Modelling airport traffic in parallel using the actor pattern

1 Introduction

This case study is focused around developing a parallel model for aircraft traffic using the actor pattern and MPI for the parallelisation. The main purpose of this is to give you experience of using the actor pattern (and other associated patterns), as well as exposing you to a potentially different way of using MPI than you have previously done.

2 Actors

To bound the problem we will limit our focus to a number of different types of entity, each of which has its own behaviour and interaction pattern. Depending upon the exact entity, there might be single or multiple instances of that entity active at any one point in time. Each entity will be modelled as a type of actor in your code.

2.1 Different entities (types of actor)

The distinct entities in our problem, each of which will form a type of actor are:

- **Airspace**: This is some *external* air traffic organisation (such as the national ATC centre) which will handover aircraft to a local airport control centre and at time request statistics back about the number of aircraft being processed by that control centre. It is this handing over of aircraft that generates them, before this they are unknown to the model. To local air traffic control tower either accepts or rejects the handover of aircraft. This is because it's not always possible for aircraft to be handed over (for instance the local control tower might be busy) resulting in a refusal being sent from the control tower to the airspace actor. If the handover request is accepted, then the airspace actor will create a new aircraft actor.
- **Local air traffic control tower**: Controls the runway and local airspace. This actor must grant permission for an aircraft to be handed over to it from the airspace. If the number of aircraft that the control tower is currently handling is too large, then the request is refused. Crucially only one aircraft can be landing on the runway at any one point in time, so the control tower must keep track of whether the runway is busy (i.e. it has approved a landing request from an aircraft and this is not yet completed) or free (can be landed on by an aircraft.)
- **Aircraft**: The aircraft actor will request landing clearance from the control tower. Depending upon whether the runway is busy or not this will be granted or refused. If permission is refused then the aircraft keeps requesting permission until the request is granted. Once permission to land is granted, the aircraft actor will contact the runway actor and inform it that it is landing, the runway actor will send an acknowledgment back and lastly the aircraft actor will send a *landed* message to the control tower. Once the control tower grants permission for an aircraft to land it marks the runway busy and only when it receives this

landed message will it mark the runway free. Once an aircraft actor has landed it can terminate.

- **Runway**: Where the aircraft land, aircraft will inform the runway they are landing and the runway actor will send a success acknowledgement message back.

In our definition of the problem, there can only be one airspace, one control tower and one runway actor, but any number of aircraft actors. The reason we have the initial handshake between the airspace and control tower to *handoff* an aircraft is to limit the number of aircraft actors. For this problem we will set a limit of 40 active aircraft actors, if there are more than this the control tower refuses the request and no new actor is created. The aircraft actor is the only dynamic actor, the other actors running for as long as the program executes, and once an aircraft has successfully landed and send it's last message to the control tower it will *die*.

There should be an initial number of aircraft actors active when the code starts up.

2.2 Interactions between actors

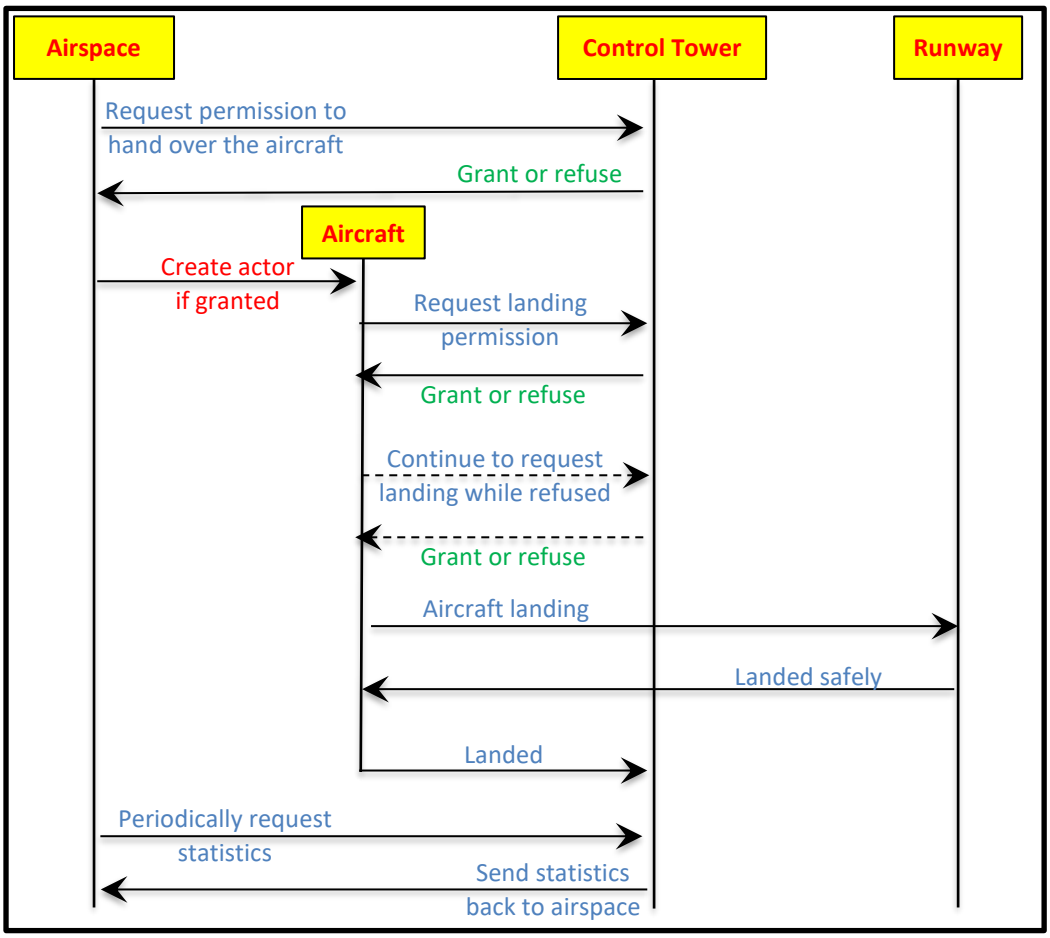

Figure 1: Interaction pattern between actors, where squares are actors and arrows indicate messages being exchanged or interaction between them

The interaction of actors is illustrated by figure 1, where the yellow boxes are actor types (the airspace, control tower and runway created when the program starts and remain until termination). The airspace actor creates any number of aircraft actors (when the control tower grants permission to hand the aircraft over to it) and this aircraft actor then requests landing permission from the control tower. This permission is either granted or refused and whilst it has been refused the aircraft will continually re-request permission from the control

tower. Once permission is granted, a message is sent from the aircraft actor to the runway informing it that it has landed and a message sent back confirming a safe landing. Lastly the aircraft actor will inform the control tower it has landed and then terminate (the UE running that actor can then be reused for another actor*.)* In between the control tower granting an aircraft actor landing permission and that actor informing the control tower it has landed, the control tower cannot grant permission to any other aircraft.

The airspace will periodically request statistics from the control tower, they are interested in the number of aircraft the control tower is currently handling and the number of successful landings. The airspace should also keep track of the total number of aircraft and the number of times the control tower has refused the *handover* request. This should be displayed periodically (once every *day* of simulation time, more on this in a minute.)

2.3 Time

In this problem time is coarse grained, and only the airspace actor needs to be aware of time. Mainly this is just so we don't swamp the control tower with *handover* requests and immediately run out of actors. It is up to you, but we suggest issuing a hand-over request every quarter of a second or so.

Statistics should be requested from the control tower, and once these have been received displayed, every ***day*** and for this time period I would suggest every 10 seconds. The code should ***terminate*** after 10 days (i.e. 100 seconds.) These are the suggested numbers, certainly you can set them to whatever value you want, but the key is to have some staggering between handing over new aircraft and requesting & displaying statistics. This notion of time is also useful for a meaningful termination point.

3 Provided code

You are provided with a skeleton implementation (more on this in a minute!) and a full process pool code that you already saw in the third practical. The process pool follows the master/worker pattern and you can treat this as a black box and provided here is a summary of its C API, with the Fortran API being similar.

The process pool is useful as it allows us to reuse UEs (the workers) for different actors which is a crucial part in the dynamic creation and destruction of actors (which MPI alone isn't so good at.) An actor runs on a worker (we strongly suggest one actor per worker/UE here to make things simple) and if that actor wants to create a new actor it can request a new worker from the process pool master, and then communicate with that worker to tell it the type of actor it is. When a worker terminates (i.e. an aircraft has landed), then the worker returns to the pool and can be reused for another aircraft actor in a minute.

Function	**Description**
int processPoolInit()	Initialises the process pool (1=worker, 2=master)
void processPoolFinalise()	Finalises and process pool (called from all)
int masterPoll()	Master polls to determine whether to continue or not
int workerSleep()	Worker waits for new task (1=new task, 0=stop)
int startWorkerProcess()	Starts a new worker task and returns the rank of this
int getCommandData()	Retrieves the rank of the task created this one
void shutdownPool()	Called by anyone to shut down the pool

4 Develop this parallel actor model!

The exercise here is to develop this model, in C or Fortran. **I strongly suggest mapping one actor per UE (per MPI process)** and you can use the process pool we looked at in practical three as a mechanism for creating and destroying actors. The fact that many of our actors are static makes things a bit easier when it comes to communication, and I suggest mapping rank 1 to the airspace actor, rank 2 to the control tower actor and rank 3 to the runway actor (rank 0 is mapped to the process pool master UE.) This means that you don't need to

worry about how actors map to ranks (aircraft actors will be on ranks 4 plus. Sending messages to these aircraft actors is only ever done in response to a message from an aircraft actor, this is important as it means you can extract the MPI rank from the message status.)

We have provided a skeleton implementation which implements the general behaviour of the actors and you can start from this. Alternatively, if you are feeling adventurous, you can start from scratch. In the instructions here, we will assume that you are working from the skeleton implementation:

- In the ***workerCode*** function you will need to receive the type of actor from the parent (probably an integer) and then based on this value call into the appropriate actor function. Once you have done this I suggest working on each actor one at a time.
- The aircraft actor (***aircraft*** function) is probably a reasonably good one to start from, in some ways this is simpler because it strongly follows a handshaking style interaction, where a message is sent to a different actor and an acknowledgement message sent back. As such you can get away with using blocking MPI receive calls, and I suggest using buffered sends.
- The runway actor (***runway*** function) is a good one to tackle next, Again this is fairly simple because in the while loop you can issue a blocking receive from MPI_ANY_SOURCE (i.e. any aircraft actor) and then send the appropriate message back to the rank using a buffered send and status.MPI_SOURCE as the rank (or status(MPI_SOURCE) if you are using Fortran).
- The control tower actor (***control_tower*** function) is a bit more complex, as multiple actors can be interacting with it concurrently (i.e. the airspace or any number of aircraft actors.) You can still use a blocking receive but like the runway actor from MPI_ANY_SOURCE and then looking at the message itself to determine what action to take. In the skeleton code we have kept much of the actor logic and commented in where messages need to be put in. We also make mention of termination here, ignore this for now as we will cover it in more detail in a few minutes.
- The airspace actor (***airspace*** function) is probably the most complex. Again, the general logical structure is in place but different message requests might be in progress at the same time (e.g. a request for handing over an aircraft might have been sent and before this has been responded to, a request for statistics data from the control tower comes in.) We don't want to block the actor on one of these messages, so instead use an MPI_Iprobe to probe for a message and if one has arrived you can then receive it and handle it.
- Create your initial actors, this is done in the master part of the ***main*** C function or ***entryPoint*** Fortran subroutine (just before the calls to ***masterPoll***) you should create the airspace, control tower and runway actors. You should also create the appropriate number of initial airspace actors (the INITIAL_AIRCRAFT) definition determines this number.

Almost there! But we have looked over one very important thing before we run the code – that is termination. Termination can be tricky with the actor pattern as if an actor needs a response from another actor (for instance an aircraft requesting landing permission from the control tower) but that target actor has already deactivated then it will likely result in deadlock. To get termination right here there are a number of things we need to do. Initially, to test the steps above you can pop in an *MPI_Abort()* call, but that isn't ideal and we can terminate cleanly with some extra steps.

- When it comes to termination, in our initial implementation in the ***runway*** actor was a bit too simple. If you look back at this routine you can see that it loops continually, without any possibility to exit the while loop. Instead, before issuing the blocking receive, pop in an

MPI_Iprobe so that it only calls into the blocking receive (MPI_Recv) if there is a message outstanding. After this point you can use process pool's ***shouldWorkerStop*** call to determine whether the process pool has terminated, and if so break out of the while loop. Therefore the actor will continually be checking whether the worker should stop (i.e. one of the other actors has determined the code should terminate and shut down the process pool.)

- In the airspace actor it is not enough to simply quit out of the actor when the number of days is reached. This is because other messages for this actor (such as statistics) might be in flight and need to be received. In the skeleton code, when we determine that the maximum number of days is reached the ***finish_actor*** variable is set to 1. If you have not already, you should send a *FINISH* message to the control tower actor when the maximum number of days is reached. Then, only when ***finish_actor*** is 1, ***outstanding_pass_over*** is 0 and ***outstanding_retrieve_stats*** is 0 should the actor break out of the while loop.
- In the control tower it is not enough to quit out of the actor as soon as the airspace actor sends it the *FINISH* message. Again, this is because interaction with other actors (i.e. the aircraft) could be in process and these need to complete properly to avoid deadlock. The ***finish_actor*** is set to 1 when the *FINISH* message is received from the airspace and for termination ***finish_actor*** should be 1, ***runway_busy*** should be 0 and ***current_aircraft_number*** also 0. As well as breaking out of the while loop, we suggest that the control tower should be the actor that calls ***shutdownPool*** which will instruct the process pool to shut down.

That should be it, compile and submit your code. You will need at-least as many workers as there are possible numbers of actors (plus one for the process pool master.) Based on the settings in the skeleton code 48 cores (2 nodes of ARCHER) will be sufficient. You can increase this and change the maximum number of active aircraft being handled by the control tower to run on more cores when you are happy it's all working.

5 Advanced exercise – running out of fuel!

You will notice in the skeleton code there is a CRASHED_NO_FUEL declaration which we have ignored up until this point. Extend the model such that when an aircraft actor is created (either as part of the initial aircraft actors when the code starts up or by the airspace actor) a random amount of fuel is provided to this actor. This can be a random integer, for instance between 1 and 10. You will need to extend the size of the data being sent by the parent to a new actor when it's created. This will hold both the actor type, and also this extra metadata. As an aircraft actor waits for permission from the control tower to land (i.e. a PERMISSION_GRANTED response), the fuel should tick down (i.e. reduce by 1 every second) until it reaches zero and at this point the CRASHED_NO_FUEL message should be sent to the control tower instead and the actor terminate. The control tower should also keep track of the number of aircraft which have run out of fuel and return this as an additional statistic to the airspace actor.

6 Advanced exercise – more complex runway interaction

So far we have assumed just one runway and all landings are successful (the runway sends back the message LANDED_SUCCESS to the aircraft actor.) Let's extend this, which will make both the logic and actor interaction more complex.

- Pop in multiple runways, these will still be created when the program starts up and run for the duration of the code. But, for instance, instead of having one runway create three runway actors. The control tower will need to keep track of these multiple runways, allocate

aircraft actors to them and send back some meta data to the aircraft, along with the PERMISSION_GRANTED message to denote which runway that aircraft actor should interact with.

- Instead of assuming each aircraft lands successful and sending back the LANDED_SUCCESS message regardless to the aircraft actor, the runway actor should calculate a probability and send back a crashed message (e.g. 1 in 10) to the actor instead in certain circumstances. The aircraft actor should then immediately terminate. At the same time the runway actor should send a crashed message to the control tower actor, informing it that the aircraft has crashed and-so the control tower actor can update the state of the runway. The number of crashed aircrafts should also be tracked by the control tower actor and sent back to the airspace actor as part of the statistics.
- When an aircraft has crashed on a runway, that runway is then unavailable for a certain amount of time (e.g. a day, 10 seconds using the default settings of the skeleton code.) Therefore, once the control tower actor is informed by a runway actor that an aircraft has crashed it should keep track of this status. Here is the tricky bit, when an aircraft requests to land, if the only runway not busy (e.g. an aircraft is not landing on it) is unavailable due to a crash, then the control tower should message the runway to determine whether it is yet available or not. If so, then the status of the runway is updated in the control tower and permission granted for the aircraft actor to land. Otherwise the permission refused message sent back to the aircraft actor. This is tricky because sending the message back to the aircraft actor requires another step (interacting with the runway actor) and the control tower doesn't want to block waiting for the response from the runway actor to send the message back to the aircraft actor. Instead it will need to somehow store the fact that this aircraft permission message is pending, send a message to the runway actor and *at some point* in the future, when the status is sent back from the runway actor, the control tower then correlates this with the outstanding return message to the aircraft actor and sends that back. This interaction pattern is illustrated in figure 2.

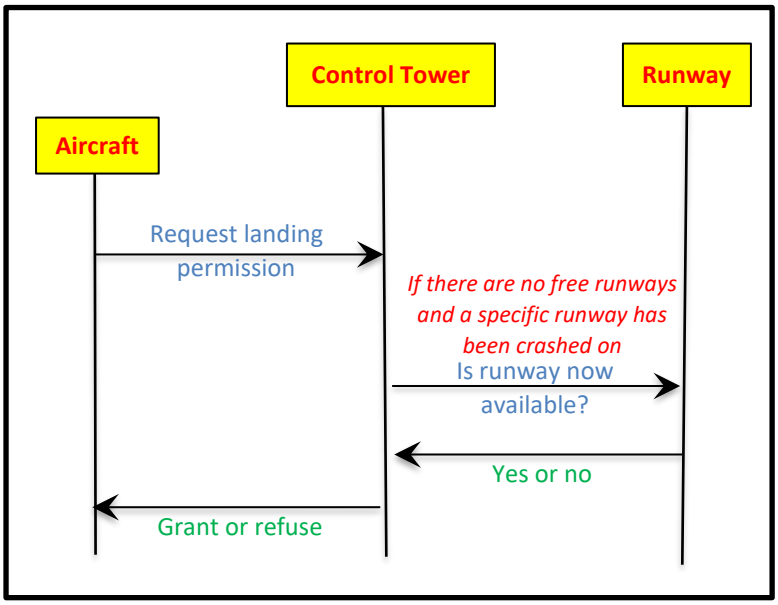

Figure 2: Interaction between actors if there are no free runways and a specific runway has been crashed upon and is unavailable