Gregory Pavlov

Computer Science

Advisor: Prof. Robert Signorile

Intelligent Entities in a Distributed Simulation Environment

Abstract

In addressing the growing model sizes used in simulation environments, I examine adding machine learning techniques to the entities in a model in an effort to produce such side effects as emergent behavior. Distributing the environment in order to increase the efficiency of the simulation also plays an important role in this thesis. The added intelligence of entities may have some affect upon the speed at which a model may be executed, as additional computation will be required. Taking a model-based approach, I attempt to solve some of the problems of interaction between components in a distributed simulation.

Introduction

Simulation is the product of science attempting to model the real world as accurately as possible in a virtual world. The goal being to make predictions regarding the result of a given decision or choice, and then see the effects upon the created realm, as the first computers were used to compute artillery shell trajectories (Fujimoto 7). As the required workload given to a specific program operating on a single machine began to grow such that a simulation could no longer run in real-time, breaking up the model and sharing the work among multiple machines became much more common. The design of each system used the modules to order the entities to different locations in the model, rather than the entities making a certain choice when required.

Discrete-event simulation uses modules to perform some task or function on an entity in the model upon arrival at the given module. The model for such a simulation appears similar to a workflow diagram, where some modules create entities, others delay, others decide the path a

given entity should take, while other remove entities from the model. A single event calendar exists such that an event is placed into an appropriate spot into a queue. The queue order is determined by the time an event should occur in regards to the simulation clock. An example of the queue would be the arrival of an entity into a given module at time t, the work this module should perform on this entity at time t+1, and the exit of this entity at time t+2.

In a continuous simulation, cellular automata are used in order to represent the real world in some fashion. Agents move from cell to cell at specific defined time intervals and depending upon the designed behavior of an agent. Each cell may perform functions that use agents, only operate when an agent is absent, or when a specific number of agents are present on the cell, similar to a module in a discrete-event simulation. Agents may also use the cell to perform some function, and may have such behavior to 'learn' to better perform the desired task.

The 'learning' performed by agents generally occurs due to a specific way in which an agent's behavior is defined. In such a definition, an agent will use some characteristic of the cell it is currently on as input into a function that exists as part of this agent, then based upon the result of the function move in some direction or perhaps perform another function. Agents will then update the function based upon the effect of the determined result.

The typical model for a continuous simulation would be a flocking demonstration. Here, each agent would be instructed to turn to the right or left a certain amount based upon another agent entering to within a predefined radius, say two cells. Using a simple model such as this with approximately ten agents, flocking behavior will emerge.

Distributed processing as used in simulation resulted from the ever growing representations that models provided. By breaking the model into smaller pieces that may be executed and spreading the workload out among multiple machines, the overall runtime of a given model can be decreased significantly. An example of such a system would be the NEC

Earth Simulator, when multiple processors and memories, the core of any computer, are used to simulate the Earth's climate.

Simulation's constant goal remains the decrease in runtime of models with the highest possible granularity. According to Ferscha, regardless of the granularity the runtime of the model may be affected by the objective or nature of the simulation (4). A model may not be suited for the simulation environment in which it has been created, and thus may experience a longer runtime.

One of the largest problems encountering simulation is granularity, or how specific and detailed to make a modeling environment. If an environment consists of small granules, then models created for this environment can be extremely specific, creating accurate results. However, a small granularity also means that the runtime of the simulation increases, as more processing is required. A large granule yields the opposite results, a quick runtime, but more vague results. Thus the granularity of a simulation environment remains a continuous issue.

The increasing size of models also remains an issue in the forefront. Distributed simulation attempts to solve this problem by breaking up the model and using the processing potential of multiple machines over a network, possibly the Internet. The overhead associated with this distribution may outweigh the possible gain, as the delay associated with sending information between machines may increase. Minimizing the amount of information sent between computers thus helps to increase the runtime.

In the past, running a simulation on computers from different manufactures could lead to different results. With the goal of distributed simulation being to reduce the runtime of a system, using different machines becomes an integral part. An object-oriented approach reduces the reliance upon a single type of machine from one manufacturer, thus increasing the flexibility of the system.

Fault tolerance represents “another potential benefit of utilizing multiple processors” (Fujimoto 5). A failure in one process may not represent the end of the simulation, as would be the case in a single-processor simulation system. If a distributed simulation environment is correctly designed and setup, the tolerance of the system can increase the overall effectiveness and efficiency of the system.

The first integral component to any simulation is the simulation clock. This represents how time will chance when the simulation executes. Users generally define a base unit of time for the simulation, anywhere from seconds to years to perhaps centuries or even larger units depending upon the type of simulation (Kelton 7). When a user creates a model, he/she must keep the model parameters within the scope of the model; using hours to simulate a system that is only valid in real life for seconds, say simulating the flight of a baseball would not be recommended. The distribution of this clock represents a large problem for distributed simulation.

The model represents the next component of any simulation. A model is composed of modules, each of which represents some function of the model, such as delaying an entity. If a model represented an assembly line, the modules would represent the pieces of the assembly line where work occurs, such as a machine that welds two pieces of metal. Entities would then represent the pieces of metal which travel through the assembly line. The more precise a model is the smaller the granularity of the system resulting in an increased runtime of the system.

The state of a simulation is preserved via the variables assigned to the modules and entities in a model. Each module stores its specific function, delay time of entities, current entities in the model, destination options, and unique name. Entities store their creation time, delayed time, and current waiting time. With each clock event, the state of the simulation is updated as entities delay or move from one module to another.

Distributed simulations usually take two different approaches to dealing with models: breaking the model up and running the entire model in parallel, or replicating the model on each machine the in system and executing the model and comparing the results. Taking the model and breaking it up has become the preferred method because this results in a decrease in runtime of the overall simulation.

Using the foundations of machine learning, distributed processing, discrete-event and continuous simulation, my thesis integrated some of the most attractive features of all four. Entities in a continuous simulation working at predefined set time intervals with the ability to 'learn' and the structure set by models in a discrete-event simulation while using parallel processing. The merging of these key concepts from provides the basis of my thesis.

Previous Work

The distribution of the simulation clock represents a key problem for simulation using multiple machines. The simulation time must be accurately and almost simultaneously relayed to every machine in the system. If each machine does not receive the time correctly, it may miss an event and fail to update its state correctly.

Fujimoto suggest that it may be more valuable and "efficient to only update the variables when 'something interesting' occurs (32). This would result in machines only updating the states of the modules currently belonging to that machine only when required as opposed to at every event. This updating is known as a logical process, and distribution of these allows for asynchronously running a synchronous simulation. This streamlining can thus improve the efficiency of the simulation and thus decrease the runtime.

The 'grain-size' of the simulation has the significant impact of affecting both the runtime and the relevancy of the produced results. The appropriate grain size depends upon the application and practical considerations. Grain size is also directly impacted by the availability

and use of existing simulation code (Logan 3). Reusing a portion of code may result in an inappropriate grain size for a model, thus increasing the difficulty in distributing a model.

The distribution of the clock also leads to the problem of synchronization of an entire distributed simulation. If the differences in time between when any two given machines in the simulation grows too large, and the simulation may begin to experience errors as a result. On a single machine, this problem remains irrelevant as the events of the simulation occur sequentially. The simplest solution to this problem as explained by Fujimoto is to use a local causality constraint, where the messages sent between the various machines time stamp each message to help maintain synchronization (52). This system is also prone to deadlock because if a cycle of messages occurs with differences in the timestamps smaller than the unit of time in the simulation, each process sending a message may wait for the other processes to complete (Fugimoto 3).

Another issue being faced is the interaction of learning entities with their environments, especially in environments which change. As an entity learns, based on some goal-directed task, the entity further specifies how it will interact with the environment, which is ideal when the environment is static. Entities thus must adapt to a dynamic environment however, as previous knowledge or results may not be relevant to the current environment. Voinea takes the approach called 'learning_while_interacting' which is based on "the assumption that in a multiagent system composed by many embodied individual agent there are simultaneous active behaviors" (66). These entities are competing or cooperating for shared resources, and thus must adapt to a changing environment.

Methodology

At the outset of my thesis, I choose a layered and model-based approach to the development, using a centralized machine, known as a Baron, for developing and distributing the

components of a model, and a network of machines, known as Peons, for the execution of the simulation.

The distribution of the clock represents the first major hurdle that must be overcome when dealing with distributed systems. Regardless of how the system operates, a simulation clock remains a vital component. Since a layered approach was used, where the model of the simulation would remain locally on the Baron with its components split between Peons, a centralized clock would be used to maintain synchronization. As a simulation would start, the Baron would send out a UDP packet via a multicast broadcast to all Peons, signaling the start of the simulation and the given runtime. As the simulation then ran, a message would be sent out again using UDP and multicast signaling for each Peon to work, and then followed by a stop message. This maintains synchronization with each machine throughout the runtime of the simulation.

The real world is extremely complex; so complex that we can only hope to realize the scale and chaos. A model represents the simplification of the real world into something that we as humans can understand. The model of the atom with a nucleus and electrons in finite states as proposed by Bohr is used to help remove some of the layers of complexity, which even today are not fully understood, in order to help us understand the affects. Using a model to represent some simplification of the real world assists simulation as well. Since we create the models used in simulations, the results of a given simulation contain greater relevance. The model-based approach thus contains many benefits relating to the components of a simulation, with a few drawbacks.

Breaking down a model into modules, resources, and entities helps to maintain the state of the system. Each component can easily keep track of its own local variables without the concern of the state of others. This allows each Peon in the simulation to only be concerned with

its local modules and entities. However, Peons must be responsible for the resources of other Peons, and thus must support the same functions that a Peon could perform on local resources on distributed resources.

Modules represent the interaction of the simulation environment with the entities. Entities travel through the model by moving from one module to the next. At each module, some delay may occur to represent some portion of the model. Entities are queued in the model in a first-come-first-served manner. If we choose to model an assembly line, the entities would represent the initial pieces on the line. As they travel down the line, they visit stations, which would be represented as modules, such as a welding station. The entity would then delay for some function of the simulation time in the module to represent the two pieces of metal being welded together. The module may have to 'seize' some shared resource in order to perform the delay; otherwise the entity is forced to wait in a queue.

Typically, models use a specific type of module to represent the delay that occurs with a resource, called a process module. The process module has access to a delay function and a resource, whether the resource is stored locally to the Peon the process module is assigned to or not. Since resources may be shared, used by multiple modules at the same time, the Peon must support the ability to access distributed resources. This is performed by assigning the resource to a specific Peon in the same manner that a module is.

When a choice between options in a model must be represented, a decide module is used. This module does not possess a delay function, but instead an expression representing how the module should route the current entity to the possible destinations. The representations I use are simply by chance and by alternating. Deciding by chance means that all possible output options have a probability of being picked assigned, and when an entity enters the module it received a random value. Depending upon which range of probabilities this value falls in determines the

destination of the entity. If instead alternating is used, the decide module simply assigns the entity to the next destination in the list of output options, using the first item in the list as the next option when the last destination option was last selected, thus creating a circular list.

Entities as discussed earlier travel from module to module based upon the assigned destination of the current module. My implementation of the entity allows it to choose its output option when entering a decide module as well as have its destination assigned depending upon the type of entity, learning or typical. Learning entities have the goal of traveling through the model in the shortest time possible, where typical entities perform as normal, obeying the modules.

Entities perform this decision making process using a perceptron (Mitchell 86). A perceptron takes as input some information about the output option, such as previous queue length, and has a single output. Each input has a given weight associated with it, and when asked to make a decision, the perceptron multiplies each weight by the input value and computes either a one or negative one. A one represents a positive decision and a negative one a negative decision. After all of the output options have each been decided upon, the first option with a positive decision is the destination. If no destination receives a positive result, then the entity is assigned to an option as a typical entity would be for that decide module.

When an entity reaches an end module, which records the statistics of all the entities of the model, it is then told to learn. This means that it compares its current time through the model to its previous time through the model. If the current time is less than the previous time, then the entity learns with a positive result. Otherwise, it learns with a negative result. Learning with a positive result refines the weights associated with the given output options appropriately; a negative means that weights are reduced in hope of an acceptable option being selected on the

next pass. The entity will then be sent back to the create module, that handles the arrival of entities to the model, in order to run through the model again.

Since learning entities run through the model multiple times, and typical entities could also exist in the simulation, while the location of given modules and resources do not change, entities may 'flow' through the model differently each time through. This results in a dynamic situation where the previous quickest path through a model may no longer be the quickest because of an increase of queue lengths from entities selecting specific output options. Entities may no longer have optimal information for making decisions.

Each Peon keeps the simulation time locally based upon the messages received from the Baron. When the simulation time reaches the appropriate runtime of the simulation, then the Peon stops executing the simulation, and begins to finalize statistics. All destroy modules then update the statistics that are gathered and send a multicast message to the Baron, who combines the statistics dynamically. This allows for statistics to be computed without the need to buffer all of the previous data send by Peons. The Baron can thus update the statistics one Peon at a time.

Results

The result of using a multicast communication system for the simulation clock means that there is no guarantee that a given Peon receives a message from the Baron. This implies that synchronization cannot truly occur, and will only appear in a stable network. This problem became apparent when also dealing with the clock speed of the simulation environment. If the clock speed was increased to below 250 milliseconds per cycle, the simulation become asynchronous as some Peons processed more information than others depending upon the components of the model currently residing there. With this system running on two machines, meaning a Baron and single Peon, the clock speed must be increased to 300 milliseconds to maintain reliability.

Since entities must maintain the necessary information for learning and deciding, this information must be passed with the entity as it moves from module to module. The current implementation uses multicast with UDP packets to send entities from one Peon to another. As the number of decide modules in a model increases, the amount of information pertaining to the weights for the perceptrons that must be stored and transmitted with each entity. As such, an entity may not fit into the conventional size of a multicast packet, producing errors for all Peons and the Baron in the system, and thus ending the current simulation unexpectedly.

Unfortunately, no emergent behavior could be observed in any of the models used due to the size restrictions on the number of decide modules. Entities would possess too much information if more than two decide modules exists, and if each decide module has more than four output options. Also, due to the lack of a graphical user interface, viewing the simulation as it runs becomes increasingly difficult. Thus, identifying emergent behavior cannot be easily performed.

An increase in the cycle time was found when large models were used with a number of Peons similar to the number of modules. This means that a large amount of information would be sent during each cycle, requiring the processing time of each machine for data transfer and not necessarily execution of the simulation. A decrease in the amount of overhead needed would help to alleviate this problem, although the necessary clock increase was only 50 milliseconds from the previous (250 to 300).

Using a model consisting of a create module which produced two intelligent entities, a create module which produces a typical entity every four simulation clock cycles, two decide modules, one alternating and one by chance, four process modules, and a single destroy module, I was able to produce results that show that the intelligent agents proceed through the model faster

than the typical agents. Intelligent agents are recycled by the destroy module, thus allowing them to run through the model more than once.

Entities, both typical and intelligent took approximately fifty five simulation time units to travel through the model which was executed for approximately three hundred simulation time units with fifty six entities on the average. Intelligent entities would travel through the model ten simulation time units quicker on the average. The number of entities includes the number of typical entities along with the number of passes by intelligent entities, as statistics are gathered from an intelligent agent each pass, thus the simulation perceives this as a new entity. The intelligent entities would have an average of nine and a half runs through the model during the course of the simulation. These results where found to be consistent using anywhere from one to nine Peons. Using more than nine Peons produced the same results as nine Peons, as any other Peons beyond the ninth did not have any modules assigned to them and thus would not compute any part of the simulation.

Entities would leave their respective create modules, and immediately enter a decide module. This module alternated between two process modules, one with a two simulation time unit delay, and the other with a four simulation time unit delay. This results in approximately equal queue lengths for each module. After leaving either process module, entities enter a second decide module.

In this model, the second decide module processes by chance and has a much higher probability, eighty percent, of picking a process module with a large delay, six simulation time units, and a twenty percent probability of picking a process module with a small delay, two simulation time units. The result of this is that large queues would occur on the process module with a large delay. Intelligent entities should therefore have, and did learn to choose modules with smaller delay and wait times.

By selecting the process modules with smaller delay times and shorter queues, the intelligent entities could then minimize the total time through the model, the goal given to each. These entities saved an average of twelve simulation time units in both total time in queues and total time waiting, defined as time delayed and time queued, when compared to the times of both intelligent and typical entities. This shows that entities can make their own decisions in a dynamic simulation environment.

Future Work

If work on this project continued in the future, I would like to see the following items addressed for improvement:

1. A reliable means of synchronization. Perhaps if IPv6 was used where reliable a reliable multicast protocol currently exists, as opposed to the current IPv4 protocol which only operates on a local area network.
2. Decide modules having more sophisticated means of selecting between output options. One such method could be based upon the queue lengths of possible output options. This would require an increase in overhead as an increase in messages between Peons would result as the decide module now requires information that may or may not be stored locally. This would also require that expressions be stored and validated in some manner, most easily in a stack-based method.
3. Increased support for expressions, allowing create modules to produce entities based upon sophisticated functions, such as exponential or logarithmic; or process modules to delay based upon these same options.
4. Entities to be sent in a more efficient manner, specifically via TCP rather than multicast. This is to allow the model sizes to grow, specifically with an increase in the number of

decide modules as each entity must keep the local information pertaining to the learning functionality and this information overflows the current size of a UDP packet.

5. Dynamic clock speeds. This is to allow the clock speed to be adjusted based upon the number of Peons, modules (specifically decide modules), shared resources, and entity creation rate.
6. Graphical user interface. The current implementation uses a simple text based user interface, which can be extremely clumsy and frustrating, especially when dealing with large models.
7. Support for submodels. This would allow for the creation of larger models from small, already existing models, thus decreasing the development time for the user (Ryde 1). This also allows for inter-enterprise simulation; businesses may then simulate the results of possible partnerships upon the global market and how each business would be affected.
8. Porting the simulation application to run on J2ME, thus allowing for the use of computation power of embedded processors along with easier networking and setup. The growth in the number of traditional computers manufactured each year pales in comparison to the number of embedded systems produced, resulting in a lower cost of an embedded system, and thus the ability to produce a distributed simulation environment at a lower cost (Bruzzone 690).

Conclusion

Using distributed simulation and machine learning techniques, I was able to show an increased efficiency in simulation of small models, along with the ability of intelligent entities to reduce their total time through a model based upon their own decisions. This was only functional with small models, having two or fewer decide modules, as it was not possible to simulate models with a larger number of decide modules due to the constraints of the system.

Emergent behavior can also be extremely difficult to identify, especially without a graphical user interface. Taking into account my suggestions for future work, this simulation system could be improved and prove extremely viable for large-scale models.

Sources

Blair, Eric; Wieland, Frederick; Zukas, Tony. (1995) *Parallel Discrete-Event Simulation (PDES): A case Study in Design, Development, and Performance using SPEEDES*. McLean, VA. IEEE

Bruzzone, Agostino; Fujimoto, Richard; Gan, Boon Ping; Paul, Ray J.; StraBburger, Steffen; Taylor, Simon J.E. (2002) *Distributed Simulation and Industry: Potentials and Pitfalls*. Uxbridge, UK

Fersha, A. (1995) *Probabilistic Adaptive Direct Optimism Control in Time Warp*. Vienna, Austria. IEEE

Ferscha, A. (1995) *Parallel and Distributed Simulation of Discrete Event Systems.* Vienna, Austria. McGraw-Hill

Fujimoto, Richard M. (2000) *Parallel and Distributed Simulation Systems*. New York, New York. John Wiley & Sons

Law, Averill M; Kelton, W. David (1982) *Simulation Modeling and Analysis*. Boston, MA. McGraw-Hill

Logan, Brian; Theodoropoulos, Georgios (1999) *A Framework for the Distributed Simulation of Agent-based Systems*. Birmingham, UK. European Simulation Multiconference

Mitchell, Tom M. (1997) *Machine Learning*. Boston, MA. McGraw-Hill

Ryde, Michael D.; Taylor, Simon J.E. (2002) *Issues Using COTS Simulation Software Packages For the Interoperation of Models*. Uxbridge, UK.

Voniea, Camelia F. (2001) *Attitude Learning in Autonomous Agents*. Bucharest, Romania. 2nd Workshop on Agent-based Simulation; Passau, Germany