DISSERTATION

DISTRIBUTED ALGORITHMS FOR THE ORCHESTRATION OF STOCHASTIC DISCRETE EVENT SIMULATIONS

Submitted by

Zhiquan Sui

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2014

ABSTRACT

DISTRIBUTED ALGORITHMS FOR THE ORCHESTRATION OF STOCHASTIC DISCRETE EVENT SIMULATIONS

Discrete event simulations are widely used in modeling real-world phenomena such as epidemiology, congestion analysis, weather forecasting, economic activity, and chemical reactions. The expressiveness of such simulations depends on the number and types of entities that are modeled and also the interactions that entities have with each other. In the case of stochastic simulations, these interactions are based on the concomitant probability density functions. The more exhaustively a phenomena is modeled, the greater its computational complexity and, correspondingly, the execution time. Distributed orchestration can speed-up such complex simulations.

This dissertation considers the problem of distributed orchestration of stochastic discrete event simulations where the computations are irregular and the processing loads stochastic. We have designed a suite of algorithms that target alleviating imbalances between processing elements across synchronization time steps. The algorithms explore different aspects of the orchestration spectrum: static vs. dynamic, reactive vs. proactive, and deterministic vs. learning-based. The feature vector that guides our algorithms include externally observable features of the simulation such as computational footprints and hardware profiles, and features internal to the simulation such as entity states. The learning structure includes basic version of Artificial Neural Network (ANN) and an improved version of ANN. The algorithms are self-tuning and account for the state of the simulation and processing elements while coping with prediction errors.

Finally, these algorithms address resource uncertainty as well. Resource uncertainty in such settings occurs due to resource failures, slowdowns, and heterogeneity. Task apportioning, speculative tasks to cope with stragglers, and checkpointing account for the quality and state of both the resource and simulation.

The algorithms achieve demonstrably good performance. Despite the irregular nature of these computations, stochasticity in the processing loads, and resource uncertainty execution times are reduced by a factor of 1.8 when the number of resources is doubled.

ACKNOWLEDGEMENT

This research has been supported by funding from the US Department of Homeland Security's Long Range program (HSHQDC-13-C-B0018, HSHQDC- 10-C-130) and the US National Science Foundation's Computer Systems Research Program (CNS- 1253908).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 DISTRIBUTED EVENT SIMULATIONS (DES)

Discrete event simulations [1] are widely used in different domains such as weather forecasting [2], disease spread control [3] and transportation simulation [4] among others. In discrete event simulations, there are three main components: entities, interactions and time units. Entities are the basic units that contain status about the simulation. Entities can be nations, herds, animals, persons, atoms, electrical components, and so on depending on the phenomena being simulated. Interactions between entities can influence their status. In discrete event simulations, these interactions are driven by events. All the interactions occur over a discrete time line in time unit increments.

Discrete event simulations may be either non-stochastic or stochastic. In non-stochastic simulations, the status of all the entities in the next time unit is computed deterministically based on the current status. In stochastic simulations, interactions and the corresponding status changes are based on the probability density functions associated with interaction. Both of stochastic and non-stochastic discrete event simulations have their applications. Non-stochastic discrete event simulations can be used in circuit design [5], network protocol design [6] and other situations involving functionality prediction to test system design and adjustment/calibration of the parameters. Instead of constructing systems in real world, simulators save both time and money for the designers. Compared to deterministic discrete event simulations, stochastic discrete event simulations are more complex. Stochastic simulations are usually used to predict the evolution of a system under some constraints. To get the information accurately, simulations usually need to run multiple times to track expected outcomes. Furthermore, the simulations usually model real-world phenomena. Thus, there might be much more entities, interactions and time units in the simulation. Correspondingly, the execution time might be much longer than non-stochastic discrete event simulations.

These simulations are time-constrained. For instance, when forecasting weather for the next day, if the simulation cannot be finished in 24 hours, no matter how accurate its prediction the forecasts are futile. Reducing the execution time of these simulations is important.

1.2 Distributed Orchestration of Stochastic DES

Parallelization is a common way to reduce the execution time of computations. The economics of cloud computing have further simplified some of the decision making. The costs for 1 machine running for 1000 hours and for 1000 machines running for 1 hour is the same. There is no reason to endure wasting long times if the computation is complex. In distribution orchestration, the processing is split into multiple parts and processing elements execute them concurrently. Based on the scale of the concurrency that is needed, there are different environment for execution, for example, multi-processor orchestration, distributed orchestration, and cloud orchestration.

In practice, multi-processor orchestration is limited by the number of processors. The most popular multi-processor solutions such as MPI, OpenMP and etc. can be unwieldy. There is a lot of effort involved in rewriting a sequential program into a parallel one if there are many communication pathways and synchronizations. Distributed orchestration [7-9] is more popular for stochastic discrete event simulation in practice. In distributed systems, there are hundreds of processing elements that are allocated in a local cluster with stable network connections with low-latency. In comparison with multi-processor solutions, the implementation of distributed orchestration is much easier. The computation and the communication aspects can be loosely coupled. Thus, it is possible to implement the distributed computation logic without knowing much about the real computation. Cloud orchestration [10][11] has gained considerable traction of late. The number of computational units can be in the order of thousands. Implementation of cloud orchestration is identical to distributed orchestration. Moreover, the resource management is isolated with users not having to worry about the maintenance of the resources. However, in distributed system (including cluster and cloud settings) there are some tradeoffs. Users have to consider fault tolerance, resource heterogeneity and slowdown among other things.

A key issue in the orchestration of DES is coping with synchronizations. In many situations, the system has to synchronize at some point to ensure the correctness of the simulation. This is because the status of the entities in one processing element may influence the status of some entities in the other processing elements. If one processing element executes without synchronizing after one time unit, the information from the processing elements that have still not finished cannot be passed to it. In such a case, the simulation result may be inaccurate. Synchronization is necessary for correctness. However, a downside of synchronization is that it keeps some processing elements idling and waiting for the others. To reduce the idling and waiting times and to mitigate the influence of the synchronizations, load balancing mechanisms must be devised. In comparison with sequential implementations, parallelization and load balancing operations introduce overheads into the system. Some of these overheads can account for significant fraction of the overall in execution time. Striking a balance between the efficiency of load balancing and reducing the overheads are two main issues in distribution orchestration.

We posit that imbalances between tasks across synchronization points adversely impact execution time. Our algorithms target reduction of these imbalances.

In our research, the hypothesis is: if we can minimize load imbalances between tasks within synchronization points, the performance of the simulation will be very good.

Intuitively, there are two ways to balance the load: reactive and proactive. Reactive load balancing algorithms balance the load after an imbalance occurs while the proactive algorithms target imbalances from occurring in the first place. In reactive strategies, we need to detect some system parameters indicative of imbalances to guide our load balancing operations. In proactive strategies, we also need to detect those parameters that contribute to execution time prediction. Thus, which parameters we should choose is an important issue.

Whether we consider the simulation as a black-box or a white box gives us different parameters. When we consider the simulation as a black-box, the execution times of the previous time steps can be measured as criteria. In the meantime, some hardware and software status such as CPU utilization, memory cost, network bandwidth, etc. also can influence the execution time. We should put these observable facts into our parameter list. If we treat the simulation as a white-box, the situation may be more complex. In the simulation, there must be some internally observable features that influence execution time. If we can account for both internally and externally observable features, we should be able to predict the execution time more accurately.

Stochastic discrete event simulations in different areas have different characteristics. They may have different execution patterns and different synchronization requirements. Moreover, a simulation may be running in different environments with resource heterogeneity, failures and slowdowns. In our approach, we will target all these aspects of resource uncertainty. We focus on a distributed/cloud orchestration of stochastic discrete event simulations. The system will be able to tune the strategies automatically based on the states of the simulations and also the execution environment while targeting load balancing and reduction of overheads.

The execution time is based on balancing loads among the processing elements (with heterogeneous capabilities). Our goal is to construct a system that keeps all the processing elements working as much time as possible during the whole simulation period. However, this is hard to achieve without paying any costs. Usually, more balance in load requires more communications and more synchronization. These operations are overheads. There is a tradeoff between overheads and load balancing. In our approach, the system explores the performance sweet spots between overheads and load balancing efficiency for different parts of the simulation in different environments autonomously.

1.3 Challenges

Stochastic discrete event simulations are **highly irregular computations** because it is impossible to know exactly what will happen in the future. Consider spatially explicit discrete event simulations such as epidemic and traffic modeling. Interactions may happen at multiple locations. Once many interactions converge at the same location, it becomes a *hotspot*. The processing element managing more hotspots will be slower than the others. If the system cannot react to these imbalances fast enough, the rest of processing elements have to wait for the overloaded task to complete. Thus, we have to adjust the load among these processing elements.

To balance the loads in processing elements, we must identify features that allow us to make scheduling decisions. Using historical information and prediction of execution time are both reasonable ways. Using the historical information such as the execution times of previous time unit is an easy way to lead the load balancing strategies. However, there will be some unpredictable differences between this information and the real execution time in the future. Such differences may well be the source of performance bottlenecks. If we want to overcome the bottleneck, we have to predict the execution time based on the current status.

Execution time prediction is rather difficult. Since simulation is stochastic there is no clear answer as to how the simulation will look like in the future. We can only predict what the simulation will likely be. However, the complex simulation itself is not the only thing we have to consider in prediction. Since the simulations are running under operating systems in real machines, the status of these software and hardware have to be considered as well. Even for the exactly same simulation, the execution time varies in different environments. Thus, the system has to consider all these aspects while predicting the execution time.

Prediction is also hard because of the distribution of training data in the system. In discrete event simulations, we cannot control what the data distribution looks like. In many simulations such as disease spread models, transportation simulations and so on, the long-execution-time iteration is rare in comparison with short-execution-time iteration because the peak does not last long. However, the importance of predicting the long-execution-time iteration is much higher. For instance, in short-execution-time iteration, the total amount of work is 5 seconds even when running only on one computational unit. Even if our prediction is totally wrong, the loss of execution time is no more than 5 seconds. However, in long-execution-time iteration, the total amount of work is 5000 seconds running on one machine. Even a 1% of the execution time imbalance will result in 50 seconds of execution time loss. Since collected data is skewed toward the short-execution-time zone, the prediction of data in this zone will be more accurate. We have to sample across all collected data. How to select this data is also a challenge.

Since the simulation is stochastic, there might be some situations that we have never seen before. In this case, the prediction might not be accurate. In these situations, we have to adjust our prediction methods to fit those situations without influencing much on the well-predicted points. Also, the system has to be able to adjust itself to make sure the simulation with stochastic paths that have never been seen before can be finish in an acceptable time.

The efficiency of the load balancing algorithm directly depends on the prediction accuracy. Also, we are not only concerned about the average prediction accuracy. More important is the prediction of the *least accurate* execution time. For example, suppose we have 100 workers in cluster. In the next simulation period, their execution time will be all 100 seconds. However, we predicted 99 of 100 as 100% accuracy while 1 as 0% accuracy which is 0 seconds. The average prediction accuracy is 99%. However, based on the prediction result, to make the whole system totally load balanced, the other 99 workers will give 1% of their work to the 0 seconds one. The result is all the 99 well-predicted workers are running for 99 seconds and the one not-well-predicted worker is running for 199 seconds. Because the all the workers have to wait for the slowest worker, in comparison with doing nothing, the execution time is 199% of the best performance. The 1% loss of the average prediction accuracy brings in about 100% performance loss.

1.4 Research Questions

This dissertation focuses on the distributed orchestration of stochastic discrete event simulations with the objective of faster completion times. We target minimizing imbalances between subtasks across synchronization points. The dissertation presents a holistic framework to solve this problem by addressing issues relating to: (1) load balancing efficiency and minimizing overheads when making these decisions, (2) the what, when, where, and how of mitigating these imbalances, (3) exploration of the spectrum from reactive to proactive load balancing, (4) coping with resource uncertainty, and (5) identification of issues on modern architectures. Specific research questions that we explore include:

1. *For each algorithm how can we make scheduling and load apportioning decisions that are fast?* Load balancing decisions can be time-consuming and the time spent doing this does not contribute to speed-up. A key requirement is that the load balancing overheads do not outpace gains in completion times due to these decisions.

2. *How can we react to imbalances between subtasks?* This involves accounting for the what, where, and when decisions. We need to identify tasks that contribute to these imbalances and mitigate them. Care must be taken to ensure that we minimize oscillations i.e. the mitigation algorithms must need result in oscillatory behavior with the system flip-flopping between imbalanced states.

3. *How can we learn from simulation itself and which features allow us to achieve this?* Simulations can be treated as a white-box or a black-box. In the case of a black-box, only externally observable features can be used to make decisions. The white box-approach involves extracting features that are internal to the simulation: this includes identification of key loop variables for example.

4. *How can we mitigate imbalances before they occur in the first place?* Care must be taken to ensure that proactive mitigation itself does not introduce performance degradation or continual rebalancing.

5. *How can we cope with resource uncertainty while orchestrating DES?* Resource uncertainty encompasses failures, slowdowns, and heterogeneity in resource capabilities. Care must be taken to ensure that this is achieved without introducing unacceptable overheads in settings where resource uncertainty is not an issue.

6. *What are the performance implications of these algorithms on modern architectures?* This includes support for hyper-threading at the chip-level and the impact of virtualization that underpins resources provisioned in Infrastructure-as-a-Service (IaaS) clouds.

1.5 Research Contributions

In this dissertation, we have designed a slew of algorithms for the distributed orchestration of stochastic discrete event simulations. We evaluate our ideas in the context of the North American

Animal Disease Spread Model (NAADSM). NAADSM is a spatially explicit, stochastic simulation that models the spread of livestock diseases such as avian influenza, foot-and-mouth disease, pseudo rabies, and exotic Newcastle disease among others.

This dissertation covers several aspects of orchestrating stochastic discrete event simulations in a distributed setting while autonomously accounting for characteristics of the simulation, uncertainty associated with the resources, and ensuring that overheads relating to load balancing do not exceed the corresponding reduction in execution time. Specific contributions include:

1. Our approach represents an exploration of both static and dynamic load balancing algorithms. While static load balancing algorithms incur the lowest overhead, their performance is limited by their inability to respond to changing conditions both at the simulation and resource level. Dynamic load balancing algorithms incur additional overhead, but cope significantly better to changes at the simulation and resource level.

2. Our approach represents a comprehensive exploration of reactive load balancing algorithms. Our algorithms for reactive load balance mitigate imbalances after they occur. The scheduling decisions damp oscillatory behavior where mitigation may itself introduce imbalances in dynamic settings. We have devised algorithms that can operate in resource constrained settings where the number of resources is preset, and also in settings where there are no such upper bounds.

3. Our approach represents an exploration of proactive load balancing algorithms. Our algorithms here focus on prevention of imbalances from occurring in the first place. We ensure that the scheduling decisions are accurate, timely, and result in shorter execution times. In fact, our proactive algorithm outperforms the best dynamic, reactive load balancing algorithm.

4. Our approach represents an exploration of predictive approaches to guide load balancing decisions. We have explored approaches which treat the simulation as a black-box and also one that treats the simulation as a white-box. In the black-box approach, we base predictions on the hardware and software characteristics associated with the simulation runs. In the case of a white-box approach, we extract features from the simulation that we believe are indicative of future execution times. Our approach uses Artificial Neural Networks (ANNs) to make these predictions. To cope with situations where the training data is non-uniformly distributed, we use a multi-stage ANN to ensure that the execution time predictions are accurate especially for cases where inaccurate predictions would lead to prolonged execution times.

5. Our approach represents orchestration in the presence of resource uncertainty. Our dynamic algorithms (both reactive and proactive) can operate in the presence of resource uncertainty due to failures, slowdowns, and heterogeneity that are common in public clusters or cloud settings. Our algorithm relies on speculative checkpointing, normalization of resources, and slowdown detection to achieve this. Most importantly, our algorithms introduce very low overheads even in situations where the resources are well behaved.

1.6 Dissertation Organization

This dissertation is organized as follows. Chapter 2 introduces the background and related work. Performance and reliability requirements will be addressed in Chapter 3. Chapter 4 includes the issues in system design. Chapter 5 introduces the issues associated with reactive load balancing algorithms. Chapter 6 contains information about execution time prediction. Chapter 7 concentrates on proactive load balancing algorithms. We will analyze the fault tolerance and slowdown detection mechanisms when the resource uncertainty exists in cluster in Chapter 8. Virtualization is introduced in Chapter 9, and Chapter 10 contains the contribution, conclusion and future work of the research.

CHAPTER 2

BACKGROUND

2.1 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 [12]. 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.

2.1.1 Datasets

To compare the simulations, we are going to use some artificial datasets and some practical datasets. In practice, if the simulation finishes too fast, there is no need to design complex strategies to make it faster. Furthermore, the overheads in this kind of simulations will dominate the execution time so that we cannot gain performance by parallelization. Thus, the datasets using in this research are all long running. Usually it takes hours to days to finish the sequential versions.

We are using a dataset that contains 660,000 herds in it as a baseline. There are 9 starting points for the disease outbreak. The simulation will last 365 simulation days. In practice, the sequential version will run for 22 hours.

2.2 Granules

Granules [13] is a framework which supports data stream exchange among distributed computational units. In Granules, the computations can be long running and the data stream exchange can be expressed as a cyclic directed graph. The computations can be written in C, C++, Java, Python and R. There is a communication bridge to connect Granules, which is written in Java, with computational components.

Granules is a non-centralized system. Each node in Granules is exactly the same. Users can specify the functionality for each node. There are interfaces that define what to do when receiving a message and which kinds of messages can be sent. With these interfaces, there is much flexibility of programming. Users can design centralized or non-centralized structure of computations freely.

Granules interleaves many computational loads onto each machine so that the resources can be utilized efficiently. Also the overhead incurred by the Granules system is quite low, both message passing among Granules nodes and from Granules node to computational components is quite fast.

In Granules, there are some interfaces that allow communications among different programming languages [14]. These interfaces make the communications from Granules to NAADSM possible. The interfaces are a lightweight communication layer and do not have a noticeable impact on computations.

2.3 Parallelization Approaches of DES

Discrete event simulations have many applications. The Atmospheric Science department at CSU developed the Regional Atmospheric Modeling System (RAMS) [15] in 1992. At that time, cluster and cloud computing techniques are not popular and their 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 [16] 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 [17]. 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 [18] 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.

In discrete event simulations, the data structure contains three components [19]. They are status of the entities, a pending event list and a global clock. The parallel discrete event simulation problems are to process the events in the list concurrently without causing causality problems. Causality problems occur when two processing elements are modifying the status of the same entity in an incorrect time order. There are several mechanisms concentrating on this problem. This can be categorized into three main strategies: split based on entities, split based on events, and failure recovery.

2.3.1 Split Based on Entities

Split based on entities is an intuitive solution for the causality problem. In this solution, each entity is assigned to only one processing element. In one processing element, the status update is sequential. Thus, the causality problem is solved automatically. However, in these approaches, the load balancing issue is important. Since the events on each entity are not equal, even if we can allocate the same number of entities on each processing element, we cannot ensure that the load is evenly distributed. Moreover, once some events in one processing element influence another, so there must be some synchronization points. Even if we can ensure the total amount of load is equal, the difference of load between synchronization points may still impede the performance. There are some approaches that concentrate on addressing the load balancing issue of split based on entities strategies.

In these approaches, there are one-shot load balancing strategies and dynamic load balancing strategies. In one-shot load balancing strategies, there are some assumptions. The synchronization frequency cannot be high and the load on each processing elements cannot change much. Dynamic strategies provide more flexibility. Between each pair of synchronization points, the processing elements have chance to adjust the load. However, such an approach requires the system to detect the load imbalance. The overheads of detections and adjustment have to be controlled.

An interesting approach to load balancing has been introduced in cosmology [20]. It partitions the 3-D domain into smaller pieces. However, there is a special assumption in cosmology: if two entities in the universe are not close enough, the force between these two entities can be ignored. Thus, even if we use orthogonal bisection to partition the domain, the communications among the processing elements are rare. It will not influence the simulation results much if we just ignore them. But the authors insist the minor differences impose difficulties in result validation. They propose a new scheme for domain decomposition. The idea is to use a space-filling fractal named Peano-Hilbert [21] curve to map the 3D space to 1D space. The idea is quite similar to the scattering strategy; however, the purpose is not balancing the load but eliminating the communications.

In transportation simulations, an example is the FastTrans simulator [3]. It simulates and routes tens of millions of vehicles on real-world road networks. In such a large-scale simulation, parallelization is necessary. FastTrans relies on several one-shot load balancing algorithms. They are based on geographical coordination with approaches relying on having either the with same number of entities, balanced event load, balanced routing load, balanced total load or scattering partitions. Among these strategies, the scattering strategy provides fairness in the computation load distribution. In the meantime, it also imposes much more communication overheads. From the overall results, we can see that the scattering strategy provides a better execution time. However, even if the fairness of the scattering strategy is 10 times more than balanced event load strategy and balanced routing load strategy, the execution time is quite close when the number of CPUs is going up as the communication overheads become more and more dominant.

More experiments have been done in [22]. It compares scattering strategy, random strategy, balanced entities strategy and balanced routing load strategy. The performance of the scattering load balancing strategy is similar to the random strategies. They are better than balanced entities strategies and balanced routing load strategies in exponential networks, normal distribution network and uniform distribution network.

However, scattering or random load balancing strategy fits only parts of the situations. In some simulations, the simulation itself requires adjacent entities to be allocated onto the same or adjacent workers. Moreover, the scattering strategy introduces more communication overheads that often cannot be ignored, because they will dominate the execution time. Thus, the scattering or random strategy is not the best solution for all the discrete event simulations.

When the number of processing elements is large, the balanced computational load strategy is close to the scattering strategy because the communication overhead can be ignored. However, how to measure the computation load is not easy. We have to find a suitable function to measure the computation load. Furthermore, even if we can find the exactly balanced way to partition the load, the frequent synchronization and the change of load between different synchronization points will impede performance. This is an unavoidable problem for all the one-shot load balancing strategies. Thus, in these situations, we have to consider the dynamic strategies.

In [23], the authors provide a solution to balance load dynamically with a load migration mechanism. This algorithm concentrates on how to measure the load on each computational unit and how to find the fastest way to balance them periodically. They measure the load by the events. But not all the events are considered the same. The events are weighted according to the distance in time of the event from the beginning of the simulation. The load balancing mechanism is to move the load to its neighbors if it contains more load than average. Thus, this algorithm fits the frequent synchronization points and improves performance significantly. However, there is no self-tuning mechanism in the system. It considers the environment stable and considers no obvious changes during execution.

2.3.2 Split based on events

Split based on events is another strategy that in applicable in simulations that generate many events and where these events are stored in a queue. We have to find out which events are available to process at each synchronization point. Then we can process those events concurrently. However, there are some constraints. First, if all the processing elements can process the events on all the entities, they have to store the status of all the entities. If the memory of one processing element is not sufficient to keep everything, these strategies cannot be utilized. Second, since the status of entities have to be synchronized at some point, generating the events and updating the status have to be done sequentially. In comparison with these sequential processes, if the concurrent computation is not dominating the execution time, performance will suffer.

There are several approaches based on this strategy. Penmatsa and Chronopoulos [24] introduce two static strategies and two dynamic strategies. Among these strategies, one static and one dynamic concentrate on minimizing the expected response time for the entire system, which fits our problem. The static strategy converts the problem into a non-linear optimization problem and solves it by using the Kuhn-Tucker theorem [25] which is a well-known technique for getting a solution in nonlinear programming. The dynamic strategy improves the static strategy with some load balancing operations between processing elements. These operations are based on the process migration approach.

Process migration [19] is a common technique to implement dynamic load balancing for split based on events. In [19], 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. They use different criteria to trigger the migration [26][27]. Campos et.al [26] gives out 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 [30], 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.

In practice, network conditions may not be ideal. Especially in clouds, there may be some latency for communications. Also, the bandwidth may be limited. Considering all these constraints, Dhakal et.al [28] describes a static algorithm and a dynamic algorithm to minimize the overall completion time. For each load balancing operation, the algorithm calculates the load balancing gain and the communication overhead. If the gain is not greater than the overhead, the operation will not be applied.

Machine learning algorithms are widely utilized in these strategies. They are usually used for load prediction and load distribution. Jun Wang et.al [29] introduces an application in radar simulations. It uses a machine learning algorithm to predict the load. The load balancing methods are based on prediction results. In [30], the authors describe a genetic machine learning algorithm to adjust the load balancing threshold. The load balancing operations are applied based on the threshold. Munetomo et.al [31] also provides a genetic algorithm. It lists all the possible operations and uses the genetic algorithm to find the best one. In [32], the authors introduce an ant-colony algorithm for dynamic load balancing. However, this is based on hierarchical load dispatching. In discrete event simulations, this topology is relatively rare.

2.3.3 Failure Recovery

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 [33][34]. 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 [35] 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 [36], 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.

2.4 Distributed and Cloud Computing Frameworks

Besides the mechanisms designed by the researchers, there are some cloud computing frameworks [73][74] which also consider the load balancing issues. In the Map-Reduce framework [37][68] , there are also some learning-based [38] or non-learning-based [39] load balancing strategies. Some of them can do well in the situations that contain frequent synchronization points. However, those tasks contain many small sub-tasks so that they can be migrated easily without any extra operations. However, in the spatially explicit discrete event simulations that we consider simulations, in each simulation time unit, migration is difficult. In Dryad [40], there are also some load balancing mechanisms. They are basically based on pipeline execution in multi-stage task scheduling. However, in our problem, the simulation is one-shot. We can only use pipeline execution when there are several simulations running together. Moreover, since the computations are similar in each simulation time unit, multi-stage scheduling will not result in performance gains. In these cloud computing frameworks, there are no self-tuning algorithms based either on the operating conditions or information about the computational tasks.

There are also some frameworks that designed specifically for discrete event simulations. GroudSim [41] is a grid and cloud simulation toolkit for scientific application based on discrete event simulation. It supports the basic structure of simulations such as entities and jobs. Moreover, it supports cost evaluation in the cloud, result tracing, probability distribution and failure recovery. These functionalities make the implementation of the simulation easier and performance better.

However, communications between processing units are not supported. This is the major limitation of this approach.

BigSim [42] is another framework that can be used for discrete event simulations. It is a parallel simulator for very large number of processors. It predicts the time of sequential code and the network performance to balance the load and minimize the communication overheads. However, if the simulation is stochastic, the execution time is not only based on the sequential code but also the current status of the simulation. Thus, this approach is not suitable for stochastic discrete event simulations.

2.5 Coping with Resource Uncertainty

While our system primarily targets cloud or cluster deployments, checkpointing also plays an important role in creating fault-tolerant grid environments. Recognizing the importance of checkpointing schemes that dynamically adapt to their environment, Chtepen et al. propose a number of heuristics for determining a checkpoint interval [43]. Unlike our system, their design does not require global state synchronization, but it does account for estimated execution time and checkpoint overhead to determine whether a job should be allowed to checkpoint its state or not.

Aurora [54][55] follows the controller-worker paradigm and also supports speculative tasks for fault tolerance and dealing with heterogeneity. Aurora shares many similarities with our framework, but does not feature a prediction-based fault tolerance module or deal with slowdown detection.

Cucuzzo et al. [56] proposes a heuristic-based method for autonomous checkpointing by each logical process in a distributed discrete event simulation. This technique accounts for simulation state information to determine when to checkpoint and in doing so completely avoids coordination overhead. However, this also results in situations where processes must “catch up” with the rest of the system if they have not recently recorded a checkpoint. Since our framework is designed for stochastic simulations, checkpointed state cannot deviate across simulation iterations.

D’Angelo [57] surveys the challenges and current trends seen in discrete event simulation parallelization and distribution, particularly noting the user-facing difficulties in running these simulations on a parallel architecture and how the trend toward cloud deployments has created a paradigm shift. For instance, distributed simulations that rely on optimistic parallelization with rollback suffer as network latencies increase; one possible solution may be conservative time synchronization [58]. The proposed multi-agent middleware solution is finer-grained than our controller-worker model, but facilitating entity migrations would also require a more involved retrofitting process for sequential simulations.

In the context of distributed event execution with realtime requirements, Feng and Lee [59] propose an actor-based system that models interactions between components as dependencies. This allows the system to provide intelligent checkpointing and perform fine-grained rollback operations when a failure occurs, contrasting with our approach of synchronizing the simulation interval across all workers. However, this process does require more information about the events and their interactions.

Another checkpoint-based approach involving the structured high-level architecture (HLA) is presented in [60]. In this case, checkpoints are stored in a universal stable storage repository by the federates, and several new publish-subscribe interactions are added to model checkpoint-related communications. As noted for the previously described technologies, this approach requires following a particular simulation structure a priori.

Failure detection research for distributed discrete event simulations has produced several innovative solutions [44][45]. Gupta, Chandra, and Goldszmidt [46] postulate that the key components of an effective failure detection mechanism are completeness, speed, and accuracy. In the case of our framework, completeness and speed are significantly relaxed in favor of ensuring accuracy to avoid needless rollback operations. Only weak completeness is required for our purposes, meaning all the components in the system do not need to know when a failure has occurred, and the heartbeat interval sets an approximate upper bound for detection times.

Rollback operations and speculative execution in discrete event simulations have also been researched by Ramirez Ortiz and Jiménez [47]. In their work, a coordinator handles snapshotting the simulation, but requires execution to completely halt while the process is carried out; our solution interleaves snapshot operations with other processing, and only requires the simulation to stop if a failure has occurred.

Xie et al. [48] focuses on data placement in a Hadoop cluster. While our particular use case does not impose high storage requirements, the computing ratio discussed in this work is applicable for heterogeneous deployments. However, this approach does not account for differences in machine usage patterns over time, and benchmarks are performed only once at the beginning of a job execution.

Lee et al. [49] improves on standard slot scheduling algorithms by maintaining a pool of core resources and accelerator resources that are dynamically adjusted at runtime, similar to our "spare worker" concept. In this case, the computing rate (CR) is calculated for each of the heterogeneous resources by running a pilot job on every possible hardware configuration. This helps identify machines that can benefit from GPU acceleration, have faster CPUs, etc., but is also a greedy approach that may cause overall throughput to decline in some situations.

Roy et al. [61] uses predictive models for workload forecasting in the context of auto-scaling elastic clouds. Specifically, moving averages are computed over a sliding window to predict incoming loads and scale accordingly. In this case, both reconfiguration costs and SLA violations are taken into account when managing the virtual machines.

Gandhi and A. Sabne describe an approach that detects system calls to identify stragglers in Hadoop [50]. Since the performance of Hadoop is based on throughput, it is reasonable to track only the disk I/O and network I/O for measurement. However, in our work, the simulation run on each worker instance is CPU intensive and there is no output in the middle of the simulation. Thus, tracking system calls is not as accurate in our situation.

Polo et al. considered machine states with a detection method similar to our approach [51]. In their work, they construct a Class Constrained Multiple-Knapsack model to maximize machine utilization. Conversely, our work uses machine learning to calculate the slowdown factor. Our goals in tracking machine states are also different: In Hadoop, the goal is to maximize throughput, while in DES, the goal is to minimize the execution time for each simulation day.

CHAPTER 3

REQUIREMENTS

In our experiments, the correctness of the simulation is the first requirement. The correctness includes not only the inner logic of the disease spread model within different sub-regions, but also whether we can run exactly the same simulation in different strategies and environments. We call it repeatable random run. It is required to compare the results. With it, we can say that we compare apples to apples in different benchmarks.

If we can hold the correctness of the simulation, there are still performance requirements and reliability requirements

3.1 PERFORMANCE REQUIREMENTS

The performance is the most important criteria for computations in distributed system. In our particular case, the execution time represents the performance directly. Using a limited number of machines, we need to deal with load balancing and overhead tradeoffs. We need to try different approaches to find these sweet spots.

Efficiency is also a factor we need to consider in this problem. For instance, if we can get only a little reduction in execution time but we have to add a lot of machines into the system, whether we should do that is an issue that must be addressed. The efficiency influences the energy cost of the system. If we run our system in the cloud, the efficiency is relative with the money cost as well.

3.2 Reliability Requirements

In the modern computing landscape, failures are an issue that must be expected and dealt with rather than simply avoided or ignored [52]. The longer a software process runs, the more likely it is to experience a hardware or software failure. This fact is only exacerbated by the trend towards distributed, multi-core architectures that take advantage of the vast processing power found in today's commodity hardware. Each additional unit of processing involved in a task increases the overall probability of a failure occurring.

Slowdown of a resource results in tasks executing on that resource executing significantly slower than other tasks executing concurrently on other resources. Resource slowdowns must be mitigated in a timely fashion because of their cascading effects. Discrete event simulations include synchronization barriers where different components of the simulation synchronize their state. Between successive synchronization barriers, the execution time is only as fast as the slowest task. This means that a single slow task greatly influences the overall execution time.

CHAPTER 4

SYSTEM DESIGN

4.1 ARCHITECTURE

At the beginning of this research, we have only a sequential version of stochastic discrete event simulation – NAADSM. The basic unit of the simulation is herds of livestock and the interactions between herds are driven by events. The smallest time period in the simulation is one simulation day. We developed a distributed version of NAADSM to start our work. In practice, we used Granules framework to implement the parallelization.

To parallelize NAADSM computations, we have to divide the NAADSM into smaller pieces. In our approach, we divide the region based on geographical regions. Since the state of each herd in the current day depends on all the herds in the previous day, there must be a synchronization point in each simulation day. In order to implement the synchronization mechanism, we use a classic Controller-Worker model to solve this problem.

4.1.1 Controller-Worker Mechanism

To orchestrate a NAADSM simulation using Granules, we employ a controller-worker model (depicted in Figure 4.1) 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.

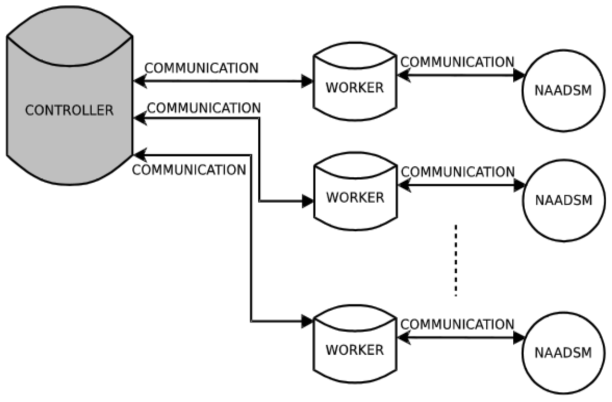

Figure 4.1 Controller-Worker Architecture

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. 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.

4.1.2 Work of Flow 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; and

- quarantining and vaccination of farms managed by the worker.

Figure 4.2 provides a sketch of the activities within a single simulation day.

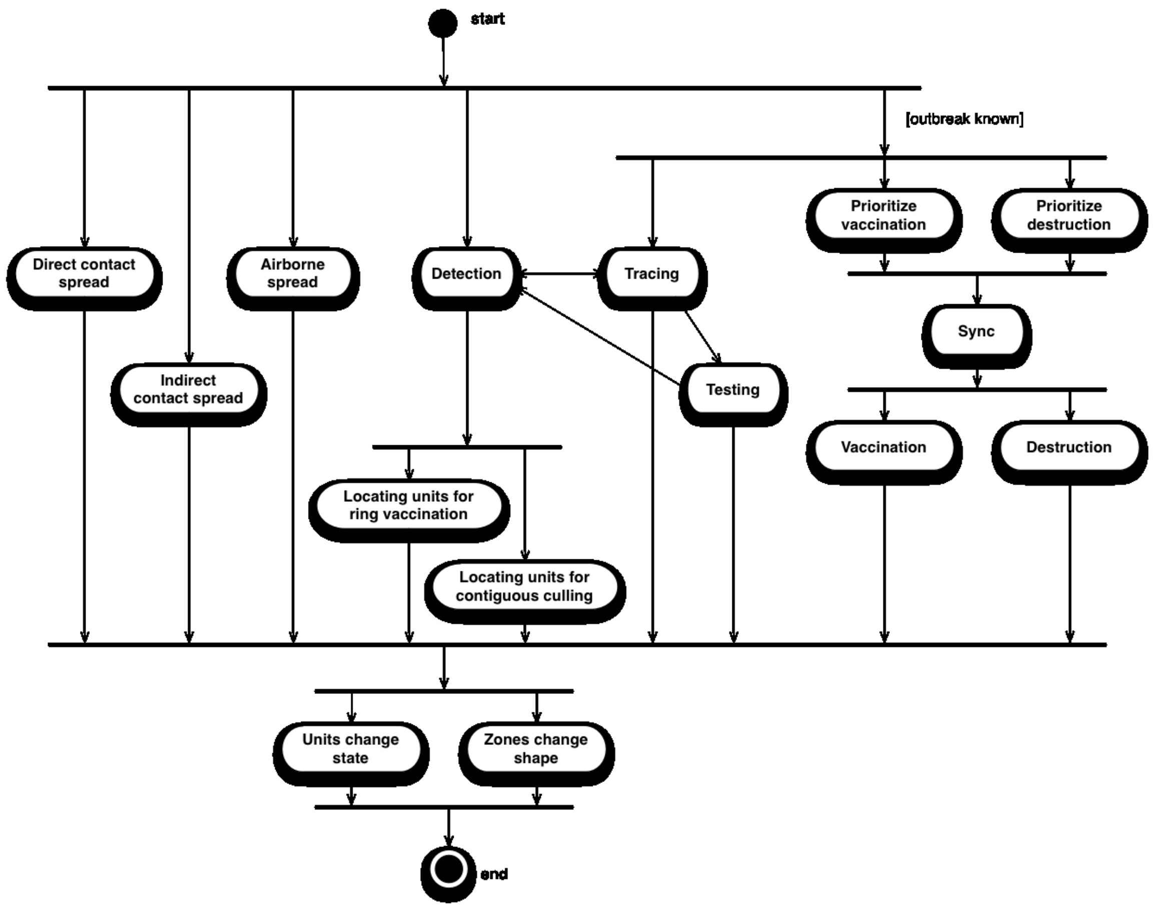

Figure 4.2 Conceptual flow of events (not considering standalone vs. distributed operation) in one simulation day.

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; and
- 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

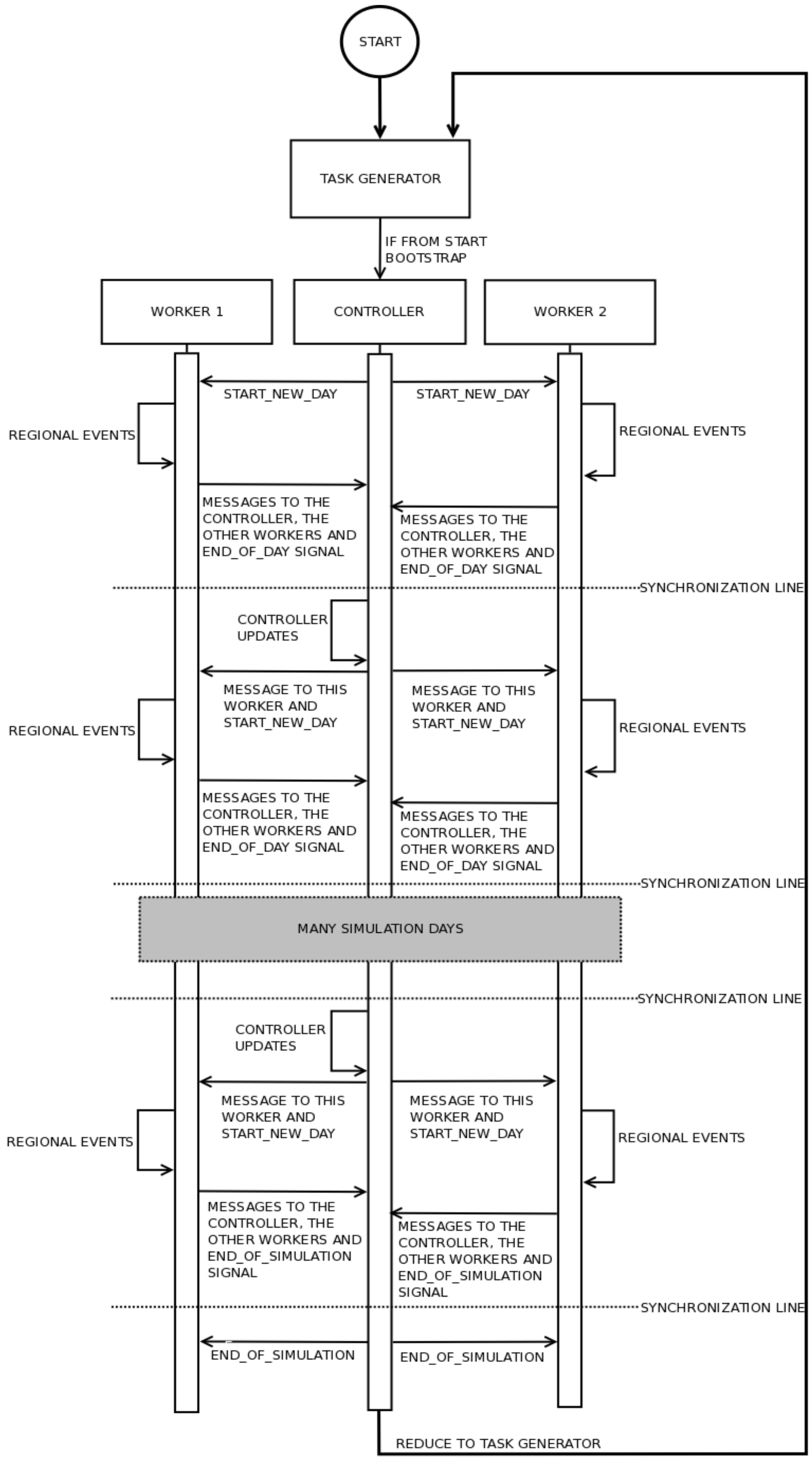
START
TASK GENERATOR
IF FROM START BOOTSTRAP
WORKER 1
CONTROLLER
WORKER 2
START_NEW_DAY
START_NEW_DAY
REGIONAL EVENTS
REGIONAL EVENTS
MESSAGES TO THE CONTROLLER, THE OTHER WORKERS AND END_OF_DAY SIGNAL
MESSAGES TO THE CONTROLLER, THE OTHER WORKERS AND END_OF_DAY SIGNAL
SYNCHRONIZATION LINE
CONTROLLER UPDATES
MESSAGE TO THIS WORKER AND START_NEW_DAY
MESSAGE TO THIS WORKER AND START_NEW_DAY
REGIONAL EVENTS
REGIONAL EVENTS
MESSAGES TO THE CONTROLLER, THE OTHER WORKERS AND END_OF_DAY SIGNAL
MESSAGES TO THE CONTROLLER, THE OTHER WORKERS AND END_OF_DAY SIGNAL
SYNCHRONIZATION LINE
MANY SIMULATION DAYS
SYNCHRONIZATION LINE
CONTROLLER UPDATES
MESSAGE TO THIS WORKER AND START_NEW_DAY
MESSAGE TO THIS WORKER AND START_NEW_DAY
REGIONAL EVENTS
REGIONAL EVENTS
MESSAGES TO THE CONTROLLER, THE OTHER WORKERS AND END_OF_SIMULATION SIGNAL
MESSAGES TO THE CONTROLLER, THE OTHER WORKERS AND END_OF_SIMULATION SIGNAL
SYNCHRONIZATION LINE
END_OF_SIMULATION
END_OF_SIMULATION
REDUCE TO TASK GENERATOR

Figure 4.3 Interactions between the controller and workers

that the simulation needs to continue. Interactions between the controller and workers are summarized in Figure 4.3.

4.1.3 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. 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. (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 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. 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.

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.

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.

4.1.4 Message Passing Mechanisms

Message passing in our system is performed via Granules. When the controller communicates with workers, all the messages are passed in a centralized way. In some of our load balancing strategies, however, we found that some communication among workers is required. Despite this modification, we still pass these messages through controller at the beginning of our experiments.

Communication overhead is an important problem in this situation because all of the workers pass messages to controller, and the controller passes messages back to the workers, 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 workers to the controller and from the controller to the workers. Since there are 64 workers, 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. In this situation, we design a direct way to pass messages from one worker directly to another worker.

4.2 Overhead Reduction

Communication overhead is an important problem. Based on Amdahl's law [70], the performance will bottleneck due to frequent synchronization and overheads on the critical path. 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 workers to the controller and from the controller to the workers. Since there are 64 workers, 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 worker may go inactive at the end of one simulation day (because it is merged with another worker) and then be re-activated on a later simulation day. When the worker 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. In contrast, in the PAW algorithm, all of the workers are active on every simulation day. When the workload is "apportioned" among the workers, each worker 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.

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 4.4. 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.

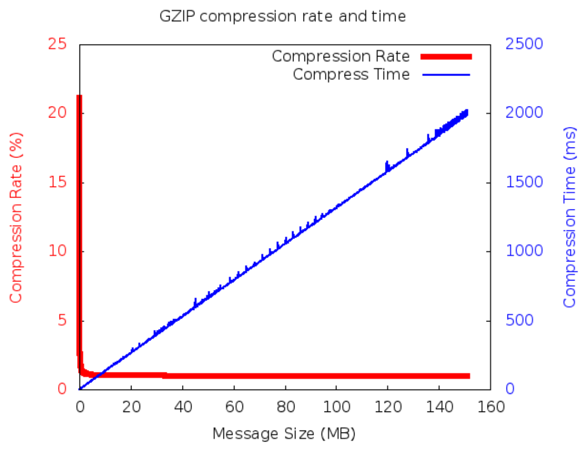

Figure 4.4 Compression rate and time for synchronization messages

CHAPTER 5

REACTIVE LOAD BALANCING STRATEGIES

5.1 CHALLENGES AND RESEARCH QUESTIONS

In discrete event simulations, there are some situations where we cannot get any information from the simulations, or the information retrieved cannot provide insights on future execution time. In these situations, we prefer reactive load balancing strategies. The main challenge is how to make reactive strategies intelligent. In this part of research, we explore the following research questions:

- *If we treat the discrete event simulation as a black box, what are the externally observable features of the simulation that we can use to inform orchestration of processing loads?*
- *With a limited number of resources, how can we utilize the resources as evenly as possible without knowledge/predictions of computational loads in the future?*

5.2 STATIC LOAD BALANCING STRATEGY

In static strategy, we divide the whole region into smaller sub-regions at the first simulation day in each simulation. This is a one-shot solution. Because there is no load balancing operations during execution, the loads on different workers are rather imbalanced. The performance of this strategy is shown in Figure 5.1.

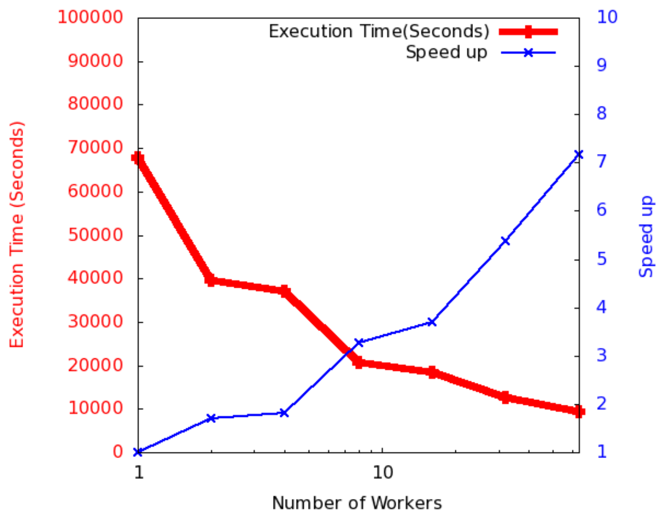

Figure 5.1 Speedups for static load balancing strategy versus number of workers on a logarithmic scale

Figure 5.1 shows the execution time and speed-up for 1, 2, 4, 8, 16, 32 and 64 workers scenario for the same simulation.

From Figure 5.1, we can see that from 1 worker scenario to 2 workers scenario the speed-up is about 2. However, there is almost no speed-up from 2 workers scenario to 4 workers scenario. We want to check the reason why the speed-up varies so much for these scenarios. Thus, we have Figure 5.2.

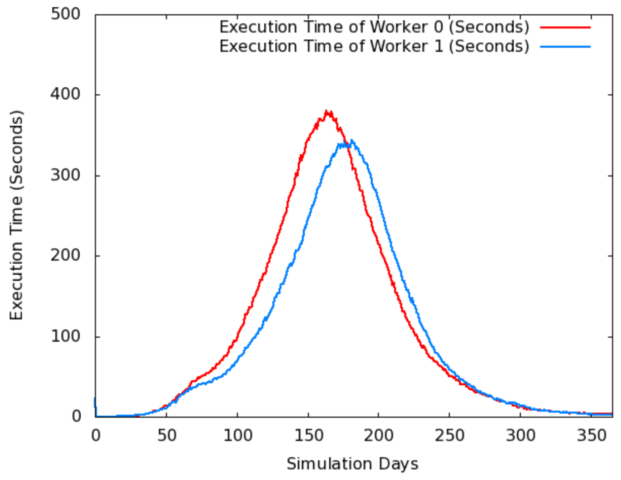

Figure 5.2 Worker execution time of 2 workers scenario

In 2 workers scenario, the execution time of these 2 workers are close. That means the execution load is almost equally divided. That is why the speed-up in 2 workers scenario is almost ideal.

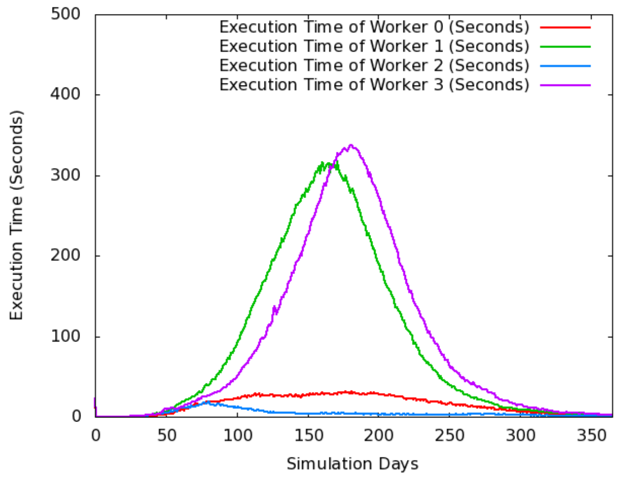

Figure 5.3 Worker execution time of 4 workers scenario

Similarly, Figure 5.3 depicts execution time for 4 workers scenario. We can see that the worker 0 and worker 2 do little work in comparison with worker 1 and worker 3. Thus, we can say that the main reason that causes the poor performance in static load balancing strategy is the imbalanced load division. Also, if we want better performance, we have to apply some other load balancing strategies which can adjust the load during runtime.

5.3 Dynamic Split Load Balancing Strategy

From the result of static load balancing strategy, we can see that the utilization of the resources is not balanced well. Thus, we designed dynamic split load balancing strategy to split the load during runtime. In comparison with static load balancing strategy, this strategy only splits the long-execution-time workers. Moreover, because there is no merge operation in this strategy, once a worker is divided, the new worker will be allocated to a new node until the simulation ends. In the meantime, the simulation hotspots migrate to different regions often during execution. Thus, we really want the split operations to be done on the slowest simulation days. Also, we want to utilize the resources fully. We don't want to see that some resources are not used in the most time of simulations. So there is a trade-off when split the workers. The algorithm of split is shown below:

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

lowThreshold = LOW_CONSTANT * rate;

highThreshold = HIGH_CONSTANT * rate;

for(i=0;i<activeWorkerNumber;i++){
    if (spareWorkerNumber<0 || executionTime[i] < lowThreshold) {
        continue;
    } else if (activeWorkerNumber==1 ||
            executionTime[i] > highThreshold ||
            (executionTime[i] > lowThreshold &&
            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.

If there are only a few resources running for this strategy, the strategy cannot perform well due to the resource are occupied too fast. Thus, we just tested 32 workers scenario and 64 workers scenario with the tuned parameters. The results are shown in Figure 5.4.

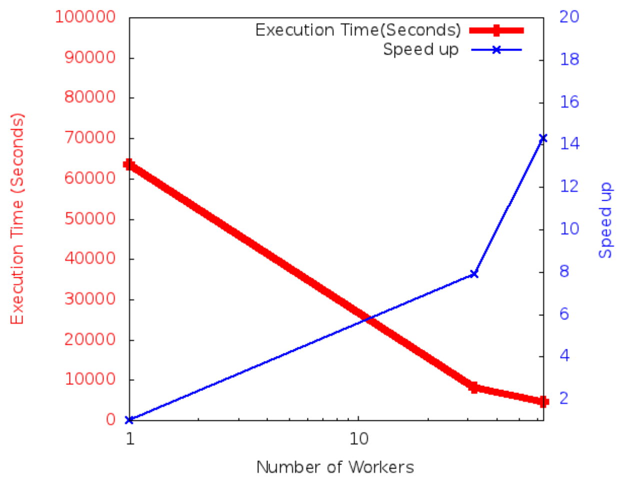

Figure 5.4 Speedups for dynamic split strategy

We can see that, the speed-up for 32 workers is about 8 and the speed-up for 64 workers is about 14. In comparison with static load balancing strategy, the performance is much better. However, in comparison with ideal speed-up, the efficiency is still low. We analyze the execution time with load balancing efficiency in Figure 5.5.

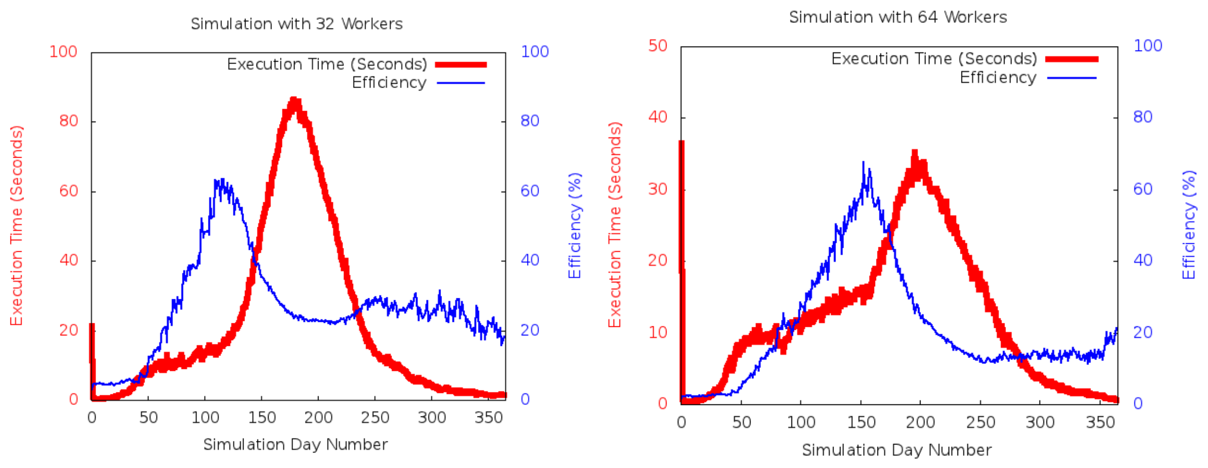

Figure 5.5 Execution time and load balancing efficiency of 32 and 64 workers scenarios in dynamic split strategy

We can see that the load balancing efficiency is a little bit off in comparison with the slowest execution time point. In 64 workers scenario, the efficiency peak is closer to execution time peak. Thus, the speed-up is growing up from 8 to 14. However, we can see that the execution time peak in 32 workers scenario is a little before day 200, but when we split slower in 64 workers scenario, the peak is a little after day 200. Also, we can see that all the execution time peaks are coming with the efficiency valley. That is because the execution hotspots migrate and the split operations before do not play as well as when they are splitting. No matter how we tune our parameters, the efficiency cannot be much better. Also, it is hard to tune all the parameters once we have a new scenario. Thus, we need some better strategy that can manage the resource utilization more efficiently and automatically.

CHAPTER 6

EXECUTION TIME PREDICTION

Stochastic discrete event simulations are highly irregular computations because it is impossible to know exactly what will happen in the future. Consider spatially explicit discrete event simulations such as epidemic and traffic modeling. Interactions may happen at multiple locations. Once many interactions converge at the same location, it becomes a hotspot. The processing element containing more hotspots will be slower than the others. If the system cannot react to these imbalances fast enough, many processing elements have to waste their time on waiting. Thus, we have to adjust the load among these processing elements.

To balance the loads in processing elements, we have to explore some criteria. Using historical information and prediction of execution time are both reasonable ways. Using the historical information such as the execution times of previous time unit is an easy way to lead the load balancing strategies. However, there will be some unpredictable differences between this information and the real execution time in the future. Once our load balancing strategy is well-designed, these differences may be the bottleneck of the performance. If we want to overcome the bottleneck, we have to predict the execution time based on the current statuses.

Execution time prediction is rather difficult. It is because the simulation is stochastic so there is no certain answer to how the simulation will look like in the future. We can only predict what the simulation will likely be. However, the complex simulation itself is not the only thing we have to consider in prediction. Since the simulations are running under operating systems in real machines, the status of these software and hardware have to be considered as well. Even for the exactly same simulation, the execution time varies in different environments. Thus, the system has to consider all these aspects while predicting the execution time.

Prediction is also hard because of the distribution of data in the system. In discrete event simulations, we cannot control what the data distribution looks like. In many simulations such as disease spread models, transportation simulations and so on, the long-execution-time iteration is rare in comparison the short-execution-time iteration because the peak does not last long. However, the importance of predicting the long-execution-time iteration is much higher. For instance, in short-execution-time iteration, the total amount of work is 5 seconds even running only on one computational unit. Even if our prediction is totally wrong, the loss of execution time is no more than 5 seconds. However, in long-execution-time iteration, the total amount of work is 5000 seconds running on one machine. Even 1% of the execution time imbalance will results in 50 seconds of execution time loss. Since more collected data is in the short-execution-time zone, the prediction of data in this zone will be more accurate. We have to sample across all collected data. How to select this data is also a challenge.

Since the simulation is stochastic, there might be some situations that we have never seen before. In this case, the prediction might not be accurate. In these situations, we have to adjust our prediction methods to fit those situations without influencing much on the well-predicted points.

Also, the system has to be able to adjust itself to make sure the simulation with stochastic paths that have never been seen before can be finish in an acceptable time.

This prediction is rather difficult; however, prediction accuracy is very important in this research. The load balancing strategy directly depends on the prediction accuracy. Also, we are not only concerned about the average prediction accuracy. More important is the prediction of the least accurate execution time. For example, suppose we have 100 workers and one worker represents one computation unit. In the next simulation period, their execution time will be all 100 seconds. However, we predicted 99 of 100 as 100% accuracy while 1 as 0% accuracy which is 0 seconds. The average prediction accuracy is 99%. However, the other 99 workers will give 1% of their work to the 0 seconds one to make sure the load is balanced. The result is all the 99 well-predicted workers are running for 99 seconds and the one not-well-predicted worker is running for 199 seconds. Because the all the workers have to wait for the slowest worker, in comparison with doing nothing, the execution time is 199% of the best performance. The 1% loss of the average prediction accuracy brings in about 100% performance loss.

Prediction is also important in coping with resource uncertainty. Within fault tolerant mechanisms, we need to predict the execution time and the checkpoint overhead. Only when we know how much execution time we will lose if some resource fails and how much overhead it will take if we checkpoint can we make a correct decision whether we should checkpoint the current state. Furthermore, within the slowdown detection mechanism, we need to predict the how each machine will perform in the next simulation day. With this information, we can correctly launch new tasks to the fast machines and the speculative tasks for the workers in slower machines.

In conclusion, the prediction is important for both load balancing performance and coping with resource uncertainty.

6.1 Challenges and Research Questions

The main goal of the execution time prediction is to guide the load balancing algorithms and alleviation methods corresponding with resource uncertainty. One challenge for prediction is how to avoid the outliers since one single outlier may bring in a huge imbalance. Another challenge is how to cope with the huge amount of training data within a non-uniform distribution because each single simulation will produce tens of thousands of training data and most of them are short in execution time.

There are several research questions that we explore in the context of prediction:

- *In proactive strategies, how can we use non-uniform distribution training data to predict execution time?*
- *Do neural networks as a prediction method provide sufficient accuracy without introducing unacceptable timing overheads?*
- *Are there situations where further improvements in prediction accuracy do not result in a corresponding improvement in performance?*

6.2 Data Collection

To make predictions, we need to collect data from the simulation. For different predictions, the training data are different. The prediction data are both collected from hardware and software.

6.2.1 Data Collection from Hardware

In our public cluster, resources are heterogeneous and we wish to launch tasks on the best available resources. We rank resources by CPU speed: our simulation is a CPU intensive task, so memory and disk I/O are not expected to be the dominating factors. We measured CPU speed with a benchmark called the RAW benchmark suite. Because we are using a public cluster, we attempted to avoid having other users' processes interfere with the speed measurement by 1) selecting a benchmark from the suite that has a short running time (we chose the FFT benchmark), 2) running the benchmark at several different times of day, and 3) choosing the fastest recorded time. We hypothesize that the fastest test is the one with the least interference.

Beyond CPU speed, we also record CPU utilization, memory and network use, and disk I/O. Each instance of our test simulation uses a single CPU core and a predictable amount of memory, which provides a baseline estimate of the cost of launching another worker process. The criteria we developed to identify resources that can run a worker instance without interfering with other users' processes are given in the following subsections.

- CPU

We account for both user space processing (UserTime) and kernel processing (SystemTime). We also measure the number of active processes (load average). The constraints we developed are:

- SystemTime <= 35%, UserTime + SystemTime <= 70%
- Load Average / Total number of cores <= 3

These values help ensure that CPU resources are sufficiently idle before scheduling new tasks on a particular machine. Percentages are taken across all available cores.

- Memory

We consider both existing tasks that may be running in resource-constrained situations as well as the conditions for scheduling new tasks. Acceptable memory conditions are:

- Memory Consumed <= 70% (existing tasks)
- Memory Consumed <= 50% (new tasks)

Additionally, we inspect the percentage of I/O requests related to swap operations (paging). If swapping accounts for even a small portion of I/O, then the resource lacks memory. Therefore, we only consider resources that are not actively swapping.

- Disk I/O

For disk-based I/O, we inspect the average waiting time of all requests (await) and the average processing time of all requests (svctm). If latency is low, the difference between these two values is small; we require the difference to be less than or equal to 1. We also consider disk utilization, which must be at or below 30% of capacity averaged over a configurable window.

- Network I/O

We ensure that the network latency between processing resources and orchestration components (Speculator and Controller) is low with a configurable threshold. In our cluster, we considered machines with a ping latency above 5ms to be experiencing slow network conditions.

6.2.2 Data Collection from Software

There are many parameters in the discrete event simulation that may influence execution time in each simulation time unit. However, exposing them all is costly in time. Thus, we need to find which of them play more significant roles in predicting the execution time.

At the beginning, we will try to expose as many parameters as possible. From those parameters, we will use some filter or embedded methods to judge which of them are more significant in prediction. After that we will calculate the execution time of exposing each parameter. Based on the execution time cost and significance in prediction, we will determine which parameters to use in prediction. Also, we should judge whether there is correlation among those parameters. All the redundant parameters will be removed from the feature list.

Among all the exposed output variables, we picked 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 6.1.

Table 6.1 Prediction features from simulation

Feature Number	Feature Content
1	# farms managed by this worker
2	area in km2 managed by this worker
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

Feature Number	Feature Content
11	# adequate exposures by indirect contact
12	# adequate exposures by airborne spread
13	# farms vaccinated today
14	# vaccination tasks queued
15	area in km2 of the zones

There are many parameters in the discrete event simulation that may influence execution time in each simulation time unit. However, exposing them all and maintaining them is costly in time. Thus, we need to balance the overhead and the prediction accuracy.

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 worker 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.

6.3 LEARNING ALGORITHMS

6.3.1 Artificial Neural Networks (ANN)

In our work, we use ANN to predict our execution times in the next simulation day. The ANN is a good predictor for the regression problem. In our problem, we take the output variables from the NAADSM and the execution time from the previous simulation day as the prediction features.

However, we have to tune the parameters for the ANN to make the optimal prediction. For example, we tried number of hidden layers from 1 to 5 and number of nodes in hidden layers from 10 to 50. In this approach, we tuned the parameter in Table 6.2.

Table 6.2 Parameters for ANN

Java Version of ANN Package	Encog
Propagation Method	Resilient Propagation

Number of Hidden Layers	1
Number of Nodes in Hidden Layer	30
Ending Condition	For every 10 iterations, compared with the current error with the previous one, the difference is less than 1E-6

We put all the variables that are received from NAADSM and the execution time from the previous simulation day into the ANN to predict the execution time for the next simulation day. The training data is collected from 10 simulations we run without prediction. The number of training data points is about 200,000. The testing data is from a simulation running with prediction. The number of testing data points is about 20,000. And about 5,000 of them more than 5 seconds. The prediction result is shown in Figure 6.1.

From Figure 6.1, we show the prediction accuracy for the execution points more than 5 seconds. Because in our strategies, the merge threshold is 5 seconds, the data points whose execution times are less than 5 seconds are not helpful for the strategies. Thus, we remove all those points. The prediction accuracy is shown in Table 6.3.

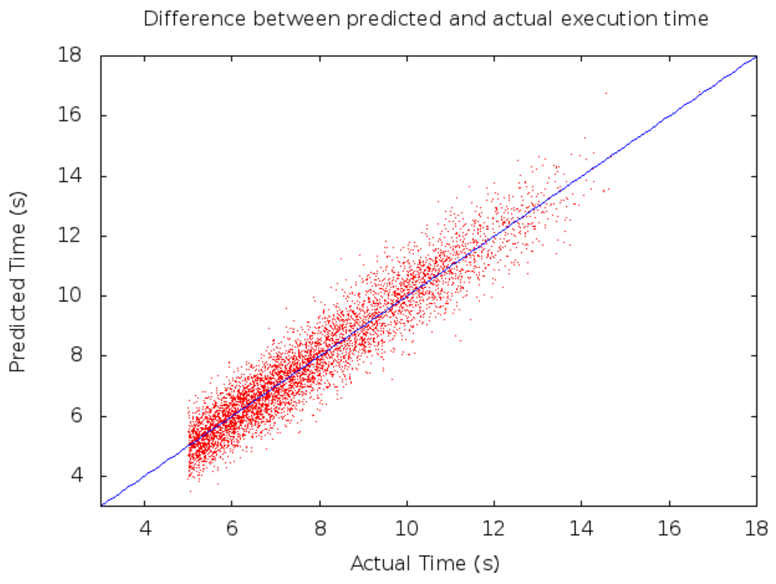

Figure 6.1 Difference between predicted and actual points

Table 6.3 Prediction accuracy based on time slots

Time slot	Prediction errors	# of samples
5s-8s	6.9%	3196
8s-11s	6.1%	1834
11s-14s	5.3%	619
>14s	6.3%	25

The data in Table 6.3 are collected from testing data. We can see that the prediction errors are all less than 7%. Except the points greater than 14 seconds, the prediction errors decrease when the execution times increase. Because the number of points greater than 14 seconds is only 25 (0.44% of the total number of samples), the prediction accuracy is acceptable.

6.3.2 Multi-Stage Neural Networks (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 an improved version of ANN. The basic idea of MSNN 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. Based on the granularity requirement, 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.

In our work, we use a MSNN to predict the execution time of a simulation day. Our prediction accuracy is depicted in Figure 6.2; 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.

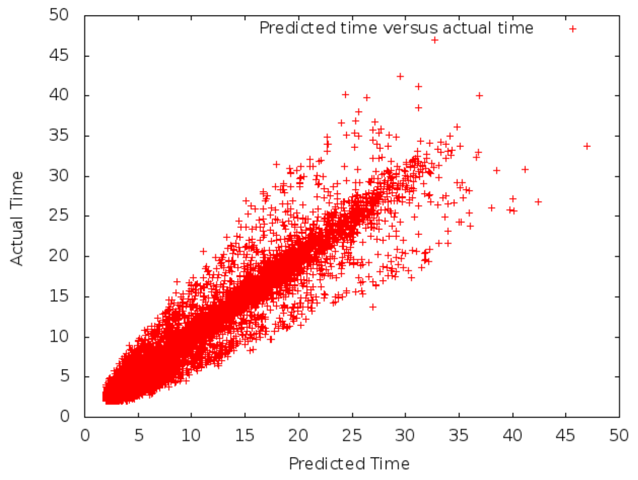

Figure 6.2 Prediction Accuracy

CHAPTER 7

PROACTIVE LOAD BALANCING STRATEGIES

7.1 CHALLENGES AND RESEARCH QUESTIONS

In proactive load balancing strategies, the load balancing mechanism is more complex. The more complex the mechanism is, the more overhead it may introduce. The first challenge of the proactive strategies is overhead control. The task of minimizing the overhead while implementing a smart mechanism to reduce the imbalance is hard. This is especially true since the most important goal for all the strategies is to minimize the load imbalance. Since we have prediction of the execution times, determining how to use this information to minimize the load imbalance becomes the second challenge.

In proactive load balancing strategies, we explore the following research questions.

- *How can we use improved execution time prediction to improve simulation performance?*
- *How can we ensure that the overheads associated with complex load balancing mechanisms do not outweigh gains in execution time reduction?*

7.2 DYNAMIC SPLIT AND MERGE (DSM) LOAD BALANCING STRATEGY

In dynamic split strategy, the efficiency decreases a lot when there are no more resources available. In dynamic split and merge strategy, we are trying to utilize the resources in a more efficient way. We do not only consider when to split the slow workers, but also merge the adjacent fast workers.

Even though there might be some lightly-loaded worker in the system, we ensure that each pair of adjacent workers is not lightly-loaded on average. Thus, the load balancing efficiency is relatively higher than dynamic split strategy.

The algorithm of dynamic split and merge strategy is:

```
if(maxTime>startToMergeThreshold){
    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 main idea is to split slowest workers if it is over the split threshold and merge two adjacent workers if the total execution time of them is less than the merge threshold. We have to avoid the situations that two workers keep on merging and splitting all the time because when we split it, the total execution time is less than the slowest worker, but when we merge them together, it is the slowest worker. Thus, we cannot set the merge and split threshold as 100%. In practice, we set both of them as 75%. Moreover, we never merge the workers when the slowest worker is faster than a threshold. This is because merge and split operations are not free. They bring in some overheads. In practice, we set the startToMergeThreshold as 5 seconds.

There will be a dynamic balance of workload during execution. Thus it is meaningless to test this strategy with too few workers. So we start the number of workers from 8. The results are shown in Figure 7.1.

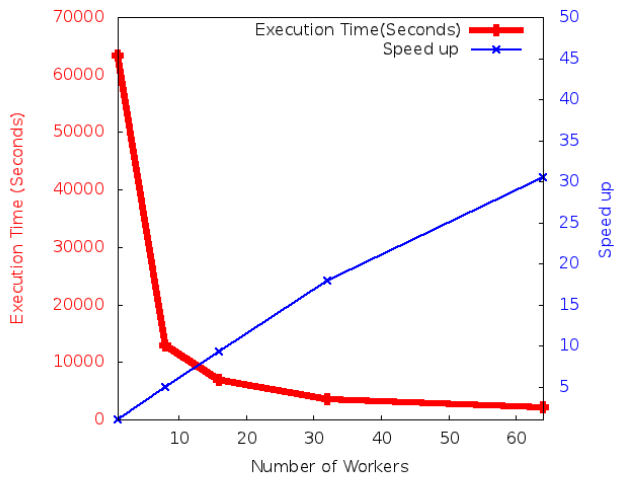

Figure 7.1 Speedups for dynamic split+merge strategy

In Figure 7.1 we can see that the speed up grows up as almost linear and the speed-up for 64 workers is around 30. We measure the growth trend of the line. It shows when we double the number of workers, we will get about 1.8 speed-up. Also we can see that, the speed-up from 32 workers to 64 workers is a little worse than the others. It is because when the execution time is faster, the overheads of communication and load balancing operations start to play a more important role.

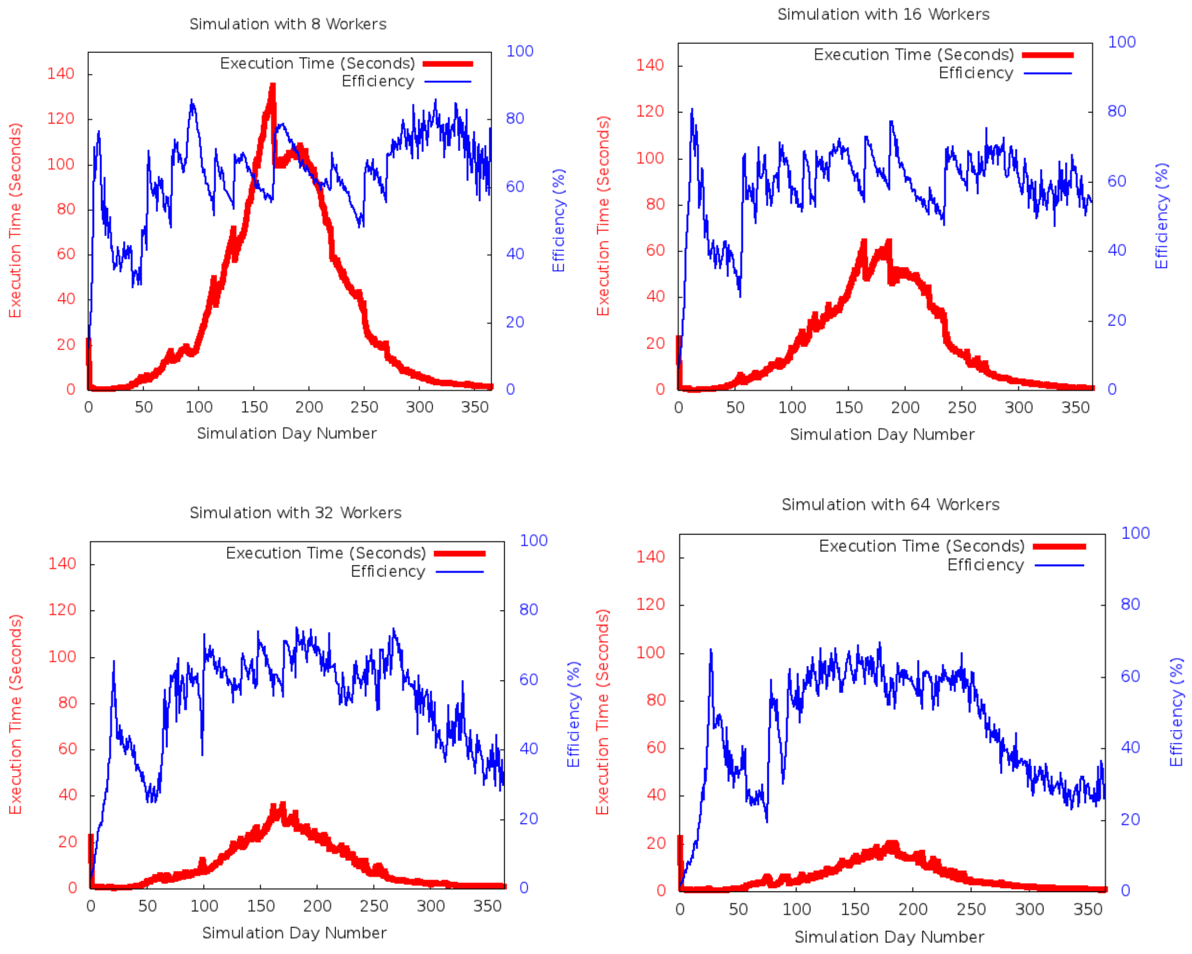

Figure 7.2 Execution time and load balancing efficiency for 8, 16, 32 and 64 workers scenarios in dynamic split and merge strategy

In Figure 7.2, we can compare the execution time and efficiency for 8, 16, 32, 64 workers scenarios. From all these figures, we can see that the load balancing efficiency is growing up at the beginning of the simulation due to the split operations. However, when the efficiency reaches a peak it starts to drop down. It is because the execution time is less than 5 seconds, there are no merge operations to optimize the resources. However, when the execution time reaches 5 seconds at some point around day 50, the merge operations start. After that, the load balancing efficiency goes up and stays on a relatively high level in the peak execution time period. However, the efficiency is going up and down periodically. This is because one load balancing operation makes the efficiency better and the efficiency drops down gradually due to the hotspots migration until another load balancing operation is applied. We can obviously see that the period is longer if there are fewer workers. It is because more workers give out more flexibility and more probability to merge workers to optimize the load balancing among resources.

The dynamic split and merge strategy is much better in both speed-up and efficiency than the other two strategies. It utilizes the resources in a much more efficient way. However, in some circumstances DSM suffers from a performance bottleneck. We divide the simulation work by geography, with each worker managing a geographically contiguous area. Therefore when we consider merging fast workers, we can only merge two workers that manage adjacent areas. There might be an idling worker between two slow workers but we cannot merge it. For example, consider the case of 64 workers where the execution time follows the pattern 70 seconds, 0 seconds, 70 seconds, 0 seconds, etc., plus one worker that takes 100 seconds. In this case, the load balancing efficiency is only about 35%, and because the fast workers are not adjacent to each other, DSM cannot improve the performance.

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 worker is split, the split is done such that the two new workers 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 workers 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.

7.3 Proactive Apportioning of Workloads (PAW) Strategy

Our PAW algorithm [67] addresses deficiencies in the DSM algorithm. If imbalances appear, this algorithm merges the entire workload together and splits it equally among all the workers. 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 7.3.

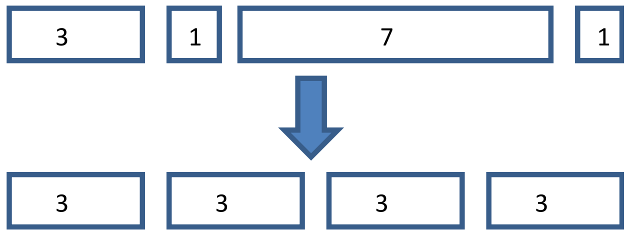

Figure 7.3 Example of the PAW algorithm

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 workers. In contrast, in the PAW algorithm, all of the workers 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 workers, 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 6.3.

The third key consideration for the PAW algorithm is the split mechanism. For instance, if we want to split one worker with a 40%-60% split, how should we do this? In the DSM algorithm, we tried splitting the worker such that the geographical area managed was divided 40/60, and we also tried splitting the worker 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.

In comparison with the DSM algorithm, the PAW algorithm fares better in scenarios with fewer workers. When there are only 2 or 4 workers, 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 7.4.

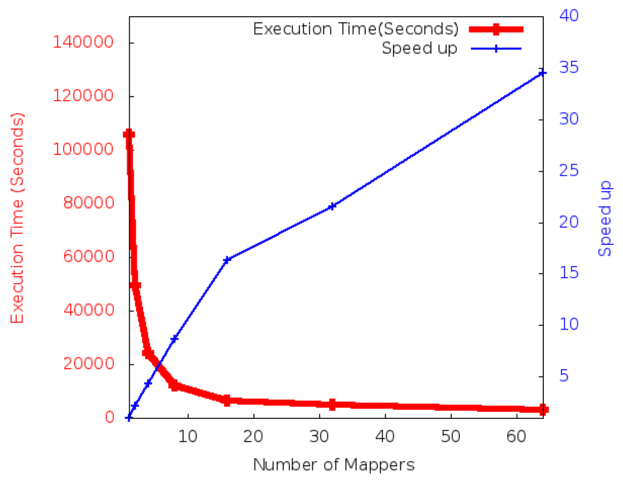

Figure 7.4 Speed-up for PAW algorithm

From Figure 7.4 we can see that the speed-up for 2, 4, 8 and 16 workers scenario is ideal. Furthermore, there is even a “super” speed-up for runs with smaller numbers of workers. The reason for this is the changing ratio of execution time to communication overhead time. If there are only 4 workers, the split and restore messages for the apportioning operation just need to be passed among 4 workers; but when the number of workers increases to 64, the number of messages passed for the apportioning operation increases. With the simulation work divided among 64 workers, 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 workers, the execution time of the slowest simulation day is always hundreds of seconds.

Figure 7.5 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 workers is small.

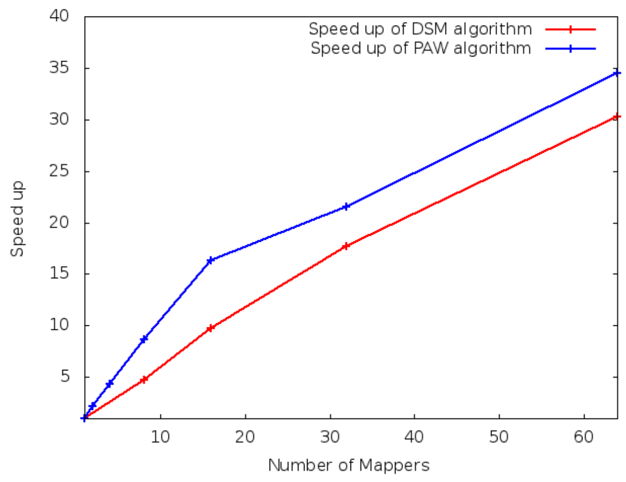

Figure 7.5 Speed up comparison for 2 load balancing algorithms

7.3 LEVERAGING HYPERTHREADING

In practice, we also tried some hyperthreading techniques to improve our performance. In previous experiments, we are running one worker on each physical core. Here we enable hyperthreading on each core. In this case, each core has two execution pipelines. Thus, the number of workers doubles but the resources we are using are still the same. The performance is shown in Figure 7.6.

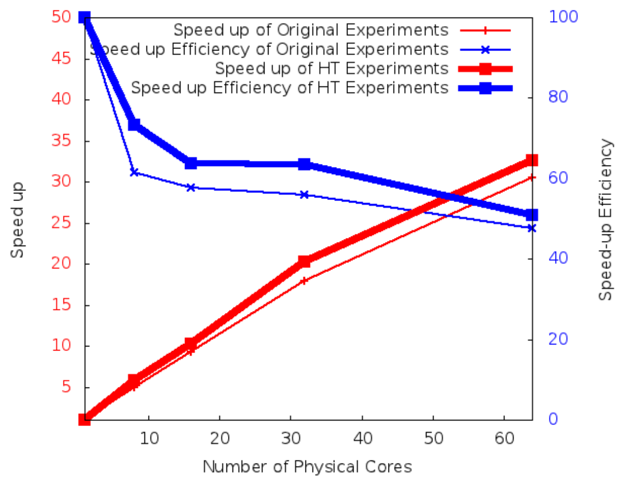

Figure 7.6 Speedups for hyperthreading experiments

From Figure 7.6, we can see that the speed-up and load balancing efficiency of hyperthreading are both better than the original implementation. With hyperthreading the number of workers doubles so that there is more flexibility for split and merges. Because the experiments do not reach the memory limit of the hardware, the performance is better because the utilization efficiency of CPU is better. The memory limitation is the main reason why we cannot have more pipelines for each physical core.

CHAPTER 8

Coping with Resource Uncertainty

In the modern computing landscape, failures are an issue that must be expected and dealt with rather than simply avoided or ignored [52]. The longer a software process runs, the more likely it is to experience a hardware or software failure. This fact is only exacerbated by the trend towards distributed, multi-core architectures that take advantage of the vast processing power found in today's commodity hardware. Each additional unit of processing involved in a task increases the overall probability of a failure occurring.

Our approach [66] to mitigating resource failures involves designing an autonomous fault tolerance agent to oversee the execution of distributed discrete event simulations. Unlike conventional distributed fault tolerance approaches, our system must cope with constantly changing, stateful computations and ensure global consistency throughout the system in the event of a failure. This requires system-level optimizations along with reliable prediction mechanisms that enable a dynamic, proactive approach to providing fault tolerant execution for our subject simulation. We rely on an adaptive strategy that determines when and how checkpoints should be requested in order to reduce the amount of duplicate work and overhead incurred from state collection. This is made possible by forecasting state changes and having the system plan accordingly.

Slowdown of a resource results in tasks executing on that resource executing significantly slower than other tasks executing concurrently on other resources. Resource slowdowns must be mitigated in a timely fashion because of their cascading effects. Discrete event simulations include synchronization barriers where different components of the simulation synchronize their state. Between successive synchronization barriers, the execution time is only as fast as the slowest task. This means that a single slow task greatly influences the overall execution time.

To mitigate resource slowdowns we rely on detection and circumvention. We predict resource slowdowns using artificial neural networks based on several factors, including the number of context switches, number of collocated processes, memory consumption, and paging. We use speculative tasks to alleviate slowdowns by launching slow tasks on faster machines. [65]

Resource heterogeneity in distributed settings occurs naturally as a result of how machines comprising a cluster are added to, upgraded, and retired. Coping with resource heterogeneity requires us to rank resources and continually monitor them to detect slowdowns. A critical aspect in dealing with resource heterogeneity is how tasks are apportioned. Large, compute-intensive tasks must be launched on faster machines. Such apportioning is needed to minimize imbalances between the tasks that comprise a simulation, resulting in faster completion times.

There are several challenges in ensuring autonomous execution of discrete event simulations in the presence of resource failures.   These include:

1. Computations composing the simulation are stateful: Though the simulation is stochastic, outcomes depend on the state built into each task within the simulation.

2. Simulation stalls during failures: Since the simulation is distributed over a collection of resources, the tasks composing the simulation must coordinate state among themselves at various synchronization barriers (the end of simulation day) throughout the execution. Here, the failure of a single machine can stall the entire simulation.

3. Overheads for failure recovery. The failure mitigation schemes should not introduce unacceptable overheads. The simulation orchestration should continue to have high speed-ups when failures do not occur.

8.1 RESOURCE FAILURE

8.1.1 Challenges

There are several challenges in ensuring autonomous execution of discrete event simulations in the presence of resource failures. These include:

- Computations composing the simulation are stateful: Though the simulation is stochastic, outcomes depend on the state built up at each task in the simulation.

- Simulation stalls during failures: State coordination is performed at synchronization points (the end of a simulation interval). Here, the failure of a single machine can stall the entire simulation.

8.1.2 Research Questions

Creating a fault-tolerant, distributed implementation of a discrete event simulation that requires stateful, interdependent tasks led us to pose a number of research questions that I have addressed in my work:

1. *Can we incorporate support for failure resilience while minimizing overheads? Since failures are infrequent, our goal should be to achieve fault tolerance without introducing unacceptable overheads into the system.*

2. *Can these overheads be minimized while also retaining the ability to cope with multiple, concurrent failures, which can either be permanent or transient?*

3. *Can failures be detected efficiently? Simply executing duplicate tasks in parallel is not an adequate scheme for dealing with failures in our system, so active detection of failures is required.*

4. *Will simply collecting state information from individual tasks be efficient means to provide fault tolerance? How can state collection be optimized?*

5. *In the event of a failure, can we minimize the amount of duplicate work?*

8.1.3 Speculator

In our system, the Speculator is responsible for all activities related to fault tolerance and speculative execution. This includes detecting failures, launching new Workers, rolling the simulation back to a consistent state, and managing resources. The Speculator can also be used to suspend a simulation, save its state to disk, and then resume execution — even on completely different hardware.

Along with these fault tolerance features, the Speculator also plays a role as an autonomous manager of system events, deciding the frequency of system state collection and allocation of resources that can be used by the Controller. These decisions allow the Speculator to provide its services without imposing a significant performance penalty on the execution of the simulation.

One key aspect of our system architecture is that communication only occurs once per simulation time unit, which is a simulation day in the case of NAADSM. This means that querying a Worker and receiving a result is a two-day process. Because of this constraint, the Speculator must be highly proactive; simply reacting to events as they take place is not sufficient due to the ever-changing state of the simulation.

As a simulation progresses, state updates are accumulated during each simulated day. In the case of a failure at any resource, the simulation cannot progress any further because all Workers must report state updates that may have effects outside their own territory before continuing. The challenges that arise in this situation are twofold: detection of failures, and failure recovery.

8.1.3.1 Failure Detection

To facilitate failure detection, the components in our system transmit small messages, called heartbeats, to the Speculator at regular intervals. Heartbeats contain system statistics, such as load information, CPU utilization, available memory, and disk activity, which are used when making fault tolerance decisions.

When a resource has not sent a heartbeat message after a configurable failure timeout threshold, the Speculator assumes it has failed. This assumption is confirmed by instructing the Controller to also attempt to contact the resource. In our tests, a heartbeat interval of 5 seconds resulted in a failure being detected in about 7.82 seconds on average.

To detect the special case of a Controller failure, the Speculator ensures that: (1) the time of the last heartbeat message from the Controller is greater than the failure timeout threshold and (2) each Worker has submitted a state bundle, signaling the end of the current simulation day. If the Controller has not instructed the Workers to begin the execution of the next day, then it has failed. The Speculator starts a new Controller instance on a different resource and transmits the current system state information to it, resuming execution.

Finally, it is possible that the Speculator itself experiences a failure. In situations where this may occur, multiple instances of the Speculator can be started transparently and operate in parallel; a simple election algorithm is used to determine the currently active instance.

8.1.3.2 Failure Recovery

Workers in our system maintain both local state and global state. Global state information is exchanged during each new simulation day, but local state is only used at the individual Worker level. With no fault tolerance considerations, this leads to an unrecoverable simulation in the event of a Worker failure. To cope with the possibility of a Worker failure, local state information is sent to the Speculator as a checkpoint. Checkpoints contain enough local state information for the Speculator to re-launch a Worker process. Unfortunately, simply re-launching a Worker is not enough to resume a simulation; all Workers must be executing the same simulation day, so the entire simulation must be rolled back to a consistent state.

Further complicating failure scenarios, there is no guarantee that the number of active Workers will stay constant during the execution of the simulation; in fact, it is highly likely that there will be a different number of active Workers at any point in time due to the system's dynamic splitting and merging functionality for load balancing. This means that the Speculator must also start up or suspend the appropriate number of Worker instances during a rollback operation.

Since the Speculator monitors the resources' system status information and the simulation state as well, it can reason about what information it will need to provide fault tolerance without impacting the overall performance of the system. For example, given a particular disease, the simulation may progress rapidly and then slow down as the disease spreads and becomes more computationally expensive. Figure 8.1 illustrates this situation, showing execution times in a 64-Worker simulation run of our Midwest scenario. In the early stages of a simulation, the Speculator does not need to request checkpoints frequently because a restart of the entire simulation would have a minimal impact on overall execution time. This property makes setting a hard “checkpoint interval” inefficient, and led us to develop a fault tolerance component designed around making intelligent, autonomous decisions about when and how checkpoints should be collected.

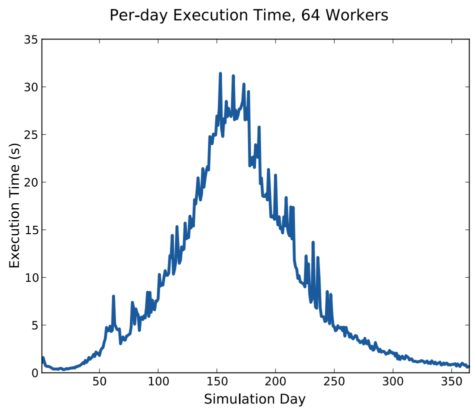

Figure 8.1 Per-day execution time of a 64-Worker Simulation of foot-and-mouth disease in Midwest USA.

8.1.4 Fault Tolerance Policy

The Speculator has several options at its disposal for providing fault tolerance, each with their own tradeoff space. How and when the Speculator requests checkpoints ultimately determines the performance impact of our approach and its scalability as more Workers are added. While thresholds can be set to guide the Speculator’s choices, hard rules tend to be brittle and ineffective when faced with different simulation and disease types. To provide a flexible model for determining the appropriate checkpointing policy, we predict the per-day execution time of the simulation, as well as the cost of generating a checkpoint. These future costs can then be evaluated against the likelihood of failures to produce a fault tolerance strategy.

8.1.5 Fault Tolerance Overhead

In order to evaluate the worst-case performance overhead incurred from our fault tolerance system, we ran two identical iterations of our Midwest scenario: one with no fault tolerance functionality enabled, and one with checkpoint requests occurring every simulation day. Figure 8.2 shows the per-day cost of collecting checkpoints in our system; the overhead accounted for a five-minute increase in execution time, but the majority was during non-critical execution at the end of the simulation. These costs can be avoided with accurate forecasting, as our benchmarks have shown.

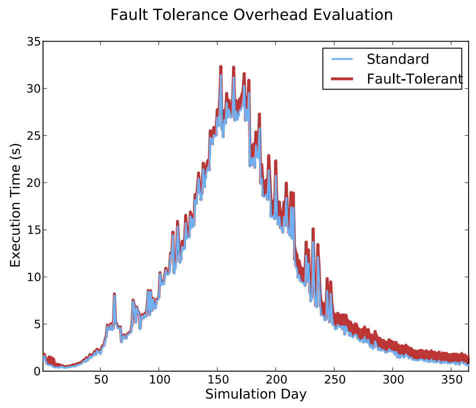

Figure 8.2 Shaded areas above the standard execution times represent overhead from checkpointing.

8.2 RESOURCE SLOWDOWNS

When simulations execute over a collection of resources, uncertainty stems from: resource heterogeneity, how tasks are apportioned among heterogeneous resources, load spikes at the resources, and load variations due to the information exchange at synchronization barriers within a simulation.

It is not unusual for a resource to have spikes in the number of processes that are executing concurrently, leading to increased contention and thus strains on CPU, memory and disk utilization. We need to be able to detect such resource slowdowns and circumvent them via apportioning of tasks or by launching speculative tasks that duplicate the processing being performed on the slow resource.

Since multiple workers execute each simulation day in parallel, imbalances may exist. Tasks that complete earlier must wait until the slowest task completes; only then is it possible to synchronize state among the tasks and proceed to the next simulation day. In general, speed and throughput drop as imbalances occur.

8.2.1 Challenges

Our research objective is faster, distributed execution of simulations in the presence of such uncertainty. There are several challenges involved in accomplishing this:

- The number and frequency of synchronization barriers is high: Distributed simulations often have multiple synchronization barriers during the course of execution. These synchronization barriers are used to exchange information about state-changes within particular subtasks that have an impact on the entire simulation. In our example case of epidemiological modeling, subtasks synchronize their simulation state at the end of each simulation day.

- Apportioning tasks among heterogeneous resources: Since the simulation is stochastic, the time spent by each task on a particular simulation day varies. The execution times cannot be calculated in a deterministic fashion either before or during the simulation run. Apportioning of tasks must account for two objectives: (1) minimization of imbalances and (2) avoidance of resource slowdowns.

- Detecting and circumventing resource slowdowns: There is no way to determine what percentage of a task has been completed: once a task begins work on a particular simulation day, the only information available is whether the task is still executing, has completed, or has failed. Slowdown detection must be done as the task is executing and must be inferred based on factors such as CPU utilization, memory availability and utilization, disk I/O, and network I/O.

- Mitigating imbalances in workloads: Imbalances may exist between synchronization barriers. These imbalances may either be due to the scope of a particular task or the machine that it is executing on. If an attempt is not made to mitigate these imbalances, they will persist for the next synchronization barrier and could get worse, leading to longer completion times.

- Interference uncertainty: The interference (due to other concurrent processes) at a machine is unpredictable. This interference could be periodic or bursty, and can occur at any machine at any time. The greater the interference, the greater the CPU and memory contention, leading to correspondingly longer execution times.

8.2.2 Research Questions

Research questions that we explore in the context of orchestrating simulations in the presence of resource heterogeneity and slowdowns include the following:

- *There is a mix of resources in the system. Can we profile these resources and rank them accordingly?*
- *How can we apportion tasks so that we exploit this heterogeneity? Can we distribute tasks across resources such that the total execution time for a simulation day is minimized?*
- *How can we cope with resource slowdowns? To accomplish this we must be able to identify slowdowns and initiate counter measures to mitigate them.*
- *How can we ensure fast completion times in such settings?*

8.2.3 Methodology: Circumventing Slowdowns and Coping with Heterogeneity

Our approach targets several aspects of coping with resource uncertainty. These include: profiling resources, apportioning tasks while taking into account both the complexity of the task and resource profile, identification of resources that are slowing down, and launching speculative tasks on faster resources to circumvent stragglers.

We gather resource capabilities such as the number of cores and available, physical memory, and along with a software benchmark use these profiles to normalize and rank resources. We use this ranking of resources to launch compute intensive tasks on best available resources, which are likely to provide the shortest completion times for the task. We also launch straggler tasks based on this resource ranking.

We apportion tasks over the available resources based on the expected completion times for tasks between synchronization barriers. The objective of this apportioning is to minimize imbalances between tasks, i.e., to ensure that tasks complete at more or less the same time between synchronization barriers. Tasks that are likely to be compute-intensive and long running must be executed on faster machines.

Since the simulation is stochastic, we cannot deterministically calculate the task completion times; rather, we need to predict them. We predict execution times at a fine-grained level: for each task for each simulation day. We also predict completion times for tasks between synchronization barriers. For the simulation that we consider, this prediction is based on disease biology (incubation period, infectious period, etc.), geographical scope of the region being managed by the particular task, and the disease prevalence within the region at the particular synchronization barrier. We use artificial neural networks to make this prediction.

Next, we identify tasks that are likely to impact execution time. These could be tasks that have a disproportionate share of the processing workload (even though they may execute on faster machines) or may be executing on slower machines. This allows us to identify tasks that are likely to take longer to execute and introduce imbalances where other tasks must wait at the synchronization barrier. We use the synchronization barriers to gather this information and also to apportion workloads to mitigate imbalances.

Another issue that needs to be accounted for is stragglers, which are tasks with longer than average completion times. There are two aspects related to dealing with stragglers. First, we need to identify tasks that are likely to be stragglers. Second, we must circumvent stragglers by launching speculative (backup) tasks on faster resources.

We use speculative tasks as a mechanism to mitigate resource slowdowns. Doing so will require us to solve three problems: the which, where and when problems. Deciding which tasks to launch will involve detecting which workers are running more slowly than expected and which are likely to slow down soon. Deciding where to launch the tasks will involve resource state prediction and considerations around data locality and network topology. Deciding when to launch will involve solving a tradeoff: we want to respond swiftly to changing conditions, but avoid excessive overhead from too-frequent launches.

To identify tasks that are likely to be stragglers, we predict when resources will slow down. We use an ANN to predict slowdowns based on tracking CPU utilization, the number of processes, and memory utilization including paging. Since execution times also increase because of an increase in CPU-bound processes that result in increased context switches, we track them as well.

We then launch speculative tasks for stragglers on better machines. Our approach identifies: (1) a set of machines that have spare capacity to take on speculative tasks, and (2) the stragglers that are most likely to benefit the simulation (i.e., faster executions of these tasks would decrease imbalances and result in shorter overall execution time).

In addition to considering the factors that can slow down a simulation, we will also consider the ideal case, when resources are well behaved and there is no interference from other users. We will quantify the overhead added by the speculative launch mechanism, and investigate whether there can even be some performance gain in this situation.

8.2.4 Harnessing Resource Profiles

Our algorithm is built around a sorted list in which each node represents a machine, and the list is sorted by the current speed of the machine. The current speed is the product of the "normalized speed" (as measured by the benchmark program) and a slowdown factor. The slowdown factor combines information about the short-term usage of the machine and the long-term trend. The long-term trend is the measured historical pattern of usage at certain times of the day.

Each node of the list contains the items below:

1. Number of available resources
2. Number of occupied resources
3. Information about the hardware

a) CPU information

b) Memory information

c) Disk I/O information

d) Network I/O information

4. Normalized speed α_i and slowdown factor Si.

The Controller and Speculator use this list to find the fastest machine. When launching speculative tasks, the system will compare the predicted execution time of the task on the current machine and the fastest spare machine in the list. Since the list is sorted during each update, finding the fastest few machines for speculative launching is fast.

8.2.4.1 List updating mechanism

The list is updated frequently, with the timing of the updates based on two heartbeats. The more frequent heartbeat occurs every 10 seconds, and it triggers a check of the CPU use, memory use, network use, and disk I/O. If the memory use, network use, or disk I/O is over threshold as defined in the previous section, then the slowdown factor is set to 0. A slowdown factor of zero will make the computed current speed zero, thus placing the machine at the bottom of the sorted list and preventing new speculative tasks on the machine. Otherwise, the measures of CPU use are combined into a vector and fed to an ANN to predict the slowdown factor.

The less frequent heartbeat occurs every 5 minutes, and it triggers a check of the historical trends data. We measured the usage of all machines in our cluster for 24 hours to discover patterns. For example, some machines are used for classes during the daytime, but are idling at night. There are also programs launched at specific times of the day by administrators and by other research groups. If the historical usage data shows that the machine will soon be occupied, the slowdown factor is set to zero. The process of list updating mechanism is shown in Figure 8.3.

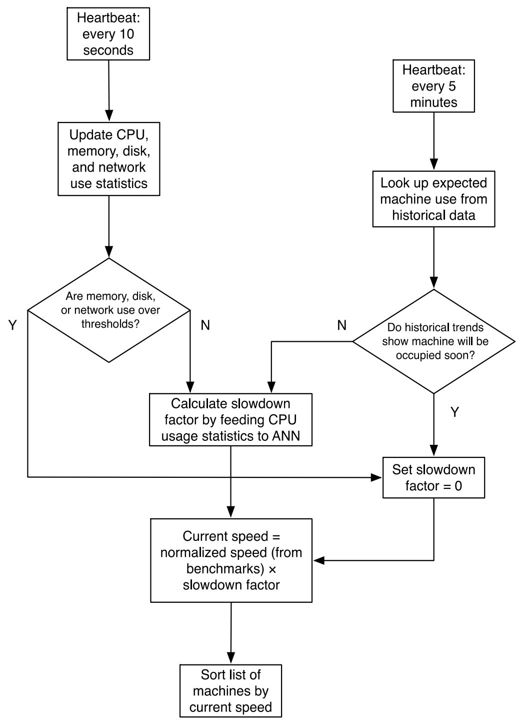
Heartbeat: every 10 seconds
Update CPU, memory, disk, and network use statistics
Are memory, disk, or network use over thresholds?
Y
N
Heartbeat: every 5 minutes
Look up expected machine use from historical data
Do historical trends show machine will be occupied soon?
N
Y
Calculate slowdown factor by feeding CPU usage statistics to ANN
Set slowdown factor = 0
Current speed = normalized speed (from benchmarks) × slowdown factor
Sort list of machines by current speed

Figure 8.3 List Updating Mechanism

8.2.4.2 Launching speculative tasks

Before launching a speculative task, we consider whether the operation is likely to provide a performance gain. Launching a speculative task will impose overheads, specifically, transferring a checkpoint over the network. We assume the network transfer latency for checkpoint has a linear relationship with the checkpoint size.

The predicted time for a task to run on the current machine i is represented as T. Given the normalized speeds and the slowdown factors found for machines i and j, and the transfer latency T_{trans}, then the predicted time to transfer a checkpoint to machine j and execute the task on machine j is $\frac{\alpha_j S_j}{\alpha_i S_i}T + T_{trans}$. If $\frac{\alpha_j S_j}{\alpha_i S_i}T + T_{trans} < T$, we launch the speculative task and we predict the performance will be better.

There are two ways the system decides to launch speculative tasks. Between simulation days, the system will decide whether or not to launch speculative tasks. At this time, speculative tasks can start at the same time as the normal tasks, avoiding additional latency costs. The second way to launch a speculative task is based on the heartbeat that checks machines' current speed. If a machine is operating slower than our system would expect, we predict whether performance would be improved by launching speculative tasks for the all of the tasks currently on the slow machine onto the fastest machine. This helps avoid toggling, where loads constantly shift between two machines.

Our system already contains a fault tolerance process in which checkpoints are requested when necessary. However, the slowdown detection mechanism requires a checkpoint for each simulation day on each worker to enable fast and dynamic migration of processes. Therefore, workers also store a local checkpoint on their disk after each synchronization point.

When speculative workers are launched, they get added to the worker list alongside the other workers that are responsible for the same task. Whichever worker finishes the task first will be kept active for subsequent simulation days; the other workers that were managing the same task are "cleaned up" and returned to the worker pool. When requested, workers can directly transmit their state information to another worker instance to create a speculative task. This avoids the need for centralized communication through the Controller or Speculator.

8.2.5 Performance evaluation of managing heterogeneity and circumventing slowdowns

As a first step, we measured the overhead of our slow worker detection mechanism. In comparison with the original Granules-NAADSM framework, the new framework adds a checkpoint operation every simulation day. This new step could have a negative impact on execution time.

In this experiment, each worker requested a complete checkpoint message from the NAADSM simulator and stored it on the local disk. There is no network communication or data compression used in this process. We measured the time gap between the checkpoint request and feedback from the simulation. The overhead versus execution time is shown in Figure 8.4. We see that the overhead time is almost constant, around 500ms for all simulation days. In comparison with the execution time of more than 5 seconds on an average simulation day, this overhead is small.

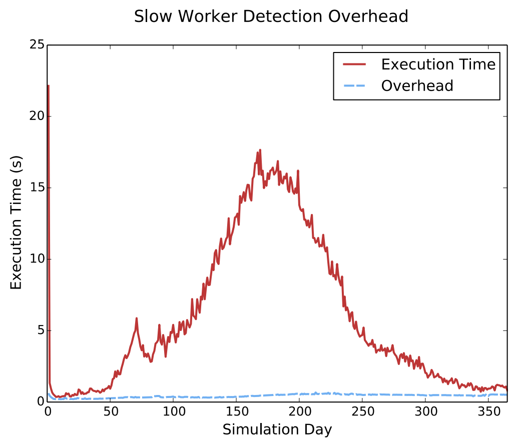

Figure 8.4 Execution time vs. overhead

Our test environment was a cluster of heterogeneous machines, with considerable variation in CPU speed, total memory, and network speed. During the experiments, there might or might not be interference from other users. In our experimental design, we show results from both situations. Even if there is no interference from other users, our new framework can still provide improved execution times. Before In previous versions of our distributed orchestration system, worker processes were launched in a random pattern. With the slowdown detection mechanism, machines are ranked by current speed, and so worker processes will be launched onto the fastest available machines. The comparison of performance with random machine selection vs. selection of the fastest machine is shown in Figure 8.5.

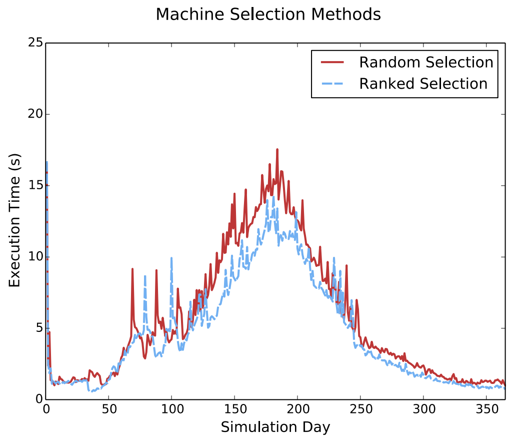

Figure 8.5 Ranked selection comparison

The total execution time with random machine selection is 2034.5 seconds, and with selection of the fastest machine is 1684.7 seconds, an improvement of 17.2%. (The peaks around simulation day 100 occur because there is a merge threshold of 5 seconds. The system does not do merge operations when the execution time is less than 5 seconds to avoid a situation where split and merge overheads dominate the execution time.) As shown here, slowdown detection works well even when there is no interference from other users of the cluster.

In the next experiment, we added interference into the cluster by launching resource-intensive programs on specific machines before starting the simulation. We found that the split-merge strategy performs well even without the speculative launch mechanism. This is depicted in Figure 8.6. From Figure 8.6, we can see that the performance with some fixed interference is similar to the performance without interference, even when we are not using the speculative launch mechanism. The reason for this is the machine selection mechanism avoids using the busy machines.

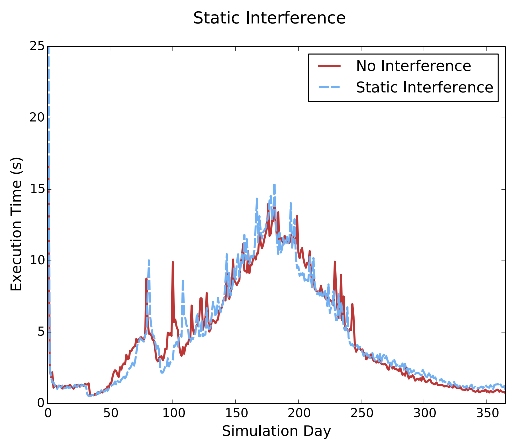

Figure 8.6 Execution time comparison with/without interference

In the next experiment, we launched resource-intensive programs randomly during the execution of the simulation. The performance comparison with and without the speculative launch mechanism is shown in Figure 8.7. Under conditions of unpredictable interference, the total execution without the slowdown detection mechanism is 2216.7 seconds. With the slowdown detection mechanism, the total execution time is 1924.0 seconds, an improvement of 13.2%.

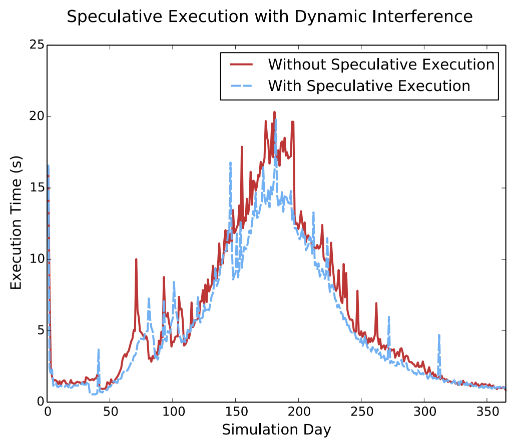

Figure 8.7 Execution time comparison with/without slow detection mechanism

CHAPTER 9

PERFORMANCE IN MACHINE SETTINGS

Virtualization [71][72] is an important technology in distributed system and cloud computing. Some of the machines in a cluster or cloud are so powerful that running one process on it only utilizes a very small amount of its resources. One approach is to share resources of a machine through virtual machines (VMs) so that resources can be utilized more effectively. However, there are some tradeoffs if we are using virtual machines.

- The Xen [69] hypervisor introduces communication overheads. These costs are unavoidable as the performance of an application running on Xen is slower than running on physical hardware.
- The I/O request mechanism in Xen is interleaved for different domains. However, if there are so many domains that I/O in domain0 meets a bottleneck, some I/O requests are delayed. This problem is mentioned by Kang et al. [53].
- In the Xen system, all the resources allocated to domains are isolated. Two processes running on the same machine but not the same virtual machine cannot share resources. For example, if there are two processes running on one machine which has 2G memory and one process needs 1.5G memory but the other needs almost no memory, they can both run. If both of them are running on the virtual machine with 1G memory, the process that needs 1.5G memory will run slowly and degrade performance.

Since there are so advantages and disadvantages using virtual machines. We also run our experiments on virtual machines to observe whether the performance is good or not.

9.1 Experimental Results on Virtual Machines

We 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 [75] on Fedora 20.

First, we ran a scenario on 1 worker in the VM setting and contrasted the execution time with that in a physical machine setting. This is depicted in Figure 9.1. Virtualization overheads become more noticeable during peak loads of the simulation.

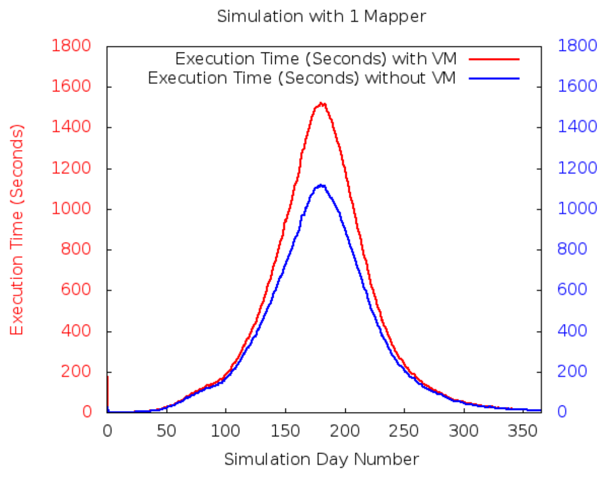

Figure 9.1 Execution time comparison w/o VMs for 1 worker scenario

We also executed the scenario with 32 workers; this is depicted in Figure 9.2. 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 workers increased. For example, the overhead (i.e. the increase in execution time for the exact same outbreak with the same number of workers) for the 1 worker scenario is 26.5%, but for the 32-worker scenario this reduces to 3.9%, and for the 64-worker 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.

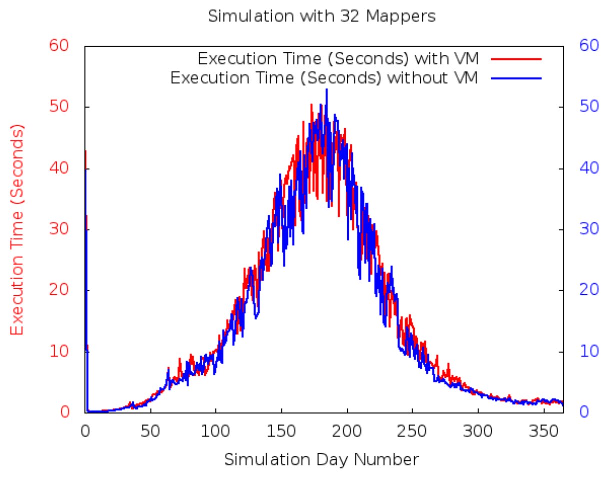

Figure 9.2 Execution time comparison w/o VMs for 32 workers scenario

CHAPTER 10

CONTRIBUTIONS AND FUTURE WORK

10.1 CONCLUSION

This dissertation describes the design of algorithms for the distributed orchestration of stochastic discrete event simulations. The primary objective in these algorithms is to ensure faster completion times for the simulations. This must be achieved while accounting for stochastic loads at subtasks that comprise the simulation and also in the presence of resource uncertainty encompassing failures, slowdowns and heterogeneity. We have implemented these algorithms into our Granules stream processing runtime and profiled the performance of these algorithms of physical and virtual machines. Specific insights that were gleaned over the course of this dissertation include the following:

1. Between synchronization points the simulation is only as fast as the slowest worker. Targeting imbalances across subtasks between synchronization points ensures that the average waiting times for individual tasks (over the duration of the simulation) is reduced, leading to faster completion times.

2. While static load balancing algorithms introduce the lowest costs for making scheduling decisions, they are unable to respond to changes in computational loads at subtasks. Changes in the computational loads occur either due to the stochastic nature of the simulation or due to resource uncertainty such as slowdowns or contentions.

3. Dynamic load balancing algorithms are able to cope with changes in both the simulation as well as resource uncertainty.

4. Reactive algorithms work well in situations where load imbalances are rare. The system then only needs to focus on identifying performance hotspots in a lightweight fashion and alleviating them.

5. Proactive algorithms are well suited for situations where load imbalances are not rare. These algorithms are well-suited to cope with situations where there might be short spikes in load that result in significant slowdowns.

6. Since the load changes are short-lived it is important to ensure that the scheduling and load balancing decisions are made in a timely fashion.

7. Algorithms for coping with resource uncertainty must ensure that the overheads to circumvent them do not introduce unacceptable overheads in situations where resource uncertainty does not exist.

8. However, care must be taken to ensure that the overheads in the scheduling decisions do not outpace gains in execution time reduction.

9. Scheduling algorithms can sometimes produce oscillatory behavior where the system oscillates continually between two states i.e. decisions made in one round are negated by a

conflicting decision made in the subsequent round and so on. Algorithms must have thresholds in place to damp such oscillations.

10.2 CONTRIBUTION

The dissertation has resulted in a collection of algorithms for the distributed orchestration of stochastic discrete event simulations. While the primary objective is the reduction of overall execution times, the dissertation also addressed issues relating to resource uncertainty – slowdowns, failures, and heterogeneity – that have a negative impact on execution times.

Our specific contributions include:

1. Our approach has broad applicability to discrete event simulations. Our approach requires minimal modifications to the discrete event simulation unlike approaches based on MPI or OpenMP.

2. To our knowledge, this is the first attempt at using a Stream Processing based system to orchestrate discrete event simulations.

3. The discrete event simulation, NAADSM, is among the most difficult classes. The computations are highly irregular, the tasks need to preserve geographical contiguousness, and the computational loads are stochastic. Since our approach works on such a difficult DES, it could also apply on most of the others.

4. The dissertation includes a suite of algorithms that explore the trade-offs and approaches of orchestrating discrete event simulations. The spectrum that these algorithms have explored include: (1) static and dynamic scheduling, (2) reactive and proactive scheduling, (3) learning and non-learning based scheduling, (4) scheduling based on approaches that view the simulation as a black-box and white-box, and (5) the ability to leverage hardware threads.

5. We have also devised algorithms that allow us to cope with resource uncertainty in the form of resource failures, slowdowns and heterogeneity. Our use of a speculative checkpointing strategy allows us to delay checkpointing till such time that the costs for not doing so are exceeded by the costs for performing a simulation rollbacks.

6. Our slowdown circumvention algorithm normalizes resources, and ensures that resources that are more powerful take on a proportionally greater share of the processing load. The slowdown circumvention algorithm also accounts for resources with high contention, which in turn leads to higher context-switches, and reduced performed. Speculative tasks that are launched to account for stragglers account for the quality of the resource as well as the likelihood that the speculative task completes before the straggler.

7. We have profiled the overheads introduced in virtualized infrastructures.

Ultimately, using our suite of algorithms as the number of processing elements double, we achieve a change in speed of 1.8. In comparison with naive algorithms such as static split and dynamic split, our performance gain is about 112.1%. We achieve this by ensuring that the overheads associated with the scheduling decisions do not outpace gains in execution time reductions.

Finally, several of the load balancing algorithms and strategies that we have devised are broadly applicable to traditional applications that execute in distributed setting.

10.3 FUTURE WORK

There are a few avenues for further research.

There is room for improvement in the prediction methods. In our current approach, simulation output variables are used for prediction by the load balancing algorithms, while hardware parameters are used for slowdown predictions. In the future, we may combine these parameters to get better prediction accuracy.

For our load apportioning algorithms, we would like to explore an approach that results in better partitioning. Currently, we find that population density correlates with load density; as such, our partitions are based on that. A refinement that adjusts the splits based on state at each task would further reduce imbalances across synchronization points.

Finally, we would also like to explore tuning and customization of these algorithms to other discrete event simulations and identify which ones provide the best performance sweet spot.

BIBLIOGRAPHY

[1] Matloff, N. (2008). Introduction to discrete-event simulation and the simpy language. Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August, 2, 2009.

[2] Chicago de Toro A, A., “Assessment of Field Machinery Performance in Variable Weather Conditions Using Discrete Event Simulation”, Department of Biometry and Engineering, Swedish University of Agricultural Sciences, 2004

[3] Deelman, Ewa, Boleslaw K. Szymanski, and Thomas Caraco. "Simulating Lyme disease using parallel discrete event simulation." In Proceedings of the 28th conference on Winter simulation, pp. 1191-1198. IEEE Computer Society, 1996.

[4] Thulasidasan, Sunil, Shiva Kasiviswanathan, Stephan Eidenbenz, Emanuele Galli, Susan Mniszewski, and Philip Romero. "Designing systems for large-scale, discrete-event simulations: Experiences with the FastTrans parallel microsimulator." In High Performance Computing (HiPC), 2009 International Conference on, pp. 428-437. IEEE, 2009.

[5] Gonsiorowski, Elsa, Christopher Carothers, and Carl Tropper. "Modeling large scale circuits using massively parallel discrete-event simulation." In Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on, pp. 127-133. IEEE, 2012.

[6] Mouftah, H.T.; Sturgeon, R.P., "Distributed discrete event simulation for communication networks," Selected Areas in Communications, IEEE Journal on , vol.8, no.9, pp.1723,1734, Dec 1990

[7] Zhiquan Sui and Shrideep Pallickara. A Survey Of Load Balancing Techniques for Data Intensive Computing. Handbook of Data Intensive Computing. Chapter 6, pp 157-168. Springer. 2011. ISBN 978-1-4614-1415-5.

[8] Zhiquan Sui, Neil Harvey and Shrideep Pallickara. On the Distributed Orchestration of Stochastic Discrete Event Simulations. Concurrency and Computation: Practice & Experience. John-Wiley. Vol.26(11), pp 1889–1907. 2014.

[9] Jayadev Misra. 1986. Distributed discrete-event simulation. ACM Comput. Surv. 18, 1 (March 1986), 39-65.

[10] Seo, Chungman, Youngshin Han, Haeyoung Lee, and Chilgee Lee. "Implementation of cloud computing environment for discrete event system simulation using service oriented architecture." In 2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, pp. 359-362. 2010.

[11] Kurt Vanmechelen, Silas De Munck, Jan Broeckhove, "Conservative Distributed Discrete Event Simulation on Amazon EC2," Cluster Computing and the Grid, IEEE International Symposium on, pp. 853-860, 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), 2012

[12] 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.

[13] 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.

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

[15] 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.

[16] 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

[17] Butkus, Albert, Kevin Roe, Barbara L. Mitchell, and Timothy Payne. "Space Surveillance Network and Analysis Model (SSNAM) Performance Improvements." In DoD High Performance Computing Modernization Program Users Group Conference, 2007, pp. 469-473. IEEE, 2007.

[18] 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, Ed.: S. Ryan, The Maui Economic Development Board., p.E34, Sep 2010

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

[20] Springel, V. (2005), The cosmological simulation code gadget-2. Monthly Notices of the Royal Astronomical Society, 364: 1105–1134.

[21] Hilbert curve, http://en.wikipedia.org/wiki/Hilbert_curve

[22] Thulasidasan, Sunil, Shiva Prasad Kasiviswanathan, Stephan Eidenbenz, and Phillip Romero. "Explicit spatial scattering for load balancing in conservatively synchronized parallel discrete event simulations." In Proceedings of the 2010 IEEE Workshop on Principles of Advanced and Distributed Simulation, pp. 150-158. IEEE Computer Society, 2010.

[23] Ewa Deelman and Boleslaw K. Szymanski. 1998. Dynamic load balancing in parallel discrete event simulation for spatially explicit problems. In Proceedings of the twelfth workshop on Parallel and distributed simulation (PADS '98). IEEE Computer Society, Washington, DC, USA, 46-53.

[24] Penmatsa, Satish, and Anthony T. Chronopoulos. "Dynamic multi-user load balancing in distributed systems." In Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pp. 1-10. IEEE, 2007.

[25] Karush-Kuhn Theorem , https://en.wikipedia.org/wiki/Karush-Kuhn-Tucker_conditions

[26] Campos, Luis Miguel, and Isaac D. Scherson. "Rate of change load balancing in distributed and parallel systems." Parallel Computing 26, no. 9 (2000): 1213-1230.

[27] Maeng, Hye-Seon, Hyoun-Su Lee, Tack-Don Han, Sung-Bong Yang, and Shin-Dug Kim. "Dynamic load balancing of iterative data parallel problems on a workstation cluster." In High Performance Computing on the Information Superhighway, 1997. HPC Asia'97, pp. 563-567. IEEE, 1997.

[28] 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

[29] Wang, Jun, Jian-Wen Chen, Yong-Liang Wang, and Di Zheng. "Intelligent load balancing strategies for complex distributed simulation applications." In Computational Intelligence and Security, 2009. CIS'09. International Conference on, vol. 2, pp. 182-186. IEEE, 2009.

[30] Dantas, Mario AR, and Alex R. Pinto. "A load balancing approach based on a genetic machine learning algorithm." In High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on, pp. 124-130. IEEE, 2005.

[31] Munetomo, Masaharu, N. K. Takai, and Yoshiharu Sato. "A stochastic genetic algorithm for dynamic load balancing in distributed systems." In Systems, Man and Cybernetics, 1995.

Intelligent Systems for the 21st Century., IEEE International Conference on, vol. 4, pp. 3795-3799. IEEE, 1995.

[32] Lakshmanan, Geetika T., and Robert E. Strom. "Biologically-inspired distributed middleware management for stream processing systems." In Middleware 2008, pp. 223-242. Springer Berlin Heidelberg, 2008.

[33] Fujimoto, Richard M. Time warp on a shared memory multiprocessor. No. UUCS-88-021A. UTAH UNIV SALT LAKE CITY SCHOOL OF COMPUTING, 1989.

[34] 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.

[35] 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.

[36] 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, Ed.: S. Ryan, The Maui Economic Development Board., p.E34, Sep 2010

[37] 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.

[38] Liu, Yang, Maozhen Li, Nasullah Khalid Alham, Suhel Hammoud, and Mahesh Ponraj. "Load balancing in MapReduce environments for data intensive applications." In Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on, vol. 4, pp. 2675-2678. IEEE, 2011.

[39] Kirpal A. Venkatesh; Kishorekumar Neelamegam; R. Revathy, "Using MapReduce and load balancing on the cloud: Hadoop MapReduce and virtualization improves node performance", http://www.ibm.com/developerworks/cloud/library/cl-mapreduce/#ibm-pcon

[40] 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.

[41] Ostermann, Simon, Kassian Plankensteiner, Radu Prodan, and Thomas Fahringer. "GroudSim: an event-based simulation framework for computational grids and clouds." In Euro-Par 2010 Parallel Processing Workshops, pp. 305-313. Springer Berlin Heidelberg, 2011.

[42] Zheng, Gengbin, Gunavardhan Kakulapati, and Laxmikant V. Kalé. "Bigsim: A parallel simulator for performance prediction of extremely large parallel machines." In Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, p. 78. IEEE, 2004.

[43] Maria Chtepen, Filip HA Claeys, Bart Dhoedt, Filip De Turck, Piet Demeester, and Peter A Vanrolleghem. 2009. Adaptive task checkpointing and replication: Toward efficient fault-tolerant grids. Parallel and Distributed Systems, IEEE Transactions on 20, 2 (2009), 180-190.

[44] Rami Debouk, Stéphane Lafortune, and Demosthenis Teneketzis. 2000. Coordinated decentralized protocols for failure diagnosis of discrete event systems. Discrete Event Dynamic Systems 10, 1-2 (2000), 33-86.

[45] Wenbin Qiu and Ratnesh Kumar. 2006. Decentralized failure diagnosis of discrete event systems. Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on 36, 2 (2006), 384-395.

[46] Indranil Gupta, Tushar D. Chandra, and Germán S. Goldszmidt. 2001. On scalable and efficient distributed failure detectors. In Proceedings of the twentieth annual ACM symposium on Principles of distributed computing (PODC '01). ACM, New York, NY, USA, 170-179.

[47] Jorge Luis Ramírez Ortiz and Ricardo Marcelín Jiménez. 2011. Fault-tolerant Distributed Discrete Event Simulator Based on a P2P Architecture. In SIMUL 2011, The Third International Conference on Advances in System Simulation. 21-26.

[48] Xie, Jiong, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, James Majors, Adam Manzanares, and Xiao Qin. "Improving mapreduce performance through data placement in heterogeneous hadoop clusters." In Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, pp. 1-9. IEEE, 2010.

[49] Lee, Gunho, Byung-Gon Chun, and Randy H. Katz. "Heterogeneity-aware resource allocation and scheduling in the cloud." Proceedings of HotCloud (2011): 1-5.

[50] R. Gandhi and A. Sabne. Finding stragglers in hadoop. Tech Report. 2011

[51] Jordà Polo, Claris Castillo, David Carrera, Yolanda Becerra, Ian Whalley, Malgorzata Steinder, Jordi Torres, and Eduard Ayguadé. 2011. Resource-aware adaptive scheduling for mapreduce clusters. In Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware (Middleware'11), Fabio Kon and Anne-Marie Kermarrec (Eds.). Springer-Verlag, Berlin, Heidelberg, 187-207.

[52] David Patterson, Aaron Brown, Pete Broadwell, and others. 2002. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science.

[53] Hui Kang, Yao Chen, Jennifer L. Wong, Radu Sion, and Jason Wu. 2011. Enhancement of Xen's scheduler for MapReduce workloads. In Proceedings of the 20th international symposium on High performance distributed computing (HPDC '11). ACM, New York, NY, USA, 251-262

[54] Park, Alfred, and Richard M. Fujimoto. "Aurora: An approach to high throughput parallel simulation." In Principles of Advanced and Distributed Simulation, 2006. PADS 2006. 20th Workshop on, pp. 3-10. IEEE, 2006.

[55] Park, Alfred, and Richard Fujimoto. "A scalable framework for parallel discrete event simulations on desktop grids." In Grid Computing, 2007 8th IEEE/ACM International Conference on, pp. 185-192. IEEE, 2007.

[56] D. Cucuzzo, S. D'Alessio, F. Quaglia, and P. Romano. A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation.

Proceedings of the International Symposium on Distributed Simulation and Real-Time Applications, 2007, pp. 227–234.

[57] G. D’Angelo. Parallel and distributed simulation from many cores to the public cloud. Proceedings of the International Conference on High Performance Computing and Simulation (HPCS ’11), 2011.

[58] K. Vanmechelen, S. De Munck, & J. Broeckhove (2013). Conservative distributed discrete-event simulation on the Amazon EC2 cloud: An evaluation of time synchronization protocol performance and cost efficiency. Simulation Modelling Practice and Theory, 34, 126-143.

[59] T. H. Feng, and E. A. Lee. Implementation of Real-Time Distributed Discrete-Event Execution with Fault Tolerance. UC Berkeley Tech Report. 2007.

[60] M. Eklof, F. Moradi, and R. Ayani,.A framework for fault tolerance in HLA-based distributed simulations. Proceedings of conference on Winter simulation, 2005, pp. 1182–1189.

[61] N. Roy, A. Dubey, and A. Gokhale. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. 2011 IEEE International Conference on Cloud Computing (CLOUD). July 2011.

[62] Matthew Malensek, Sangmi Pallickara, and Shrideep Pallickara. Galileo: A Framework for Distributed Storage of High-Throughput Data Streams. Proceedings of the IEEE/ACM Conference on Utility and Cloud Computing. Melbourne, Australia. 2011.

[63] Message Passing Interface (MPI), http://en.wikipedia.org/wiki/Message_Passing_Interface

[64] Open Multi-Processing (OpenMP), http://en.wikipedia.org/wiki/OpenMP

[65] Zhiquan Sui, Matthew Malensek, Neil Harvey and Shrideep Pallickara. Autonomous Orchestration of Distributed Discrete Event Simulations in the Presence of Resource Uncertainty. (under review) ACM Transactions on Autonomous and Adaptive Systems (TAAS)

[66] Matthew Malensek, Zhiquan Sui, Neil Harvey and Shrideep Pallickara. Autonomous, Failure-resilient Orchestration of Distributed Discrete Event Simulations. Proceedings of the ACM Cloud and Autonomic Computing Conference. Miami, USA. 2013.

[67] Zhiquan Sui, Neil Harvey and Shrideep Pallickara. Learning Based Distributed Orchestration of Stochastic Discrete Event Simulations. (To appear) Proceedings of the IEEE/ACM Conference on Utility and Cloud Computing. London, UK. 2014.

[68] T. Sandholm and K. Lai. MapReduce optimization using regulated dynamic prioritization. In Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), pages 299–310, 2009.

[69] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03). ACM, New York, NY, USA, 164-177

[70] Amdahl’s law, http://en.wikipedia.org/wiki/Amdahl's_law

[71] Hyser, C., Mckee, B., Gardner, R., Watson, B.J.: Autonomic virtual machine placement in the data center. HP Labs Technical Report HPL-2007-189, February 2007

[72] R. P. Goldberg. 1973. Architecture of virtual machines. In Proceedings of the workshop on virtual computer systems. ACM, New York, NY, USA, 74-112.

[73] Zaharia, Matei, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: cluster computing with working sets." In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, 2010.

[74] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." In NSDI, 2011.

[75] Kivity, Avi, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. "kvm: the Linux virtual machine monitor." In Proceedings of the Linux Symposium, vol. 1, pp. 225-230. 2007.