A Parallelized & Asynchronous Forward Simulation Framework to Inform Multi-Robot UAV Coordination via Akka Streams

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

Abstract

The essence of many approaches for solving and optimizing stochastic control process, such as those that arise in multi-robot coordinated operations, is to consider many candidate policies. For many dynamic robots this in turn requires a huge number of forward simulations, especially when the model is complex.

We investigate the use of Akka Streams to parallelize robot simulation. This allows for processing of data streams generated by multiple sensors on multiple robots in situations where it is not possible to pause the simulation during calculations or (network) communication. Their use helps to mitigate the effects of network and calculations latency and provides an expressive DSL (Domain Specific Language) to define the stream transformation graphs. It also helps reduce coordination errors and improve overall distributed simulation stability.

1 Introduction

From a squadron of flying drones to a school of submarines exploring together, designing controllers so as to enable automatic interactions between independent (autonomous) robotic vehicles is of interest in many application domains. A strategy for automatically constructing these controllers, or more generally their control policies, is to frame the problem as a Markov Decision Process (MDP) [Howard, 1960]. The essence of this stochastic control approach is a forward adaptive search in which many candidate policies (each of which are themselves a set of control actions to take over the a state-space) are evaluated – via a forward model – to find the expected value of this possible policy for the task. This is then used to construct the next subsequent policy.

For many complex robots this forward model is too complex to express as a closed-form analytic (linear) dynamical systems and is this evaluated for each step of each case via simulation. Each robot in a typical dynamical case can require millions to tens of millions of simulation evaluations [Haarnoja *et al.*, 2018]. Moreover, this would need to be repeated for every robot in the system. Indeed, the underlying problem, a decentralized stochastic game [Redulla and Singh, 2018], is a class of decentralized Markov decision process (DEC-MDP) which has been shown to be NEXP-complete and NP-hard [Papadimitriou and Tsitsiklis, 1986; Bernstein *et al.*, 2002].

There are many high fidelity simulators that provide accurate modeling of physics, such as AirSim [Shah *et al.*, 2017], V-REP [Rohmer *et al.*, 2013], and MuJoCo [Todorov *et al.*, 2012] (which is the engine often used with simulation systems such as OpenAI Gym [Brockman *et al.*, 2016] and the DeepMind Control Suite [Tassa *et al.*, 2018]). The issue is that the number of simulations needed to train one robot (leave along multiple robots interacting which is an entire complexity class harder) is huge and can be on order of 200 million evaluations. With the availability of affordable cloud computing services, this might seem to motivate the use of a distributed large-scale, high-fidelity simulation; however, this introduces the issue that all this has to be synchronized.

That is, the training of the MDP and simulation models in most cases is 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 (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 Figure 4 and Figure 5. 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 Figure 4 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. Figure 5 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 by 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. For example, AirSim requires a modern GPU, significant amount of RAM and at least 8-core CPU [Shah *et al.*, 2017]. 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 real-time (asynchronous, non-blocking) parallelization. Instead of handling the tasks sequentially within a single time loop, they can be processed via separate sub-process handling updates from each sensor, different parts of calculations, and communication of actions back to the agents. Note, no specific implementation is implied here. The process is not tied to the specific processor core or thread. Indeed, even the processes within a single pipeline can be parallelized (e.g. image recognition on the camera feed can be processed in parallel as soon as the data are received so the next image can start processing before the previous has completed). Handling and processing multi-channel communication with multiple robots is a necessary stepping stone to bridging the gap between simulation and reality [Loquercio *et al.*, 2019; James *et al.*, 2019].

2 Software Framework

As noted, stochastic dynamic control processes, such as those in multi-vehicle Unmanned Aerial Vehicle (UAV) robot coordination is a MDP. As the the transition model/dynamics and/or reward function are not always known, this problem is often treated reinforcement learning (RL) problem [Sutton and Barto, 2018]. This has recently been extended to continuous control domains [Henderson *et al.*, 2018] via algorithms such as the Deep Deterministic Policy Gradient (DDPG) [Lillicrap *et al.*, 2015] and Soft Actor-Critic (SAC) [Haarnoja *et al.*, 2018].

RL methods work in tandem with a simulator that provides observations, which is then used by the learning algorithm to shape the policy that 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.

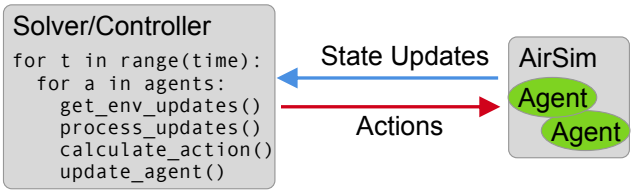

Figure 1: Example of a simple solver interacting with the simulator.

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 Akka and Actor Model

Actor abstraction is a parallel programming model based on actors and messages. Immutable messages are passed between actors asynchronously and delivered into the recipient's mailbox. Each message is then completed in a single threaded fashion. This is also true for the internal state of the actor - it can only be modified and accessed by the actor [Lawlor and Walsh, 2016]. Actor systems easily express a wide range of computational paradigms, and provide a natural extension of programming into concurrent (parallel) systems. This naturalness and ease of expression means that programs to solve complex problems do not add even more complexity of their own. This direct relationship of the code to the problem domain also makes it easier to optimize algorithms based on knowledge of that domain [Agha *et al.*, 1997].

Communication with AirSim

AirSim uses MessagePack [Furuhashi, 2014] for communication. MessagePack is an RPC (Remote Procedure Call) implementation that provides fast and efficient

communication between components written in different languages. At the time of writing, the last update of the JVM implementation was done 7 years ago[1] and it also depends on a no longer supported version of MessagePack serializer[2]. The RPC implementation doesn't provide any error handling which makes it impossible to properly handle errors returned by AirSim. This implementation cannot be used for camera feed because the images returned by `simGetImage` call exceed the allowed value size. Lastly, mapping between its Value types and native Scala types is either missing or is problematic and slow.

As such the a custom RPC implementation using a routing pool of Akka TCP clients was used. The solver uses ask messages to to send commands and receive responses in a non-blocking manner. A Scala msgpack4s[3] library was used for serialization/de-serialization.

2.3 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]

2.4 Domain Specific Language (DSL)

Besides technical reasons, there is also additional motivation for the reactive streams. They provide a DSL (Domain Specific Language) [Fowler, 2010] to define the stream transformation graphs in a expressive way. In a few lines of code it's very easy to express how the data sources are transformed and how data stream flows between the components and where it's terminated.

This is illustrated in Figure 2. Location streams for both actors are merged together using a merge component and then fed into the relative distance calculator. The result is then broadcasted into 2 parallel streams, one for each agent. Based on the relative position, the solver calculates actions in the form of steering thetas. Each steering theta is then broadcasted into 3 parallel streams - the first one to update the AirSim, second to feed the updated theta back to the solver and the last one to persist the steering decision data.

3 Experiments

The same algorithm was tested in 4 different setups - sequential model with simulator paused for calculations and communication (this represents the ground truth), sequential model with simulator running throughout, partially parallelized model using futures to encapsulate communication/calculations for each agent at every time step, and fully parallelized model using Akka Streams. Each model was run under 2 different network setups[4]. The first set of experiments was using a direct internet connection which provides stable average latency of 36ms. The second set was run over VPN with average latency of 65ms, fluctuating between 20ms and 200ms (Figure 3 shows the latency details).

A fixed algorithm was chosen as a policy to avoid side-effects that could be introduced by stochastic nature of RL training. 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[5].

Four observations were measured: recency of location

[1] https://github.com/msgpack-rpc/msgpack-rpc-java

[2] Version 0.6.6 of MessagePack for Java (https://github.com/msgpack/msgpack-java)

[3] https://github.com/velvia/msgpack4s

[4] The simulator is running on a GPU on GCP (Google Cloud Platform) located in Sydney (zone australia-southeast1) while the solver was located in Brisbane (about 1000km away).

[5] The location data for each player is shared in every implementation to reduce the number of requests because high number of concurrent requests tends to overload AirSim.

```
eLocationsE ~> merge
pLocationsE ~> merge

merge ~> relativeDistanceFlow(eRelPositionActor) ~>
  broadcastRelDistance ~> calculateEvadeTheta ~> eBroadcast ~> evadeAirSim(airSimPoolMaster)
                                                 eBroadcast ~> updateTheta(Constants.e, eRelPositionActor)
                                                 eBroadcast  ~> eSaveSteeringDecision
  broadcastRelDistance ~> calculatePursueTheta ~> pBroadcast ~> pursueAirSim(airSimPoolMaster)
                                                  pBroadcast ~> updateTheta(Constants.p, eRelPositionActor)
                                                  pBroadcast ~> pSaveSteeringDecision
```

Figure 2: The graph controlling the simulation expressed using the Akka Streams DSL. Symbol ~> stands for "feeds into".

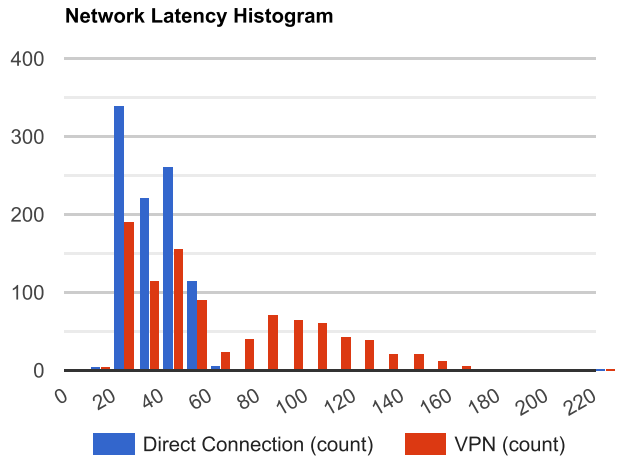

Figure 3: Histogram of the latency using direct connection and VPN.

data of the player and opponent and time since the last move by the player and opponent.

3.1 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.

3.2 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 relatively small delay of 500ms translates to a location difference of 5m.

3.3 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 code blocks for both agents are run in parallel.

3.4 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 Results

Figures 4 and 5 show the summary of the results. The first column illustrates the trajectories generated by the agents. Because this is a pursue/evade situation, the distance between the agents is an important measure. This is plotted in the charts in the second column. Charts in the third column show recency of data used for calculations and time-lag since the last move.

4.1 Direct Connection

The first image shows the ground truth with the drones orbiting after about 11s. The distance between the agents converges fast to about 8.55m.

In model 2, the delays caused by the sequential execution of the blocking calls to update location details cause much longer turning times. The effects are more visible in the pursuer's path, because it is moving faster and any delay in control will move it further off course. As a result, the distance is alternating between large and small values.

The partially parallelized model performs well, the trajectory follows the ground truth closely, albeit with a larger diameter. The distance between the agents converges to about 10m. Because both location updates run in parallel, they finish on average in 36ms and the action, sent to back to the simulator, is delayed by the same 36ms as compared to the ground truth (see the Time Since My Last Move chart). This delay causes a minor drift in agents which is what causes the slightly larger radius.

Model 4 follows the ground truth the most closely. The distance converges to 9.1 which results in a slightly more than the ground truth but less than model 3. The delays between moves are very close to 100ms just like in the ground truth model.

4.2 VPN Connection

The sequential model behaves in a similarly to the direct connection, only longer delays cause the agents to drift further away[6]. The relative distance follows the same pattern as before but with larger amplitude.

When using the direct connection and the partially-parallelized model (model 3), both blocking requests[7] (location updates for evader and pursuer) finished within the time-step window of 100ms so there were no uncompleted Futures overlapping to future iterations. Under VPN conditions, there are requests taking longer and some of Futures complete in the future iterations. This may lead to actions from older iterations overriding the newer updates and being used instead. Using older data has effects similar to increased latency. Both the trajectory and the relative distance chart are now very similar to the sequential model. The smaller delays cause smaller relative distances.

Fully parallelized model (model 4) performs closest to the ground model. There are visible distortions, however, it does converge to the expected results of orbiting agents.

The cause of the different results is the increased latency and latency fluctuations of the received data and how this latency is propagated through the rest of the system. Model 2 and model 3 do not provide any form of control or management of the latency. In model 2, the rest of the program needs to wait until the request is completed. In model 3, the slower requests don't block the faster requests, but may override them if an earlier and slower request completes on after the faster and later request. It may be tempting to counter this with e.g. lower timeout for the Future and using a sequence data structure to keep the order of Futures, but these only address some of the concerns. What is needed, is a solution able to handle the multiple variable rate data flows concurrently and provide tools, like back pressure to deal with the variable rate.

Using Akka streams, the requests run in parallel, just like in model 3, but the streams provide an order guarantee, which means that results of the requests will be emitted in the same order as they were generated. The timeout of each location update is built-in in the ask pattern and any request taking longer will be skipped transparently to the downstream consumers. The data flow needs to be managed on the other end of the pipeline as well. Too fast rate of the updates leads to crashes in AirSim. Using streams, this can be achieved through throttle and buffer strategy[8]. Using plain Futures, this may be very difficult to achieve.

5 Conclusion

Acknowledgments

We acknowledge the Microsoft AirSim open-source project for their simulation tools. We thank Anne Redulla for discussions and for her open-source simulation and differential game solvers. This research was partly supported by an Australian Research Council Discovery Project (DP160100714).

References

Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

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

Daniel S Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of markov decision processes. *Mathematics of operations research*, 27(4):819–840, 2002.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai hym. *arXiv preprint arXiv:1606.01540*, 2016.

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

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

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[6]The trajectory had to be scaled down 3x to fit display window.

[7]The action requests are non-blocking. Each subsequent action cancels the previous task and starts new command.

[8]Throttle reduces the rate at which elements are emitted and buffer strategy controls which elements are dropped.

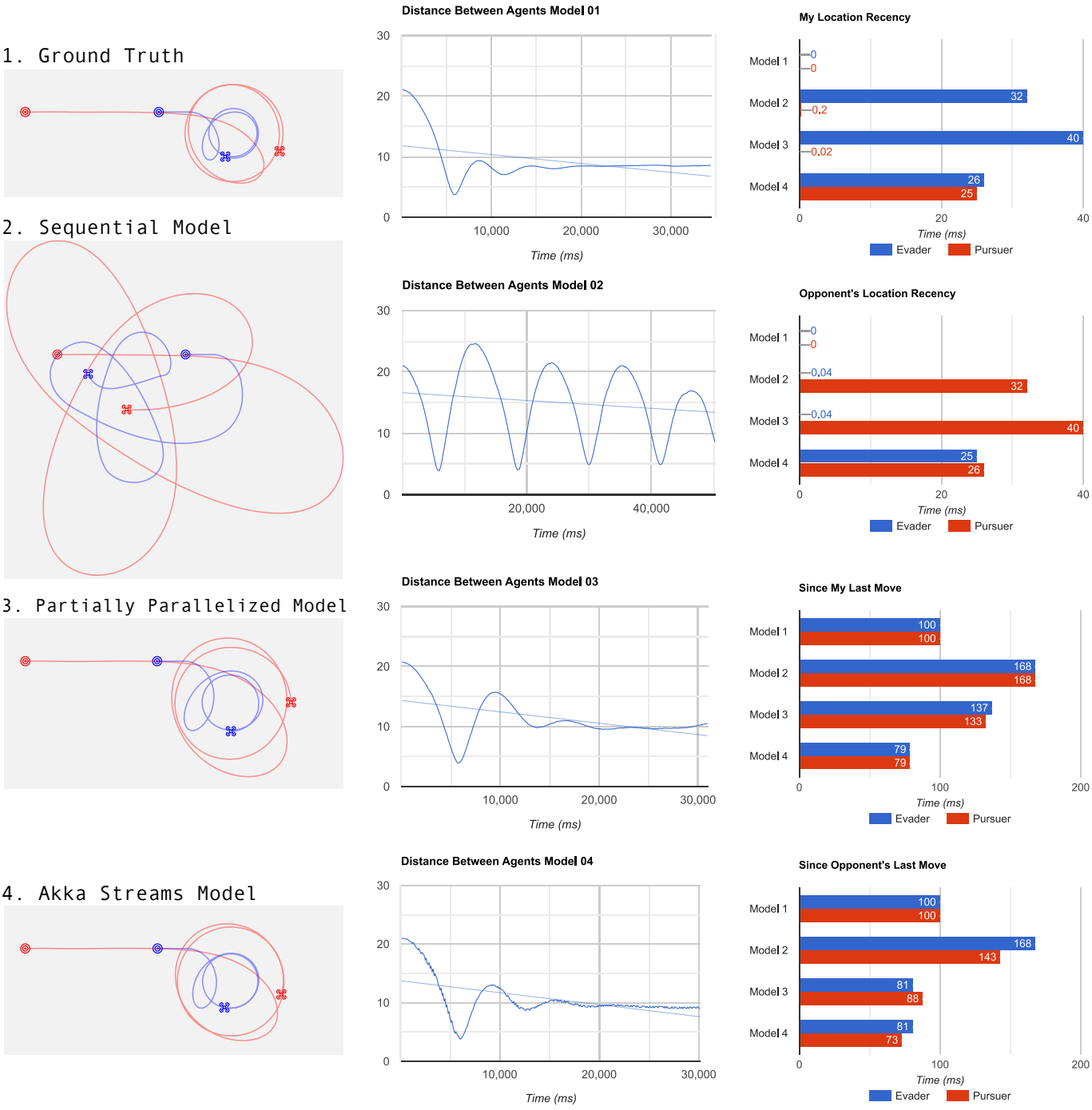

Figure 4: Results of running the models over a reliable network with latency averaging 36ms. The first column shows the trajectory followed by the agents. The second column shows the distance between the agents. The third column shows the recency of the data used for calculations and delay since the previous step.

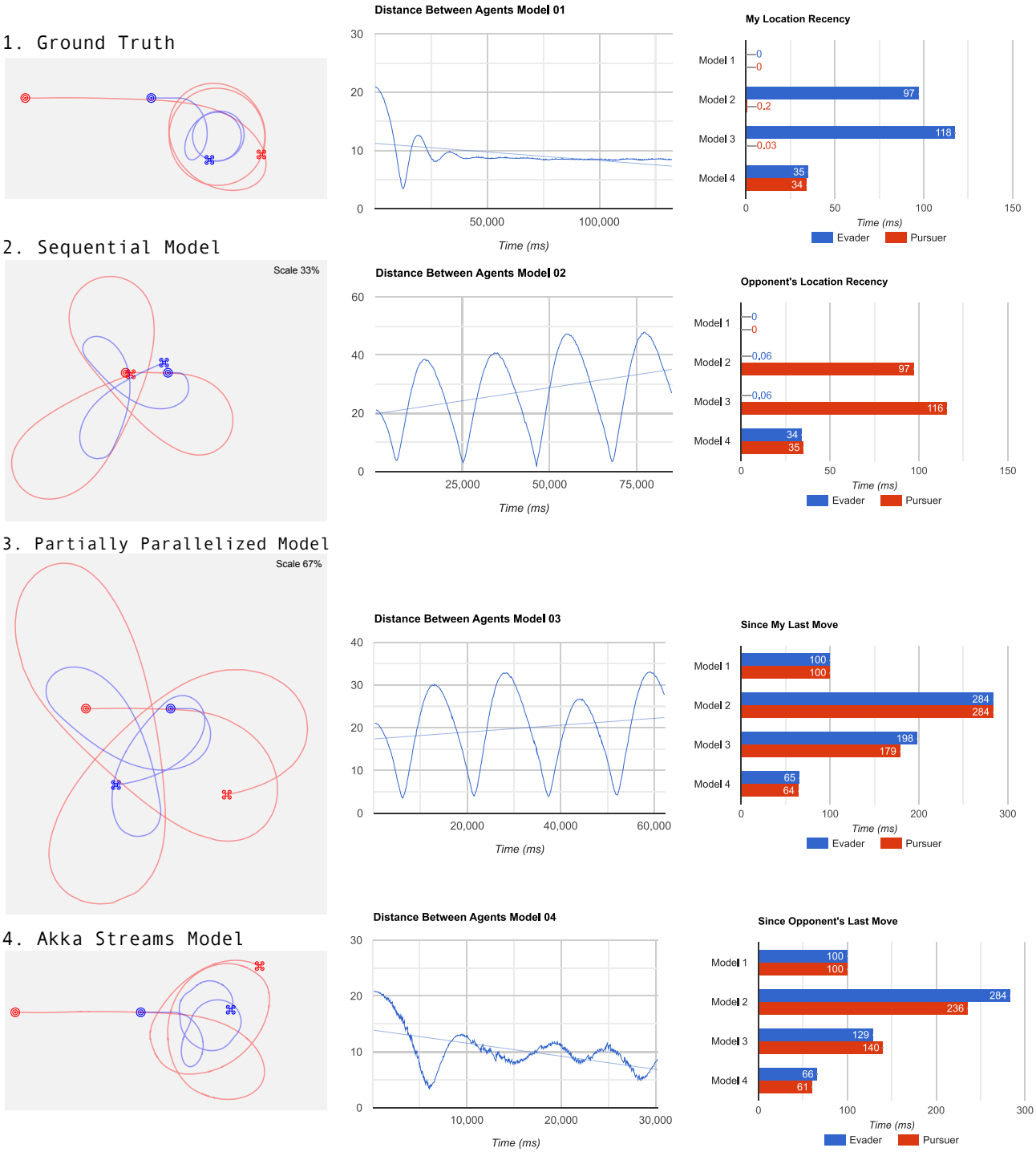

Figure 5: Results of running the models over a VPN with fluctuating latency (average of 110ms). The first column shows the trajectory followed by the agents. The second column shows the distance between the agents. The third column shows the recency of the data used for calculations and delay since the previous step.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Ronald A Howard. Dynamic programming and markov processes. 1960.

Stephen James, Paul Wohlhart, Mrinal Kalakrishnan, Dmitry Kalashnikov, Alex Irpan, Julian Ibarz, Sergey Levine, Raia Hadsell, and Konstantinos Bousmalis. Sim-to-real via sim-to-sim: Data-efficient robotic grasping via randomized-to-canonical adaptation networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 12627–12637, 2019.

Brendan Lawlor and Paul Walsh. The weekend warrior: how to build a genomic supercomputer in your spare time using streams and actors in scala. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, pages 575–580. IEEE, 2016.

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Antonio Loquercio, Elia Kaufmann, René Ranftl, Alexey Dosovitskiy, Vladlen Koltun, and Davide Scaramuzza. Deep drone racing: From simulation to reality with domain randomization. *arXiv preprint arXiv:1905.09727*, 2019.

Christos H Papadimitriou and John Tsitsiklis. Intractable problems in control theory. *SIAM journal on control and optimization*, 24(4):639–654, 1986.

2019.

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.

Eric Rohmer, Surya P. N. Singh, and Marc Freese. V-REP: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326. IEEE, 2013.

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.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning series)*. A Bradford Book, second edition, 2018.

Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.