Use of AKKA Streams To Inform Forward Simulation in Robot Control

Peter Böhm[1] and Surya P. N. Singh[2]
[1]ACME University, Australia
[2]Robotics Design Lab, The University of Queensland, Australia
peter@nextlogic.biz, spns@uq.edu.au

Abstract

We investigate the use of AKKA Streams to parallelize robot simulation. This allows for parallel processing of data streams generated by multiple sensors on multiple robots in situations where it is not possible to pause the simulation during calculations and network communication. At the same time, it encapsulates entire simulation into a self-contained set of processes. This, in turn, allows for multiple cases to be running in parallel, which may be beneficial for problems requiring forward simulation. This is seen in stochastic multirobot optimization and control problems solved using reinforced learning (RL).

1 Introduction

Typically, the reinforcement learning works with a simulator. The simulator provides observations, and then a learning algorithm has a policy which chooses an action. The simulator then processes the action and returns a reward. This simple time loop repeats until the end of the episode at which point there are more calculations (e.g. the policy is updated) and everything starts again. Because the simulator is on hold during the calculations and data transfer, there is no latency to be considered, the calculations use accurate state observations of the environment updated after every turn. When dealing with multiple agents, they are all updated at the same moment before the simulation resumes, and as such there is no delay and no inter-dependencies on the order of the agents' actions. This is shown in Figure 1.

There are many high fidelity simulators that provide accurate modeling of physics, such as AirSim [Shah *et al.*, 2017], Gazebo and OpenAI Gym. The training of the RL models is, in most cases, dependent on these assumptions of no latency and data accuracy. Situations in which it is not possible to stop the simulation during calculations and those that deal with remote agents

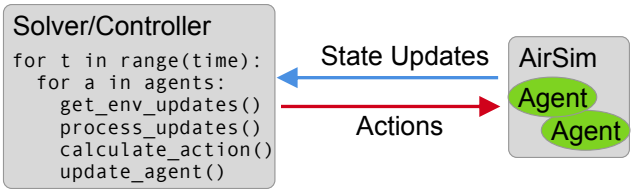


Figure 1: Example of a simple solver interacting with the simulator.

(e.g. using a cloud GPU) require a different approach. Because both calculations and network communication involve certain latency, dealing with multiple sensors attached to multiple agents can result in material delays between receiving and processing the data and delivering the actions back to the agents. This is demonstrated in Table 1 and Table 2. The images represent trajectories generated by pursue flight. The blue color represents the evader and the red color represents the evader. Trajectories in all of the images are using the same algorithm. The first image in Table 1 represents the ground truth. After every step, the simulator was paused for location updates, calculations and communication of actions back to the agents. The rest of the images show results without pausing the simulator which caused various degrees of latency, depending on the level of parallelization. Table 2 shows performance of the same algorithm using network connection with higher and much more variable latency.

While it is possible to reduce the communication delay by running everything on a single machine and thus eliminating the expensive network communication and reduce the calculation delay but simply using a more powerful machine, this approach may not scale sufficiently and it may not even be possible if external hardware is in the loop. E.g. the AirSim alone requires a modern GPU, significant amount of RAM and at least 8 core CPU. Running more computationally expensive tasks such as image recognition to process the camera feed or deep learning of policy gradient along side the

rendering engine is not possible without compromising the stability of the entire system.

We propose a different solution using parallelization. Instead of handling the tasks sequentially within a single time loop, they can be processed in parallel. This means that there is a separate sub-process [1] handling updates from each sensor, separate sub-processes handling different parts of calculations and separate sub-processes handling communication of actions back to the agents. Indeed, even the processes within a single pipe-line can be parallelized (e.g. image recognition on the camera feed can be processed in parallel as soon as they are received).

Handling and processing multi-channel communication with multiple robots is a necessary stepping stone to bridging the gap between simulation and reality. *citations needed - there's plenty papers on bridging the gap*

2 Software Framework

2.1 Simulator

This text describes integration with AirSim. AirSim is a simulator for drones, cars and more, built on Unreal Engine. It is open-source, cross platform, and supports hardware-in-loop with popular flight controllers such as PX4 for physically and visually realistic simulations. It is developed as an Unreal plugin that can simply be dropped into any Unreal environment. [Shah *et al.*, 2017]

2.2 Reactive Streams

High fidelity simulators, just like real-world implementations, generate large amount of data that needs to be processed in near real-time. Each agent can generate multiple streams (e.g. location updates, camera feed, LIDAR feed, etc.) and streams from multiple agents may need to combined (e.g. in synchronized movement of multiple vehicles). Each stream may need multiple transformations (e.g. deserialization, conversion, calculations, etc.) and those transformations may need to be parallelized to increase throughput. Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols. [rea, 2019]

Back pressure

In a system without back pressure, if the subscriber is slower than the publisher, then eventually the stream will stop - either because one of the parties runs out of memory and one of its buffers overflows or because the implementation can detect this situation but cannot stop sending or accepting data until the situation is resolved (if it can be resolved at all). Although the latter scenario is somewhat more positive than the former, blocking back pressure (the capability of a system to detect when it must slow down and to do so by blocking) brings with it all of the disadvantages of a blocking system, which occupies resources such as thread and memory usage. [Bernhardt, 2016]

[1] No specific implementation is implied here. The process is not tied to the specific processor core or thread.

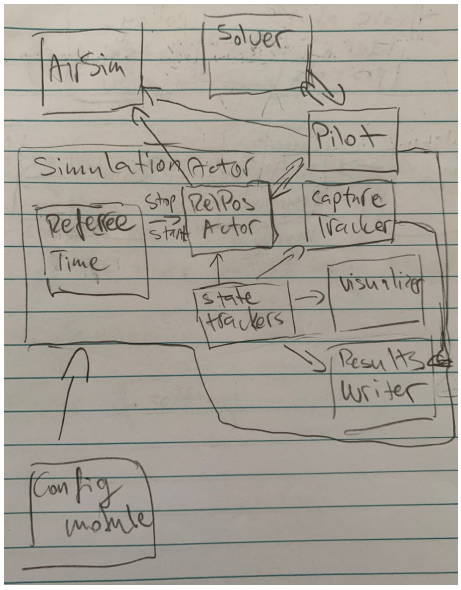


Figure 2: System design without streams.

2.3 DSL

Besides technical reasons, there is also additional motivation for the reactive stream. They provide a DSL (Domain Specific Language) [Fowler, 2010] to define the stream transformation graphs in a expressive way.

2.4 Actor Model

Something about concurrency issues and how actor model addresses them? Need for new programming model (something like Akka Docs intro). Routing

2.5 Message Broker

For situation where the solver cannot be implemented in the same program (e.g. because it is implemented in a different programming language) or on the same machine (e.g. because it requires extra resources such as GPU), it is necessary to implement remote messaging.

One way is to use some kind of RPC (Remote Procedure Call) solution. There are several available: AirSim uses MessagePack RPC (Remote Procedure Call), Google uses gRPC for its services, and there are several others. Another option is to use a message broker. The difference is that while RPC calls the remote service directly, with message broker, the services publish and consume messages via the message broker.

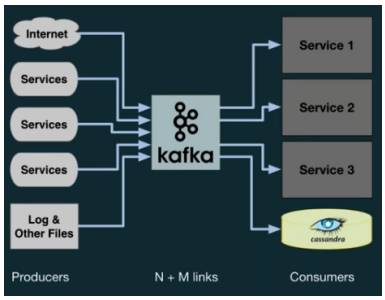


Figure 3: Message broker handles .

Kafka was developed to collect and distribute massive messages for distributed systems. It is a high-throughput distributed messaging system. Kafka can always keep stable performance even if it processes millions of messages per second. [Wang *et al.*, 2015]

Figure 3 shows how such communication works. The raw (or pre-processed) data is published to Kafka (e.g. one topic for location updates, one topic for LIDAR, etc.) and consumed by services that process these data. The results are then published to a different topic. This simplifies dependencies between services, reduces data loss when a service crashes, provides a simplicity of one "API" for communication.

3 Implementation

Each simulation runs within its own parent actor `SimulationRunnerActor`. Based on the settings provided, this actor sets up all the auxiliary services (e.g. time keeper, visualizer, capture detector) as child actors. It also sets up an Akka Stream for each agent and each tracked sensor and location updates. The simulation is terminated either by the `TimeKeeperActor` when the time has completed or if some termination event is detected (e.g. capture detected by `CaptureDetectorActor`). At this point a kill switch is activated on the streams and the `SimulationRunnerActor` receives a poison pill which stops all the child actors and shuts down the simulation.

Running the entire simulation within the context of a single parent actor allows multiple simulations to run in parallel.

3.1 Communication with AirSim

AirSim uses MessagePack [Furuhashi, 2014] for communication. Some of the methods calls are non-blocking (e.g. methods for controlling the vehicles such as `moveByVelocityZ`), however, methods returning vehicle's state, camera feed, LIDAR feed are blocking. The return time of the blocking methods is especially important when using a cloud GPU to run AirSim. Depending on the network connection and type (e.g. VPN), the latency of `getMultirotorState` (a method that returns

```
class AirSimClientActor() extends Actor {
  ...

  def startedReceive(client: Client): Receive = {
    case GetMultirotorState(vehicleName) =>
      sender ! client.callApply(
        "getMultirotorState", Array(vehicleName)
      ).toString

    case GetCameraImage(vehName, camName, imageType) =>
      val image = client.callApply(
        "simGetImage", Array[AnyRef](camName, imageType, vehName)
      ).toString
      sender ! (if (isValidImage(image)) Some(image) else None)

      ...
  }
}

val airSimClientRef = system.actorOf(Props[AirSimClientActor])

val getStateFlow =
  Flow[String]
    .map(name => GetMultirotorState(name))
    .ask[String](parallelism = 4)(airSimClientRef)
```

Figure 4: Flow transforming vehicle name to multi-rotor state string.

position, orientation, velocity and acceleration of the vehicle) can be as high as 100ms.

Because multiple components need to interact with AirSim and the interaction is stateless (the only state is the open connection, but each connection can handle any commands from any component), it is a good use-case for Akka routing. Akka Router creates a pool of actors and each of them opens a connection at the start of the simulation. If a component (e.g. position tracker) requires location update, it passes a message to the router. The router then routes the message to the actor. Once the actor receives a response from AirSim, it returns it to the component in the form of message. Because most of the components are Akka Streams and not actors, this call is encapsulated in a `Flow` using the ask pattern.

A simplified code is shown in Figure 4. `AirSimClientActor` receives messages such as `GetMultirotorState` or `GetCameraImage`, interacts with the AirSim and returns the result to the sender of the message. Flow `getStateFlow` uses `ask` pattern to interact with the actor. Parallelism argument is related to back pressure. It allows these calls to be parallelized (e.g. if some of the responses take longer to process than the query interval).

3.2 Akka Streams

Figure 5 shows implementation using Akka Streams. Each data flow is encapsulated in the stream. The incoming data is received by a merge/broadcast component that provides a publisher/subscriber functionality. The downstream components subscribe for the input they need (this is transparent to the data producers). Solvers are a special category, because they do not behave as sinks, they behave as flows (i.e. transformations). Solvers transform the input into the output in

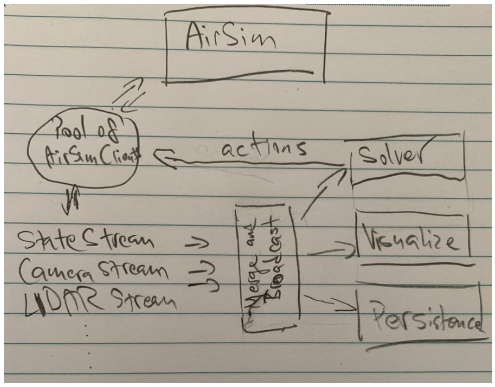


Figure 5: Implementation using Akka Streams.

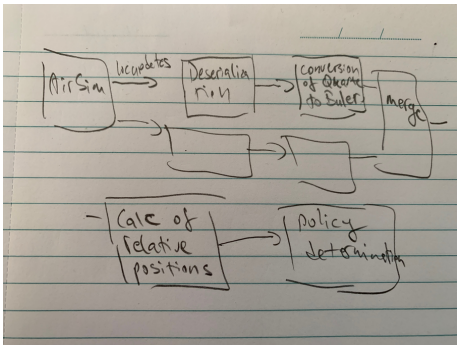


Figure 6: Graph handling location updates stream. Data is received from AirSim, deserialized, converted and fed into the solver that determines the next action.

the form of actions and these actions and then routed through the `AirSimClientActor` back to the AirSim.

3.3 Kafka

Implementation using Kafka is somewhat similar to the above solution but instead of built-in solver, the streams are terminated in Kafka. Figure 7 shows such implementation. These messages are then received by the solver (or any other program), e.g. in Python using `KafkaConsumer`. The solver generates action and this action is then posted back to Kafka on a different topic. Back in the middleware the actions are consumed and using the `AirSimClientActor` routed back to AirSim.

3.4 Persistence

The main body of the text immediately follows the abstract. Use 10-point type in a clear, readable font with

```
val producerSettings = ProducerSettings(
  system, new StringSerializer, new StringSerializer
)
val done: Future[Done] =
  Source(1 to 100)
    .throttle(1, 100.millis)
    .map(value => new ProducerRecord[String, String]("test", s"msg $value"))
    .runWith(Producer.plainSink(producerSettings))
```

Figure 7: Serializing Akka Stream to Kafka.

1-point leading (10 on 11). For reasons of uniformity, use Computer Modern font if possible. If Computer Modern is unavailable, Times Roman is preferred.

Indent when starting a new paragraph, except after major headings.

4 Experiment

The same algorithm was tested in 5 different setups - sequential model with simulator paused for calculations and communication (this represents the ground truth), sequential model with simulator running through out, partially parallelized model using futures to encapsulate communication/calculations for each agent at every time step, fully parallelized model using Akka Streams and finally fully parallelized model using Akka Streams together with Kafka message broker and calculations run on a different machine.

A fixed algorithm was chosen as a policy to avoid side-effects that could be introduced by stochastic nature of RL training *this could back-fire by leading back to the magic RL that can compensate for everything.* The algorithm was originally created for simulation of the differential games [Redulla and Singh, 2018]. The pursuer is faster, moving with velocity of 10m/s, but has a larger turning radius of 8m. The evader is slower at velocity of 5m/s, but is more agile and can make turns of any angle. The algorithm only requires position of both players and their headings[2], however, additional call to read the LIDAR data was added to approximate the effects of multiple sensor readings per agent.

Five observations were measured: recency of location data of the player and opponent, time since the last move by the player and opponent and error of the steering angle θ resulting from stale location data. Because the errors are compounding, the final trajectory is also compared with the ground truth.

4.1 Sequential model with simulator paused between turns

This represents the ground truth as the paused simulator serves as a synchronization layer that shields the algorithm from both stale data and delays in actions. The simulator starts paused. The measurements are taken, and the actions calculated and sent to the agents. After that, the simulator is run for 100ms. Then the process repeats. Because the communication with the simulator happens in the paused state it can be done in a blocking manner and the calculations start only once all the requests have finished.

[2]The location data for each player could be shared between players to save the round trip, however, we are simulating behavior of independent actors.

4.2 Sequential model with simulator running

This is the same as the previous model, but without the paused simulator acting as the synchronization layer. Because all commands run in sequence, the next request can only be sent once the previous one completes and the calculations can only start once all the requests finished. This creates a varying level of staleness among the received data that enter the calculations. It also increases the time lag between actions being sent to the agents. With the pursuer moving at 10m/s, even 500ms creates a difference of 5m.

4.3 Partially parallelized model

This model represents a simple (if short-sighted) solution to the concurrency issue. Since each agent is treated as a separate entity, at each timestamp, they are wrapped in their individual future. This means that while all communication and calculations for a single agent runs sequentially, the agents move in parallel. *maybe a diagram?*

4.4 Fully parallelized model using Akka Streams

In this model, not only the agents move in parallel, but each operation within each agent is parallelized as well. There are no inter-process dependencies: the location is updated independently, the calculations use whatever latest data available and once the new action is calculated, it is sent to the agent. The whole process moves in the interleaved manner.

4.5 Fully parallelized model with Kafka

This model builds on the previous one and adds on a message broker to allow for out of process and out of machine communication.

5 Results

6 Conclusions

Acknowledgments

The preparation of these instructions and the LaTEX and BibTEX files that implement them was supported by Schlumberger Palo Alto Research, AT&T Bell Laboratories, and Morgan Kaufmann Publishers.

References

[Bernhardt, 2016] Manuel Bernhardt. *Reactive Web Applications: Covers Play, Akka, and Reactive Streams*. Manning Publications Co., 2016.

[Fowler, 2010] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[Furuhashi, 2014] S Furuhashi. Messagepack.(2017). *URL http://msgpack. org/. Accessed*, pages 02–21, 2014.

[rea, 2019] 2019.

[Redulla and Singh, 2018] Anne Redulla and Surya PN Singh. Simulating differential games with improved fidelity to better inform cooperative & adversarial two vehicle uav flight. In *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pages 130–136. IEEE, 2018.

[Shah *et al.*, 2017] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017.

[Wang *et al.*, 2015] Zhenghe Wang, Wei Dai, Feng Wang, Hui Deng, Shoulin Wei, Xiaoli Zhang, and Bo Liang. Kafka and its using in high-throughput and reliable message distribution. In *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*, pages 117–120. IEEE, 2015.

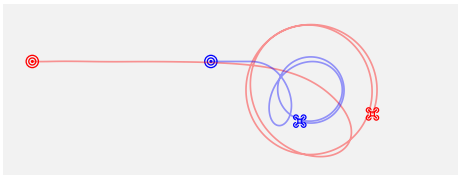
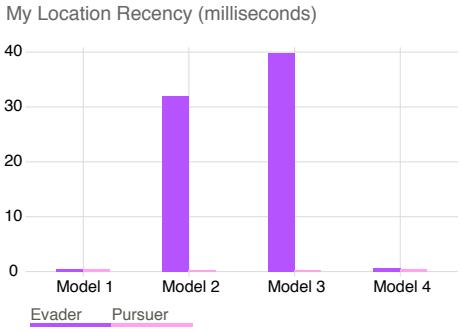
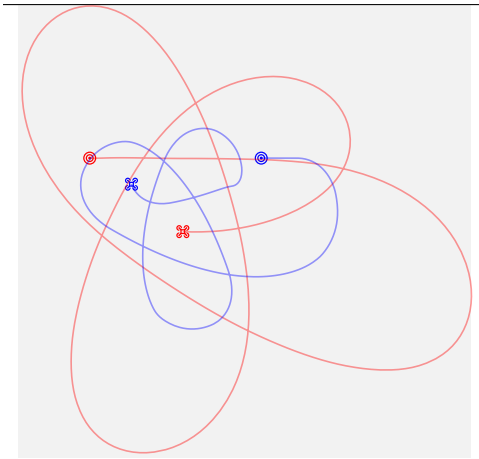
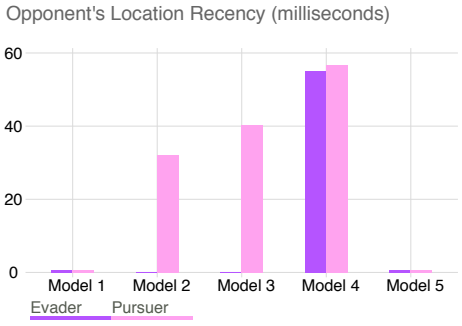
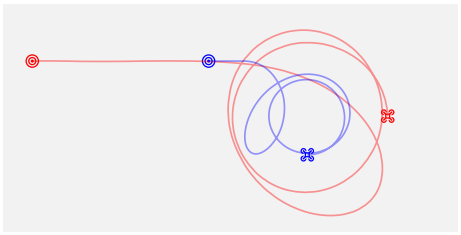
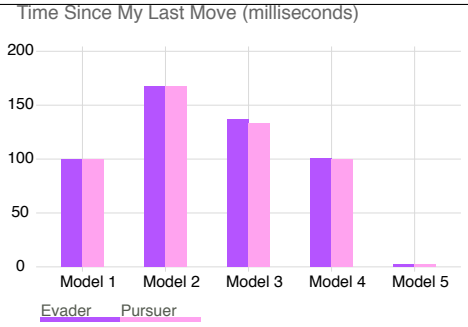
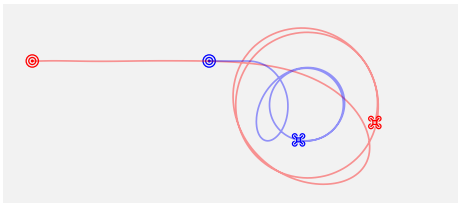
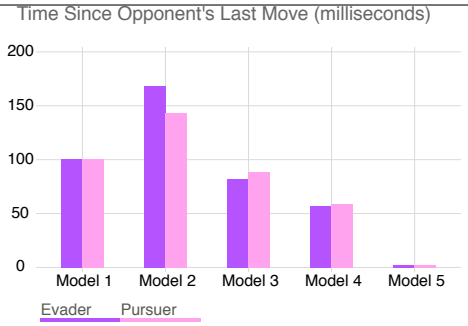
	Model 01: Sequential model with AirSim paused for calculations and communication.	
	Model 02: Sequential model with AirSim running throughout.	
	Model 03: Partially parallelized model through Futures.	
	Model 04: Fully parallelized model using Akka Streams.	

Table 1: Results of running the models over a reliable network with latency 30ms.

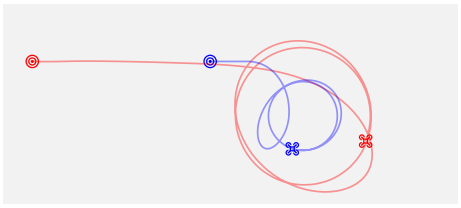
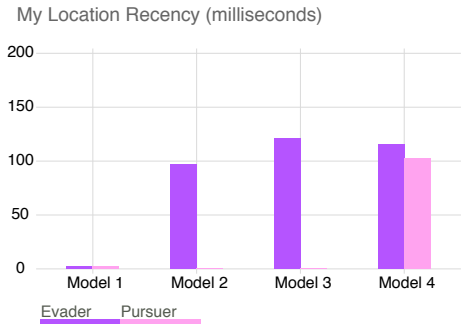
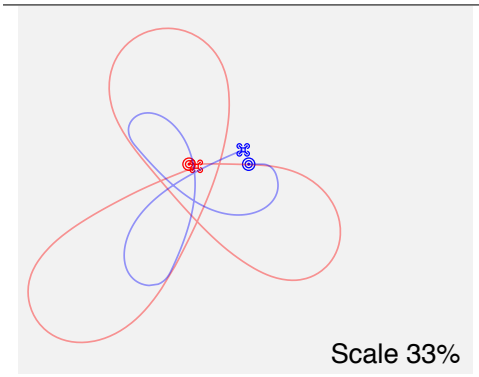
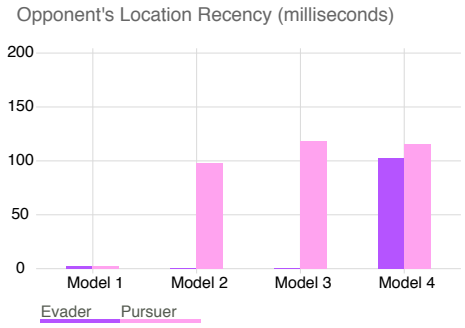
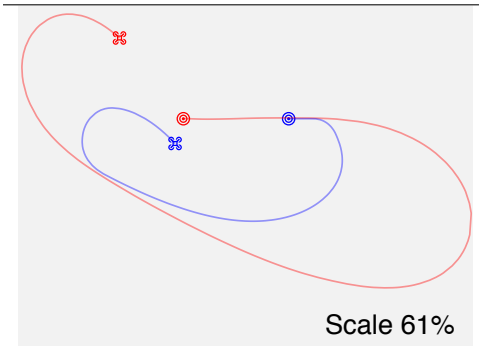
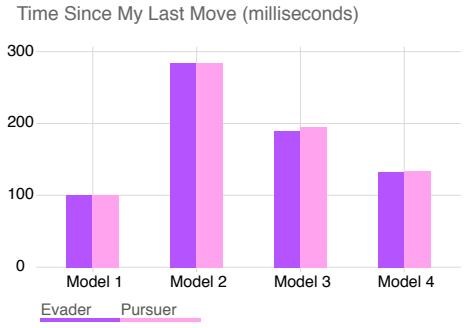
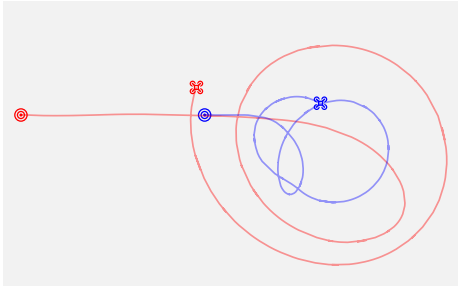
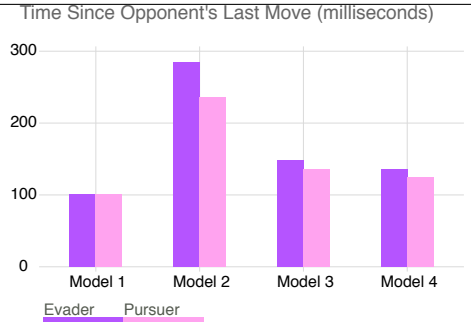
	Model 01: Sequential model with AirSim paused for calculations and communication.	
	Model 02: Sequential model with AirSim running throughout.	
	Model 03: Partially parallelized model through Futures.	
	Model 04: Fully parallelized model using Akka Streams.	

Table 2: Results of running the models over a VPN with latency fluctuating between 30ms and 120ms (average around 70ms).