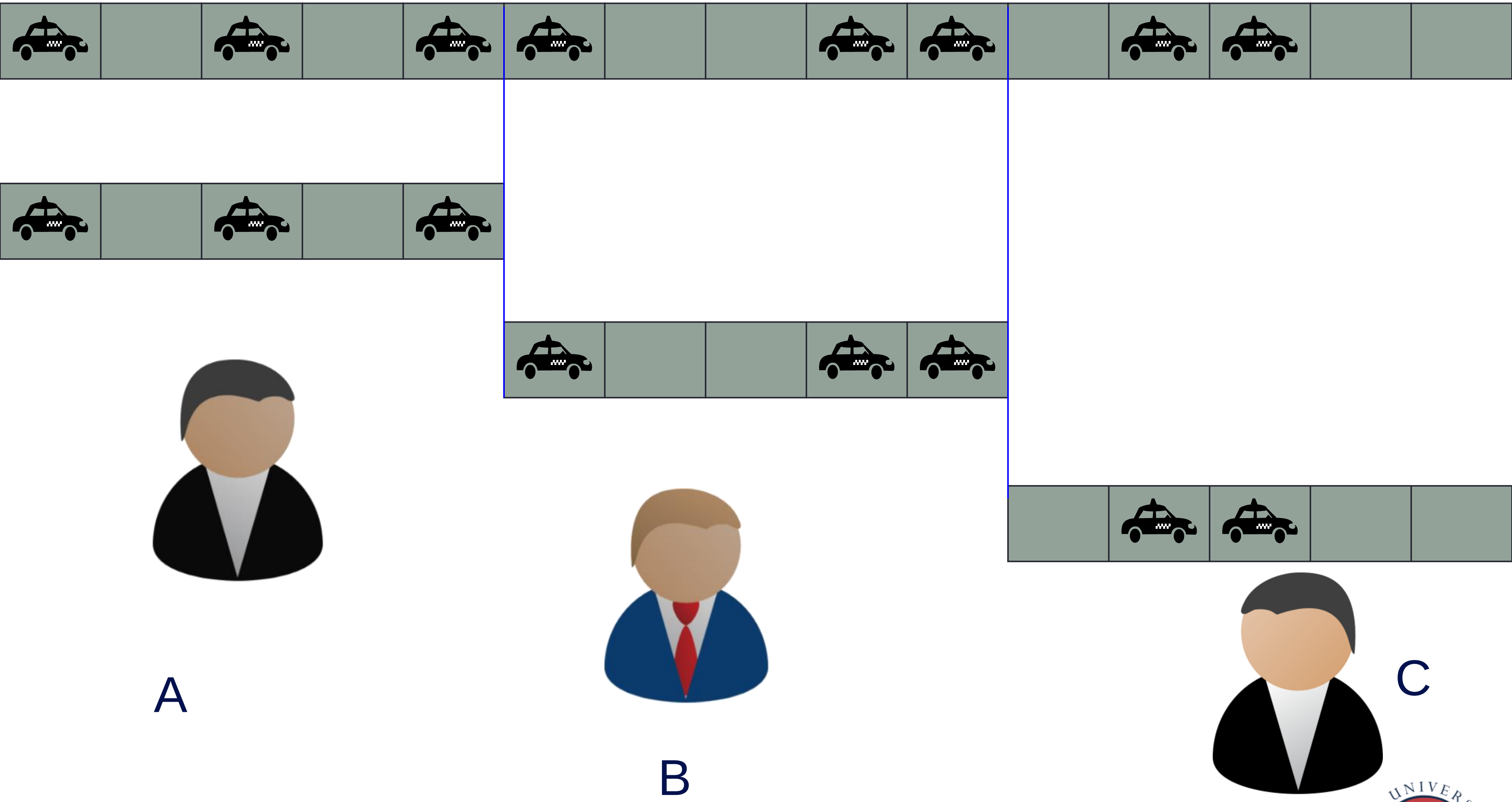
- measure speed in Car Operations Per second
  - how many COPs?
- around 2 COPs

how fast can we run the model?

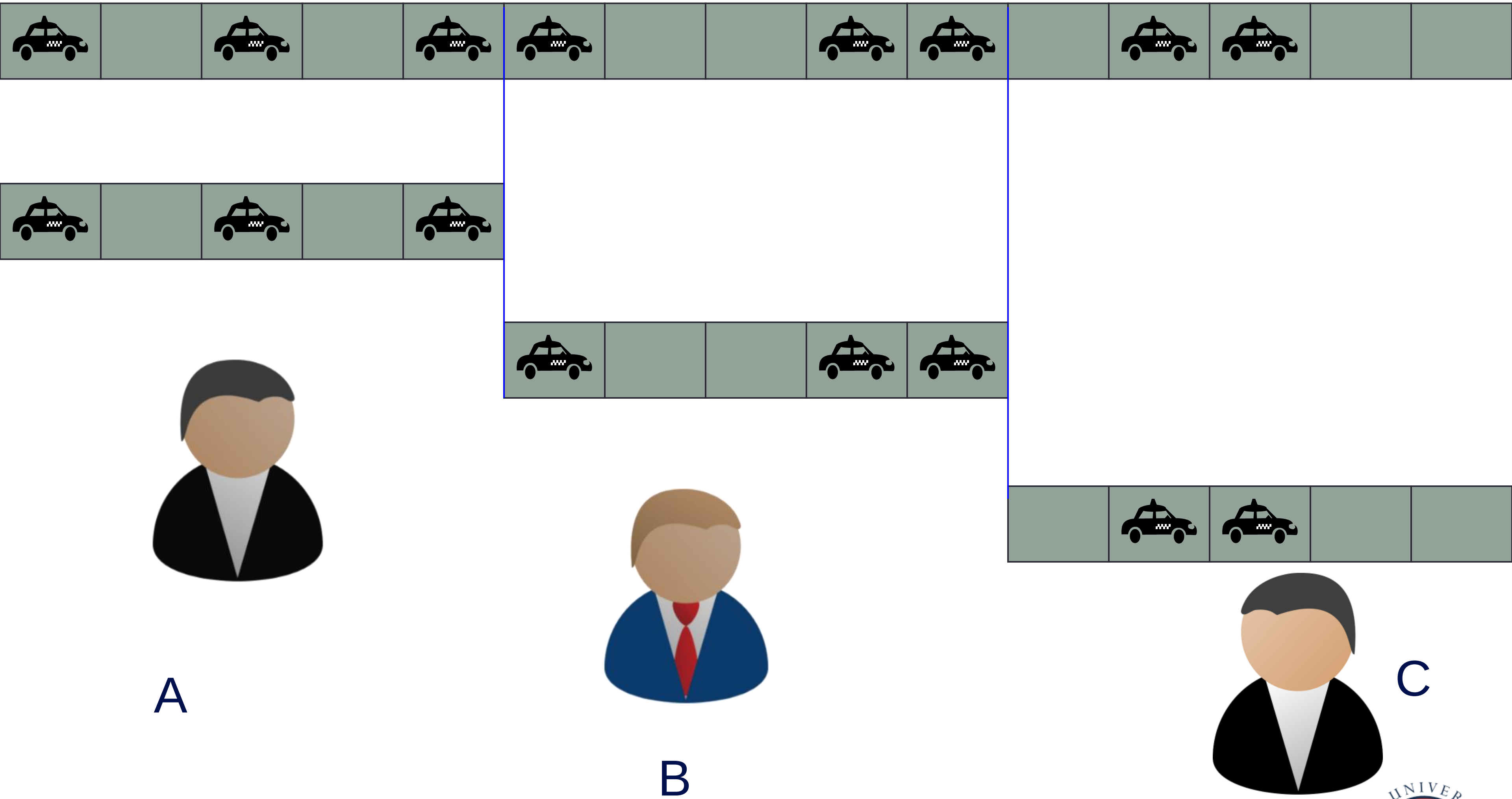
- measure speed in Car Operations Per second
  - how many COPs?
- around 2 COPs
- but what about three people?
  - can they do six COPs?



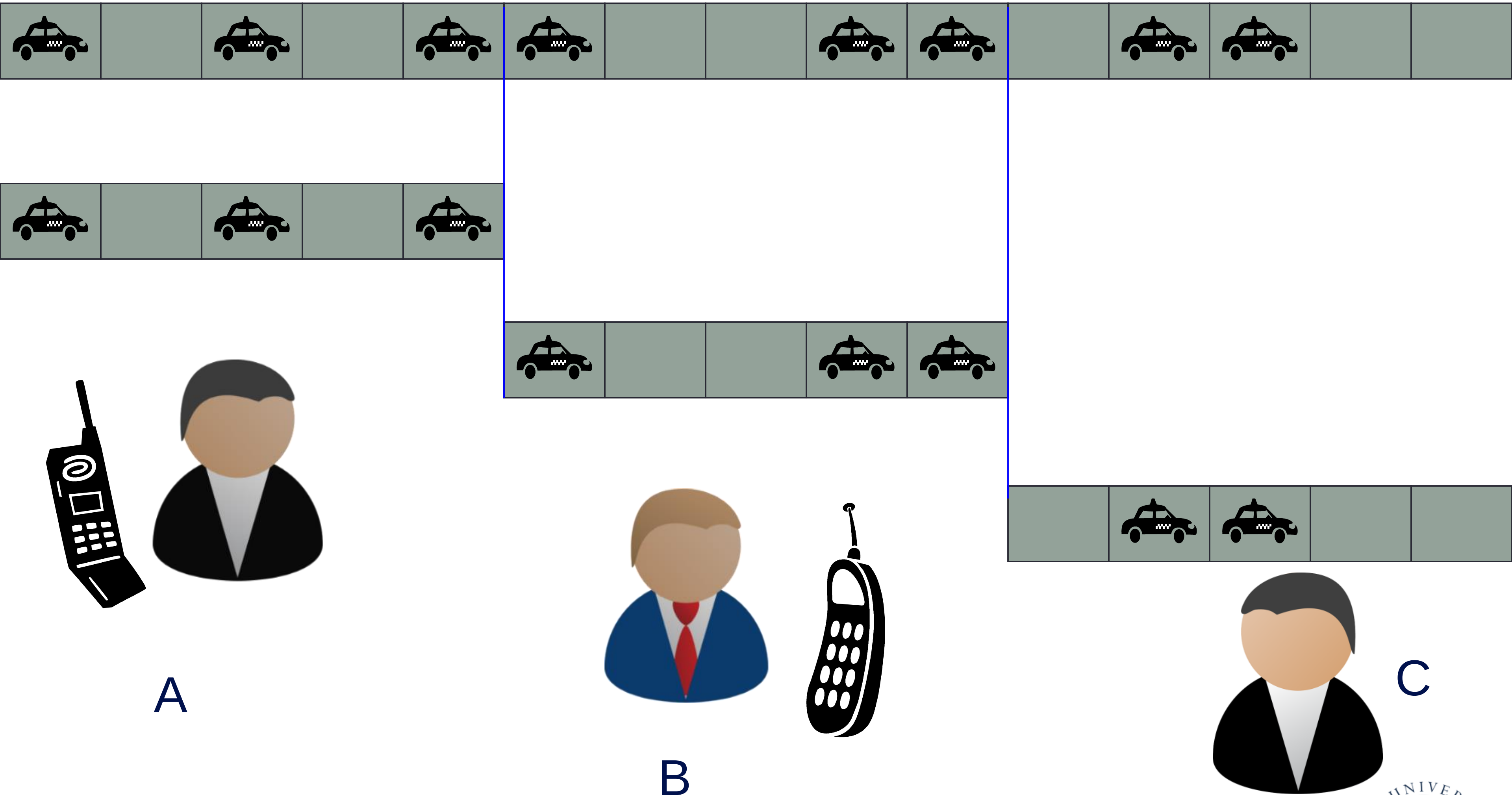

a parallel traffic model



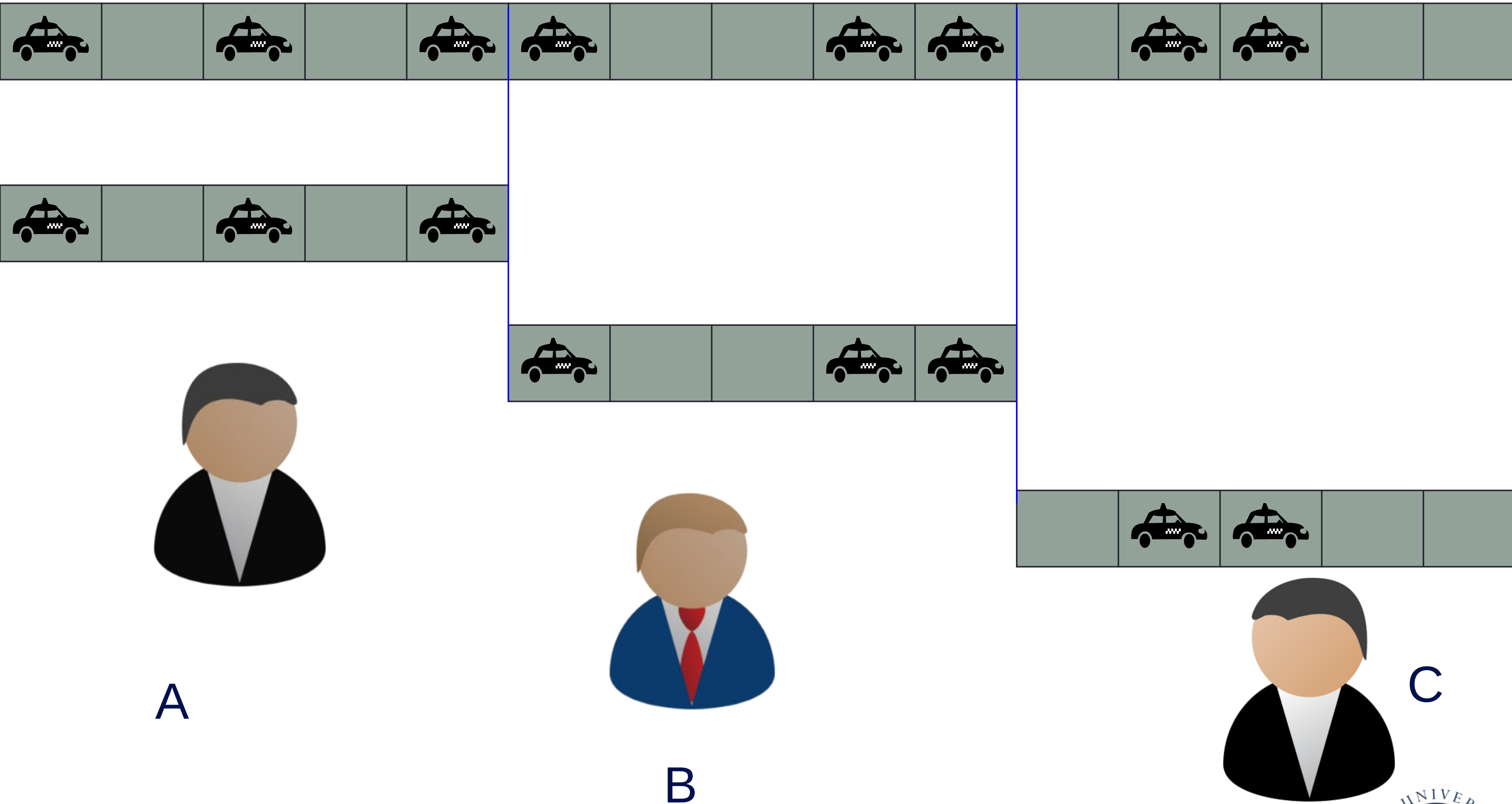
a parallel traffic model



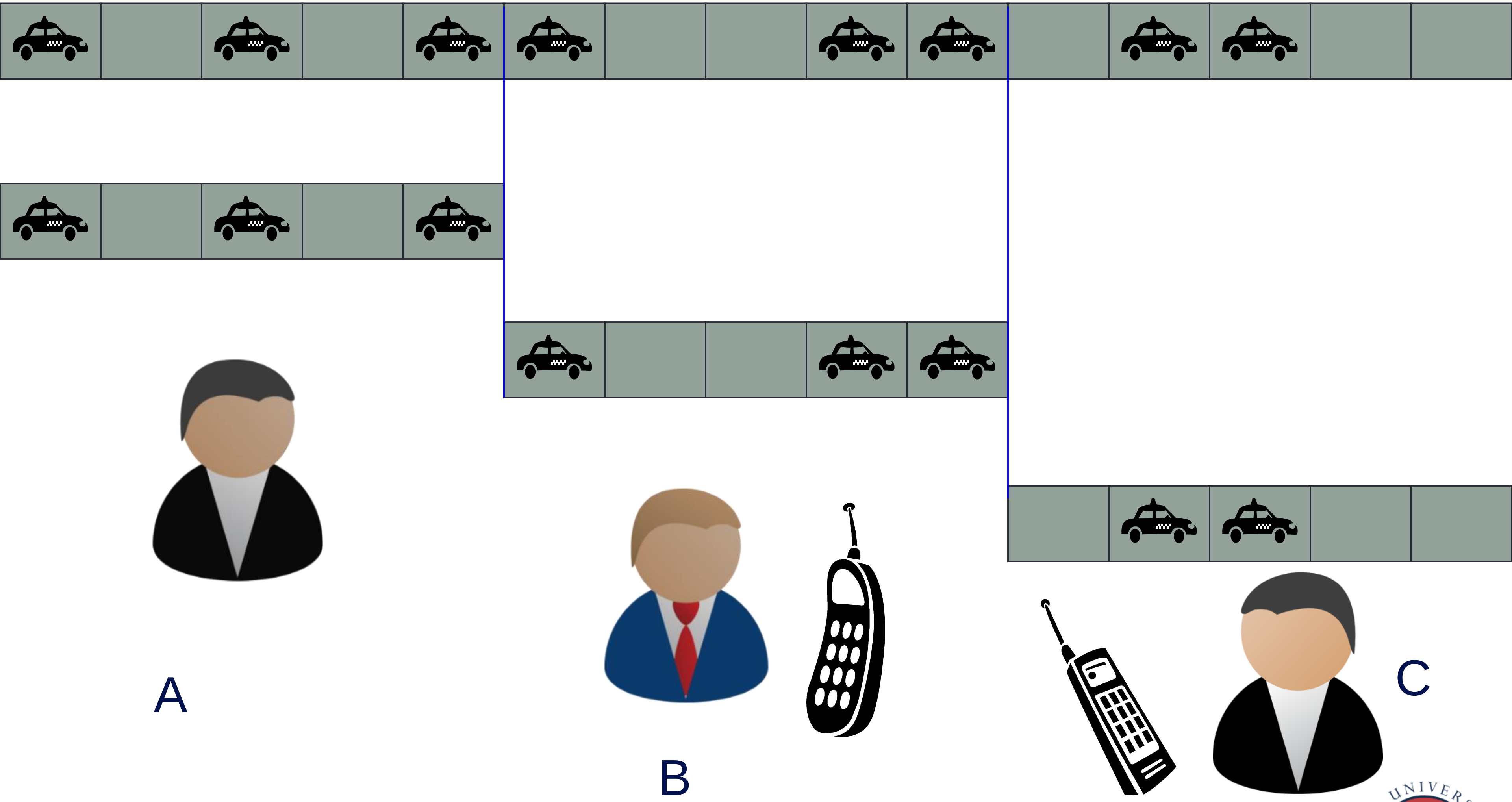
a parallel traffic model



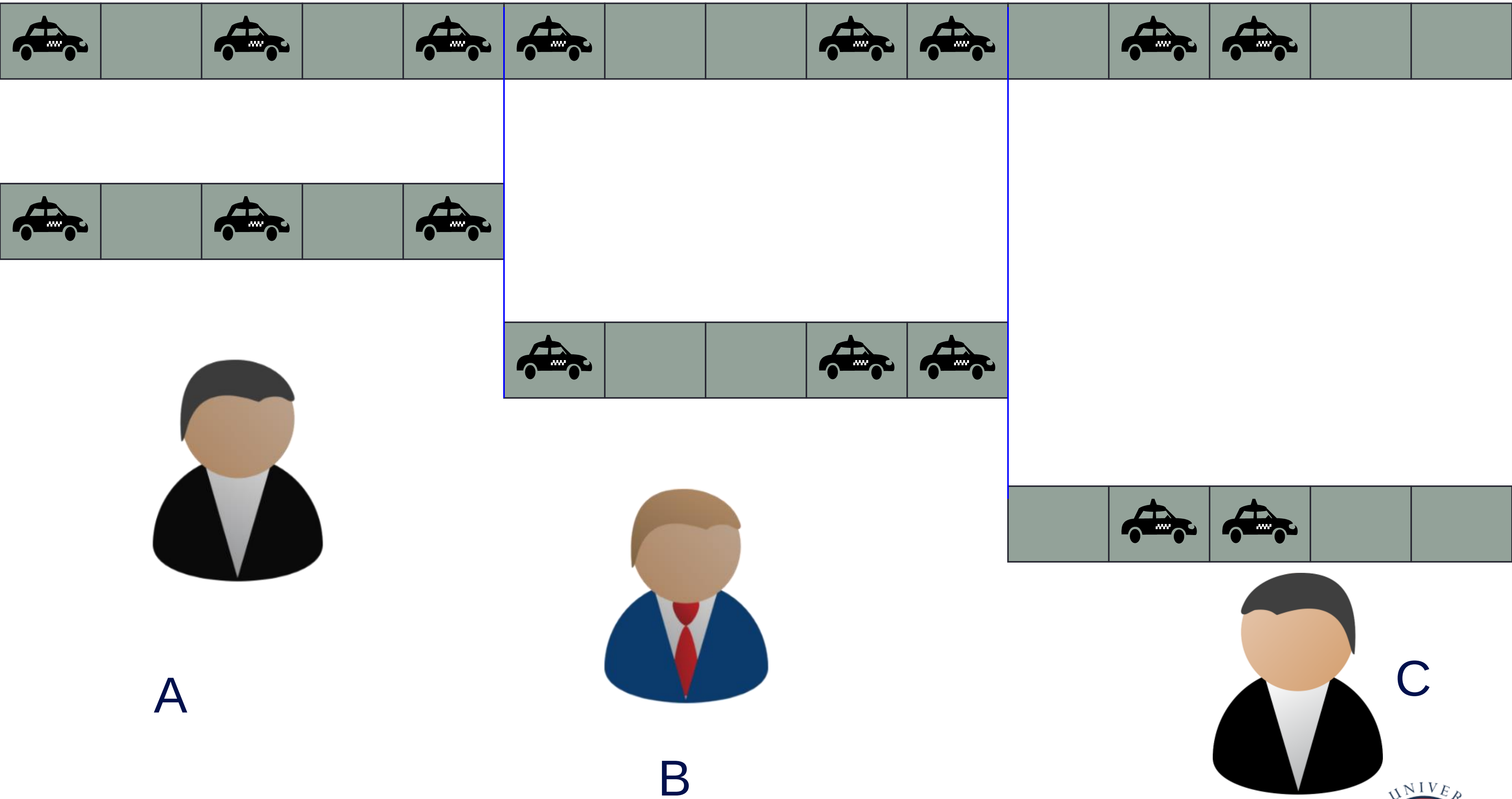
a parallel traffic model



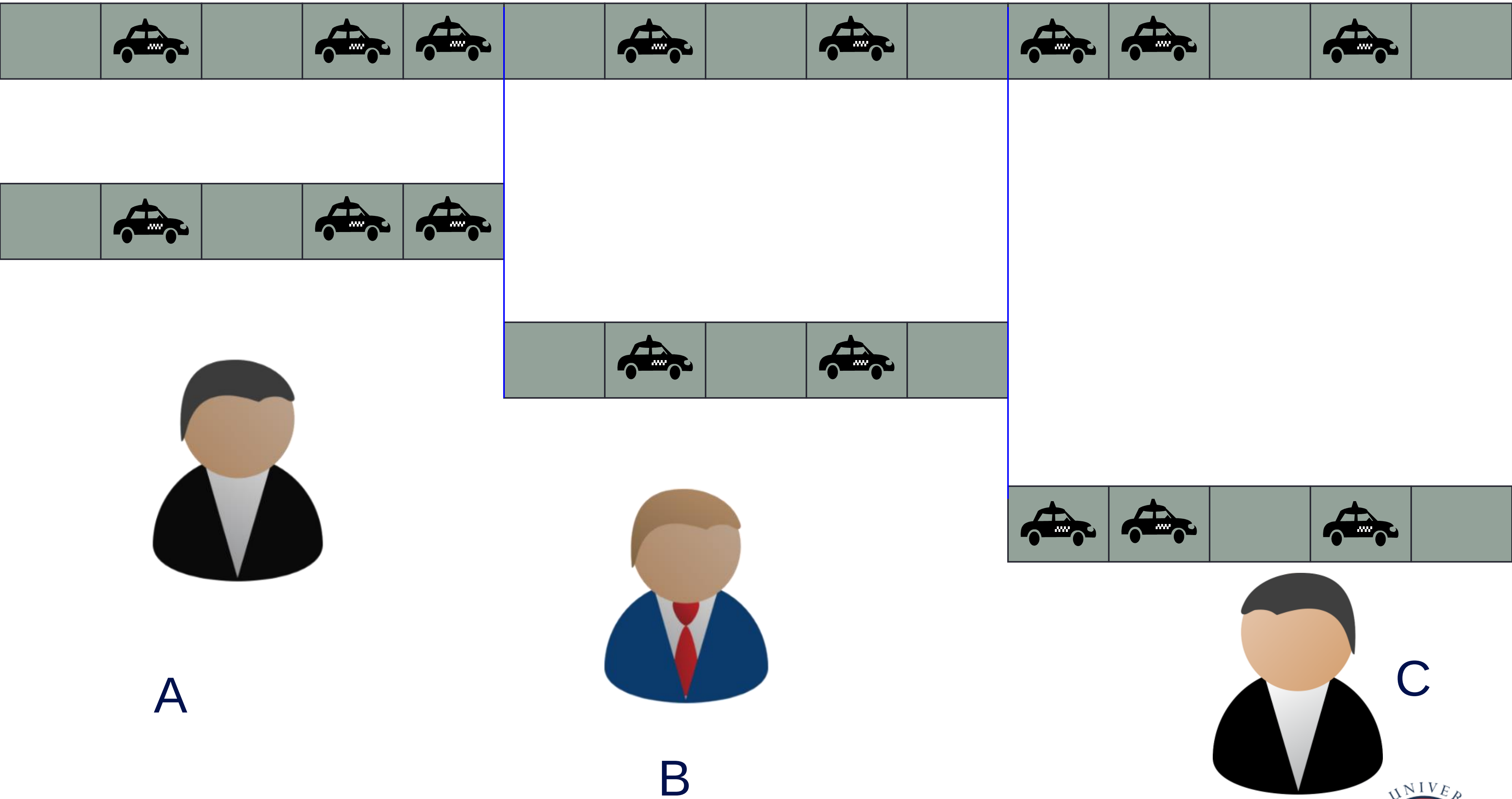
a parallel traffic model



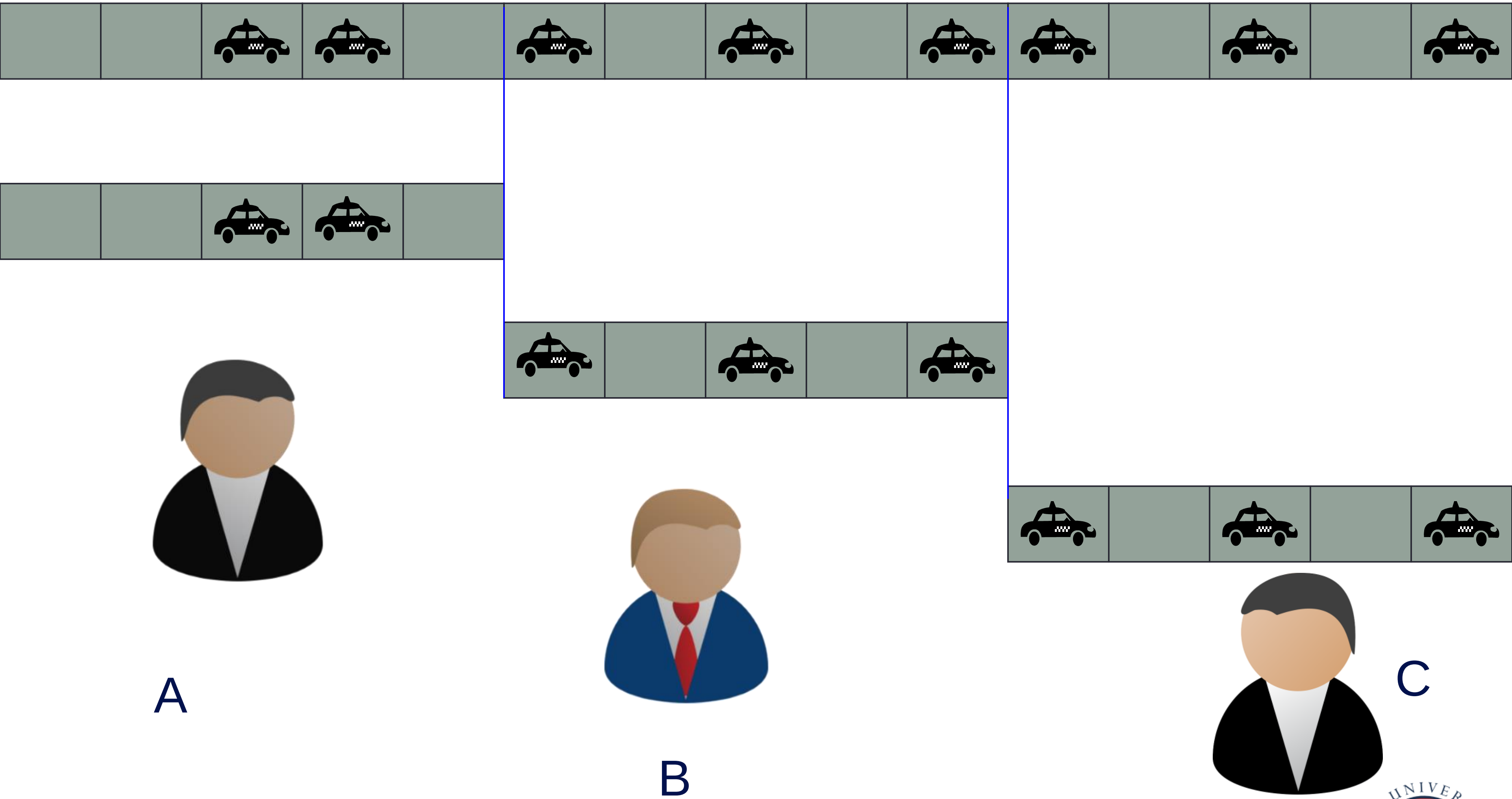
a parallel traffic model



a parallel traffic model



a parallel traffic model



Traffic Model

Parallel Solutions

The Model

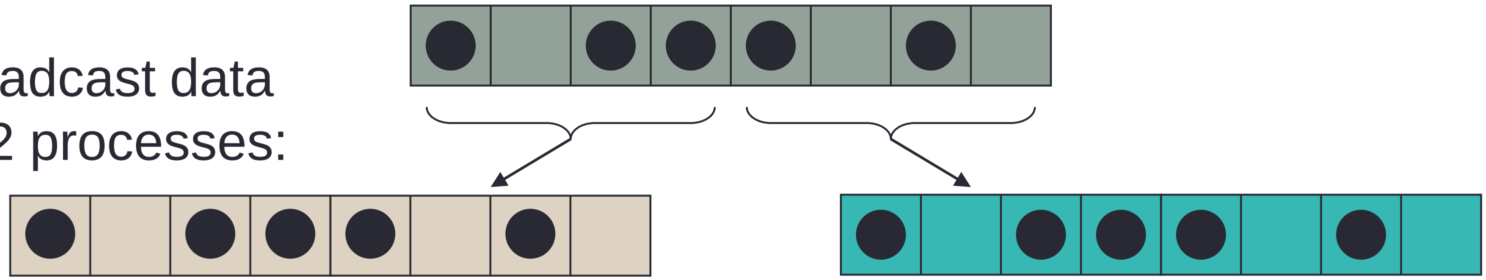
- Consider a road with *N* cells
- Simulate traffic on a ***roundabout***
  - i.e. periodic boundary counditions
- If a car moves off the right it reappears on the left
  - i.e. identify cell N+1 with cell 1, and cell 0 with cell N

Pseudo Code

```
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
```

Message-Passing Strategy (1)

Broadcast data
to 2 processes:



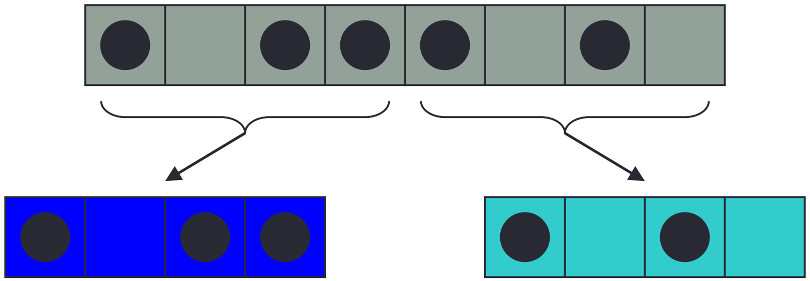
Split calculation
between 2 processes:

Process 1 Process 2

- Globally resynchronise all data after each move
  - a **replicated data** strategy
- Every process stores the entire state of the calculation
  - e.g. any process can compute total number of moves

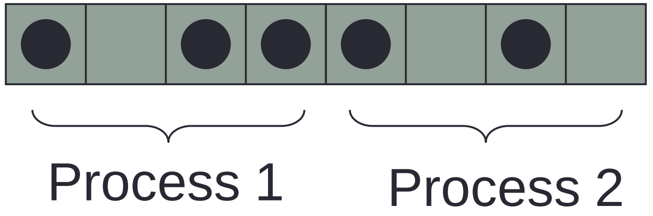
Parallelisation Strategy (2)

Scatter data
between 2 processes:
distributed data strategy



•Internal cells can be updated independently.
•Must communicate with neighbouring processes to update edge cells.
•Sum local number of moves on each process to obtain total number of moves at each iteration.

Split calculation
between 2 processes:

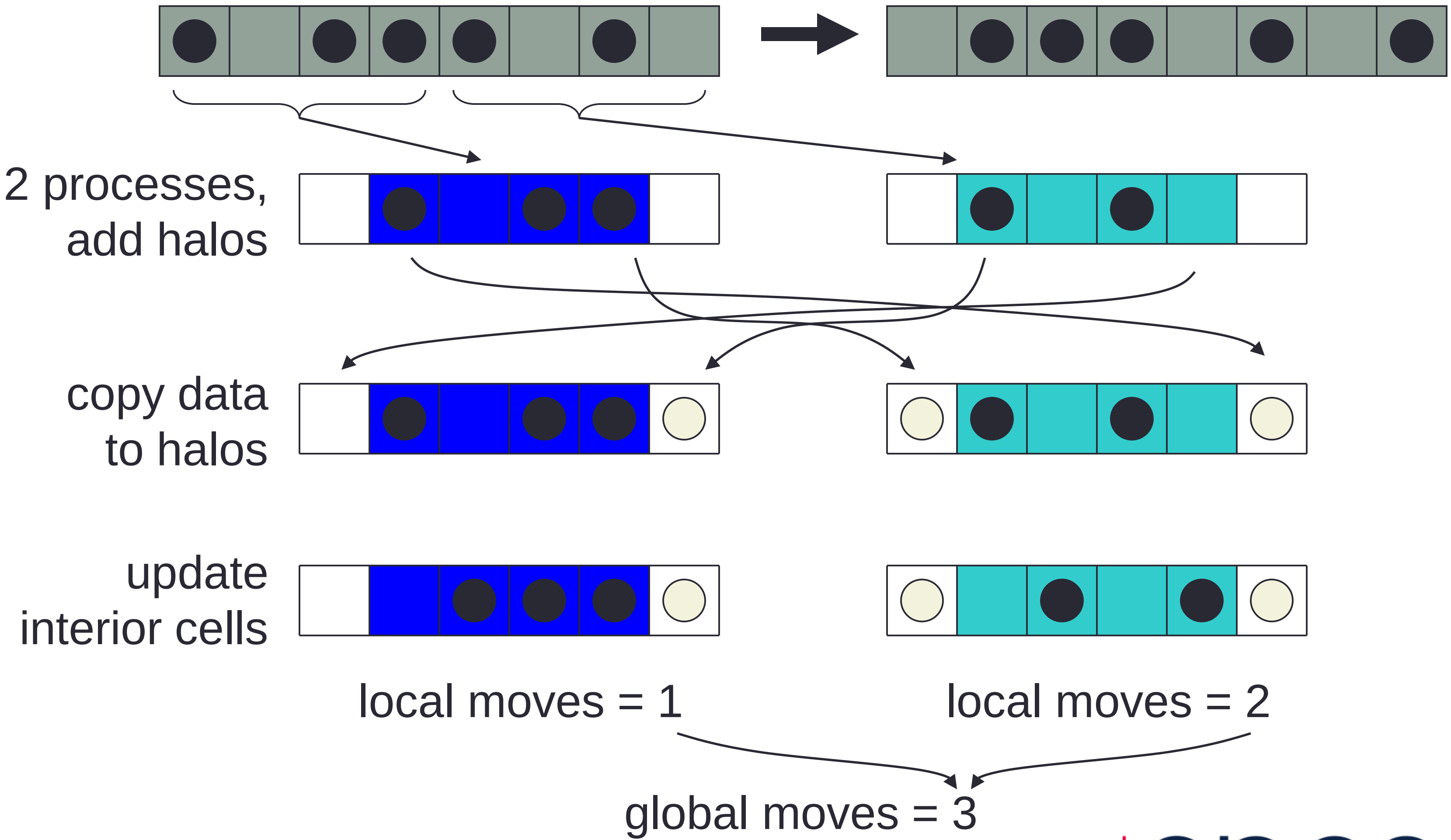


•Each process must know which part of roadway it is updating.
•Synchronise at completion of each iteration and obtain total number of moves

Parallelisation

- Load balance not an issue
  - updates take equal computation regardless of state of road
  - split the road into equal pieces of size *N*/*P*
- For each piece
  - rule for cell *i* depends on cells *i*-1 and *i*+1
  - the *N*/*P* - 2 interior cells can be updated independently in parallel
  - however, the edge cells are updated by other processors
    - similar to having separate rules for boundary conditions
- Communications required
  - to get value of edge cells from other processors
  - to produce a global sum of the number of cars that move

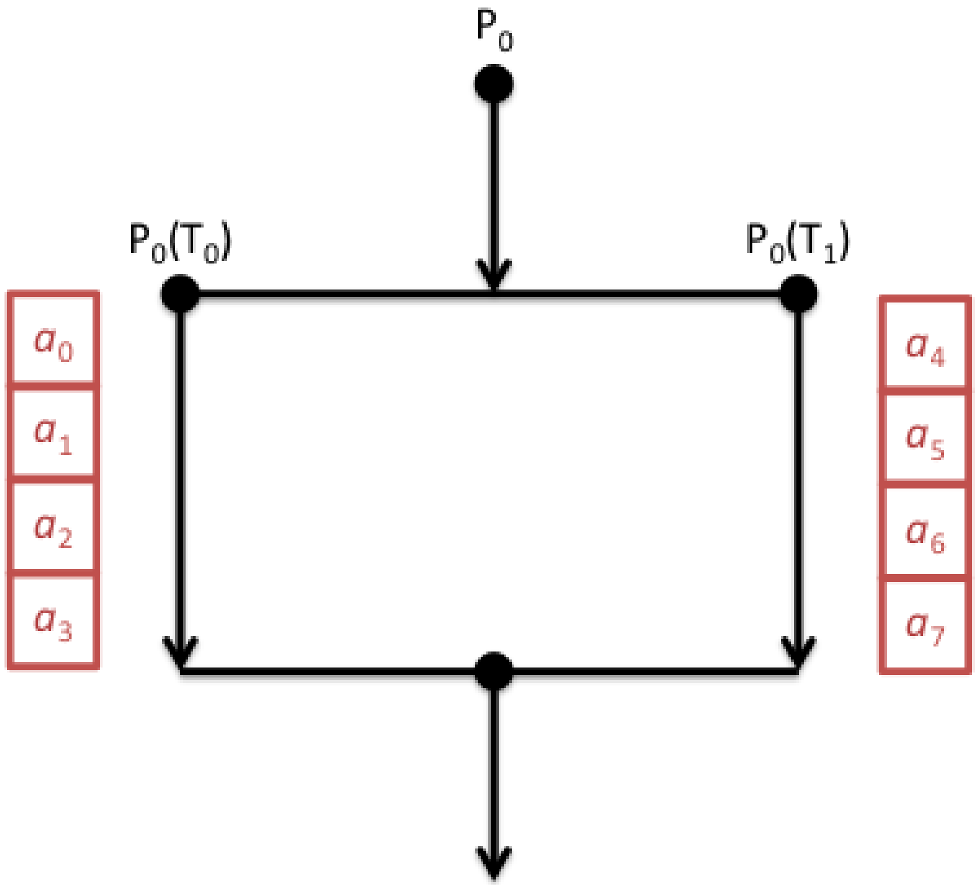
Message Passing Parallelisation



Threads Parallelisation

- Load balance not an issue
  - updates take equal computation regardless of state of road
  - split the road into equal pieces of size *N*/*T* (for *T* threads)
- For each piece
  - rule for cell *i* depends on cells *i*-1 and *i*+1
  - can parallelise as we are updating new array based on old
- Synchronisation required
  - to ensure threads do not start until boundary data is updated
  - to produce a global sum of the number of cars that move
  - to ensure that all threads have finished before next iteration

Fork-Join Model



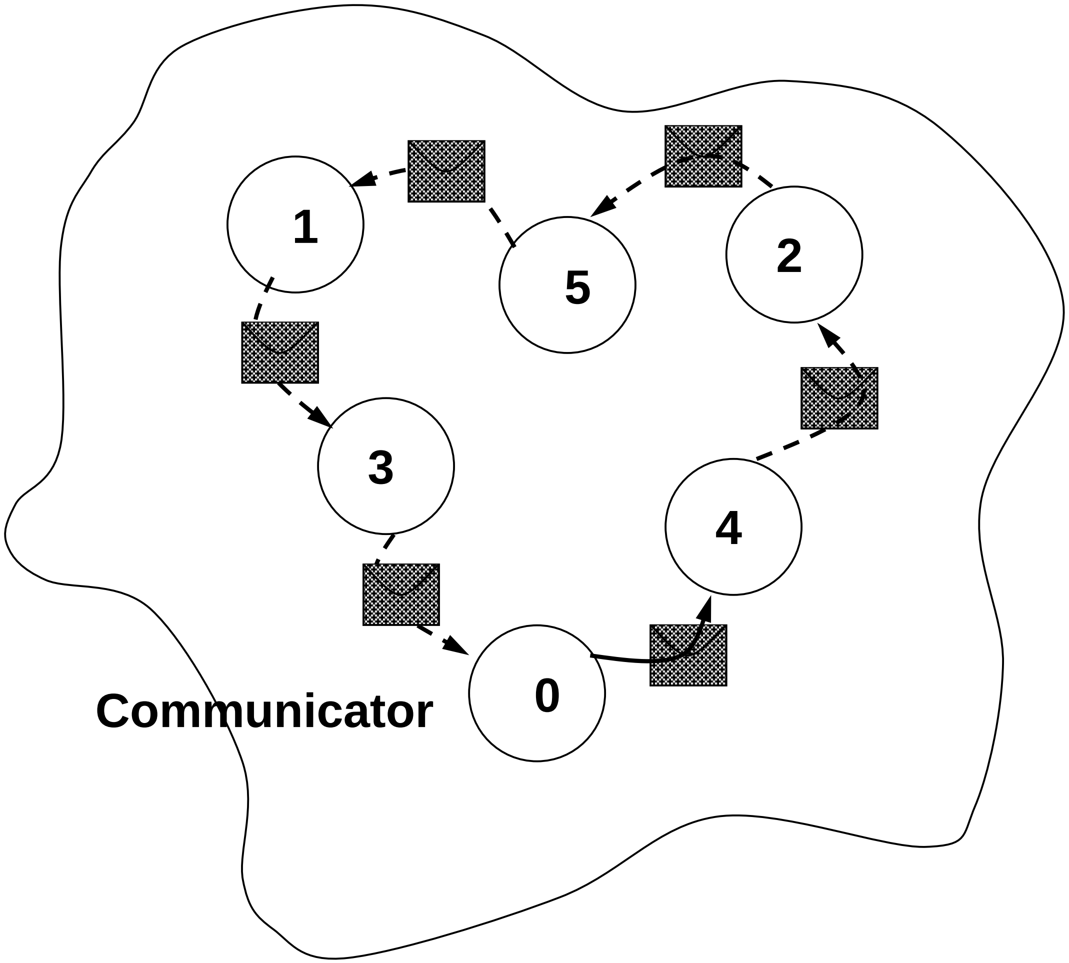
Shared Variables Parallelisation

```
serial: initialise old(i) for i = 1,2,...,N-1,N
serial: loop over iterations
  serial: set old(0) = old(N) and set old(N+1) = old(1)
  parallel: loop over i = 1,...,N
              if old(i) = 1
                if old(i+1) = 1 then ...
              if old(i) = 0
                if old(i-1) = 1 then ...
              end loop over i
  synchronise
  parallel: set old(i) = new(i) for i = 1,2,...,N-1,N
  synchronise
end loop over iterations
```

- private: i; shared: old, new, N
  - reduction operation to compute number of moves

Non-Blocking Communications

Deadlock



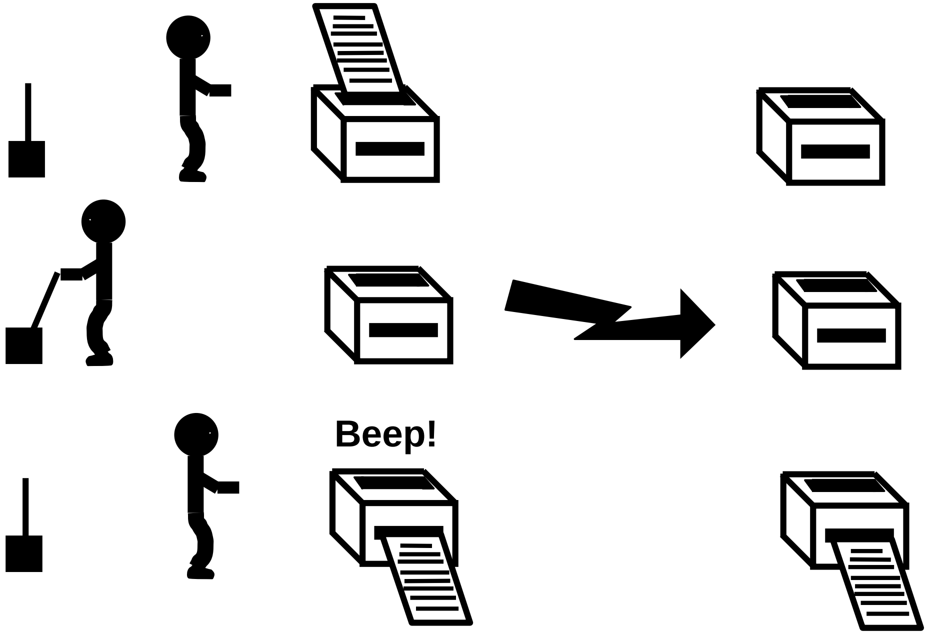
Completion

- The *mode* of a communication determines when its constituent operations complete.
  - i.e. synchronous / asynchronous

- The *form* of an operation determines when the procedure implementing that operation will return
  - i.e. when control is returned to the user program

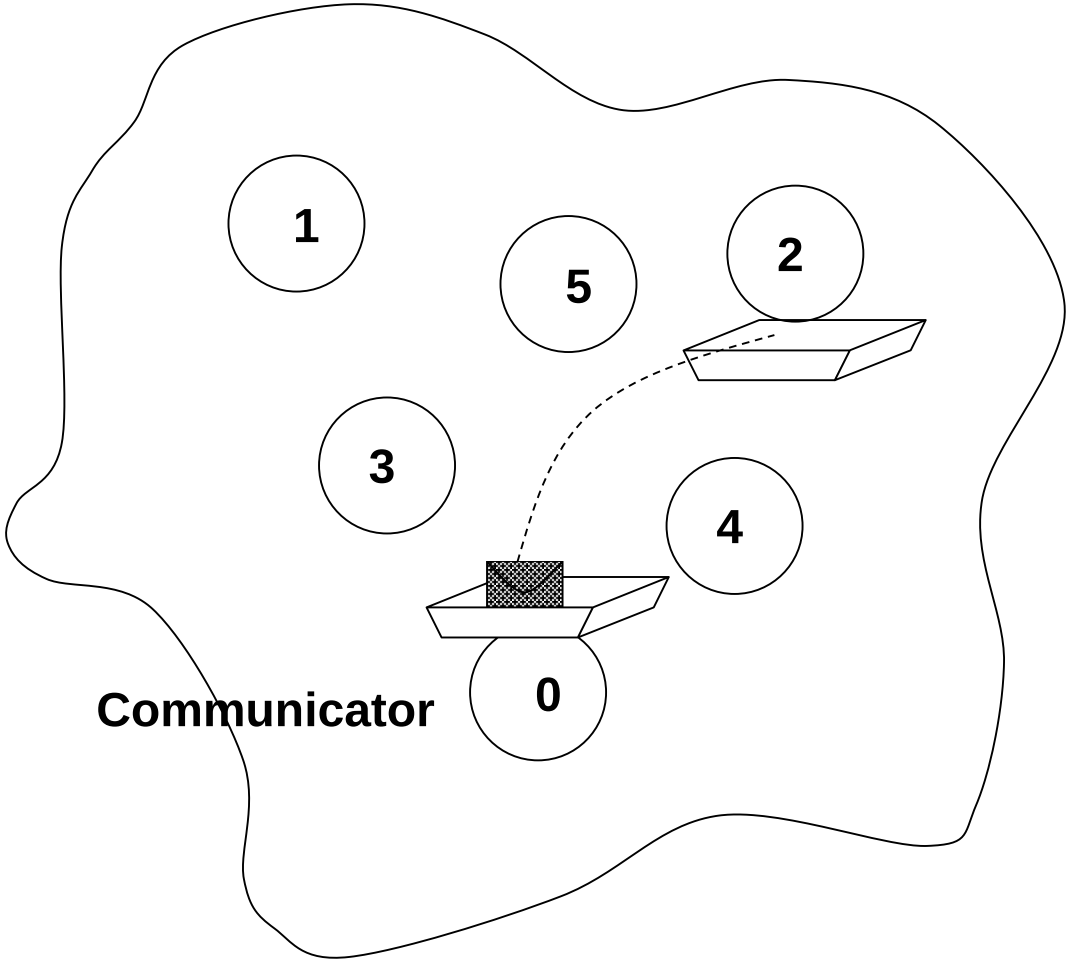
- Relate to when the operation has completed.
- Only return from the subroutine call when the operation has completed.
- These are the routines you used thus far
  - MPI_Ssend
  - MPI_Recv

- Return straight away and allow the sub-program to continue to perform other work. At some later time the sub-program can *test* or *wait* for the completion of the non-blocking operation.

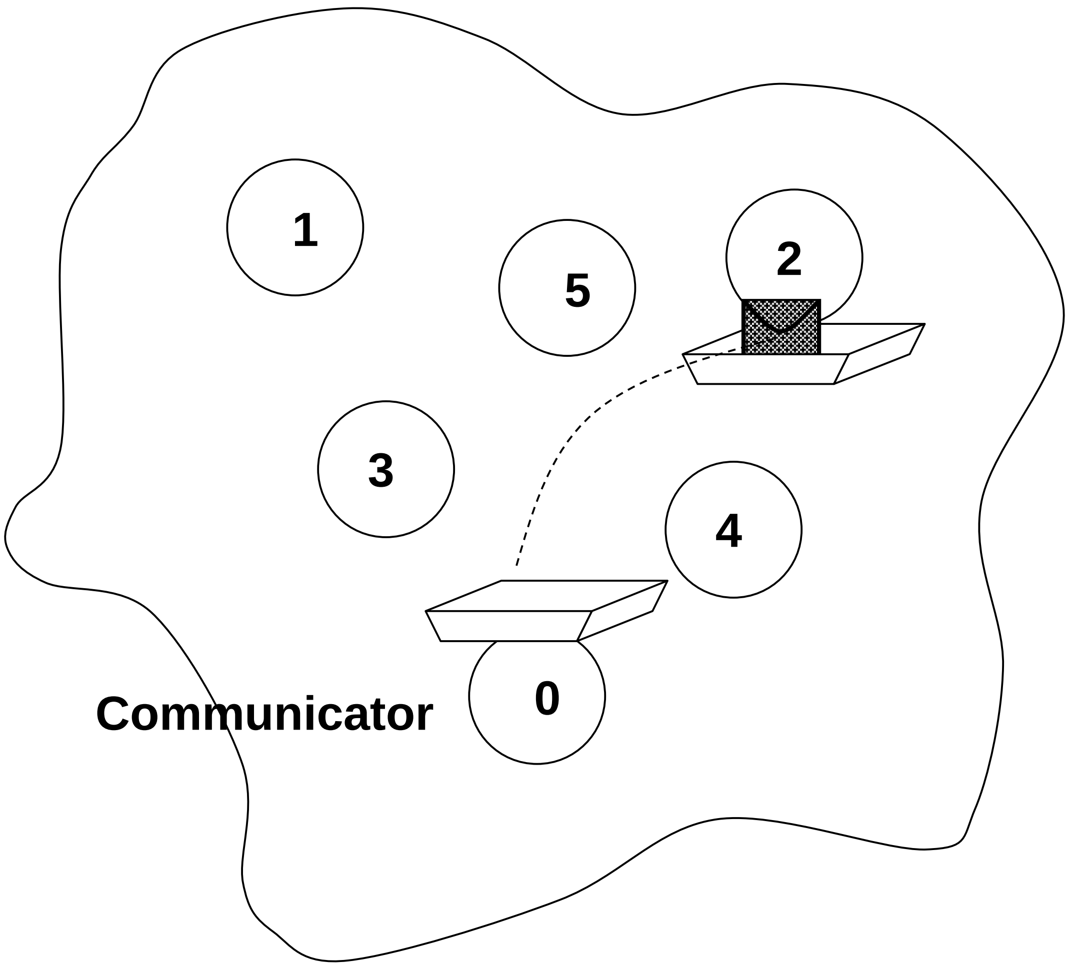


- All non-blocking operations should have matching wait operations. Some systems cannot free resources until wait has been called.
- A non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation.
- Non-blocking operations are not the same as sequential subroutine calls as the operation continues after the call has returned.

- Separate communication into three phases:
- Initiate non-blocking communication.
- Do some work (perhaps involving other communications?)
- Wait for non-blocking communication to complete.


1
5
2
3
4
0
Communicator

Non-Blocking Receive



- datatype same as for blocking (`MPI_Datatype` or `INTEGER`).
- communicator same as for blocking (`MPI_Comm` or `INTEGER`).
- request `MPI_Request` or `INTEGER`.
- A *request handle* is allocated when a communication is initiated.

- C:

```
int MPI_Issend(void* buf, int count,
           MPI_Datatype datatype, int dest,
           int tag, MPI_Comm comm,
           MPI_Request *request)

int MPI_Wait(MPI_Request *request,
             MPI_Status *status)
```

- Fortran:

```
      MPI_ISSEND(buf, count, datatype, dest,
                 tag, comm, request, ierror)

      MPI_WAIT(request, status, ierror)
```

- C:

```
int MPI_Irecv(void* buf, int count,
              MPI_Datatype datatype, int src,
              int tag, MPI_Comm comm,
              MPI_Request *request)

int MPI_Wait(MPI_Request *request,
             MPI_Status *status)
```

- Fortran:

```
      MPI_IRECV(buf, count, datatype, src,
                tag, comm, request, ierror)

      MPI_WAIT(request, status, ierror)
```

- Send and receive can be blocking or non-blocking.
- A blocking send can be used with a non-blocking receive, and vice-versa.
- Non-blocking sends can use any mode - synchronous, buffered, standard, or ready.
- Synchronous mode affects completion, not initiation.

NON-BLOCKING OPERATION	MPI CALL
Standard send	MPI_ISEND
Synchronous send	MPI_ISSEND
Buffered send	MPI_IBSEND
Ready send	MPI_IRSEND
Receive	MPI_IRECV

- Waiting versus Testing.
- C:

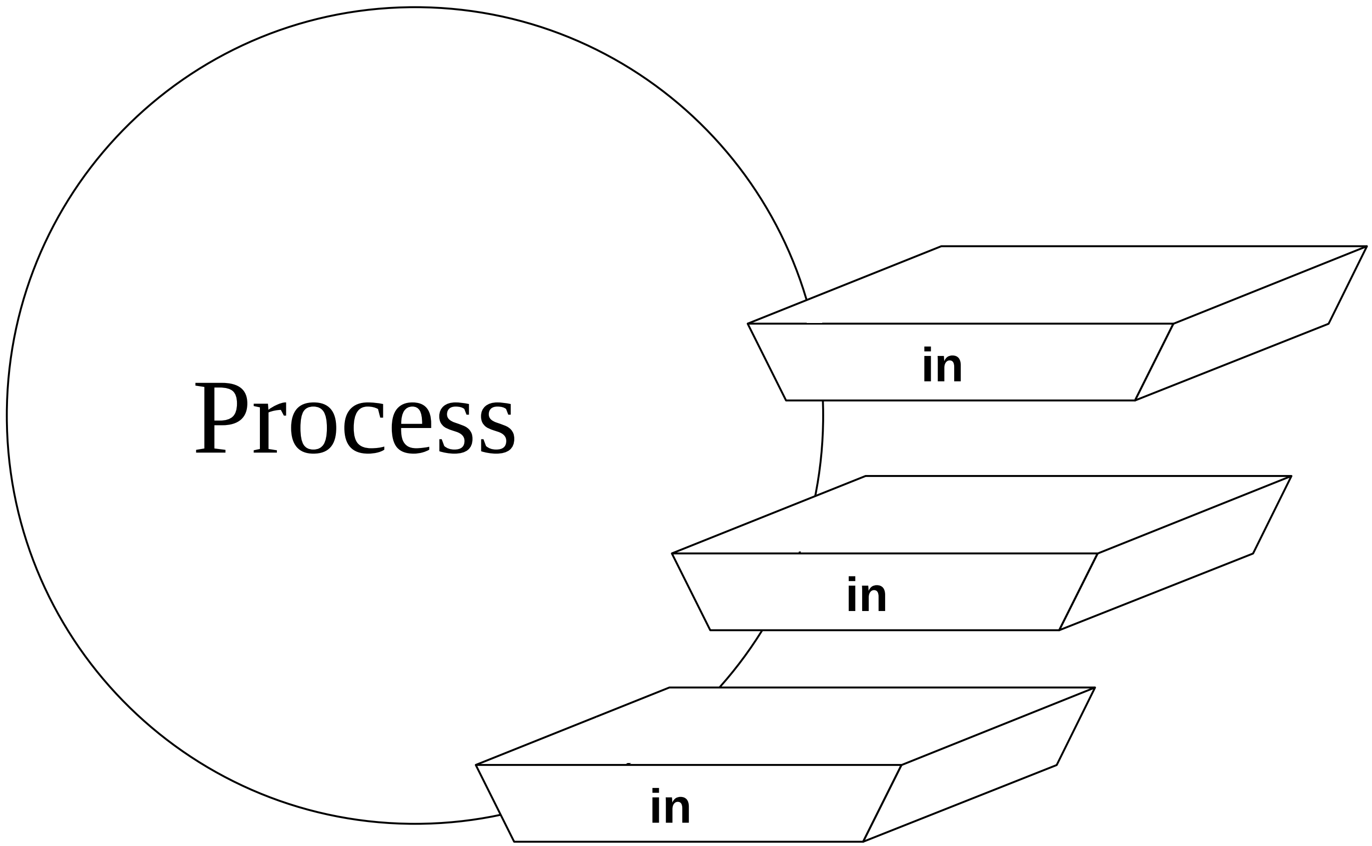
```
int MPI_Wait(MPI_Request *request,
             MPI_Status *status)
int MPI_Test(MPI_Request *request,
             int *flag,
             MPI_Status *status)
```

- Fortran:

```
MPI_WAIT(handle, status, ierror)

MPI_TEST(handle, flag, status, ierror)
```

- Test or wait for completion of one message.
- Test or wait for completion of all messages.
- Test or wait for completion of as many messages as possible.


Process
in
in
in

Combined Send and Receive

- Specify all send / receive arguments in one call
  - MPI implementation avoids deadlock
  - useful in simple pairwise communications patterns, but not as generally applicable as non-blocking

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 int dest, int sendtag,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype,
                 int source, int recvtag,
                 MPI_Comm comm, MPI_Status *status);
```

```
MPI_SENDRECV(sendbuf, sendcount, sendtype, dest,   sendtag,
             recvbuf, recvcount, recvtype, source, recvtag,
             comm, status, ierror)
```

Rotating information around a ring

- See Exercise 4 on the sheet
- Arrange processes to communicate round a ring.
- Each process stores a copy of its rank in an integer variable.
- Each process communicates this value to its right neighbour, and receives a value from its left neighbour.
- Each process computes the sum of all the values received.
- Repeat for the number of processes involved and print out the sum stored at each process.

Possible solutions

- Non-blocking send to forward neighbour
  - blocking receive from backward neighbour
  - wait for forward send to complete
- Non-blocking receive from backward neighbour
  - blocking send to forward neighbour
  - wait for backward receive to complete

- Non-blocking send to forward neighbour
- Non-blocking receive from backward neighbour
  - wait for forward send to complete
  - wait for backward receive to complete

Notes

- Your neighbours *do not change*
  - send to left, receive from right, send to left, receive from right, …
- You *do not alter* the data you receive
  - receive it
  - add it to you running total
  - pass the data *unchanged* along the ring
- You *must not access* send or receive buffers until communications are complete
  - cannot read from a receive buffer until after a wait on irecv
  - cannot overwrite a send buffer until after a wait on issend