Parallel design patterns ARCHER course

The actor pattern

Reusing this material

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

https://creativecommons.org/licenses/by-nc-sa/4.0/

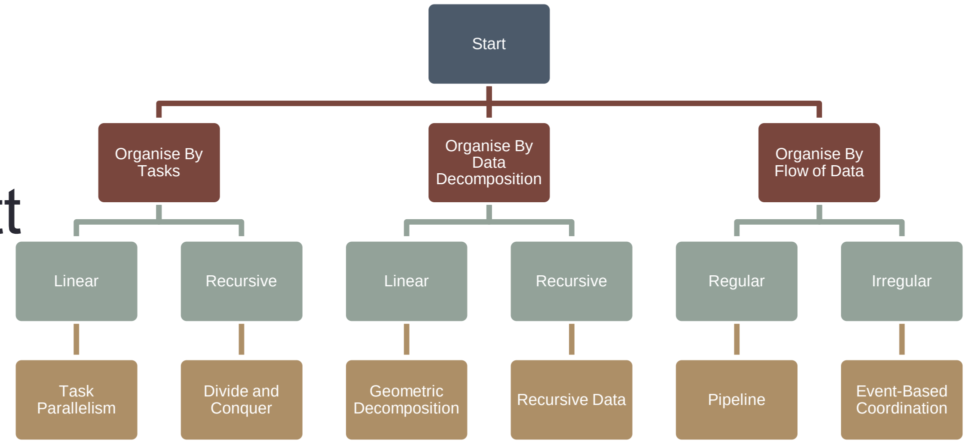
This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Acknowledge EPCC as follows: "© EPCC, The University of Edinburgh, www.epcc.ed.ac.uk"

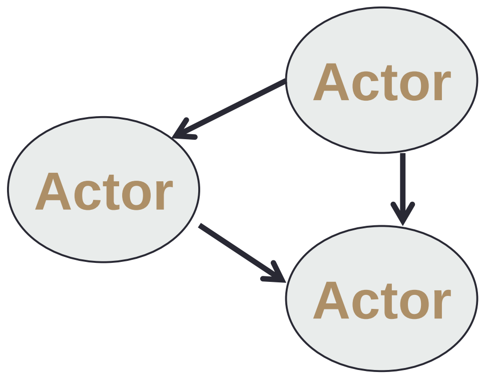
Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

The Actor Pattern

- The Actor Pattern is a *Parallel Design Pattern* in the *algorithm strategy/parallel algorithm structure* space
- It does not fit directly into the classification of other patterns
  - It is **closely related** to Event Based Coordination
    - Forces and context are basically the same as event based coordination
  - Major organising feature of the actor pattern is where parallelism will be unpredictable
- Was first described as the “actor model” by Carl Hewitt
  - In the 1970s as a way of organising software

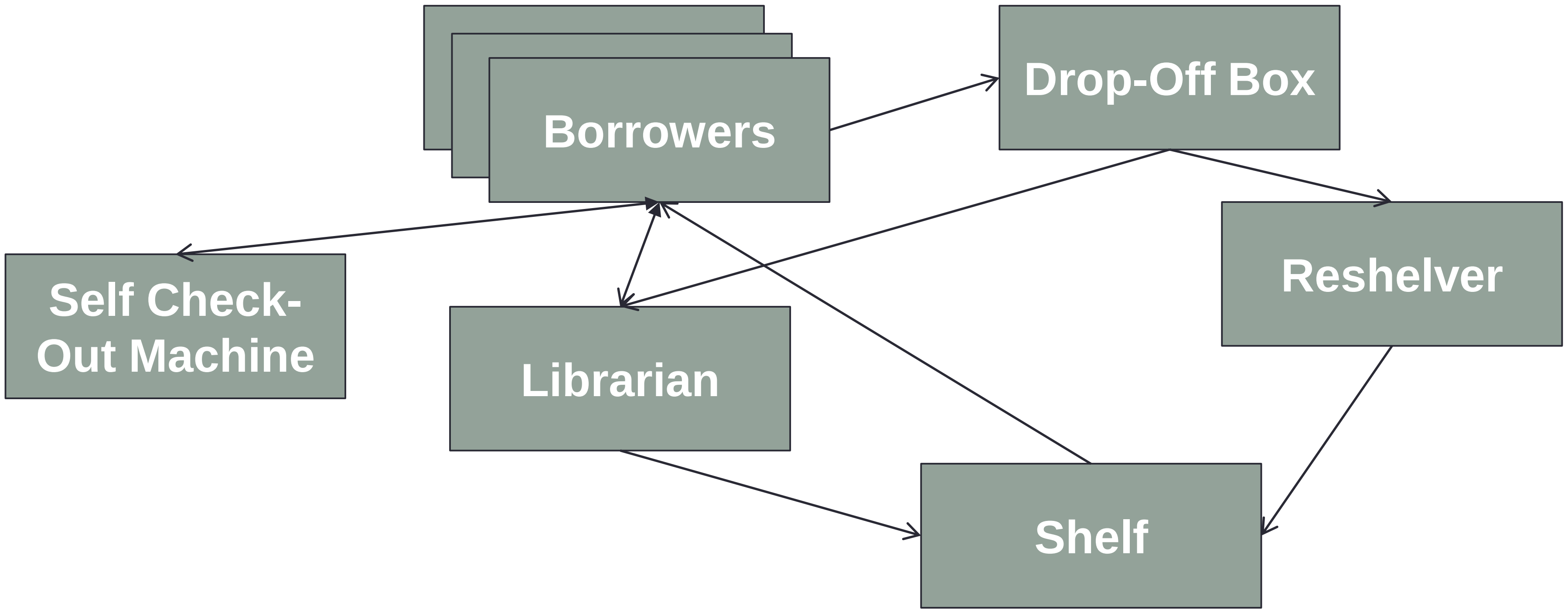


The Actor Pattern



- Like event-based co-ordination, the solution is to map real-world entities on to tasks with a 1:1 mapping
- The mapping of tasks to UEs is *often* 1:1, but it's also quite common to map several (in some cases, *many*) tasks to each UE
- Conceptually the model is of independent actors interacting *only* through the exchange of messages
  - Actor pattern uses the terminology "message"
  - A "message" is like an event, and it has an intended recipient (another actor)
  - Very similar to an MPI message but note that an MPI message is between *processes* and an Actor message is between *actors*

Contrast with Event based coordination



- In event based coordination we are concerned with books flowing between the entities (event handlers)
- In the actor patterns each of these is a different actor, communication can involve the transfer of books but also other items (such as fines from the librarian.)

Everything is an Actor

- The Actor Pattern "philosophy" is:

Everything is an Actor

- In the same spirit to
  - "everything is an object" in OO programming
  - "everything is a file" in UNIX
- This is a *way of thinking* about your problem

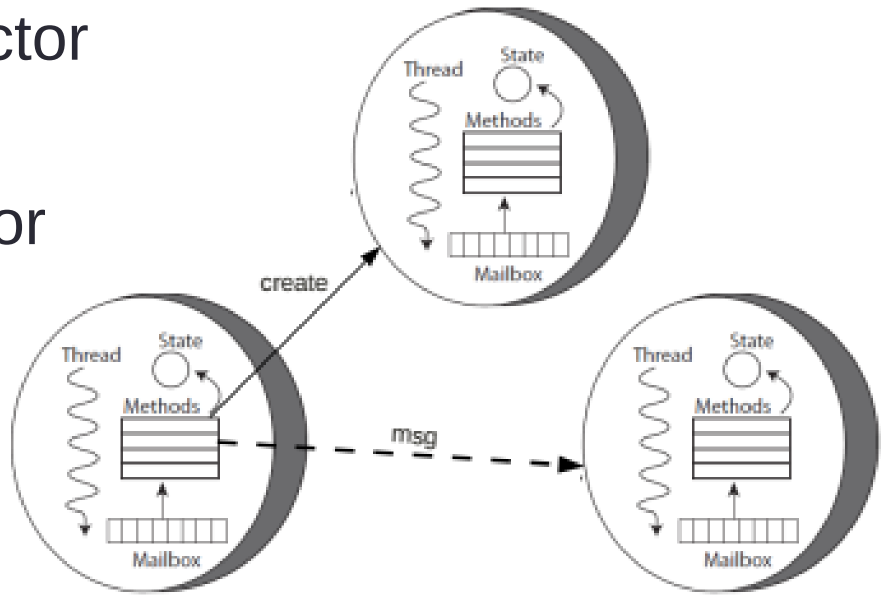
Everything an actor?

- In fact, just like with all patterns, the Actor pattern can be combined with other patterns
  - …but don’t use non-actor components just because they’re more familiar to you
  - Try to think within the actor pattern and **make everything an actor**

- Why make everything an Actor?
  - Maintains a symmetry
    - All elements in the program can interact in the same way: Through messages
    - Don’t need to add the complication of how actors communicate with non-actors
    - As soon as we start to add none actors then loose some of the advantages of this model

An actor can…

- Receive a message from any actor
- Do computational work
- Send a message to another actor
- Create a new actor
- Die



- It is entirely up to a specific actor to decide how to respond to a message from another and different actors might very well respond in different ways
- They maintain their own state

An actor should

- Be perfectly encapsulated, ideally there should not be any shared state between them
- Represent and be anything
  - Within an actor can still use other parallel patterns to help with computational work
- An actor doesn’t need to care about what other actors are doing (apart from understanding their messages.)

Things that can be actors

- Objects representing *particles*, *people, animals, books,* or *any real-life entity*
- Grid cells
  - an alternative to domain decomposition
  - useful, for example, when the actual geometry is less important and the main interaction with the grid cell is with other actors
- Global features / fields
  - Actors don't have to represent something localised in space
- Clocks
  - Actors generally act asynchronously and out-of-lockstep
  - It is sometimes useful to have some global notion of time which can be implemented by a clock Actor which sends messages to those Actors that need to be aware of global time
  - Time is often coarse grained without notion of computational steps

Benefits of the actor pattern

- Very flexible
  - As anything can be an actor then (theoretically) we can use this to model just about anything
  - If your system contains a number of different types of entities that need to interact then this can be helpful
- Actors encompass the ideas of modularity and encapsulation
  - As they are self contained and atomic, it should be trivial to add new types of actors
  - Do need a way to ensure that other actors can deal with messages from them
- Easy to conceptualise so can simplify parallelisation
  - Such as deadlock avoidance

Criticisms of the actor pattern

- There is often less order to the system
  - How do we do effective load balance if the actors have different computational requirements?
  - As actors can create other actors dynamically the state of the system can change dramatically and unpredictably.

- Can involve many messages flowing around unpredictably
  - Hard to design any locality into communication
  - Need to be careful when it comes to message ordering (as in the event based coordination model)
  - Unbounded nondeterminism, where the delay in servicing a message can appear to have no limit whilst still guaranteeing that the request will eventually be serviced.

Programming Languages & Libraries

- ABCL
- AmbientTalk
- Axum
- E
- Erlang
- Fantom
- Humus
- Io
- Ptolemy Project
- Rebeca Modeling Language
- Reia
- Rust
- SALSA
- Scala
- Scratch

- Akka
- Ateji PX
- F# MailboxProcessor
- Korus
- Kilim
- ActorFoundry (based on Kilim)
- ActorKit
- Retlang
- Jetlang
- Haskell-Actor
- GPars (was GParallelizer)
- PARLEY
- Pykka (inspired by Akka)
- Termite Scheme
- Theron
- Libactor
- Actor-CPP
- S4
- libcppa

Source: Wikipedia (http://en.wikipedia.org/wiki/Actor_model)

The practicalities of the actor pattern

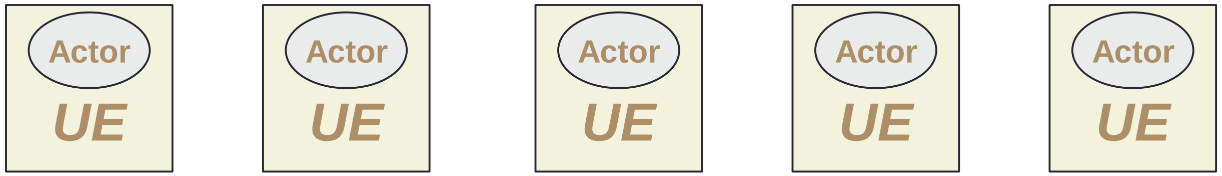
- MPI is **not** a perfect fit for the Actor pattern but when used in the right way, MPI can be used to implement the Actor pattern
  - What's more, if you want to run an Actor Pattern-based code on a massively parallel machine, you probably don't have a lot of choice
    - or at least, your other choices just have different shortcomings

- Harder things to do with MPI:
  - Creation and destruction of Actors
  - Fire-and-forget asynchronous messages

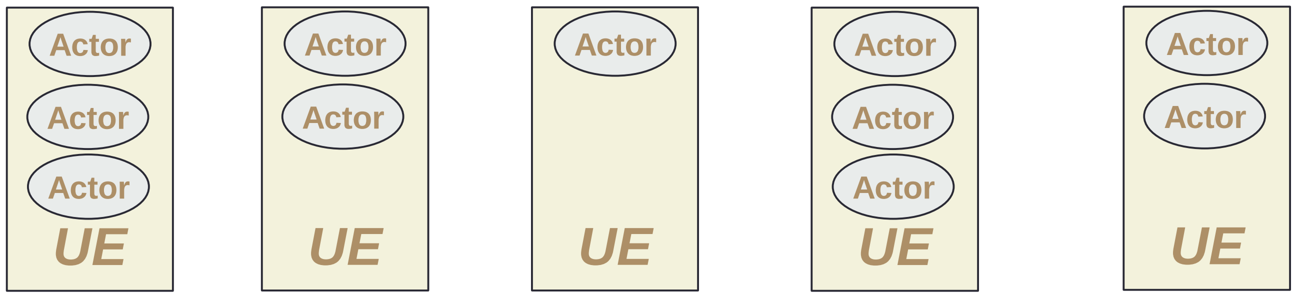
- Unnecessary aspect of MPI:
  - Program looks like it's SPMD. All actors have to start off running the same program

One Actor per UE

- Advantages
  - Conceptually more simple
  - Exposes most parallelism
  - Actor messages map directly to MPI messages
  - It’s possible (although not always desirable) to use MPI ranks to index the actors
- Disadvantages
  - Might require a very large number of UEs
  - Load balancing might become an issue

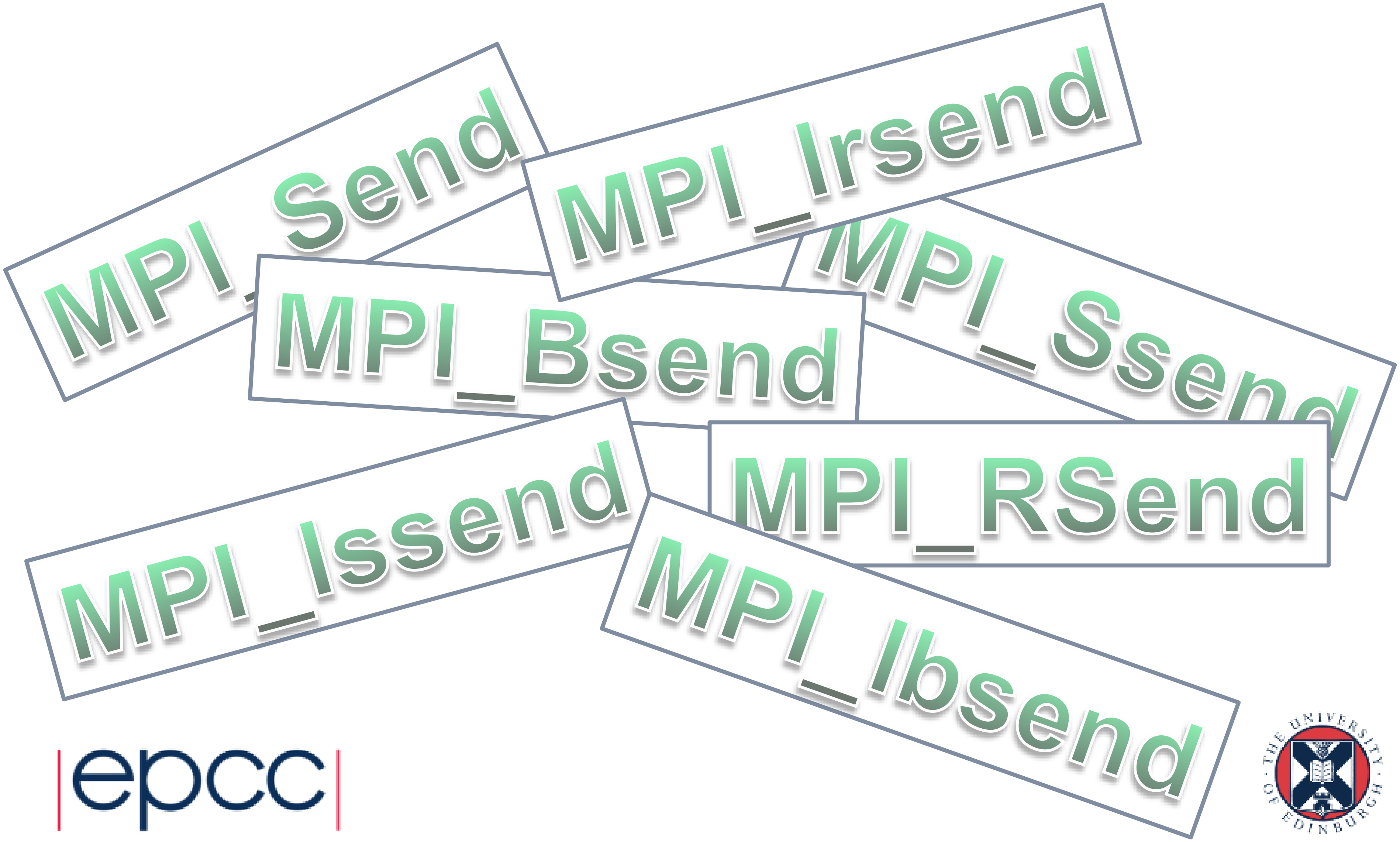


Multiple Actors per UE



- Advantages
  - Less low level parallel overhead (number of UEs can match target architecture)
  - Actor creation and destruction is simpler as don't need to create any UEs
  - Can mix actors with different computational requirements
- Disadvantages
  - You loose symmetry (actors on the same UE (local) as well as remote actors can communicate)
  - Need to provide your own messaging solution, such as an event queue for each UE.

What type of MPI message to use?



Buffered Sends for Actor Messages

- None are perfect but the best one to is a **buffered send**, as this is the closest fit to *fire-and-forget*

- `int MPI_Bsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - **buf** initial address of send buffer (choice)
  - **count** number of elements in send buffer (nonnegative integer)
  - **datatype** datatype of each send buffer element (handle)
  - **dest** rank of destination (integer)
  - **tag** message tag (integer)
  - **comm** communicator (handle)

How buffered sends work

- MPI_Bsend causes the contents of *data* to be copied into an internal MPI buffer
- As soon as the contents of *data* have been copied, the call completes and the process moves on to next line in the program
  - You can then re-use / modify *data* without the message being affected
  - It is not guaranteed that the message has been received by the receiver, and there's no way to check that the message has been received
- The programmer must specify the size of the buffer with MPI_Buffer_attach
- If the buffer is full, your program will error

MPI_Buffer_attach

- `int MPI_Buffer_attach(void *buffer, int size)`
  - **buffer** initial buffer address (choice)
  - **size** buffer size, in bytes (integer)

- For an actor pattern which could have many messages in transit, you should probably start with a large buffer size (allowing, say, hundreds of messages to be buffered)

What should I send?

- Contents of the buffer could, in general, be some kind of message structure
- In practice, if your application allows it, it can be far simpler to use a known (basic) data type, with a known count
  - For example, if you know that the only data that needs to be included with any of your messages is of integer type, and you know that you'll never need to send more than two integers in a message, you can also use an integer to define your message type and just make all of your sends and receives of the form
    - `MPI_Bsend(`*`data`*`, 3, MPI_INTEGER, …)`
    - `MPI_Irecv(`*`data`*`, 3, MPI_INTEGER, …)`

Where integer 1 is the command/type and 2 & 3 are data associated with it

Receiving Messages

- MPI requires matching sends and receives
- The actor pattern requires that an actor can get on with doing what it is doing and not have to wait for messages, we have two choices (I find the second tends to be simpler):
  - **MPI_Irecv** but the downside is keeping track of request handles and cancelling this in termination
  - **MPI_Iprobe** to check for messages and then **MPI_Recv** if a message is outstanding can be simpler – as no request handles.

- Since buffered sends are used, MPI messages can queue up, so for a simple implementation you only need to post one receive at a time
  - If you need to "look ahead" in your queue, you might want more

An Actor in MPI

Simple codes just wait here for a message

```
do {
  MPI_Irecv(message, …, request)
  while not MPI_Test(request,…){
        do_compute_work_step()
  }
  process(message)
}
```

```
do {
  MPI_Iprobe(…, outstanding, status)
  if (!outstanding) {
        do_compute_work_step()
  } else {
     MPI_Recv(message,
     process(message, …, status.MPI_SOURCE, …)
  }
}
```

- Both *do_compute_work_step* and *process* functions could include **MPI_Bsend**s to send off new messages

- If *process* is time consuming, it could just add *message* to a local message queue to be handled during *do_compute_work_step*

Receiving data from anyone

- Remember that an actor might (unpredictably) receive data from any other actor
  - It is therefore common to use *MPI_ANY_SOURCE* in place of a receiver's explicit process id
  - Can have a look at the MPI status to figure out the pid of the sender

```
MPI_Request request;
MPI_Status status;

MPI_Irecv(&message, 3, MPI_INT,
       MPI_ANY_SOURCE, …,
       &request)
MPI_Wait(&request, &status);

int source=status.MPI_SOURCE;
```

```
MPI_Status status;
int outstanding;

MPI_Iprobe(MPI_ANY_SOURCE, …,
        outstanding, &status)
if(outstanding) {
  int source=status.MPI_SOURCE;
  ……
}
```

In Fortran the status is an integer array, status(MPI_STATUS_SIZE), and the source rank is an element of this array which you can grab via status(MPI_SOURCE)

Creation and Destruction of Actors (1)

- One of the requirements of an Actor is that it can create other actors.
- Even if you’re doing a 1:1 mapping of actors to UEs you often want to avoid creating new MPI processes when creating a new actor.

- Solution: A process pool
  - At the start of the program, launch more processes than you’ll ever have actors
  - Ensure that the program never creates more actors than this limit

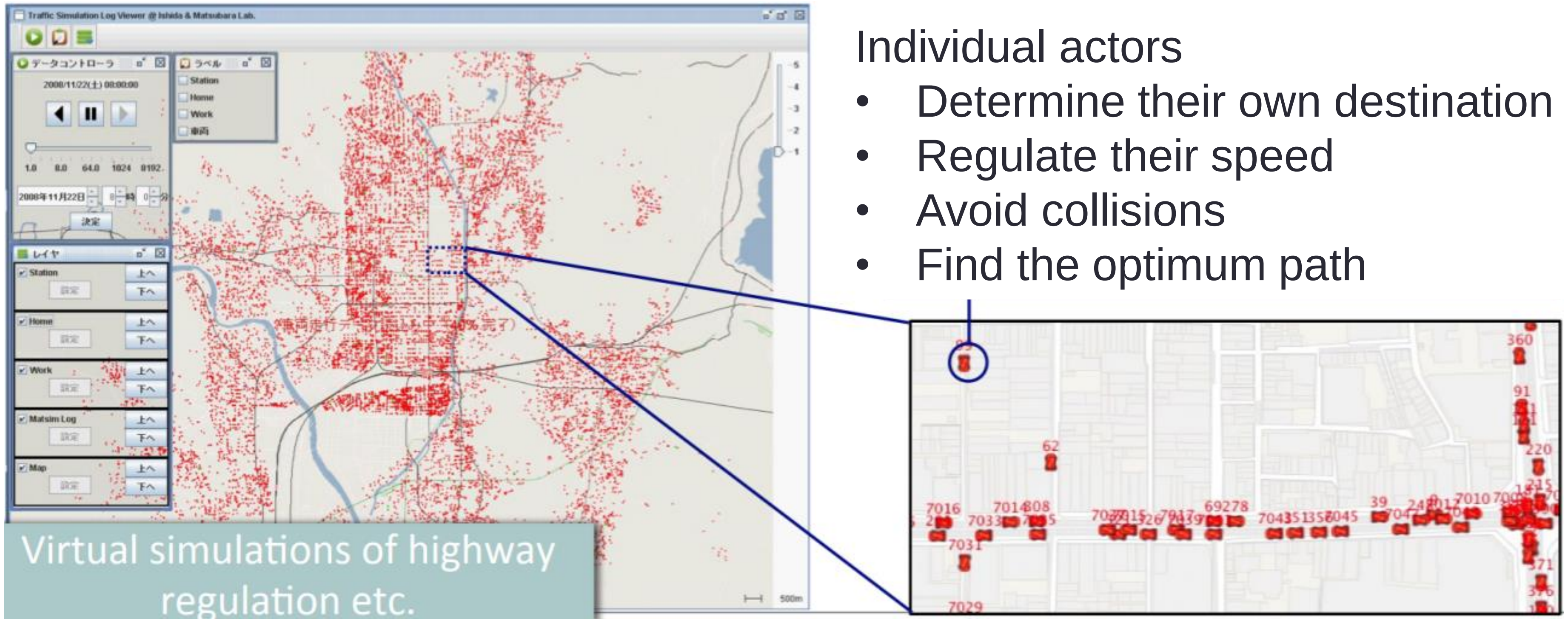
Creation and Destruction of Actors (2)

- The problem with the solution:
  - How do the actors know if there are processes left in the pool?

- The solution:
  - Use a master process to manage new actors
  - Have a special master process with its own actor whose job it is to manage the process pool
  - When an actor wants to create a new actor, it sends a message to the master with the required information, and it’s the master’s job to assign an MPI process from the process pool to the new actor
    - You effectively use a master-worker pattern with the worker’s task being: become an actor, and keep going through your event loop until you die

We will be playing with a process pool in the mergesort practical and also the actor case study

Social simulations (traffic in Kyoto)



Virtual simulations of highway regulation etc.

Individual actors

- Determine their own destination
- Regulate their speed
- Avoid collisions
- Find the optimum path

- Roads are actors (50,000)
- Cars are actors (up to 100,000)
- Time is coarse grained (run up to 200 days)

Conclusions

- We have talked about the Actor Pattern, similar to event based co-ordination but with some differences
  - Actors can create other actors and die
  - No need for external events
  - Actors can perform work not driven by messages
- A useful pattern which has the potential to be very important in the future
  - The loose synchronisation might be crucial for extremely large core counts
- MPI isn’t a perfect fit for the implementation, but can be used
  - Dynamic creation/destruction of actors is tricky