Learning Based 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 (DES) are used in situations where we need to understand or describe complex phenomena. This paper describes an algorithm for dynamic orchestration of stochastic DES. To cope with long execution times in stochastic DES settings, we use MapReduce to achieve concurrent processing of the simulation on a distributed collection of machines. The proposed algorithm proactively targets imbalances between subtasks of the simulation. It achieves this by accurately predicting future execution times for map instances and apportioning processing workloads while accounting for the overheads associated with the apportioning. Our empirical benchmarks demonstrate the suitability of our scheme.**

Keywords: *discrete event simulations; MapReduce; load balancing; proactive schemes; learning based orchestration*

I. INTRODUCTION

Discrete event simulations (DES) are widely used in different domains such as weather forecasting, transportation modeling, and epidemiology. DES are used in modeling complex phenomena by identifying *entities* of interest and all possible interactions (or *events*) that can take place between them. Stochastic DES is a special type of DES where such interactions are based on probability density functions associated with these interactions. Subject-matter experts create these probability density functions based on previously observed phenomena. Unlike deterministic DES, a stochastic DES results in a slightly different *outcome* every time it executes even with the same set of starting parameters. This is because the interactions between entities are stochastic. Since DES capture all possible interactions between entities, their expressiveness and fidelity comes at the price of long execution times.

The DES we consider in this paper is a stochastic DES that models disease spread. To cope with long execution times in stochastic DES settings, we use MapReduce to achieve concurrent processing of the simulation on a distributed collection of machines. The geographic area being modeled is split into a set of contiguous sub-regions, each of which is managed by a map instance. Each mapper generates outputs at the end of each *simulation day*. The outputs represent progression of the disease in the sub-region managed by a mapper. These outputs are processed by a reducer that serves as a coordinator. The coordinator is responsible for state synchronization between the mappers between successive simulation days. The state synchronization includes information about depletion of vaccines, movement controls including quarantines, and the number of infections.

The simulation executes as an iterative MapReduce stage with multiple, successive rounds of execution. Just as in the traditional MapReduce framework, during the Map-phase execution, individual mappers do not communicate with each other. Each mapper functions autonomously without blocking on input/communications from some other mapper.

There are some differences from the traditional MapReduce framework. The processing footprint at each mapper may be slightly different. The stochastic nature of the disease spread, and the concomitant processing loads, ensures that imbalances are very likely. The coordinator node (which serves as the reducer) is responsible for apportioning and migrating processing loads to mitigate imbalances; i.e., one of the mappers may take on processing loads that belong to another mapper. This reshuffling of processing loads is done during state synchronization at the end of the simulation day. The process repeats itself in iterative MapReduce phases until the simulation terminates when the disease has been eliminated. Finally, our map and reduce instances are *stateful* i.e. the result of processing depends on the state built up within the computation.

Synchronization points play an important role in the correctness and speed of the simulation. State synchronization allows interactions to be consistent and similar to what happens in a sequential run of the simulation. The speed implications are based on the compute imbalances between mappers. Each mapper must complete the particular simulation day before: (1) the reducer can generate a global state, (2) perform state synchronization, and (3) initiate the next iteration of the MapReduce state. Due to these synchronizations, the simulation is only as fast as the slowest mapper between each pair of synchronization points.

The research objective of this paper is to design a dynamic orchestration scheme for discrete event simulations. We achieve this by learning from previous executions of the simulation, and incorporating this knowledge into decisions that guide the distributed orchestration.

A. Research Challenges

Challenges in accomplishing learning based distributed orchestration of DES include:

1. Imbalances in event generation at mappers: Since the simulation is stochastic there may be imbalances in event generation. If disease is particularly active in a sub-region, the number of events that are generated within the corresponding mapper may be disproportionately high. The greater the number of events, the greater the processing overhead, i.e., the mapper could be a performance hotspot.
2. Stochastic movement of performance hotspots: As disease moves through the geographical area being modeled, the performance hotspots move along with it. Disease spread is based on stochastic interactions between entities.
3. Reactive approaches can be limiting. Since performance hotspots are short-lived and move continually, a reactive approach that attempts to address imbalance after the fact may not be effective. What is needed is a proactive approach that prevents imbalances from occurring. This would need to be based on predicting execution times for mappers based on feature vectors extracted from the simulation.
4. Prediction accuracy determines performance. In proactive approaches, prediction accuracy for execution times determines the efficiency of the orchestration. We have to consider not just the average prediction accuracy, but also the worst-case prediction accuracy.
5. Speed of load balancing operation. Load imbalance detection and mitigation are in the critical path of the simulation. Overheads that are greater than the accrued performance gain are inefficient no matter how balanced the mappers' loads are.

B. Research Questions

The challenges described above guide the research questions that we will explore. These include:

1. *What is the composition of the feature vector that we use to make predictions about execution times?* Such feature identification is a precursor to subsequent training of the learning structures and predictions.
2. *How can we balance the competing pulls of knowledge management and prediction accuracy?* The more detail we maintain about the simulation, the more accurate our predictions are likely to be. However, there are costs associated with maintaining and updating this information. Prediction accuracy should not be at the cost of delays in making the predictions.
3. *How can we balance the competing pulls of load balancing efficiency versus performance overheads?* Complex load balancing mechanisms that minimize imbalances may introduce unacceptable performance costs. Since load balancing operations are in the critical path of the simulation execution, it is important to ensure that the costs for load balancing do not outpace any gains due to alleviating imbalances.

C. Approach Summary

In this paper we present the design of a framework to reduce the execution time of a stochastic DES. We accomplish this via distributed orchestration of the DES. We express the DES as an iterative MapReduce pipeline with each mapper responsible for managing a certain geographical scope of the region being modeled. Given the large number of synchronization points as well as the imbalances that exist between mappers, load balancing decisions are key to ensuring faster completion times. Our approach is based on a proactive load-balancing scheme that tries to prevent imbalances from occurring in the first place. This is predicated on accurately predicting future execution times for each mapper and making load-balancing decisions that mitigate future imbalances.

Our load prediction approach uses several simulation variables as elements of the feature vector used for predictions. Elements of our feature vector track several variables that capture the prevalence and intensity of the disease outbreak. Some of these variables are used in programmatic constructs such as loop variables, recursion depth, and so on. Since the orchestration and load balancing decisions are based on these predictions, a key requirement is prediction accuracy. In our work, we use Artificial Neural Networks (ANN) [12] to predict the execution time. We found that during training of these neural networks, the data points relating to execution of the simulation are not distributed uniformly. To address this we have multiple ANNs and organize them into a Multi-Stage Neural Network (MSNN) [13]. The choice of the particular ANN with the MSNN for predictions in based on the execution time of the previous simulation day. Our experiments demonstrate the suitability of our feature vector and the MSNN in predicting execution times for mappers accurately.

Load balancing decisions must target imbalances among mappers. We achieve this by reapportioning load between the mappers. This reapportioning has costs associated with it, including state-transfer and creation of additional map instances. To minimize these overheads, we designed a P2P message passing mechanism among mappers along with data compression to reduce communication overheads.

However, pursuing prediction accuracy and efficiency in targeting imbalances without also tracking overheads can be problematic. In some situations overheads can dominate execution time and, rather than achieving a speed-up as more resources are added, performance degradations occur. Our work relies on a lightweight scheme for predicting execution times, and ensuring that load-balancing decisions including reapportioning are done only when the expected gains in execution times outpace the overheads.

D. Paper Contributions

Our algorithm for orchestration of processing workloads learns from the computational footprints of disease outbreaks to inform load balancing decisions. The algorithm focuses on proactive mitigation of load imbalances. The approach combines execution time forecasts and imbalance mitigation while accounting for overheads associated with reapportioning workloads. Our proposed algorithm is applicable to other DES as well; our approach to execution time prediction copes with situations where some of the elements within the feature vector contribute more heavily to execution times than others.

E. Paper Organization

This paper is organized as follows. In section II, we describe the background for our work. We address our prior work and inefficiencies in section III; this also provides the motivation for the current work. Section IV describes the elements that comprise our algorithm. Experimental results are described in section V. In section VI, we discuss related work. Finally, we give our conclusions in section VII.

II. Background

A. North American Animal Disease Spread Model (NAADSM)

NAADSM is a simulator for the spread and control of livestock diseases [11]. Diseases simulated within NAADSM include foot-and-mouth disease (FMD), exotic Newcastle disease, pseudo rabies, and avian influenza. It was developed collaboratively by the US 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 is a stochastic discrete event simulation.

Examples of *events* in the simulation are exposures, detections, quarantine, test results, vaccination, and culling. The probability of various events occurring is governed by probability density functions (PDFs), which are input by the modeler based on scientific evidence and observations made by epidemiologists. Figure 1 sketches the activities that can occur on each simulation day.

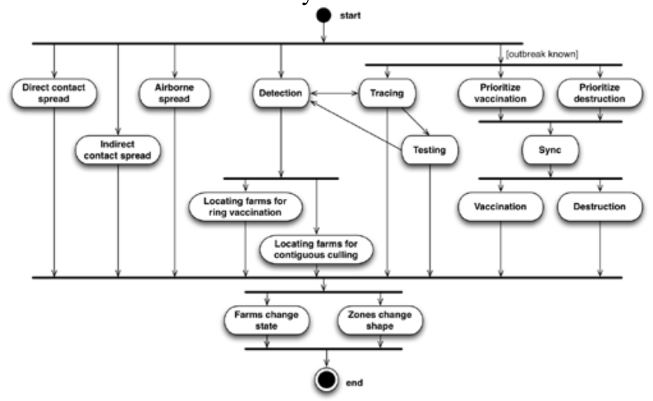

Figure 1: Activity diagram for a single simulation day.

The fundamental unit of disease 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 is a lightweight streaming runtime [10]. Granules supports two of the dominant models for cloud computing: Map-Reduce [1] and directed acyclic dataflow graphs [15]. Granules extends these models by including support for streaming datasets, stateful computations, and support for cycles within these graphs. Granules computations can be developed in C, C#, C++, Java, Python or R.

Computations can retain state across executions and have built-in lifecycle support. Users can activate computations either periodically or when data is available and enforce restrictions on the number of times that a computation can be executed. Users can programmatically specify a scheduling strategy for computations that is a combination along these dimensions. Granules manages the lifecycle and finite state machine associated with computations.

Granules maximizes utilization of a resource by interleaving the concurrent execution of thousands of computation tasks by scheduling tasks for execution only when all its scheduling constraints have been satisfied, and keeping them dormant otherwise. The system makes no assumptions about the type of resource hosting these computations: individual resources could be stand-alone workstations or nodes within a cluster, a supercomputer, a grid, or a data center (public or private cloud). Resources could also be virtualized machines.

C. NAADSM with Granules

NAADSM is adapted to work within Granules by dividing the geographic area being modeled into a set of contiguous sub-regions, each of which is treated as one Granules computation. The computations communicate once per simulation day, each computation sending out a subset of the events it generated that day. The events a computation sends out are those that could affect the sub-regions managed by other computations. For example, a movement of animals that crosses between sub-regions must be communicated. The first detection of disease inside the sub-region must be communicated, because it may initiate movement slowdowns across the entire population. Establishment of a disease control zone must be communicated, because the control zone boundary may overlap the border between sub-regions. Quarantining or culling of farms must be communicated, because neighboring computations need to know which farms are no longer capable of receiving contacts from others.

Once all of the computations have exchanged these update messages, they all proceed with the subsequent simulation day.

III. Prior Work

In our previous work [14], we designed several load balancing algorithms. Among these algorithms, dynamic split and merge (DSM) was the best solution for most situations. In DSM, the system checks the execution time of each mapper on each simulation day. If there are any currently unused (spare) mappers, the slowest mapper(s) will be split to make use of the spare mappers. In the meantime, if the total execution time of two adjacent mappers is less than 70% of the execution time of the slowest mapper, the system will merge them together. Using this algorithm, the system reaches a dynamic equilibrium.

However, in some circumstances DSM suffers from a performance bottleneck. We divide the simulation work by geography, with each mapper managing a geographically contiguous area. Therefore when we consider merging fast mappers, we can only merge two mappers that manage *adjacent* areas. There might be an idling mapper between two slow mappers but we cannot merge it. For example, consider the case of 64 mappers where the execution time follows the pattern 70 seconds, 0 seconds, 70 seconds, 0 seconds, etc., plus one mapper that takes 100 seconds. In this case, the load balancing efficiency is only about 35%, and because the fast mappers are not adjacent to each other, DSM cannot improve the performance.

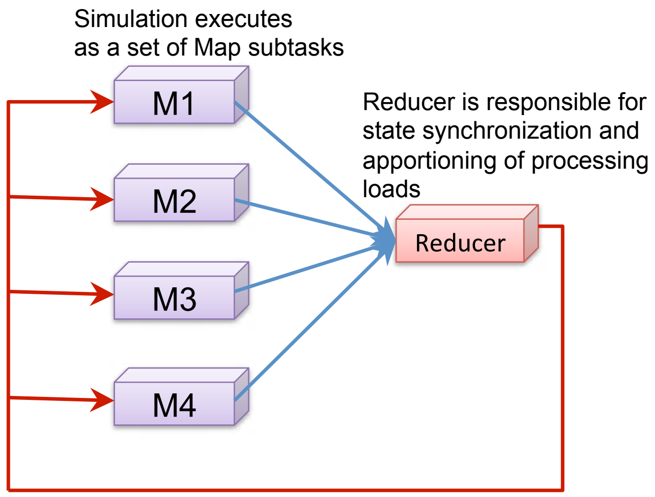
Simulation executes as a set of Map subtasks
M1
M2
M3
M4
Reducer is responsible for state synchronization and apportioning of processing loads
Reducer

Figure 2: Architecture Overview. The entire simulation executes as an iterative MapReduce stage.

There are many elements to this algorithm that could be tuned. The criteria for split and merge operations are simply the execution times from the previous simulation day, since they are generally close to the execution times of the next simulation day. When a mapper is split, the split is done such that the two new mappers manage either the same amount of area *or* the same number of farms; these approaches split the workload roughly in two, but not exactly. Communication overhead starts to impact the performance as the number of mappers increases, but we did not explore ways to reduce the overhead. Tuning these elements would not overcome the bottleneck in the algorithm. In our more new algorithms, however, we do consider these potential areas for improvement.

IV. Design of Current Work

Our architecture is depicted in Figure 2. All aspects relating to load balancing decisions are made at the reducer.

A. *Proactive Apportioning of Workloads (PAW) Algorithm*

Our PAW algorithm addresses deficiencies in the DSM algorithm. If imbalances appear, this algorithm merges the entire workload together and splits it equally among all the mappers. Ideally, the workload will be completely or close to completely balanced in each simulation day. The PAW algorithm eliminates the situation where the load is imbalanced but the system is unable to alleviate the imbalances. An apportioning operation is depicted in Figure 3.

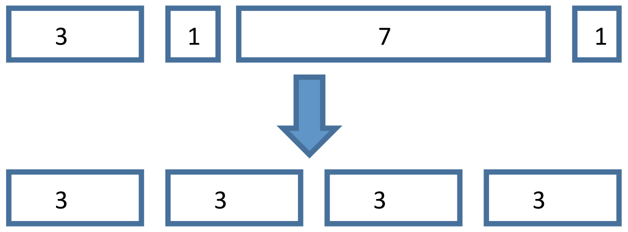

Figure 3: Example of the PAW algorithm. Each blue block represents a mapper. The number within the blocks represents the execution time. The PAW algorithm balances the execution time to minimize the waiting time of most of the workers.

There are several important considerations related to performance. First, the overhead of the apportioning operation must be small. If the operation achieves a 3 second performance gain but adds 20 seconds of overhead, it is counter-productive. In the DSM algorithm, the overhead of one load balancing operation was about 0.3 seconds, which was acceptable when a simulation day lasts more than 10 seconds. However, splits and merges were occasional operations involving a few mappers. In contrast, in the PAW algorithm, all of the mappers perform the apportioning operation at the same time. To avoid hitting network bandwidth limits, we must find ways to reduce the communication time.

The second key consideration for the PAW algorithm is accurate prediction of execution time. In the DSM algorithm, the accuracy did not need to be high; we used the execution time of the previous simulation day as a very rough prediction of the next day's execution time. However, in the PAW algorithm, since each load balancing operation involves so many mappers, any prediction error may accumulate. For this algorithm, we used an artificial neural network (ANN) and multi-stage neural network (MSNN) to predict the execution time. The details of these prediction mechanisms are introduced in IV.C and IV.D.

The third key consideration for the PAW algorithm is the split mechanism. For instance, if we want to split one mapper with a 40%-60% split, how should we do this? In the DSM algorithm, we tried splitting the mapper such that the geographical area managed was divided 40/60, and we also tried splitting the mapper such that the number of farms managed was divided 40/60. These are just approximations to truly splitting the workload 40/60. This drove us to develop a better mechanism for the PAW algorithm.

B. *Communication Message Reduction and Compression*

Communication overhead is an important problem in the PAW algorithm because all of the mappers pass messages to controller, and the controller passes messages back to the mappers, when an apportioning operation needs to happen. The bandwidth of the network may become a bottleneck. In our particular case, before doing any message reduction or compression, each apportioning message is about 25Mbytes. The message is sent from the mappers to the controller and from the controller to the mappers. Since there are 64 mappers, the total message size is 64*25*2 = 3.2Gbytes. Our bandwidth is 128Mbytes per second, so one apportioning operation takes 25 seconds, which is unacceptable: we must reduce the communication overhead.

We apply both message reduction and compression mechanisms. In the DSM algorithm, a mapper may go inactive at the end of one simulation day (because it is merged with another mapper) and then be re-activated on a later simulation day. When the mapper is re-activated, it must be restored with all of the state information it needs to return to participating in the running simulation—which may be a considerable amount of state information. Specifically, the state information would include the current disease state of every farm in the area managed, countdowns to state transitions, some state information about farms *outside* of the area managed (enough to know which can or cannot receive contacts), records of past contacts among farms (in case the simulated disease control authorities need to do "tracing", or following the path of infection backwards from a detected infected farm), the current shape of the disease control zones, and more. In contrast, in the PAW algorithm, all of the mappers are active on every simulation day. When the workload is "apportioned" among the mappers, each mapper can retain much of the state information it already has, and send/receive only as much information as it needs to give up management of certain farms and take on management of certain others. This reduces the size of the apportioning message by more than half.

We also use GZIP compression on the messages. There is a tradeoff between message size and compression overhead: if the message is small, it is not worth the cost of performing the compression. We decide whether to compress, and what the compression ratio is, based on the message size.

C. *Prediction of Execution Times*

It is possible for us to get useful information from the simulation as it runs to help predict the future execution time. Our prediction uses both the execution time of the previous simulation day and some additional features that quantify the workload for a particular simulation day; these are listed in TABLE I.

TABLE I. PREDICTION FEATURES FROM SIMULATION

Feature Number	Feature Content
1	# farms managed by this mapper
2	area in km^2 managed by this mapper
3	# adequate exposures
4	average distance of adequate exposures
5	average latent period for new infections
6	# farms detected as diseased today
7	# contacts traced today
8	# farms destroyed today
9	# destruction tasks queued
10	# adequate exposures by direct contact
11	# adequate exposures by indirect contact
12	# adequate exposures by airborne spread
13	# farms vaccinated today
14	# vaccination tasks queued
15	area in km^2 of the zones

We use an ANN as the predictor because it works well for non-linear data. Based on experiments to get the best prediction accuracy, our propagation method is resilient propagation, there is 1 hidden layer with 30 nodes, and the condition under which training of the learning structure ends is that the error between 10 iterations is less than 1E-6.

D. *Multi-Stage Neural Network (MSNN)*

Acquiring training data for our prediction methods is a challenge because we cannot control how the execution times are distributed. For instance, there might be too many training samples whose execution times (per simulation day) are around 3 seconds but too few samples whose execution times are around 30 seconds. This data distribution may make the model fit better to the short execution time samples and not as well to the long execution times samples. However, the prediction accuracy of the 30-seconds samples is more important than 3-seconds samples, because the longer the execution time, the more time is lost due to prediction errors. We use MSNN to eliminate the influence of the abnormal distribution of training data.

MSNN is a simplified version of a Resource-Allocated Network (RAN). The basic idea of RAN is to cluster the training data and construct a different ANN for each cluster. New incoming training data do not influence the predictions for data which is far away. In our case, we do not need a complex clustering algorithm because the execution time of the previous simulation day is usually similar to the execution time of the next simulation day; we can exploit this to quickly cluster the training data. We construct 4 ANNs based on the execution time of the previous simulation day. For the testing data, we check the execution time of the previous simulation day and use the corresponding ANN to predict the execution time.

E. *Overhead vs. Load Balancing*

We experimented with a split mechanism based on the states of individual farms. From observations of the simulation, the current state of a farm affects how much it adds to the computational load. For example, an infected farm can infect other farms; thus, infected farms add more to execution time than susceptible farms. The type of the farm can be a factor: for example, some types of businesses have higher shipping/receiving rates than others. And while airborne spread may influence only the immediate neighborhood of an infected farm, direct-contact spread can have a wider effect. We constructed a model for all these situations, using regression methods to find the best fit. Based on the model, we can split the area managed by a mapper such that the expected amount of work that will be generated by farms on each side of the split is in the desired proportion.

Maintaining this table of farms could potentially increase the prediction and split accuracy greatly. However, in our benchmark simulation which contains 660,000 farms, the overhead for maintaining this table is more than 10 times the execution time of the simulation itself. Thus, it is prohibitive to maintain such a table even if the load balancing is efficient.

V. EXPERIMENTAL RESULTS

For our experimental benchmarks we used a NAADSM scenario that simulated an outbreak of foot-and-mouth disease (FMD). The population was based on Kansas, USA data, but up-sampled to 660,000 farms, which is the approximate number of FMD-susceptible farms in the 12 Midwest US states.

NAADSM as originally written uses a global random number generator for stochastic decisions. This presents a problem for comparing execution time in various distributed configurations. A run performed on a single mapper will not behave the same way as a run distributed across two mappers, 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 addressed this problem by attaching a separate 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 mappers, this scheme guarantees that given the same starting seed, the same sequence of events will occur in the simulation, regardless of how many mappers the run is distributed across. Note that this was done solely for the purpose of forcing identical runs of the simulations (no matter how many mappers the run is distributed across) so that execution time can be meaningfully compared across experiments. It is *not* a mandatory step for adapting stochastic simulations to a distributed setting.

Experimental results presented in this section were obtained on a 78-node cluster connected by a 1 Gbps link. Individual machines in this cluster run Fedora OS (version 20) and have 4-core, 2.4 GHz CPUs and 12 GB of RAM.

Our performance benchmarks profile our predictions, the communication costs associated with our algorithm, and the speed-ups generated by our algorithm in physical and virtual machine settings.

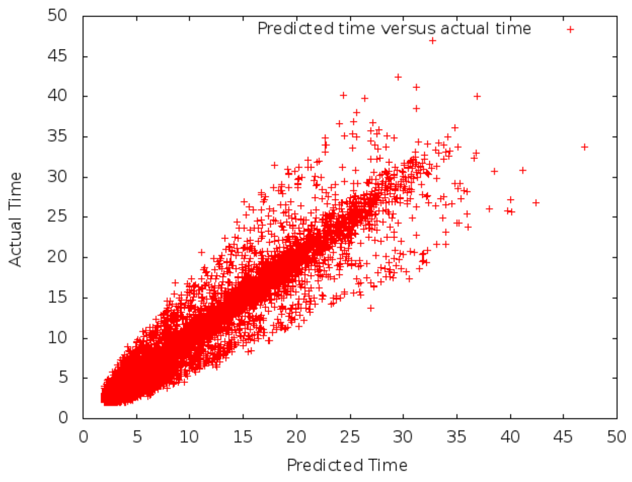

Figure 4 Prediction Accuracy

A. *Prediction Accuracy*

In our work, we use a MSNN to predict the execution time of a simulation day. In our prediction results, our focus is on points where the execution time lasts more than 2 seconds. For points with shorter execution times, prediction errors do not influence the total execution time. Our prediction accuracy is depicted in Figure 4; on average our accuracy is 85.4%. These predictions are fast, on the order of 100 microseconds. The results demonstrate that our scheme produces acceptable accuracy with low overheads.

Our prediction accuracy does not need to be extremely high. First, the overheads associated with getting extremely high prediction accuracies can be quite high. Since the execution time for each simulation day lasts only for seconds, we cannot afford prediction methods with overheads in the orders of seconds even if the predictions are perfect. Second, even if we predict the execution time perfectly, we might still not be able to balance the load perfectly. This is because our re-shuffle mechanism is based on herds' density. For instance, if we decide to partition a sub-region using a 70-30 split, the partitioning will keep 70% of herds in left sub-region and 30% in the right one. However, the execution time is closely but not perfectly aligned with herd density. Thus, even if the 70-30 is the perfect prediction result for partitioning, we might not be able to partition the sub-region to achieve exact execution times.

B. *Compression Performance*

We profiled our use of compression to reduce message sizes during synchronization. We used gzip as our compression algorithm. The synchronization messages are highly amenable to compression. We tracked both the compression rate (defined here as compressed size divided by uncompressed size) and compression time; this is depicted in Figure 5. For larger messages, the compressed message is about 1% of the size of the uncompressed message. The compression time increases almost linearly with the message size. When the message size is about 150Mb, the compression time is about 2 seconds.

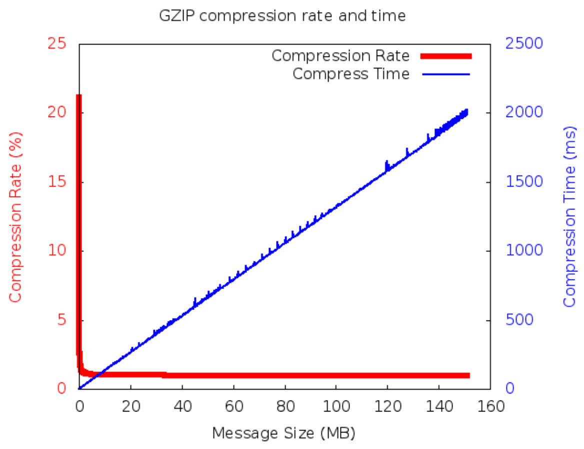

Figure 5 Compression rate and time for synchronization messages

C. *Performance of Simulation*

In comparison with the DSM algorithm, the PAW algorithm fares better in scenarios with fewer mappers. When there are only 2 or 4 mappers, DSM has no way to adjust the load even if the system is aware of the existence of imbalances. However, the PAW algorithm can re-shuffle the load to make it more balanced. The speed-up figure for PAW algorithm is shown in Figure 6.

From Figure 6 we can see that the speed-up for 2, 4, 8 and 16 mappers scenario is ideal. Furthermore, there is even a "super" speed-up for runs with smaller numbers of mappers. The reason for this is the changing ratio of execution time to communication overhead time. If there are only 4 mappers, the split and restore messages for the apportioning operation just need to be passed among 4 mappers; but when the number of mappers increases to 64, the number of messages passed for the apportioning operation increases. With the simulation work divided among 64 mappers, the maximum execution time of a simulation day is only about 24 seconds. The communication overhead time of 3 seconds is more than 10% of the execution time. With fewer mappers, the execution time of the slowest simulation day is always hundreds of seconds.

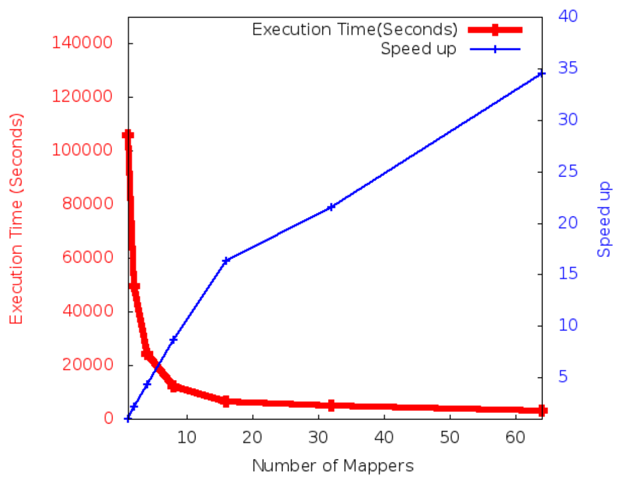

Figure 6 Execution Time and Speed-up for PAW algorithm

Figure 7 compares the two load balancing algorithms. From this figure we can see that the PAW algorithm is always superior to the DSM algorithm, but especially so when the number of mappers is small.

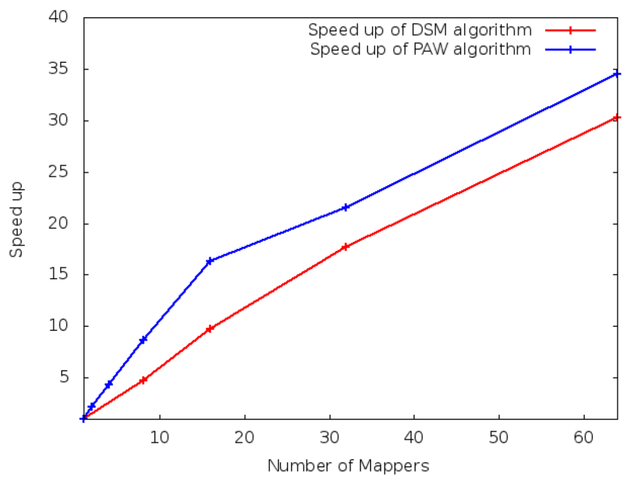

Figure 7 Speed up comparison for 2 load balancing algorithms

D. *Performance on Virtual Machines*

In this paper, we optimized the execution time for one run of simulation. In practice, the stochastic simulation has to be run multiple times and the results averaged to achieve the most likely result. Since the multiple simulations will be executed at the same time, the scale of the cluster will likely be extremely large. In practice, we will run these simulations in cloud settings.

In cloud based settings, resources are typically provisioned as virtual machines. Thus, we also need to test our experiments in virtual machine settings to identify performance and execution issues.

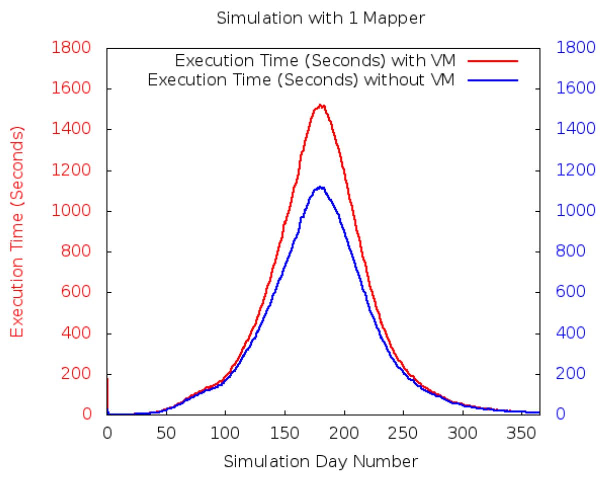

Figure 8 Execution time comparison w/o VMs for 1 mapper scenario

We also profiled the performance of our algorithm in a virtualized environment. Our virtualized environment was set up on the same cluster we used for our earlier tests. We used the KVM hypervisor on Fedora 20.

First, we ran a scenario on 1 mapper in the VM setting and contrasted the execution time with that in a physical machine setting. This is depicted in Figure 8. Virtualization overheads become more noticeable during peak loads of the simulation.

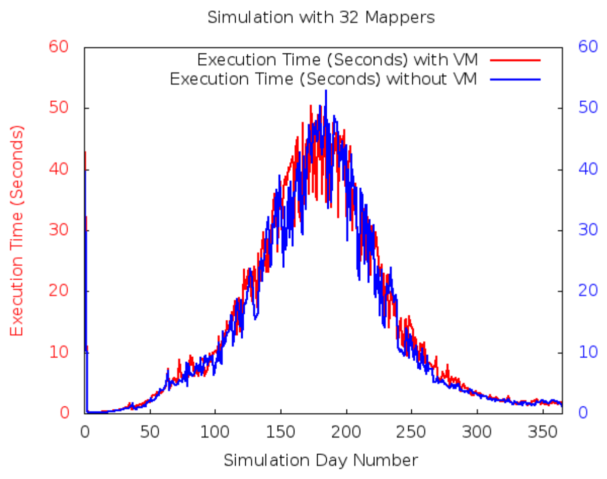

Figure 9 Execution time comparison w/o VMs for 32 mappers scenario

We also executed the scenario with 32 mappers and 64 workers; this is depicted in Figure 9 and Figure 10. The execution time varies considerably from day to day. This indicates that the load balancing mechanism is adjusting the load frequently and the bottleneck now becomes the load balancing efficiency. When comparing virtual machine performance with physical machine performance, we found a general trend towards lower overheads as the number of mappers increased. For example, the overhead (i.e. the increase in execution time for the exact same outbreak with the same number of mappers) for the 1 mapper scenario is 26.5%, but for the 32-mapper scenario this reduces to 3.9%, and for the 64-mapper scenario the overhead is 5.4%. From our experiments we can conclude that a single VM does not perform as well as the physical machine, but when the number of VMs increases, the performance in the VM setting improves in relation to that in the corresponding physical machine setting.

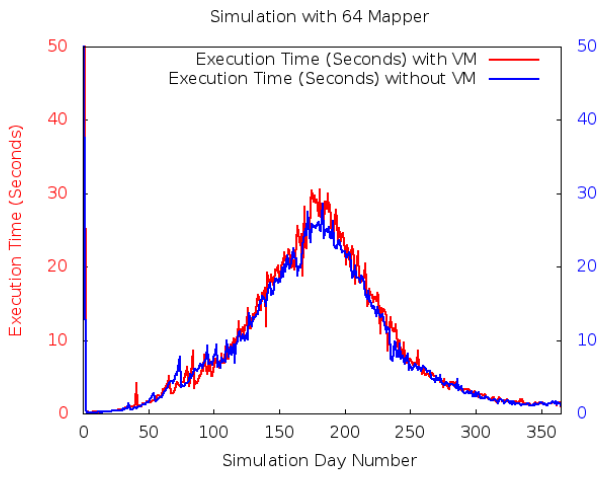

Figure 10 Execution time comparison w/o VMs for 64 mappers scenario

VI. RELATED WORK

Discrete event simulations are widely used in different areas. In atmospheric sciences there is the well-known Regional Atmospheric Modeling System (RAMS) [16] that was developed in 1992. At that time, cluster and cloud computing techniques are not popular and thir parallel orchestrations were based on MPI. In the circuit design area, Bagrodia, R, etc. have designed a parallel simulation environment for complex systems named Parsec [17] in 1998. In this system, the authors consider both conservative and optimistic strategies. This approach also considers the load balancing problem and communication overheads. The Space Surveillance Network (SSN) also uses discrete event simulations and a parallelization mechanism has also been designed [18]. This parallelizes the primary functional areas: Probability of Detection (PoD) which takes around 80% of the execution time. Since the PoD computations are completely independent, the parallelization is rather simple. Another approach is addressed in [19] where a comparison of the conservative and optimistic strategies has been performed. The authors prefer the optimistic strategies with rollback mechanism; however, details of how this was implemented are light.

There are many researchers concentrating on execution in distributed systems. Using frameworks is a convenient way to approach implementations. Google's MapReduce framework is a suitable approach for orchestrating many large scale problems. It divides the problems into a Map phase and a Reduce phase, and solves each phase in a distributed way. Hadoop [2] is the most dominant implementation of the MapReduce framework. It accomplishes the basic functionality of MapReduce, and supports managing, tracking and relaunching tasks. Furthermore, an approach to conquer heterogeneity in the Hadoop framework is introduced in [3]. However, in these frameworks, the results from the Map phase are written to disk and used by the Reduce phase directly, and there is no support for a cyclic communication pattern. In our simulation, each simulation day can be treated as an MapReduce cycle. But since we have multiple simulation days, the traditional MapReduce framework is not sufficient, hence our use of the Granules framework to implement a controller-worker model to provide the iterative MapReduce structure we need for our research. Granules uses NaradaBrokering for stream disseminations [27].

Existing research in parallel discrete event simulations covers many domains, such as weather forecasting, traffic simulation [4], chemical plant modeling [5], urban congestion [6], and telecommunication network management [7]. The characteristics of our disease spread model are similar to these domains in many aspects (e.g., a large number of interacting entities, frequent synchronization requirements) so some of the load balancing mechanisms from the literature can be considered. There are several load balancing mechanisms introduced in [8] and [9]. However, our requirements for geographical contiguousness and frequent communications between non-adjacent sub-regions for disease spread by direct contact (i.e., animal movement) differ from the underlying assumptions in this other work. Thus, we needed to develop new load balancing mechanisms.

Failure recovery is another mechanism to address the causality problem. It does not require any synchronization. When the causality problem occurs, the system will rollback the event. This mechanism is efficient when the causality problem is rare and the rollback is not expensive. For instance, one entity only influences a few adjacent entities in one simulation time unit. If the causality problem is detected fast enough, the rollback operation only involves a few entities. Under this circumstance, the failure recovery mechanism provides performance. Fujimoto introduces a direct cancellation mechanism in [23][24]. This uses shared memory to cancel the incorrect computation of events. Once an event depends on another event, a pointer will be left. If the system found the first event is incorrect executed, it is fast to cancel the dependence event with that pointer. The application domain for failure recovery in [25] is aviation. In this approach, the system schedules events in a "look ahead" manner. The events are dispatched as far ahead as possible. This enhances parallelization since the probability of rollback is significantly reduced. Moreover, even if the rollback occurs, the number of events that need to be rolled back is minimal. Thus, the performance of failure recovery is improved in this approach. In [26], an improved mechanism for rollback recovery is proposed. Instead of rollback from the failure status, a shared memory system is used to maintain the information. Using a dirty-bit mechanism, the information is only updated and scheduled when necessary. It reduces the overhead of rollback and synchronization mechanism. In our research, since the state of each herd may be influenced by all the other herds, the rollback cost is too huge. Thus, we have to synchronize instead of rollback.

Thulasidasan et al [8] have focused on the load balancing optimization using: even distribution of entities, computational load divisibility, and the scatter partition strategy; the scatter algorithm clearly outperforms the other two strategies. However, in our case, because geographical contiguousness of subregions managed by the mappers must be preserved, we cannot use the scatter partition strategy.

Process migration [20] is another common technique to implement dynamic load balancing for split based on events. In [20], the authors introduce migration requirements, mechanisms and characteristics. Also it summarizes many examples of process migration. The authors posit that all process migration problems can be summarized into when to migrate which process where. This ties into distributed scheduling policies such as sender-initiated policy, receiver-initiated policy, and a symmetric policy that combines aspects of the previous two. These policies are suitable for different environments. Sender-initiated policy fits the situation when the network is idling while receiver-initiated policy works better when the network is busy. The symmetric policy balances the communication overheads by looking for idling and busy workers and works well in both situations.

There are several approaches based on process migration, that use different criteria to trigger the migration [21][22]. Ref [21] outlines a dynamic load balancing strategy. In this approach, each processing element maintains two local tables with information representing its view of the system's load distribution. This algorithm uses a distributed approach to disseminate the global information and a low overhead update mechanism. Thus, the load will be migrated among processing elements based on the tables. In [22], a centralized agent periodically checks for imbalances in the system and finds a suitable time for migration. The imbalance detection mechanism is based on a threshold. Process migration is a good technique when the number of tasks is high and the costs for migration low since the entire process needs to be migrated. However, in our case, process migration is not appropriate given the fine-grained tasks.

VII. Conclusion and Future Work

Proactive load balancing of DES during distributed orchestration entails prediction of execution times for the constituent tasks and attempting to mitigate imbalances before they worsen. Apportioning workloads to mitigate imbalances care must be done only when the costs for doing so do not outpace the expected gains in reduced waiting times for sub tasks. Costs associated with ensuring prediction accuracy for execution times can be controlled by incorporating feature vector pruning schemes; this can be accomplished by identifying the predictive value associated with features within the feature vector – in the case of ANNs this can be determined by the weights associated with inputs after the training process is completed.

Our future work will focus on improvements in prediction accuracy and apportioning of workloads. Currently, the feature vector for our predictions includes simulation output variables. We plan to augment this vector with the static and dynamic profile of machines where the tasks are executing. This information would be used not just for predictions but also to guide apportioning schemes to reduce the influence of unexpected slowdowns of machines.

ACKNOWLEDGMENT

This research has been supported by funding from the US Department of Homeland Security's Long Range program (HSHQDC-13-C-B0018) and the US National Science Foundation's Computer Systems Research Program (CNS-1253908).

REFERENCES

[1] 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.

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

[3] 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.

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

[5] 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.

[6] 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

[7] 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

[8] 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

[9] 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.

[10] 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.

[11] Harvey, N., et al. The North American Animal Disease Spread Model: A simulation model to assist decision making in evaluating animal disease incursions. Preventive Veterinary Medicine, 82 (2007), 176-197.

[12] Christopher M. Bishop. 2006. Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[13] John Platt. 1991. A resource-allocating network for function interpolation. Neural Comput. 3, 2 (June 1991), 213-225.

[14] Zhiquan Sui, Neil Harvey and Shrideep Pallickara. On the Distributed Orchestration of Stochastic Discrete Event Simulations. (To appear) Concurrency and Computation: Practice & Experience. John-Wiley.

[15] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. SIGOPS Oper. Syst. Rev. 41, 3 (March 2007), 59-72.

[16] Cotton, W.R., R.A. Pielke, Sr., R.L. Walko, G.E. Liston, C.J. Tremback, H. Jiang, R.L. McAnelly, J.Y. Harrington, M.E. Nicholls, G.G. Carrió.P. McFadden, 2003: RAMS 2001: Current status and future directions. Meteor. Atmos Physics, 82, 5-29.

[17] Bagrodia, R.; Meyer, R.; Takai, M.; Yu-An Chen; Zeng, X.; Martin, J.; Ha Yoon Song, "Parsec: a parallel simulation environment for complex systems," Computer , vol.31, no.10, pp.77,85, Oct 1998

[18] Butkus, A.; Roe, K.; Mitchell, B.L.; Payne, T., "Space Surveillance Network and Analysis Model (SSNAM) Performance Improvements," DoD High Performance Computing Modernization Program Users Group Conference, 2007 , vol., no., pp.469,473, 18-21 June 2007

[19] Jefferson, D.; Leek, J., "Application of Parallel Discrete Event Simulation to the Space Surveillance Network", Proceedings of the Advanced Maui Optical and Space Surveillance Technologies Conference, held in Wailea, Maui, Hawaii, September 14-17, 2010

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

[21] 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

[22] Hye-Seon Maeng; Hyoun-Su Lee; Tack-Don Han; Sung-Bong Yang; Shin-Dug Kim; , "Dynamic load balancing of iterative data parallel problems on a workstation cluster," High Performance Computing on the Information Superhighway, 1997. HPC Asia '97 , vol., no., pp.563-567, 28 Apr-2 May 1997

[23] Fujimoto, R.M. Time Warp on a shared memory multiprocessor. Trans. Sot. for Comput. Simul. 6, 3(July 1989), 21 l-239.

[24] Fujimoto, R.M. Performance of Time Warp under synthetic work-loads. In Proceedings of the SCS Multi conference on Distributed Simulation22, 1 (January 1990), pp. 23-28.

[25] Frederick Wieland. 1998. Parallel simulation for aviation applications. In Proceedings of the 30th conference on Winter simulation (WSC '98), D. J. Medeiros, Edward F. Watson, John S. Carson, and Mani S. Manivannan (Eds.). IEEE Computer Society Press, Los Alamitos, CA, USA, 1191-1198.

[26] Jefferson, D.; Leek, J., "Application of Parallel Discrete Event Simulation to the Space Surveillance Network", Proceedings of the Advanced Maui Optical and Space Surveillance Technologies Conference, Wailea, Maui, Hawaii, Sep 2010

[27] G. Fox, S. Pallickara, M. Pierce, H. Gadgil. Building Messaging Substrates for Web and Grid Applications. Philosophical Transactions of the Royal Society: Mathematical, Physical and Engineering Sciences. Volume 363, Number 1833, pp 1757-1773. Aug, 2005.