On the Distributed Orchestration of Stochastic Discrete Event Simulations

Zhiquan Sui
Computer Science Department
Colorado State University
Fort Collins, CO, USA
simonsui@cs.colostate.edu

Neil Harvey
School of Computer Science
University of Guelph
Guelph, Ontario, CA
neilharvey@gmail.com

Shrideep Pallickara
Computer Science Department
Colorado State University
Fort Collins, CO, USA
shrideep@cs.colostate.edu

***Abstract*— Discrete event simulations are a powerful technique for modeling stochastic systems with multiple components where interactions between these components are governed by the probability distribution functions associated with them. Complex discrete event simulations are often computationally intensive with long completion times. This paper describes our solution to the problem of orchestrating the execution of a stochastic, discrete event simulation where computational hot spots evolve spatially over time. Our performance benchmarks report on our ability to balance computational loads in these settings.**

Keywords-Discrete Event Simulations; Distributed Systems; Stream Processing Systems; Epidemiological Simulations

I. INTRODUCTION

Discrete event simulations are a powerful technique for modeling stochastic systems with multiple components where interactions between these components are governed by the probability distribution functions associated with them. Discrete event simulations have been deployed in domains such as economics, traffic modeling, disease epidemic modeling, and weather forecasting.

Complex discrete event simulations are often computationally intensive with long completion times. Running them on a distributed set of machines can speed up such simulations. However, such an approach introduces additional challenges. First, we must be able to divide the simulation into components that can execute concurrently. Second, since these components need to interact with each other, *e.g.*, to synchronize information or to trigger events in other components, coordinating these communications is important.

In this paper we describe how a distributed stream processing system, Granules [2], was used to orchestrate the execution of a discrete event simulation. The simulation that we consider is NAADSM (North American Animal Disease Spread Model), which models the outbreak of epidemics among livestock. Granules has three features that make it particularly well-suited to orchestrating the execution of discrete event simulations: (1) Support for computations that produce and consume events as streams, (2) support for computations that can have multiple rounds of execution with state retention across successive execution rounds, and (3) the ability to activate a computation when data is available on any of its input datasets.

There are several reasons why efficient, distributed execution of our epidemic simulation is hard.

1. The computations are irregular. The simulation is stochastic and events within the simulation (relating to infections, movements, detections, etc.) are generated based on the probability distributions associated with these events in real settings. Events must be processed and the accompanying computational footprints depend on the state of the simulation and the event itself.
2. The simulation is memory intensive, so there is a limit on the number of partitions that can be hosted on a single machine.
3. Synchronization across the simulation happens frequently. Dependencies exist across workers since the state of a region is influenced by the state of the other regions for a given simulation day. Events generated in one region that impact another must be processed within computations that manage the impacted region.
4. Computational hot spots emerge stochastically depending on the intensity of the disease breakout in a region. This results in a corresponding increase in the number of events that are generated and their processing accompanying footprints. The computational hot spots move spatially depending on the paths taken by the disease. The hot spots may straddle multiple workers that manage different contiguous regions and can move from worker to another over the course of the simulation.

We posit that while orchestrating such simulations one must account for spatial features and also how the computational loads evolve temporally. The nature of these simulations result in computational imbalances at nodes that are responsible for different portions of the simulation. We describe three algorithms that address the spectrum of possibilities for distributed orchestration of such spatially explicit simulations. We have devised a measure of the computational imbalance across different nodes and also describe how to alleviate such imbalances. The responsiveness of the algorithms to alleviate such imbalances is directly proportional to the speed ups that are achieved.

Paper Contributions:
This paper's contributions are three-fold. First, it proposes a framework for orchestrating the execution of long-running stochastic simulations using a distributed stream processing system. To the best of our knowledge this is the first such attempt and is an innovative aspect of this research. Second, it describes a scheme to alleviate computational imbalances that emerge due to spatially evolving performance hotspots to improve overall efficiency and speed-ups. Our benchmarks are comprehensive and we contrast the performance of different algorithms. Third, since our approach required no changes to the epidemiological model, we expect our approach to be broadly applicable to other discrete event simulations where computational loads evolve spatially over time.

The remainder of this paper is organized as follows. In section II, we provide an overview of NAADSM and Granules. The system structure and mechanisms are described in section III. We then introduce our partitioning strategies and the corresponding experimental setup and performance measurements in section IV. Section V describes some related work in the discrete event simulation area and load balancing area. Finally, in section VI we present our conclusions and future work.

II. BACKGROUND

A. North American Animal Disease Spread Model (NAADSM)

NAADSM is a project developed by the U.S. Department of Agriculture, the Canadian Food Inspection Agency, Colorado State University, the University of Guelph, and the Ontario Ministry of Agriculture, Food and Rural Affairs. NAADSM simulates the spread and control of livestock diseases [1]. Diseases simulated within NAADSM include foot and mouth disease, exotic Newcastle disease, pseudo rabies, and avian influenza. The project is an open-source effort and the software is available for download from http://www.naadsm.org/.

NAADSM is a stochastic discrete event simulation. The probability of the occurrence of various events is governed by probability density functions (PDFs), which are input by the modeler based on scientific evidence and observations made by epidemiologists. The fundamental unit of spread and control is a farm. Each farm has a state with respect to the disease, such as susceptible, infected, or immune. NAADSM simulates both spatial and temporal aspects of disease spread and control. Examples of spatial activities are movement of animals between farms and establishment of disease control zones. The temporal aspect encompasses the progression of individual farms through disease states, and propagation of the disease between farms over simulation days.

B. Granules

Granules [2] is a distributed stream processing system for processing data generated by sensors and programs in real time. In Granules computations can be expressed as MapReduce [3] or as directed cyclic graphs [6] and the runtime orchestrates these computations on a set of available machines. Individual computations can specify a scheduling strategy that allows them to be scheduled for execution when data is available or at regular time intervals. Computations in Granules can have multiple, successive rounds of execution across which they can retain state. A computation can change its scheduling strategy during execution, and the runtime will enforce the new scheduling strategy during the next round of execution. A computation is also allowed to specify a ceiling on the maximum number of times that it can be scheduled for execution. The runtime allows a computation to specify a hybrid scheduling strategy that is a combination of data availability, periodicity, and the maximum number of execution.

Granules is an open-source effort and computations can be developed in a number of programming languages such as C, C++, C#, Java, Python and R [22]. Some of the domains that Granules has been deployed in include bioinformatics, brain-computer interfaces [13], multidimensional clustering algorithms [23], handwriting recognition [14], and epidemiological modeling.

C. *Problem Statement*

A NAADSM simulation must be run many times to build up a picture of the distributions of its output variables. A single run of the simulation may be computationally intensive, particularly if the simulation features a large number of farms infected simultaneously, a large amount of interaction between farms, and/or a large number of disease control zones being established to enforce movement controls and quarantines.

Given the need to run each simulation many times, an obvious approach is to simply divide the desired number of simulation runs among a set of processors. This is how NAADSM has typically been deployed in supercomputing environments. In these cases, each run proceeds sequentially on one underlying processing element and the user must wait for this sequential execution to complete for results to be available.

Distributed orchestration of NAADSM involves executing a given simulation run concurrently on multiple machines. This would allow a given simulation run to complete faster than in the sequential case. Distributed orchestration may also be useful in cases where the number of available processors is greater than the desired number of runs of a simulation. Consider running a simulation N times. With 2N processors, simply dispatching individual runs of the simulation one to a processor will leave half of the processors idle. However, if individual runs of the simulation could also be divided across processors, all of the processors could be employed and results would be available sooner. As an aside, it might appear that reduced memory use would also be a good argument for distributing one simulation run across several machines; however, as elaborated in Section III.C, we did not anticipate substantial memory savings from partitioning the simulation because some bulky data items, like farm location and farm type data for the full study population, still need to be retained across all workers.

III. System Structure and Mechanism

A. *Controller-Worker Model*

To orchestrate a NAADSM simulation using Granules, we employ a controller-worker model (depicted in Figure 1) with one controller and many workers. Simulation tasks are divided among the workers, while the controller keeps the workers synchronized. The controller and workers execute concurrently on different machines.

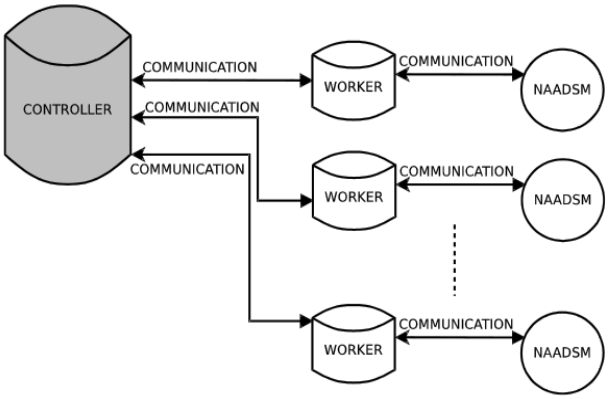


Figure 1. Controller-Worker Architecture

The structure is suitable for this particular problem because each worker has to send synchronization information to all the other workers in each simulation day (as discussed in Section III.B below). Assume there are N workers in the system. If the system relies on peer communications between workers, each worker has to broadcast 1 message and receive N-1 messages. Thus, there will be N(N-1) messages going through the network. In our controller-worker structure, workers send its own synchronization information to the controller, and the controller combines them into one message and broadcasts it. In each simulation day, there are only 2N messages going through the network.

Each worker knows the location and type of every farm in the population, but manages only a subset of the farms. The division of farms among workers is a geospatial partitioning with each worker managing a non-overlapping portion of the study area where the disease spread is being modeled. The choice to have workers manage *non-*

overlapping portions of the study area eliminates some potential choices for partitioning methods, such as assigning randomly-chosen subsets of farms to each worker. However, some aspects of disease control can follow political boundaries (*e.g.*, control over resources and task priorities) so with an eye to possible future extensions in that domain, we made an early decision to use non-overlapping spatial partitioning.

B. Flow of Work and Events

NAADSM is a discrete event simulation application. Any interesting occurrence or interaction in the simulated world is an event. For example, an *exposure* (such as movement of animals or contact via people, trucks, or equipment) that may transmit disease from one farm to another is an event and *vaccination* of a farm is also an event. Some events can trigger other events; for example, *detection* of disease at a farm may lead to *tracing* which other farms have had contact with the infected farm, which in turn may lead to more *detection*.

We have defined an extensible *wire format* so that all of NAADSM's events (and any new types of events in future versions) may be marshaled and unmarshaled between Granules computations. The following sections introduce some of the events in the system, in the order they occur in the flow of execution.

1) Starting the Simulation Day

A START_NEW_DAY event, sent from the controller to the workers, signals the workers to begin the sequential execution of NAADSM logic for one simulation day. All workers begin each simulation day in lockstep *i.e.* all the active workers begin day 1 of the simulation at the same time, then begin day 2 of the simulation at the same time, and so on.

A day is the natural barrier at which to keep workers in lockstep because a day is the basic time-step in a NAADSM simulation. If an effective exposure occurs from infected farm "A" to susceptible farm "B" on day d, farm "B" will start its incubation period on day *d*+1. If a farm is identified on day *d* as requiring vaccination, the vaccination will occur (at the earliest) on day *d*+1. Synchronization at intervals of less than one day can be avoided. Synchronization at intervals of more than one day would create possibilities for inconvenient out-of-order events. For example, if a worker that is on day 5 of the simulation were to receive an event informing it that one of the farms it manages received an exposure on day 3, the worker would need to backtrack to day 3 to include the effects of that additional infected farm.

2) A Single Simulation Day

A NAADSM simulation day consists of disease spread and disease control elements. During a typical simulation day, a worker may generate events such as:

- *Exposures* (*e.g.*, animal movements; indirect contacts via people, trucks, or equipment; airborne spread) originating from farms managed by the worker.
- *Detections* of disease by farm owners or veterinarians, on farms managed by the worker.
- A public *announcement* of the first detection of disease.
- *Requests to vaccinate* farms, based on various strategies. For example, requesting vaccination for all farms within a fixed distance of a detected infected farm.
- *Quarantining and vaccination* of farms managed by the worker.

Figure 2 provides a sketch of the activities within a single simulation day.

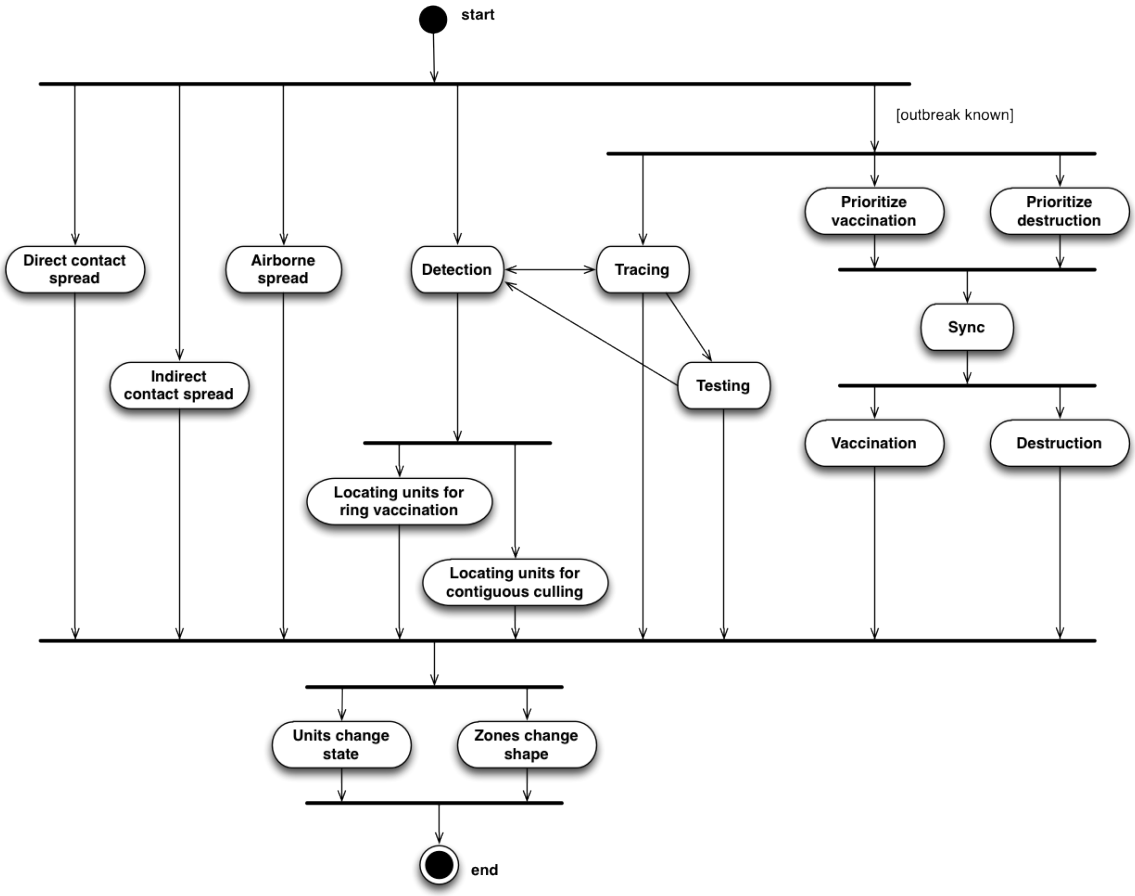


Figure 2. Conceptual flow of events (not considering standalone vs. distributed operation) in one simulation day.

In a distributed setting, a worker generates all the events originating from farms in the area it manages for one simulation day without any communication with the controller or other workers. While doing so, the worker records events that may affect farms outside of the area the worker manages and passes this to the other workers after all the local work has been done.

3) Continuing or Ending the Simulation

The last event in the stream sent from a worker to the controller is an END_OF_DAY event. In the END_OF_DAY event, each worker sets a flag indicating whether it believes the simulation termination conditions have been reached. Termination conditions are specified by the modeler as part of the simulation parameters, and may include the following:

- A fixed number of days have been completed.
- The first detection has occurred.
- All disease spread has completed *i.e.* no incubating or infectious farms remain in the population.
- All control measures have been completed. For example, no more farms are queued to be vaccinated.

If every worker has set the termination flag, the controller sends an END_OF_SIMULATION event, telling the workers to shutdown. Otherwise, the controller bundles together the events it received from each worker and streams that bundle out to all workers and the simulation progresses. In this way, each worker receives necessary updates about events that occurred in the areas managed by the other workers. Such events include exposures, public announcements, and vaccinations; a short description of each follows.

Consider the case of exposures where the source farm was managed by another worker, but the recipient farm is managed by this worker. The worker must now initiate the progression of disease in the recipient farm. Even

exposures that do not cross the boundaries between geospatial areas managed by different workers are communicated because other workers may need that information later on to carry out *tracing*. Tracing is the process of finding the farms that have had contact with a detected infected farm, and the farms that have had contact with those farms, and so on. (The pattern of contacts can potentially be traced all the way back to the start of the simulation, although in practice modelers will specify a time period of interest, such as all contacts within the last 2 weeks, that limits the tracing.)

There can also be public announcements of detection of disease in other workers' areas. This may affect exposure rates (if animal movement slowdown is among the disease control measures implemented in the simulation) and detection rates (if detection increases now that awareness exists that disease is present in the population) in subsequent simulation days.

We may also have situations where requests to vaccinate farms generated by the other workers. For example, if the disease control strategy includes vaccination of all farms within a fixed distance of a detected infected farm, that circle of vaccination may cross the boundaries between areas managed by different workers.

Finally, after sending out this synchronizing information to the workers, the controller sends the workers another START_NEW_DAY event to continue the simulation if the controller determines that the simulation needs to continue. Interactions between the controller and workers are summarized in Figure 3.

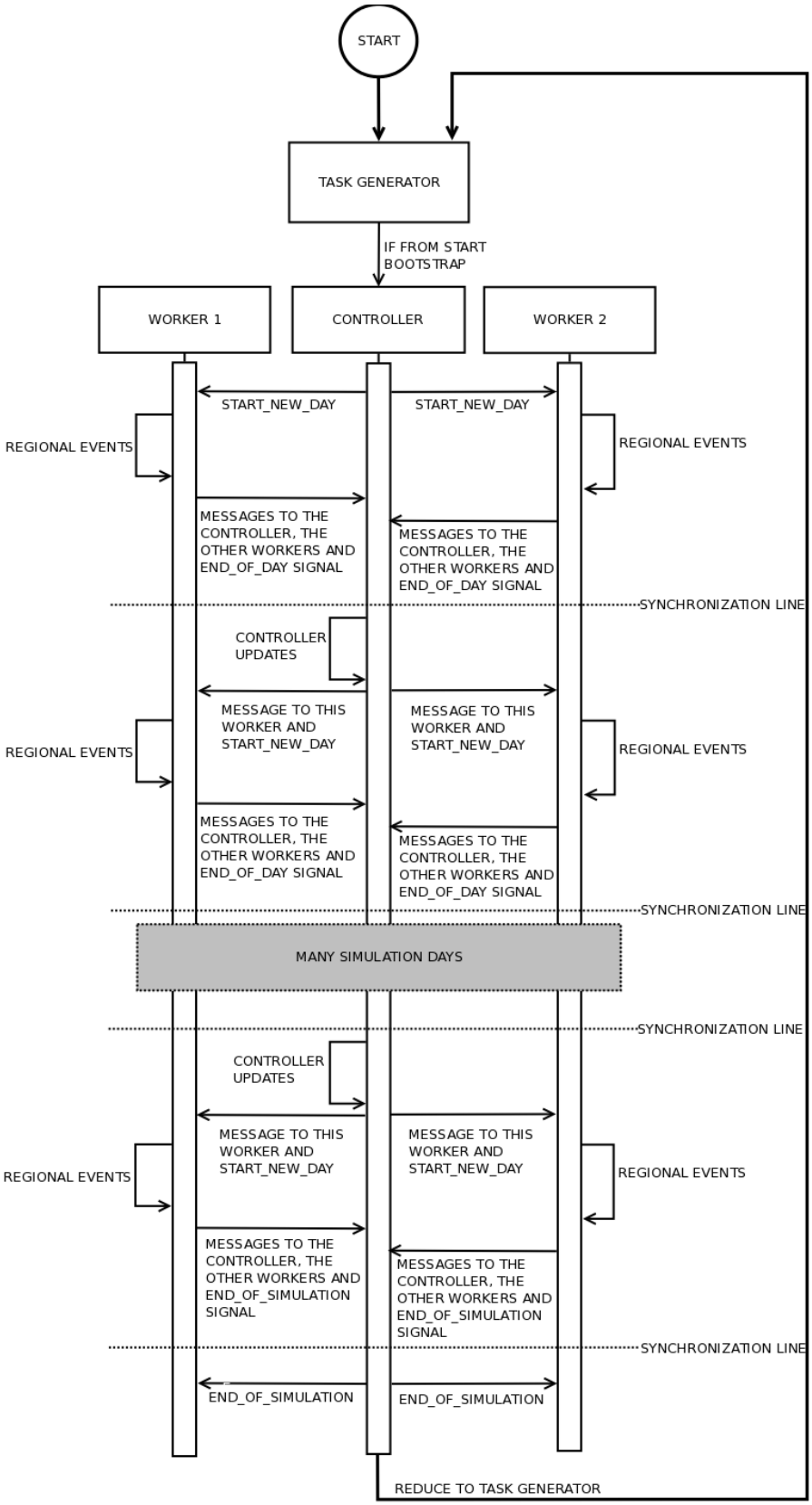


Figure 3. Interactions between the controller and workers

C. Adapting NAADSM to Work with Granules

We had two key goals for orchestration. First, we wanted to ensure that we make no changes to the NAADSM logic. Second, we wanted to maintain loose coupling between NAADSM and Granules, so that future versions of NAADSM might be substituted into the system with minimal effort. In our system, we completely accomplish these two goals.

In some ways NAADSM was already well suited to work with Granules. The design of the simulator is modular, with modules communicating via events following the Publish-Subscribe design pattern. The simulator modules are unconcerned with where the events they receive originate, or where the events they produce go. The events may all be produced and consumed within the memory space of a single process, or they may be exchanged over a network with other computers; it makes no difference to the simulator modules. Thus we were able to stream events to distributed computations that were responsible for managing different subsets of farms.

Inside NAADSM, supporting streaming of events required the addition of a *bridge*. The bridge translates incoming events in the binary "wire format" into objects of the appropriate type, and translates outgoing event objects into the wire format. In the Publish-Subscribe paradigm, this bridge is simply another object that *publishes* incoming events (takes them from the "wire" and injects them into NAADSM's run loop) and *subscribes* to outgoing events (copies them from NAADSM's run loop and sends them out over the "wire").

Giving an external entity (the Controller) control over the progress of the simulation was a minor modification. Previously, NAADSM had a loop that iterated over simulation days, injecting START_NEW_DAY events into the run loop; now it has a loop that listens for START_NEW_DAY events from the bridge. A simple module that stands in for the bridge and produces START_NEW_DAY events will allow NAADSM to run standalone if not connected to a Controller.

Distributing responsibility for farms was accomplished by adding a flag to indicate that a farm is or is not managed by the current instance of the NAADSM executable. These flags are set or cleared during the split and merge operations discussed in Section IV. This change, too, is backwards compatible with a standalone setting, in which all the farms are marked as managed by the single running instance of NAADSM.

A few aspects of NAADSM's design required more adaptation to function in a distributed framework.

The Unix version of NAADSM was written for a batch computing environment in which the modeler submits a job, and later receives completed outputs from the job, with no interaction in between. In contrast, Granules is a highly asynchronous framework, in which computations may receive messages over their communication streams at any time. This mismatch was resolved by having the worker component, which is coded in Java, handle the asynchronous communications and act as a buffer for incoming messages. Communication between the Worker component and the NAADSM executable is then done synchronously, at specific points in NAADSM's run loop, via the `stdin` and `stdout` streams of the NAADSM processes. Consistent with our goal of loose coupling, this avoids the need to make deep changes in the NAADSM code (such as transforming it into a multithreaded program) just to support interaction with Granules.

Partitioning the simulation by geography presented a challenge. Some interactions between farms do not have distance constraints associated with them, owing to the model's original scope of limited scale simulations. For example, suppose the model makes a stochastic decision that a given farm is going to make a shipment to another farm of type "X". Suppose further that the only available recipient farm of type "X" is very far away (perhaps farms of that type are rare, or the nearby ones are already under quarantine). The model will make that shipment rather than rejecting it on the basis of distance. Whether this behavior should change to suit larger scale simulations is a question for model development, but the scope of the current work did not include altering or extending the original model; the cost of this choice is that each worker must maintain a small amount of status information about *all* farms in the population, even those very far away. (This was alluded to briefly in Section II.C above, in which we mentioned that we did not anticipate substantially reduced memory use from partitioning a simulation run across machines.)

NAADSM as originally written uses a global random number generator object. This presents a problem for comparing execution time in various distributed configurations. A run performed on a single worker will not behave the same way as a run distributed across two workers, even if the same random number seed is used, because the same sequence of samples will not be drawn from the random number generator in these two cases. We solved this problem by attaching an independent random number stream to each farm, initialized by combining a global seed

and the farm's unique ID. All stochastic decisions pertaining to a farm are made using the farm's own random number stream. Because a single farm is never divided across workers, this scheme guarantees that given the same starting seed, the same sequence of events will occur in the simulation, regardless of how many workers the run is distributed across. Note that this is done solely for the purpose of forcing identical runs of the model (no matter how many workers the run is distributed across) so the result of runtimes can be meaningfully compared across distribution experiments. It is important to remember that this is *not* a necessary step in adapting a program like NAADSM to be managed by Granules in a distributed setting.

An important aspect to note in Figure 3 and the discussion in Section III.B above is that the synchronization step between workers does not quite occur on the boundary between simulation days. Rather, it occurs at a point where each worker has processed all of the day's events (exposures, detections, *etc.*) *originating from the subset of farms that it manages*. Only after the worker exchanges updates with the other workers during the synchronization step does the worker have a complete picture of all of the day's events that affect the farms it manages; then the worker can proceed to the next simulation day. So we can say that inter-worker synchronization occurs when each worker contains a partially complete simulation day, rather than on the day boundary. In terms of programming consequences, this means that we must be careful of when stochastic decisions are made. In a standalone setting, stochastic decisions can be made throughout the simulated day, whenever is most convenient for the programmer. In a distributed setting with once-a-day synchronization, all stochastic decisions *that can affect the following day's events in another worker* must be made prior to the synchronization step. Fortunately, it is easy to relocate stochastic decisions to earlier in the flow of execution if needed, without affecting the model's behavior. For example, consider the decision of how long a farm's contagious period will last. You could decide on that number when the farm is originally exposed, or when the farm makes the transition from incubating to contagious. At the conceptual modeling level, it makes no difference when you make that random number draw, but at the implementation level it does make a difference in terms of how early that piece of data is available in the system.

The most significant amount of code added to NAADSM for this project is for marshaling and unmarshaling data structures. In section IV below, we explore strategies for dynamic migration of an in-progress simulation from one compute node to another, with the goal of splitting up the work on overloaded nodes and merging the work from under-loaded nodes. Many simulator modules inside NAADSM maintain private state information; for example, the module that handles vaccination contains queues of farms that need to be vaccinated. Each module must be able to pack its private state information into binary format, and unpack it, in order to support migration of an in-progress simulation. Each module defines its own private "wire format" for its in-simulation data, building on the same routines for marshaling and unmarshaling basic data types that are already used in the wire format for events. When the simulator is instructed to pack up its data for migration, each simulator module produces a block of binary data representing its private state. The bridge concatenates these blocks of binary data, adds an index (so that the blocks can be sent to the correct simulator modules upon unpacking later), and sends the result out over the "wire" just like any other event. In this way, we can take the same communication stream infrastructure that moves individual simulation events between workers and use it to migrate complete simulation state from one computer to another.

In an environment where data is being frequently split up and merged, opportunities abound for erroneous data duplication and/or data loss. As a general approach, we found it useful to classify each data item in NAADSM as either global, meaning that the data item is known to and identical across all workers, or private, meaning that each worker may store a different value for the data item, reflecting only its own local knowledge. This classification was frequently cross-checked during implementation. Two specific examples of situations that required extra care follow.

Note in Section III.A that the controller in our architecture bundles the daily updates from the individual workers and then broadcasts that message bundle to all workers. This reduces network traffic (one broadcast message vs. individual messages sent to each worker), but it also means that each worker will receive its own updates echoed back in the broadcast. It is trivial to make a worker ignore its own updates, but consider the case where worker "1" has just been divided into workers "1" and "2". Worker 1 of course knows to ignore its own updates echoed back; worker 2, having only just begun its existence as a (half-) clone of worker 1, must also know to ignore updates from worker 1 in the first broadcast it receives, lest unintended data duplication occur.

Consider also a case where an infected farm "A", managed by worker "1", sends a shipment of animals to farm "B", managed by worker "2". It might seem that worker 1 can safely forget about this shipment after generating it:

after all, it is worker 2 that will be responsible for applying the shipment to farm B, changing farm B's state to infected at the next day boundary, and so on. And indeed, it is good programming practice to free or destroy data items as soon as they are not needed. However, there is a potential for data loss here: if worker 2 is merged into worker 1 before the next day boundary, then worker 1 *will* gain responsibility to manage farm B. Care must be taken to ensure that the shipment information is not lost: either by applying it to worker 2 before the merge occurs, or retaining it for a little longer in worker 1 *just in case* a merge occurs. Although it sounds counterintuitive, we tracked more possibilities for data loss bugs in merging operations than in partitioning operations, simply because of cases where information seemed disposable because it was "another worker's responsibility" – but merging could imminently change that.

D. Optimizations

Within each simulation day, each worker produces many events to sends to the controller. Rather than send these events as separate messages, we combine them into a larger data structure. Empirically we have found that sending a large message (comprising multiple, small events) introduces less overhead than sending a large number of small events. Each worker sends and receives only one message from the controller for each simulation day.

IV. PARTITIONING STRATEGIES

In the following sections we describe 3 progressively more complex partitioning strategies: static split, dynamic split, and dynamic split and merge. We then compare results for each approach. Experimental results presented in this section were obtained on a 48-node cluster connected by a 1 Gbps link. Individual machines in this cluster run version 14 of the Fedora OS and have 4 cores, 2.4 GHz CPU, and 12 GB RAM. The total number of machines involved in the benchmark varied depending on the number of workers involved. One machine was always set aside for the controller. The thread pool in Granules was configured to match the number of cores available on each machine: four. The maximum number of workers assigned to a machine was four, and each executing on different cores. Our configurations during the benchmarks involved 2, 4, 8, 16, 32 and 64 workers utilizing the corresponding number of cores on up to 16 different machines.

The scenario that we are simulating involves modeling the outbreak of Foot-and-Mouth Disease (FMD) in 660,000 farms, which is roughly the number of farms with FMD-susceptible livestock in the 12 Midwest states, according to 2008 data from the National Agricultural Statistics Service. The population was up-sampled from information for a single state so density and clustering should display a usable level of realism. The population contains seven farm types, with interaction parameters customized for each pairing of farm types. Nine of the farms are infected initially and the disease spread is configured to occur by airborne, direct contact and indirect contact.

A. Static Split Strategy

In the static split strategy, the population is divided among workers before the simulation begins. Workers receive SPLIT events from the Controller, instructing the worker to divide the population along a certain geographical line and assume management of one part of it. If two workers are available, a single SPLIT event is sufficient to divide the population between the workers. Additional SPLIT events can be used to divide the population into smaller parts when more workers are available, as illustrated in Figure 4.

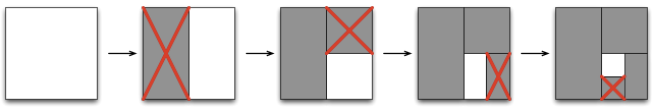
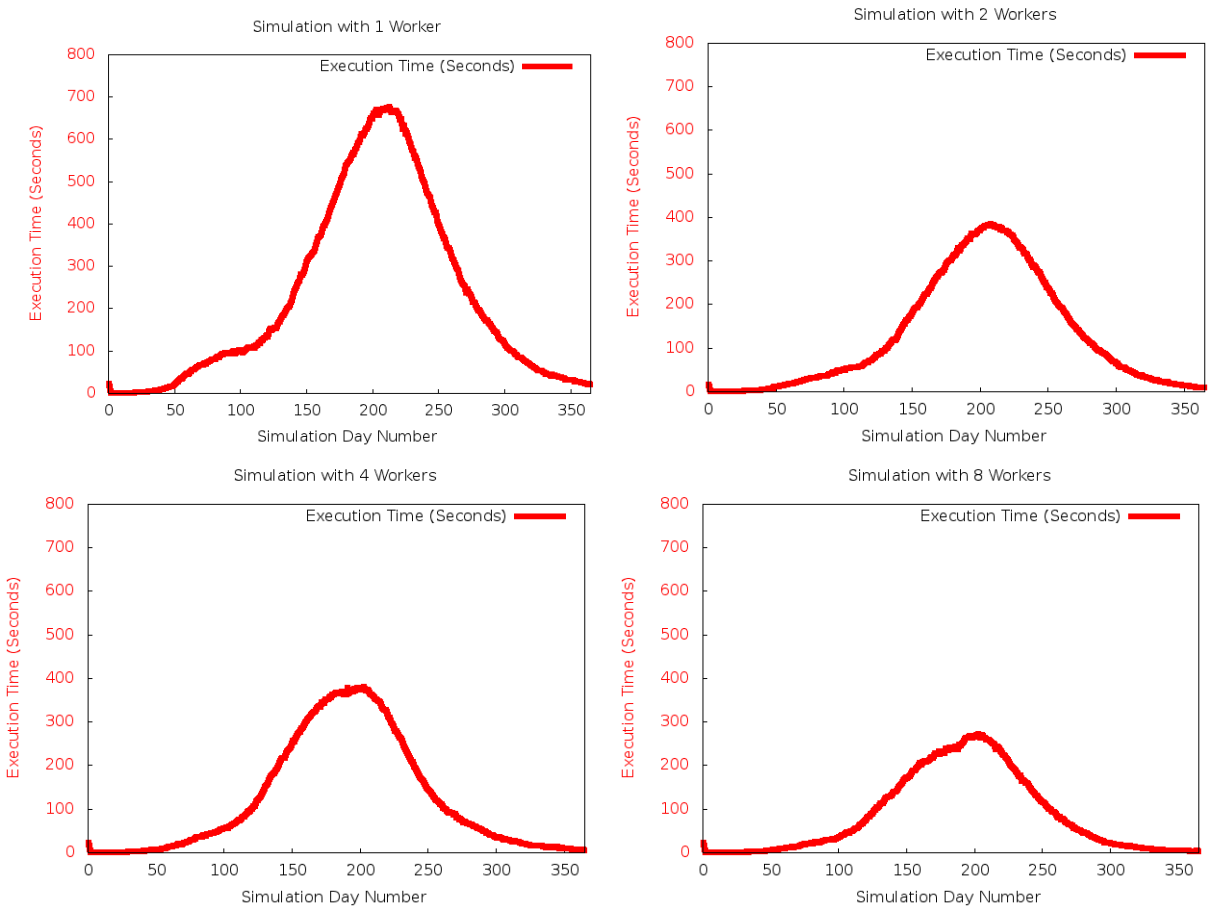


Figure 4. If 16 workers are available, a series of 4 SPLIT events is sufficient to tell a worker which part of the population to manage.

Figure 5 summarizes the execution times for the simulation with different numbers of workers, as well as the speed-up as the number of workers increases. In general, as the number of workers increases the simulation time decreases. However, the speedup that we see is not ideal.

Figure 6 depicts the execution time for the simulation with different numbers of workers. As the disease spreads in a region there is a corresponding increase in the computational load. In general as the number of workers increases there is a corresponding reduction in the execution times for the simulation.

Figure 5. Speed-up vs. Number of Workers



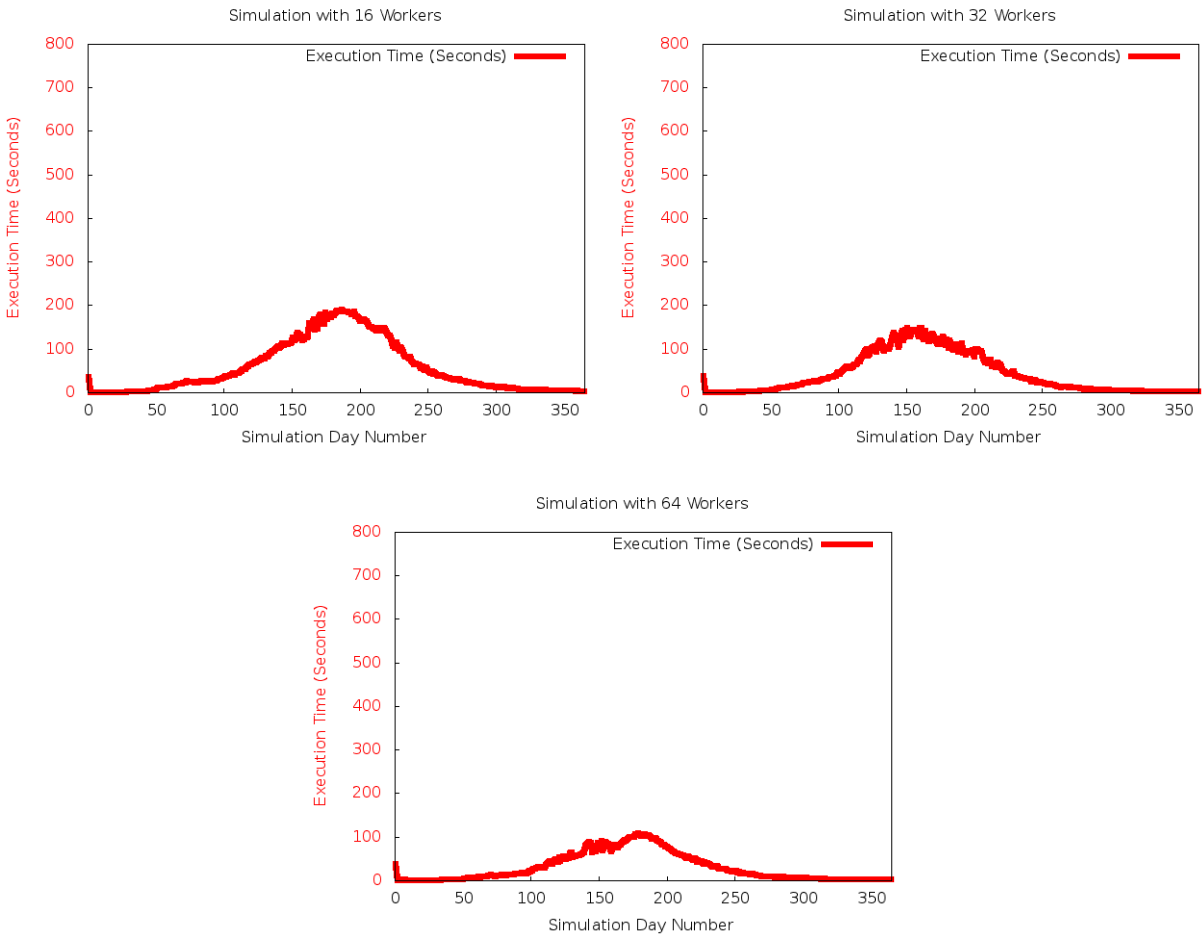


Figure 6. Execution Time for 1, 2, 4, 8, 16, 32, and 64 Workers

One reason for this suboptimal speed-up is the imbalance in the computational load. The simulation involving 1 worker executes for about 70,000 seconds, while that involving 2 workers executes for about 40,000 seconds. However, the speedup in the 4-worker setup does not continue the pattern. Because the assignment of farms to different workers is based on geography, variations in the density of farms and clustering of the disease cases can lead to a subset of the workers bearing a greater share of the computational burden.

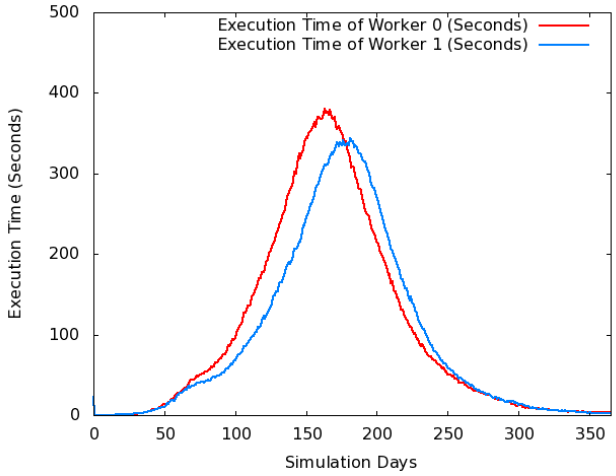


Figure 7. Execution Time of 2 Workers

This can be clearly seen in **Figure 7**, which depicts the execution time per worker per simulation day for the 2-worker setup. We can see that the computational loads at these two workers are almost the same so the speedup is almost ideal. Our partitioning policy is to subdivide an area into 2 equal geospatial halves. In a situation where there are few workers, load imbalances between the workers may not be that serious.

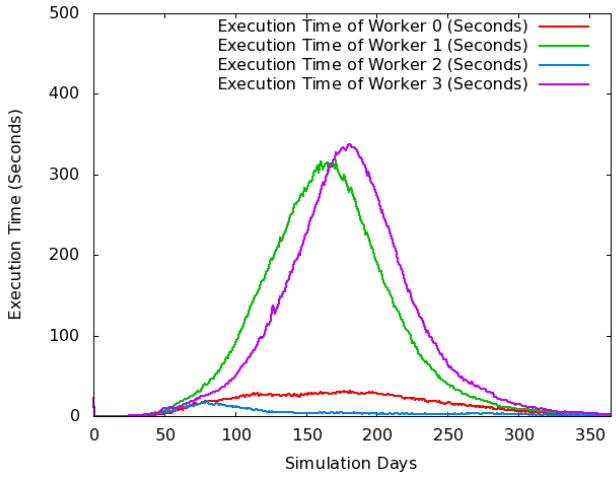


Figure 8. Execution Time of Workers 0-3 in 4 Workers Experiment

However, as workers are added to increase concurrency, the imbalances increase. As discussed earlier, the simulation is synchronized based on the passage of simulation days, *i.e.*, a worker will not advance to processing the next simulation day until *all* workers have finished processing that simulation day. In other words, the worker that has the highest computational load governs the progression of individual days in the simulation. In our 4-worker setup depicted in **Figure 8**, Worker 1 and Worker 3 did most of the work while Worker 0 and Worker 2 spent most of their time waiting for Worker 1 and Worker 3 to finish processing a given simulation day. So in this case the performance of the 4-worker setup is almost the same as that of the 2-worker setup. Figure 5 shows an overall trend that as the number of workers increases two-fold there is a corresponding 1.5 fold speed-up.

In the static split strategy we found that once the distribution of computational density became uneven at a worker it stayed that way, *i.e.*, the heaviest-loaded worker continues to be computationally intensive for most of the simulation.

We also benchmarked the total overhead introduced by the Controller-Worker pattern by tracking the total amount of time that was spent in issuing an advance simulation day event over the course of the simulation. Without the Controller in place, in a completely standalone setting, these advancements would have happened immediately after the last sub-region had indicated that it had finished its simulation day. The total overhead introduced by the Controller-Worker pattern was in the order of 0.4-4.6 seconds for the different scenarios.

B. Dynamic Split Strategy

In the static split strategy, the population of farms was divided among the available workers before the simulation began; each worker managed an equal-sized area (but not necessarily an equal number of farms). This changes in the dynamic split strategy.

In the dynamic split strategy, the Controller tracks the execution time of the workers as the simulation progresses, and targets the slowest worker(s) for partitioning. Over many simulation runs we observed that the execution time for a given day was generally similar to the execution time of the previous day. Based on this observation, the controller determines whether to split a worker by comparing the execution time for the current day to that of the previous day. The decision threshold is important: a split is irreversible in this strategy, so once a worker is split, an additional compute resource is occupied until the simulation finishes. If the decision threshold is too high, compute resources may go unused; if the threshold is too low, all the compute resources may become occupied early, leaving no room for further partitioning when computational imbalances reveal themselves later in the simulation.

Since the splits are irreversible, deciding when to split is a balance between achieving speed-ups for not just the next several simulation days but also throughout the simulation. The value of compute resources changes over time. On the first day of the simulation, there is no reason not to partition the whole region into two. After the first day, the controller should consider partitioning slow workers: the earlier the compute resources are brought into the simulation, the more parallelization is utilized. Thus the decision threshold starts from a very low value. As more of the available compute resources are used, the threshold should become higher and higher. An ideal threshold would ensure that all of the compute resources are working in parallel during the most compute-intensive stage of the simulation, and that the computational load is distributed evenly among workers.

The dynamic split strategy algorithm is depicted below:

$$\texttt{rate} = \frac{\text{activeWorkerNumber}-1}{totalWorkerNumber-2} + 1;$$

```
lowThreshold = LOW_CONSTANT * rate;
highThreshold = HIGH_CONSTANT * rate;
for(i=0;i<activeWorkerNumber;i++){
  if (spareWorkerNumber<0 ||) {
      continue;
  } else if (activeWorkerNumber==1||
         executionTime[i]>highThreshold||
         (executionTime[i]>highThreshold&&
         executionTime[i]>1.5*secondMaxTime)){
     splitWorker(i);
     splitWorkerNumber++;
     spareWorkerNumber--;
  }
}
activeWorkerNumber+=splitWorkerNumber;
```

`LOW_CONSTANT` and `HIGH_CONSTANT` are two parameters for which reasonable values can be obtained experimentally by using a trial run. The constants are scale-specific (number of herds), stable, and do not need to be reset. The value of *rate* is in the interval [1, 2). The primary idea is that the fewer compute resources are available, the harder it is to perform another split.

Since the splits are irreversible, the effects of the dynamic split strategy are not visible – i.e., it is indistinguishable from the static split strategy – with a smaller number of workers. For these experiments, we benchmarked with a 32-worker scenario and a 64-worker scenario. The execution time and load balancing efficiency are shown in **Figure 10**. In these experiments, since the split operations are based on the execution time of the current day, they increase the load balancing efficiency for the next day. However, the computational hotspots move geospatially as the simulation progresses and in this strategy once a split is applied, there is no way to adjust it. So, after the efficiency peak it decreases gradually before stabilizing. Thus even though this solution is better than a static strategy the load balancing issue still persists. In **Figure 11** we can see that the speed-up for 64 workers is about 14; the static scheme achieves only a 7-fold speed-up for the same scenario with 64-workers.

We devised a *load balancing efficiency* metric that captures how balanced the workload is for a given simulation day. **Figure 9** illustrates this idea. The red bars are execution time for each worker. Because of the workers have to advance to the next day in lockstep, workers that finish early have to wait for the others to complete. The blue areas represent the idling time for the workers that finish early. Load balancing efficiency is defined as the red area divided by the area of the whole rectangle. We chose this calculation over simpler measures such as the ratio of the fastest and slowest workers because it gives a better picture of the average utilization ratio of resources. Consider a system with 100 workers in which the fastest worker takes almost zero time and the other workers all take the same amount of time. A ratio of fastest-to-slowest runtime would rate the efficiency of this outcome at almost zero, whereas our calculation would rate the efficiency at 99%.

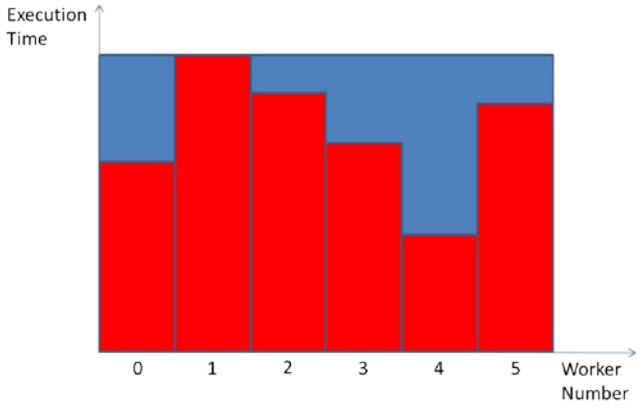


Figure 9. Load Balancing Efficiency Example

In Figure 10, we show not only the execution time of each simulation day but also its load balancing efficiency. The dynamic split strategy performs better than the static split strategy. However, it introduces a new problem. The irreversible split process can activate many workers that end up mostly idling during the simulation run. The next section describes our solution to this problem.

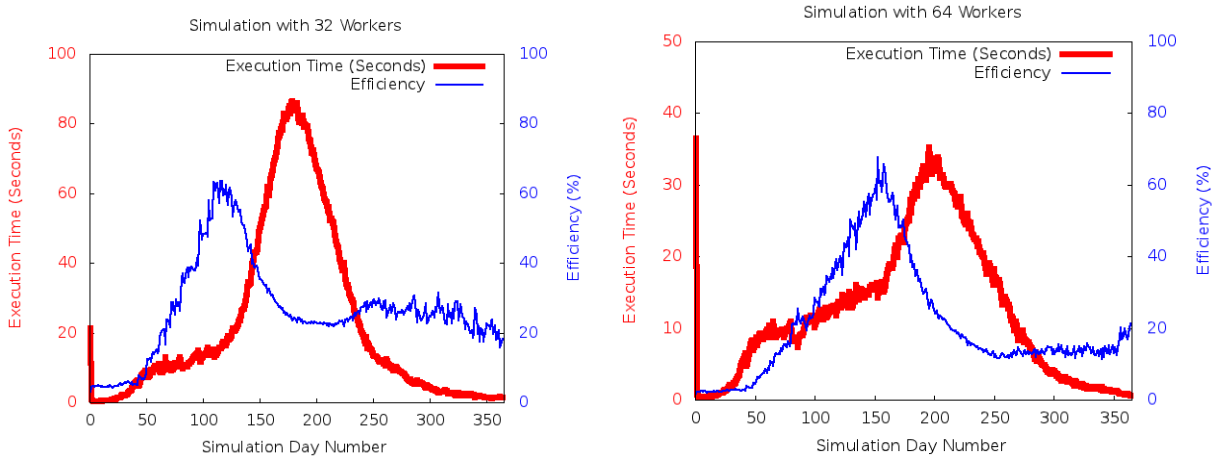


Figure 10. Execution Time and Load Balancing Efficiency for 32 and 64 Workers

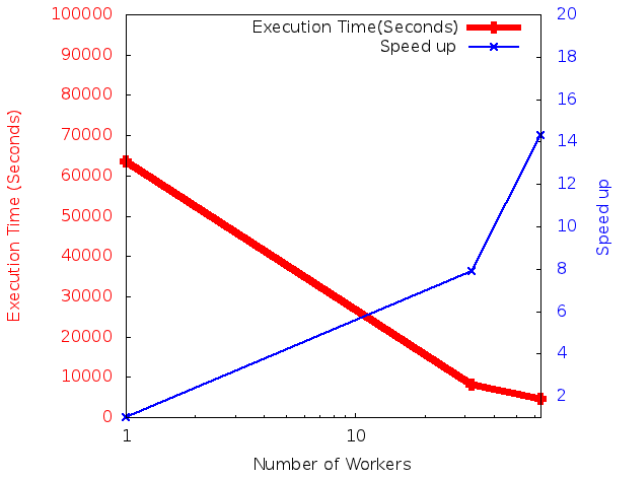


Figure 11. Speed-up for Dynamic Split without Merge Strategy

C. Dynamic Split+Merge Strategy

The dynamic split+merge strategy adds a merge operation to reverse split operations. In this strategy, we partition heavily-loaded workers into two and merge two adjacent lightly-loaded workers into one. Here, the geospatial area managed by workers varies dynamically throughout the simulation, and the creation of workers is not as constrained as in the dynamic split strategy. Early in the simulation, we partition the work quickly so that more parallelization is utilized. As the simulation progresses, the heavily-loaded workers are partitioned into smaller and smaller pieces and the lightly-loaded workers are merged into larger and larger regions. In the end, each worker does almost the same amount of the work, and the system as a whole will achieve a dynamic equilibrium status. The system will even be able to self-adjust as the stochastic nature of the disease spread results in movements of the computational hot spots.

The algorithm for the dynamic split+merge strategy is:

```
for(i=0;i<activeWorkerNumber-1;i++){
  pos = activeList.get(i).getWorkerNum();
  posAd = activeList.get(i+1).getWorkerNum();
  if(executionTime[pos]+executionTime[posAd]<mergeThreshold*maxTime)
      mergeThemTogether(pos, posAd);
}
sortByExecutionTime(activeList);
for(i=0;i<activeWorkerNumber-1;i++){
  if(activeList.getNumberOfSpareWorkers()==0)
      break;
  if(executionTime[pos]>splitThreshold *maxTime)
      splitWorker(pos);
}
```

The merge threshold and split threshold are constants for a given run of the simulation. In practice, we set them as 75% to get the best performance. When the sum of execution times of two adjacent workers is less than 75% of the execution time of the slowest worker, we merge them. Conversely, if a worker takes more than 75% of the execution time of the slowest worker and there exists at least one spare worker in the system, we split that worker (giving preference to the very slowest worker, then the next-slowest worker, and so on).

The performance of the dynamic split+merge strategy is shown in **Figure 12**. We can see that the speedup is almost linear: when we double the number of workers in the system, we achieve a speedup of about 1.8. A setup with only 8 workers does not perform as well as setups with more workers, because fewer workers do not provide enough flexibility to quickly balance the workload via split and merge operations. The speedup begins to level off with higher numbers of workers, because the overheads for synchronization and communication play an increasingly important role.

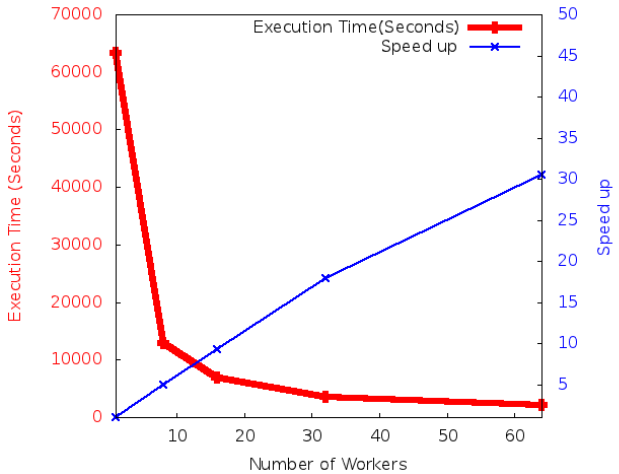


Figure 12. Speed-up for Dynamic Split+Merge Strategy

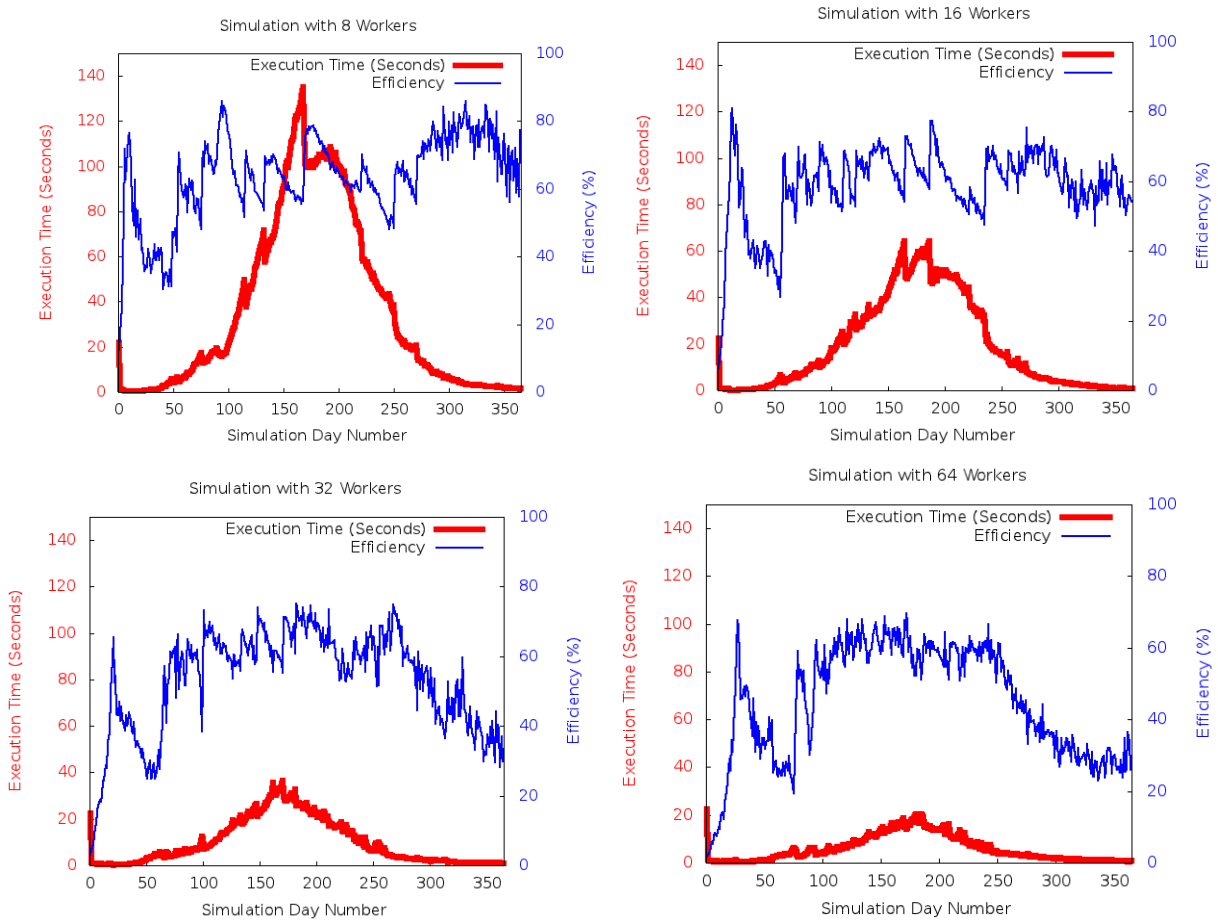


Figure 13. Execution Time and Load Balancing Efficiency for 8 through 64 Workers

Figure 13 shows the execution time and load balancing efficiency for an 8-worker setup through a 64-worker setup. From **Figure 13** we can see that the load balancing efficiencies are about 50% to 70% during the busiest period. At the beginning and ending periods of the simulation, however, the efficiency may drop to a lower rate. This is because the Controller will not merge workers when the slowest worker is faster than a given threshold, to avoid the overhead of the merge operation.

We can see that the system is more responsive when there are more workers. In the 8 workers scenario in **Figure 13**, there are several sections where the efficiency is decreasing for many simulation days, then bounces up over the next few days. However, in the 64 workers scenario in **Figure 13**, when the efficiency drops, it will improve for the next simulation day.

Because the split and merge operations occur between two simulation days, these operations introduce some overhead. This overhead is proportional to the number of events occurring in the simulation, not how many workers are active in the system. In the 64-worker setup, the execution time without split and merge is 2005 seconds and the total execution time including split and merge functions is 2078 seconds. The overhead is 73 seconds. As more workers are used, the execution time will decrease and the overhead will become more dominant.

The multi-worker version of NAADSM has slightly more work to do in each day than the original version. The original version never needs to marshal and unmarshal events as the multi-worker version does. Additionally, some computational shortcuts employed in the original (which relied on complete knowledge of the simulation state) cannot be used in the distributed version.

D. Leveraging hyperthreading

In our previous experiments, each 4-core machine had four workers on it with each worker executing on a separate core. Here we enabled hyperthreading on these cores (each core has two execution pipelines) and ran eight workers on each machine. Increasing the number of workers on a machine beyond 10 introduced memory bottlenecks and interference between computations that degraded performance.

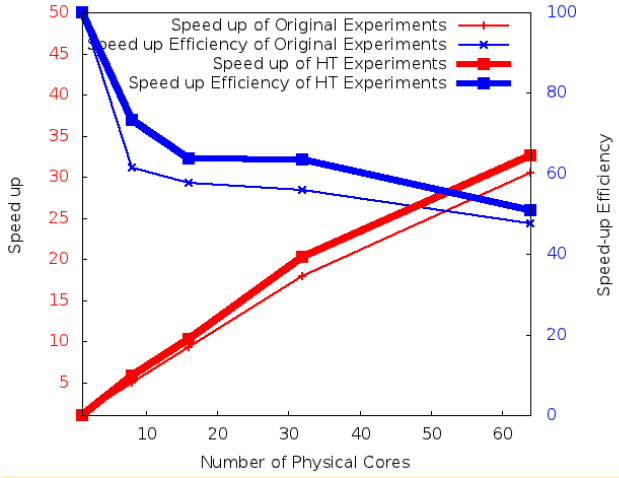


Figure 14. Speed-up of when enabling hyperthreading

Figure 14 shows our results with eight workers on a machine, where each core supports two workers. Enabling hyperthreading and increasing the number of workers improves speedup and efficiency. Moreover, when there are eight cores, the performance gain is more obvious than the other scenarios. When hyperthreads are disabled, there are only four workers on one machine; for a fixed set of machines, there may not be enough workers for the algorithm to respond and balance the load in a timely fashion. Enabling hyperthreading increases the number of workers, and correspondingly the CPU utilization, to better balance the load with the same number of available cores. In the case of 64 cores for hyperthreading the performance improvement is not substantial because the communication and synchronization overheads between the workers increases with a large number of workers resulting in reduced performance gain.

V. Related Work

There are several approaches to managing distributed executions in distributed settings such as grids and clouds. The MapReduce framework [3] developed by Google is particularly suitable for orchestration of large information retrieval tasks. Hadoop [4] is an implementation of the MapReduce framework with support for managing, tracking, and relaunching tasks that comprise individual jobs. An approach to make Hadoop more reliable in a setting involving heterogeneous resources with diverse capabilities is proposed in [5]. Microsoft's Dryad provides a powerful framework for using computational graphs to evaluate complex SQL queries over a distributed database [6]. However, basic MapReduce and Dryad do not allow multiple, successive rounds of execution without storing intermediate state to disk. In Granules computations can be expressed as MapReduce and also as graphs that have cycles in them. Conceptually, the Controller-Worker pattern that we have used can be thought of as MapReduce stage with a feedback loop.

Parallel discrete event simulation is one approach to decrease the execution time; this is especially true for time constrained applications such as weather forecasts, and traffic simulations [7]. These have been used in settings such as reliably modeling of chemical plants [8] and modeling urban-traffic congestion [9]. Benveniste et al deal with the diagnosis of an asynchronous event system, which is motivated by the problem of event correlation in telecommunications network management [10]. In each of these cases, the simulation is coded as a parallel application from the very beginning. To the best of our knowledge, this is the first time a general-purpose distributed stream processing engine has been used to orchestrate a discrete event simulation.

There are some optimizations for discrete event simulations. Thulasidasan et al have focused on the load balancing optimization [11]. They have explored geographical partitions based on the following strategies: even distribution of entities, computational load divisibility, and the scatter partition strategy. Their results show that the

scatter partition strategy outperforms the other two strategies. However, in our problem, because we need to preserve geographical contiguousness of subregions we cannot use the scatter partition strategy; while their static load distribution strategies are unsuitable for coping with situations where the hotspots emerge stochastically. Deelman et al focus on a strategy for dividing regions, computing load metrics, and process migration for a discrete event simulation [12]. In their settings, they treat the whole region as a circle and allow logically defined adjacent regions to pass load to each other. In each simulation time unit, the computational load for the next time unit is known and this makes it easy to pass a particular amount of work to the neighbors – this allows the system to ensure that each logical process does exactly the same amount of work for a given simulation time unit. However, in our case, deterministically deriving the computational load for each logical process is not possible given the stochastic nature of the simulation.

Load balancing is an important issue when orchestrating processing on a collection of machines. Both static load balancing and dynamic load balancing strategies are widely used. Shivle et al describe several static load balancing strategies [15]. However, these strategies are based on known subtasks, which simplifies the problem considerably. There have been several attempts to exploit dynamic load balancing strategies, but these strategies require significant computational overheads during execution time to perform such rebalancing [16, 17, 18, and 19]. However, in our problem, since the execution time for each day is relatively short we cannot afford additional overheads for these computations.

VI. Conclusions & Future Work

In this paper, we described our framework for orchestrating the execution of a NAADSM simulation on a distributed collection of machines. Our results have demonstrated the feasibility of using Granules to perform such orchestrations. We presented results from static split, dynamic split, and dynamic split+merge load balancing schemes. The static split scheme resulted in a situation where the region with the highest computation footprint determined the overall speed of the simulation. The dynamic split scheme performed better; however, it could not adapt to the computational workload changing over time during the execution. The dynamic split+merge scheme overcame the bottlenecks of the other two solutions, made better use of resources, and achieved higher efficiency during the busiest time in the simulation. However, in situations where few worker machines are available, the dynamic split+merge scheme does not react quickly enough to a changing workload. Future work will examine the possibility of fast migration of parts of the workload to adjacent workers.

Our future effort will be focused on leveraging learning based-predictions to better balance the workload. Learning based-prediction is widely used in load balancing area [20, 21]. The challenge will be to keep the overheads for rebalancing low as the loads continually evolve.

Acknowledgment

This research has been supported by funding (HSHQDC-10- C-130) from the US Department of Homeland Security's Long Range program.

References

[1] Harvey, N., Reeves, A., Schoenbaum, M.A., Zagmutt-Vergara, F.J., Dubé, C., Hill, A.E., Corso, B.A., McNab, W.B., Cartwright, C.I., & Salman, M.D. (2007). The North American Animal Disease Spread Model: A simulation model to assist decision making in evaluating animal disease incursions. Preventive Veterinary Medicine, 82, 176-197.

[2] Shrideep Pallickara, Jaliya Ekanayake and Geoffrey Fox. Granules: A Lightweight, Streaming Runtime for Cloud Computing With Support for Map-Reduce. Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2009). New Orleans, LA.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Proc. 6th Conf. Symp. Operating System Design and Implementation (OSDI 04), Usenix Assoc., 2004, pp. 137-150.

[4] Apache hadoop website. http://hadoop.apache.org/, April 2010.

[5] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., and Stoica, I. Improving MapReduce Performance in Heterogeneous Environments. In Proceedings of OSDI. 2008, 29-42.

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," presented at the Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, Lisbon, Portugal, 2007

[7] Richard M. Fujimoto. 1990. Parallel discrete event simulation. Commun. ACM 33, 10 (October 1990), 30-53.

[8] Bikram Sharda and Scott J. Bury. 2008. A discrete event simulation model for reliability modeling of a chemical plant. In Proceedings of the 40th Conference on Winter Simulation (WSC '08), Scott Mason, Ray Hill, Lars Mönch, and Oliver Rose (Eds.). Winter Simulation Conference 1736-1740.

[9] Thulasidasan, S.; Kasiviswanathan, S.; Eidenbenz, S.; Galli, E.; Mniszewski, S.; Romero, P.; , "Designing systems for large-scale, discrete-event simulations: Experiences with the FastTrans parallel microsimulator," High Performance Computing (HiPC), 2009 International Conference on , vol., no., pp.428-437, 16-19 Dec. 2009

[10] Benveniste, A.; Fabre, E.; Haar, S.; Jard, C.; , "Diagnosis of asynchronous discrete-event systems: a net unfolding approach," Automatic Control, IEEE Transactions on , vol.48, no.5, pp. 714- 727, May 2003

[11] Thulasidasan, S.; Kasiviswanathan, S.P.; Eidenbenz, S.; Romero, P.; , "Explicit Spatial Scattering for Load Balancing in Conservatively Synchronized Parallel Discrete Event Simulations," Principles of Advanced and Distributed Simulation (PADS), 2010 IEEE Workshop on , vol., no., pp.1-8, 17-19 May 2010

[12] Deelman, E.; Szymanski, B.K.; , "Dynamic load balancing in parallel discrete event simulation for spatially explicit problems," Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on , vol., no., pp.46-53, 26-29 May 1998.

[13] Ericson, K.; Pallickara, S.; Anderson, C.W.; "Analyzing Electroencephalograms Using Cloud Computing Techniques," Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on , vol., no., pp.185-192, Nov. 30 2010-Dec. 3 2010

[14] Ericson, K.; Pallickara, S.; Anderson, C.W.; Handwriting Recognition using a Cloud Runtime.Colorado Celebration of Women in Computing 2010. Golden, USA.

[15] S. Shivle, R. Castain, H.J. Siegel, A.A. Maciejewski, T. Banka, K. Chindam, S. Dussinger, P. Pichumani, P. Satyasekaran, W. Saylor, D. Sendek, J. Sousa, J. Sridharan, P. Sugavanam, and J. Velazco, "Static mapping of subtasks in a heterogeneous ad hoc grid environment," in Proc. of 13th HCW Workshop, IEEE Computer Society, 2004.

[16] Dejan S. MiloJicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. ACM Computing Surveys, 32(3):241–299, September 2000

[17] Penmatsa, S.; Chronopoulos, A.T.; , "Dynamic Multi-User Load Balancing in Distributed Systems," Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International , vol., no., pp.1-10, 26-30 March 2007

[18] Miguel Campos, L.; Scherson, I.; , "Rate of change load balancing in distributed and parallel systems ," Parallel and Distributed Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP, IEEE Computer Society 1999. Proceedings, vol., no., pp.701-707, 12-16 Apr 1999

[19] Dhakal, S.; Hayat, M.M.; Pezoa, J.E.; Cundong Yang; Bader, D.A.; , "Dynamic Load Balancing in Distributed Systems in the Presence of Delays: A Regeneration-Theory Approach," Parallel and Distributed Systems, IEEE Transactions on , vol.18, no.4, pp.485-497, April 2007

[20] Jun Wang; Jian-wen Chen; Yong-liang Wang; Di Zheng; , "Intelligent Load Balancing Strategies for Complex Distributed Simulation Applications," Computational Intelligence and Security, 2009. CIS '09. International Conference on , vol.2, no., pp.182-186, 11-14 Dec. 2009

[21] Dantas, M.A.R.; Pinto, A.R.; , "A load balancing approach based on a genetic machine learning algorithm," High Performance Computing Systems and Applications, 2005. HPCS 2005, IEEE Computer Society 2005. 19th International Symposium on , vol., no., pp. 124- 130, 15-18 May 2005

[22] Kathleen Ericson, Shrideep Pallickara: Adaptive heterogeneous language support within a cloud runtime. *Future Generation Comp. Syst*. 28(1): 128-135 (2012)

[23] Ericson, K.; Pallickara, S.; , "On the Performance of Distributed Clustering Algorithms in File and Streaming Processing Systems, *Fourth IEEE/ACM International Conference on Utility and Cloud Computing* , vol., no., pp.33-40, 5-8 Dec. 2011