PARALLEL DISTRIBUTED DISCRETE EVENT SIMULATION OPTIMIZATION USING COMPLEXITY AND DEEP LEARNING

by

EDWIN A. CORTÉS
B.S. Mechanical Engineering University of Central Florida, 1996
M.S. Mathematics University of Central Florida, 2003

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Industrial Engineering and Management Systems
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2015

Major Professor: L. Rabelo
ABSTRACT

Parallel distributed discrete event simulation (PDDES) is the execution of a discrete event simulation on a tightly or loosely coupled computer system with multiple processors. The discrete-event simulation model is decomposed into several logical processors (LPs) or simulation objects that can be executed concurrently using partitioning types such as spatial and temporal. PDDES is exceedingly important for the reduction of the simulation time, increase of model size, intellectual property issue mitigation in multi-enterprise simulations, and the sharing of resources.

One of the problems with PDDES is the time management to provide flow control over event processing, the process flow, and the coordination of different logical processors to take advantage of parallelism. Time Warp (TW), Breathing Time Buckets (BTB), and Breathing Time Warp (BTW) are three time management schemes studied by this research. For a particular PDDES problem, unfortunately, there is no clear methodology to decide a priori a time management scheme to achieve higher system and simulation performance.

This dissertation shows a new approach for selecting the optimal time synchronization technique class that corresponds to a particular parallel distributed and
discrete simulation with different levels of simulation logic complexity. Simulation complexities such as branching, parallelism, function calls, concurrency, iterations, mathematical computations, messaging frequency, event processing, and number of simulation objects interactions were given a weighted parameter value based on the cognitive weight approach. Deep belief neural networks were then used to perform deep learning from the simulation complexity parameters and their corresponding optimal time synchronization scheme value as measured by speedup performance.
To my wonderful family and my great friends.
# TABLE OF CONTENTS

LIST OF FIGURES .................................................................................................................. ix

LIST OF TABLES .................................................................................................................... xii

CHAPTER ONE: INTRODUCTION ......................................................................................... 1

1.1 Background .................................................................................................................. 1

1.2 Problem Statement ..................................................................................................... 12

1.3 Research Question ..................................................................................................... 13

1.4 Research Goals .......................................................................................................... 13

1.5 Research High Level Objectives ................................................................................ 13

1.6 Research Contributions ............................................................................................. 14

1.7 Document Outline ..................................................................................................... 15

CHAPTER TWO: LITERATURE REVIEW ............................................................................ 17

2.1 Introduction ................................................................................................................ 17

2.2 Parallel Discrete Event Simulation Engines .............................................................. 17

2.3 Simulation Optimization ........................................................................................... 31

2.4 Deep Belief Neural Networks .................................................................................... 35

2.5 Summary and Gaps ................................................................................................... 43

CHAPTER THREE: RESEARCH METHODOLOGY ............................................................ 45
3.1 Methodology Introduction .................................................................................. 45
3.2 Methodology ........................................................................................................ 46
3.3 Gap Analysis.......................................................................................................... 47
3.4 Refined Research/Preliminary Methodology and Development Tools .......... 48
3.5 Methodology, Testing, and Validation, and Conclusions .................................. 49

CHAPTER FOUR: DEVELOPMENT OF DBN FOR PATTERN RECOGNITION ..... 51

4.1 Pre-Processing Telemetry ................................................................................... 52
4.2 Training Data: Matrix v ....................................................................................... 53
4.3 Deep Belief Neural Networks: Algorithms ......................................................... 54
4.4 DBN Training Parameters ................................................................................... 56
4.5 DBN Optimization Parameters ........................................................................... 59
4.6 Pattern Detection .................................................................................................. 61
4.7 Analysis of Neuron Activation during Pattern Detection ................................... 66
4.8 Summary .............................................................................................................. 87

CHAPTER FIVE: PROGRAMS IN PARALLEL DISTRIBUTED DISCRETE EVENT
SIMULATION AND THEIR CHARACTERIZATION ................................................. 89

5.1 Parallel Distributed Discrete-Event Simulators .................................................. 89
5.2 Programming in WarpIV ..................................................................................... 94
5.3 Complexity .......................................................................................................... 106
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>New Methodology</td>
<td>114</td>
</tr>
<tr>
<td>CHAPTER SIX: TESTING</td>
<td></td>
<td>116</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>116</td>
</tr>
<tr>
<td>6.2</td>
<td>Performance Criterion Used</td>
<td>116</td>
</tr>
<tr>
<td>6.3</td>
<td>Selected Problems (Rationale)</td>
<td>117</td>
</tr>
<tr>
<td>6.4</td>
<td>Training Session</td>
<td>117</td>
</tr>
<tr>
<td>6.5</td>
<td>Testing Performance</td>
<td>120</td>
</tr>
<tr>
<td>6.6</td>
<td>Analysis</td>
<td>121</td>
</tr>
<tr>
<td>CHAPTER SEVEN: CONCLUSIONS AND FUTURE RESEARCH</td>
<td></td>
<td>123</td>
</tr>
<tr>
<td>7.1</td>
<td>Conclusions</td>
<td>123</td>
</tr>
<tr>
<td>7.2</td>
<td>Contributions to the Body of Knowledge</td>
<td>127</td>
</tr>
<tr>
<td>7.3</td>
<td>Future Research</td>
<td>127</td>
</tr>
<tr>
<td>APPENDIX A: SPACE SHUTTLE MAIN ENGINE</td>
<td></td>
<td>130</td>
</tr>
<tr>
<td>APPENDIX B: DEEP BELIEF NEURAL NETWORK NEURON MODEL</td>
<td></td>
<td>132</td>
</tr>
<tr>
<td>LIST OF REFERENCES</td>
<td></td>
<td>134</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1: Fixed Time Buckets (adapted and modified from warpiv.com) ...................... 3
Figure 2: The process of rollback in TW (adapted and modified from warpiv.com) .......... 7
Figure 3: The Event Horizon (adapted and modified from Steinman, 2001) ................. 9
Figure 4: BTB Processing Cycle (adapted and modified from Steinmann, 2001) .......... 10
Figure 5: BTW cycle in three nodes (adapted from warpiv.com) .................................. 12
Figure 6: SPEEDES and its simulation and event objects (adapted and modified from Steinmann 2001) ........................................................................................................ 23
Figure 7: SPEEDES Main Program (adapted and modified from Steinmann 2001) ....... 24
Figure 8: Logical Process with Aircraft and Radars as SOs ........................................... 28
Figure 9: PDDES 5 Node Distributed Simulation Example ............................................ 30
Figure 10: Examples of events and their relationships – two events: Check and Status Update .............................................................................................................................. 31
Figure 11: Two RBMs ..................................................................................................... 36
Figure 12: Dissertation Research Methodology .............................................................. 45
Figure 13: Proposed methodology (processes/message passing and the process/shared memory schematics are adapted from Jeff Steinman (warpiv.com)) ...................... 50
Figure 14: Deep learning & pattern detection .................................................................. 51
Figure 15: RBM Neuron Learning: Gibbs Sampling and Weight Update ....................... 55
Figure 16: Representative DBN .......................................................................................... 57
Figure 17: Propagation of Neuron Activation Probabilities ........................................ 62
Figure 18: Epsilon Threshold ............................................................................................. 63
Figure 19: Method One: Detection by Epsilon Threshold .............................................. 64
Figure 20: Method Two: Detection by Comparison ....................................................... 65
Figure 21: Main Fuel Valve: Leak Detection Sensors .................................................... 70
Figure 22: Engine Temperature Instrumentation at MFV ............................................... 71
Figure 23: Nominal MFV Temperatures (5 Missions) ..................................................... 72
Figure 24: STS-135 MFV Leak at $\beta_1$ and $\beta_2$ ......................................................... 75
Figure 25: Output Neuron $\Delta p/\Delta t$ ........................................................................ 76
Figure 26: Output Neuron $p(\cdot)$ and Its Derivatives .................................................... 76
Figure 27: Neuron Activation $p(\cdot)$ vs. STS-135 Leak ................................................ 77
Figure 28: SSME Simplified Flow Diagram .................................................................... 79
Figure 29: Engine Start Transient – 5 seconds ............................................................. 82
Figure 30: STS-100 Simulated Anomalous Pattern in $\beta_9$ ............................................ 85
Figure 31: Nominal and Anomalous Output Neuron Activation .................................... 86
Figure 32: Anomaly Detected by $P_N-P_A$ ..................................................................... 87
Figure 33: Simulation Engines Investigated .................................................................... 90
Figure 34: Aircraft range detection scenario using two types of simulations objects (radar and aircraft) ........................................................................................................... 95
Figure 35: UML schematics of the development with two types of Simulation Objects (Aircraft and Radar) and two events (i.e., Scan and Trajectory Update)................. 99

Figure 36: The implementation of the problem with different components .................. 100

Figure 37: Area selected for the verification & validation simulation......................... 101

Figure 38: Combined Speedup chart for BTW, BTB, and TW .................................. 106

Figure 39: Cognitive Weights Sample Calculation.................................................. 108

Figure 40: Classification of optimistic synchronization scheme with DBN................. 115

Figure 41: Root Mean Square error and Cross Entropy error for hidden layer 1 ..... 119

Figure 42: Root Mean Square error and Cross Entropy error for hidden layer 2 ..... 119

Figure 43: Root Mean Square error and Cross Entropy error for hidden layer 3 ..... 120

Figure 44: Confusion matrix for two DBNs.............................................................. 121

Figure 45: Pattern recognition methodology for the identification of an optimistic time management and synchronization scheme for PDDES................................. 125

Figure 46: The Space Shuttle Main Engine............................................................... 131

Figure 47: DBN Neuron Model.............................................................................. 133
LIST OF TABLES

Table 1: Contrastive Divergence Primer ................................................................. 56
Table 2: Deep Belief Neural Network Parameters for Learning ................................. 57
Table 3: Fixed DBN Parameters for Learning ......................................................... 58
Table 4: DBN Parameters for Optimization ............................................................ 59
Table 5: Case Studies by Pattern ............................................................................. 68
Table 6: MFV Telemetry – Nominal .......................................................................... 71
Table 7: DBN Initial Parameters for Learning ............................................................ 73
Table 8: Final DBN Parameters ................................................................................. 73
Table 9: MFV Telemetry – Leak Pattern ..................................................................... 74
Table 10: Engine Start Transient Telemetry Vectors for DBN Input ........................... 81
Table 11: DBN Initial Parameters for Learning ............................................................ 83
Table 12: Final DBN Parameters ................................................................................. 83
Table 13: Computer systems specification ............................................................... 103
Table 14: The utilization of the different systems during the experiments ................. 103
Table 15: Experiment results for each computing configuration and time management
and synchronization scheme (BTW, BTB, and TW) ............................................. 104
Table 16: Cognitive weights ..................................................................................... 107
Table 17: Vector that defines a PDES problem ......................................................... 110
Table 18: Vector that defines the PDDES problem of Section 5.2 with 4 Global Nodes and 1 Local Node using Block .......................................................... 113

Table 19: TW has the minimum wall clock time .......................................................... 114

Table 20: PDDES DBN Parameters ........................................................................... 118

Table 21: Vector that defines a PDDES problem....................................................... 126
CHAPTER ONE: INTRODUCTION

1.1 Background

Parallel and distributed discrete event simulation (PDDES) is the execution of a 
discrete event simulation (DES) on parallel or distributed computer systems. It is 
important to state that it is a single discrete event simulation program (e.g., just one with 
irregular and data dependent nature). PDDES has several advantages:

1. Increase Speed (i.e., reduced execution time) due to the parallelism

2. Increase Size of the discrete event simulation program and/or data generation

3. Heterogeneous Computing

4. Fault Tolerance

5. Usage of unique resources and Multi-Enterprise/Geographical Distributed 
   Locations


There are several computer systems which can handle PDDES. These computer 
systems can be tightly coupled and loosely coupled. Tightly coupled systems are a
multiprocessor computer system which communicates through shared memory modules. On the other hand, a loosely coupled system is a multiprocessor computer system where each processor has its own local memory and the processors communicate via messages.

1.1.1 Conservative and Optimistic Schemes

Simulation Objects must interact in certain fashion in order to accomplish an efficient parallel and distributed execution with perfect integrity. Several innovative techniques have been developed in order to solve this challenging problem from two viewpoints: Conservative and Optimistic.

1.1.2 Conservative Viewpoint

The Conservative Viewpoint executes events for Simulation Objects when it can be assured that no other event with an earlier time stamp will be received by the Simulation Object. The following three restricting specifications are essential (Steinmann, 2001):

1. Simulation objects can only interact with other Simulation Objects as specified by connectivity rules established during initialization.

2. Events scheduled from one Simulation Object to another must always have time tags that monotonically increase over time.
3. Events scheduled between Simulation Objects must never occur closer in time than an identified time interval (i.e., Lookahead).

The most important approach in the conservative domain is Fixed Time Buckets (Fujimoto, 2000). Fixed Time Buckets, Figure 1, allow events to be scheduled and processed asynchronously at arbitrary times by permitting a simulation object to schedule events in another simulation objects only if occurs tighter in time than a specified value known as the global Lookahead (L) of the simulation.

![Figure 1: Fixed Time Buckets](adapted and modified from warpiv.com)

1.1.3 Optimistic Viewpoint

The Optimistic Viewpoint uses a different strategy for obtaining parallelism by
aggressively processing events without regard for accuracy. Rollback techniques are implemented to undo events that might have been processed out of order whenever straggler event messages are received from other Simulation Objects. In this manner, events are executed “optimistically”. While the optimistic approach places no restrictions on how Simulation Objects can interact, the biggest drawback is that models must be developed in a rollbackable manner. Optimistic event processing is able to achieve optimal execution of the chain of dependent events that limit the performance of a simulation.

1.1.4 Most Sophisticated and Utilized Optimistic Approaches

There are several schemes that computer scientists have developed in order to implement Optimistic Approaches in Parallel Discrete Event Simulation. The most sophisticated and utilized ones are Time Warp (TW), Breathing Time Buckets (BTB), and Breathing Time Warp (BTW). These approaches are the ones researched in this dissertation.

1.1.5 Time Warp (TW)

The TW event management provides an efficient rollback mechanism for each Simulation Object. The simulation time of each Simulation Object is defined as the time
stamp of its last executed event, or the time of the event it is presently executing. When a Simulation Object receives a straggler event in its past, it rolls the Simulation Object back to its last correctly processed event. These events that were rolled back are either reprocessed or “rolled forward.” A rolled back event can be safely rolled forward if the straggler event does not modify any of the Simulation Object’s state variables that were accessed by the event when it was originally executed.

TW does not rollback the entire node when a Simulation Object receives a straggler message. Instead, only the affected Simulation Object is rolled back. Of course, during the rollback, all events scheduled by those events that were rolled back must also be retracted, potentially causing secondary (or cascading) rollbacks. Each event must therefore keep track of its generated events until the event itself is committed. Retraction messages, used to withdraw incorrectly scheduled event messages in Time Warp, are called antimessages (Fujimoto, 2000).

It is important to provide a state saving mechanism to support the rollback process. Rolling back an event returns the Simulation Object to the state it had before the event was processed. One state-saving technique that is commonly used for rollbacks is to save a full copy of the entity’s entire state before the event is processed: full state saving (Steinmann, 1990). However, there is another mechanism that has demonstrated to be more efficient: incremental state saving. With incremental state saving, state-modifying
operations performed by events transparently generate rollback items that are automatically collected in a queue managed by the event’s rollback manager. These rollback items keep track of affected state and allow each operation to be rolled back in the reverse order that they were performed.

A very important concept in Optimistic time management is the Global Virtual Time (GVT). GVT is defined as the time stamp of the “earliest unprocessed event or message within the simulation that is still in transit” (Steinmann 2001). GVT states when an event can be committed. In other words, those events with time stamps less than GVT were correctly processed and will never be rolled back. The objective is to update GVT across all nodes as often as possible without affecting the processing of the simulation with excessive synchronization. From a computational viewpoint (Steinmann 2001): The best performance is on true parallel machines with shared memory and using a ultra-high-speed communications infrastructure. In addition, it can also perform well in local area networks with manageable latencies.

Figure 2 details the process of rollback and the cascading of antimessages.
Rollback can be started when a Simulation Object receives a straggler message (one which tag is before the current simulation time of the Simulation Object). This straggler message will make several processed events invalid (the ones from the time tag of the straggler event to the current simulation time of the Simulation Object). TW rolls back each invalid event and then processes the straggler event. As each event is rolled back, antimessages may be generated from the corresponding events, which can cause further rollbacks. The antimessage received by another event that has already been processed will generate more rollbacks and antimessages due to the processes being considered invalid. As explained by Steinmann
“Of course, as events are rolled back due to the arrival of antimessages, they too might have incorrectly generated messages that must be canceled by releasing yet further antimessages”. This secondary antimessage generation is known as cascading antimessages.

1.1.6 Breathing Time Buckets (BTB)

BTW is a hybrid between the Fixed Time Buckets algorithm and TW. Unlike TW, “messages generated while processing events are never actually released until it is known that the event generating the messages will never be rolled back” (warpiv.com). This means that messages which cause invalid events with potential antimessages are not released. Therefore, BTB is a hybrid in the following sense:

- BTW is TW without antimessages.
- BTW processes events in time window cycles like Fixed Time Buckets however cycles are not fixed.

The Event Horizon is an important concept in BTW. The event horizon is the point in time where events generated by the simulation turn back into the simulation. At the event horizon, all new events that were generated through event processing at the
previous “bucket” could be sorted and merged back into the main event queue. Parallelism can be exploited because the event processed in each event horizon cycle has time tags earlier than the cycle’s event horizon. Therefore, it is important to calculate the Global Event Horizon to avoid problems with events that will be scheduled in others Simulation Objects (Steinmann 1995). The local event horizon (Figure 3) only considers the event horizon for events being processed on its node, while the global event horizon factors all nodes. Once all of the nodes have processed events up to their local event horizon, they are then ready to be synchronized. The next step is to compute the global event horizon as the minimum local event horizon across all nodes. Once GVT is determined, all events with time stamps less than or equal to GVT are committed.

Figure 3: The Event Horizon (adapted and modified from Steinman, 2001)
A potential problem is the some of the nodes may have processed events that went beyond GVT. An event processed by the respective Simulation Object must be rolled back when a newly generated events is received in its past (See Figure 4). Rollback is very simple in this case and involves discarding unsent messages that were generated by the event and then restoring state variables that were modified by the event. Therefore, antimessages are not required due to the restrictions in releasing messages.

![Figure 4: BTB Processing Cycle](https://via.placeholder.com/150)

**Figure 4: BTB Processing Cycle (adapted and modified from Steinmann, 2001)**

### 1.1.7 Breathing Time Warp (BTW)

BTW is another hybrid algorithm for time management and event synchronization that tries to solve the problems with TW and BTW (Steinmann, 1993):
1. TW has the potential problem of rollback and cascading antimessage explosions.

2. BTW has the potential problem of a higher frequency of synchronizations.

Cascading antimessage explosions can occur when events are close to the current GVT. Because events processed far ahead of the rest of the simulation will likely be rolled back, it might be better for those runaway events to not immediately release their messages. On the other hand, using TW as an initial condition to bring BTB reduces the frequency of synchronizations and increases the size of the “bucket”.

The process of BTW is explained as follows (Steinmann, 2001):

1. “The first $N_{\text{risk}}$ events processed locally on each node beyond GVT release their messages right away as in TW. After that, messages are held back and the BTW starts execution.”

2. “When $N_{\text{gvt}}$ events are processed, or when the event horizon is determined, each node requests a GVT update. If a node ever processes $N_{\text{opt}}$ events beyond GVT, it temporarily stops processing events until the next GVT cycle begins.” $N_{\text{risk}}$, $N_{\text{gvt}}$, and $N_{\text{opt}}$ are defined flow control parameters by the simulation engineer. An example of a typical processing cycle for a four-node execution is provided in Figure 5.
Figure 5: BTW cycle in three nodes (adapted from warpiv.com)

1.2 Problem Statement

Discrete event simulation on parallel and distributed processors is very different from the single processor scheme implemented in the traditional programs such as ARENA (arenasimulation.com) and Simio (simio.com). In order to implement a simulation problem in a parallel discrete distributed simulation paradigm, the simulation program would have use time synchronization schemes that are optimistic in nature (Time Warp, Breathing Time Buckets, and Breathing Time Warp). These techniques have been developed in order to implement optimistic time synchronization schemes, each with its respective strengths and weaknesses. Prior to a PDDES execution, a selection of an optimal time synchronization scheme is desired in order to achieve the
most optimized distribution and execution of the simulation object among the computer cores processing the distributed simulation problem. However, there is no mechanism or efficient rules to decide a priori the best approach at a given PDDES problem (i.e., software implementing the simulation objects plus the hardware configuration to be utilized).

1.3 Research Question

This research will propose to answer the question: Can a pattern classification mechanism using deep belief neural networks and measures of complexity be designed and optimized so that it can be used as a detector of the best optimistic scheme for a Parallel Distributed Discrete Event Simulation configuration of hardware and software?

1.4 Research Goals

The overall goal of this research is to investigate the feasibility of designing and using measures of complexity and deep learning to classify the best optimistic scheme for a Parallel Distributed Discrete Event Simulation Program.

1.5 Research High Level Objectives

*Objective 1*: Develop a multivariate model of the features of a parallel distributed
discrete event simulation program using deep belief neural networks.

Objective 2: Develop a framework to establish the fundamental parameters needed to recognize patterns of the different time synchronization schemes such as Time Warp, Breathing Time Buckets, and Breathing Time Warp when applied to parallel distributed discrete event simulation problem.

Objective 3: Implement pattern recognition simulation optimization by selecting the right time synchronization scheme.

1.6 Research Contributions

This investigation significantly contributes to the body of knowledge encompassing parallel distributed discrete-event simulation. The developed framework provides a method for using neural networks and measures of software complexity to recognize patterns in parallel distributed discrete-event simulation programs and computational hardware resources and configurations.

This research’s framework provides a systematic approach to using deep belief neural network for pattern classification and simulation optimization. The pattern detection techniques outlined by the framework can be implemented within a data analysis system that can be “self-contained” and can be deployed in any hardware system.
Finally, this dissertation’s simulation optimization techniques outlined by the framework could be implemented within existing distributed simulation systems to predict simulation performance characteristics from complexity measures.

1.7 Document Outline

This document has seven chapters. Chapter one provides overall information that sets the stage for this research. It covers the research goals, objectives, research question and problem statement. Chapter two presents a review of the literature to date. The literature review encompasses refereed published research on several interdisciplinary technologies, disciplines, and techniques necessary for the design, development, and implementation of the mathematics and algorithms pertinent to performing deep learning and complexity in software. Chapter Three describes this research methodology and its sequential flow. Chapter Four develops a deep learning scheme. Using case studies, chapter four implements the processes for neuron deep learning and applies empirical analyses of DBN output probabilities to demonstrate DBN capabilities as pattern detectors. Chapter Five demonstrates the development of the vector to be used to characterize a parallel discrete-event simulation program. Chapter Six describes the
results and the corresponding analysis. Finally, Chapter Seven summarizes results and provides conclusions and recommendations based on the results of this research.
CHAPTER TWO: LITERATURE REVIEW

2.1 Introduction

This survey of literature presents previous work related to the fundamentals applicable to the interdisciplinary design and implementation of this dissertation’s research work. The literature review starts with concepts related to the parallelism, multiprocessing, and distributed nature inherent in different Parallel and Distributed Discrete Event Simulation (PDDES) engines. It presents some of the major work in simulation optimization. In addition, it investigates the nature of Deep Belief Neural Networks (DBN) for Deep Learning.

2.2 Parallel Discrete Event Simulation Engines

Parallel and distributed discrete event simulation is the execution of a discrete event simulation (DES) on parallel or distributed computer systems. Traditional discrete event simulation systems typically formulate and analyze sequential events and processes in a single computer. In PDDES simulation objects interact among themselves and schedule events across parallel and/or distributed computer systems in order to accomplish an efficient parallel and distributed execution with perfect integrity. Several
innovative techniques have been developed in order to solve this challenging problem from two viewpoints: Conservative and Optimistic as explained in Chapter One.

A number of PDDES engines were identified and reviewed during our survey efforts and they are listed as follows:

- Georgia Tech Time Warp (GTW)
- Rensselaer’s Optimistic Simulation System (ROSS)
- Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES)
- WarpIV Simulation Engine

The listed parallel processing computing engines have the capabilities to implement high performance parallel simulation executives for discrete-event simulation applications. The parallel computations are performed through the implementation of optimistic synchronization techniques for time advance or speed-up of simulation programs.

2.2.1 Georgia Tech Time Warp (GTW)

Georgia Tech Time Warp (GTW) (Das et al. 1994 Fujimoto, 2000) is a general purpose parallel discrete event simulation executive capable of implementing parallel
computing algorithms using optimistic synchronization techniques. The optimization techniques implemented in this engine utilizes direct cancellation, fast GVT algorithms, on-the-fly fossil collection and memory-based to maximize performance and control optimism.

The GTW kernel was used for the development of different parallelization applications in the areas of telecommunication, battle management and the aviation sector. The parallel and distributed simulation (PADS) laboratory in Georgia Tech University used the GTW kernel to enable decision making of different battle management scenarios that analyzed different threat evaluations and weapon assignments as a proof of concept application. The parallelization efforts used the Time Warp optimization synchronization protocol.

2.2.2 Rensselaer’s Optimistic Simulation System (ROSS)

The Rensselaer’s Optimistic Simulation System (ROSS) is a parallel discrete-event simulator that executes on shared-memory multiprocessor systems (Carothers, 2000). The ROSS kernel was modeled after the time warp Georgia Tech GTW kernel. Its implementation mainly uses the Time Warp optimization mechanism for developing parallelized discrete-event simulations.
The ROSS kernel is capable of performing simulations in supercomputers and concluding implementations for large-scale simulation models. To achieve high parallel performance, ROSS uses a technique call “Reverse Computation” in which the optimization mechanism is implemented, not by state-saving, but by literally allowing to the greatest possible extent events to be reverse (Yaun et al., 2003). Their main research efforts have been on testing the parallel performance for the Time Warp synchronization protocol.

One example of their research efforts and of the ROSS capabilities included demonstrating the scalability of the Time Warp optimization technique on the variants of the IBM Blue Gene supercomputer (http://www.cs.rpi.edu/~chrisc/ross.html). They have presented a design for a robust performing Time Warp simulator over a variety of communication loads, and extremely large processor counts that reach up to 131,072 processors.

2.2.3 SPEEDES: Synchronous Parallel Environment for Emulation and Discrete-Event Simulation

The Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES) is a general purpose parallel and distributed discrete-event
simulation framework (speedes.com). This simulation framework was developed to serve as the core infrastructure for several DoD simulation systems. It was developed in the early 1990’s by NASA AMES Research Center engineers. The framework uses the Standard Simulation Architecture (SSA), which is defined by the government, for the encapsulation of critical functionality and extending capability through higher-level abstraction (Steinmann, 2001).

The SPEEDES simulation kernel provides event management capabilities to provide optimistic, rollback-based and event management schemes that can be used to support multiple management schemes. The kernel has different time management capabilities and modes for parallel processing:

- Sequential
- Time_Buckets
- Breathing_Time_Buckets (BTB)
- Time_Warp (TW)
- Breathing_Time_Warp (BTW)

The SPEEDES simulation framework provides a set of modeling constructs that promote reuse through object-oriented encapsulation mechanisms. The simulation objects are distributed to different processors by their corresponding object managers. These managers are responsible for creating and destroying their simulation objects. SPEEDES
is based on the concepts of “Simulation Objects” in which algorithms and data structure are available to support sequential, conservative, and optimistic execution modes of simulation events and messages (Steinman, 1992). Its ability to provide flow control for optimistic processing and message sending made SPEEDES the PDDES engine of choice for a number of U.S. government programs and projects including the Missile Defense Agency (MDA).

2.2.3.1 SPEEDES PDDES Engine

SPEEDES provides a set of C++ classes and API as the framework. As the part of framework SPEEDES provides a base class “SpSimObj”. Any class which needs to schedule an event or needs to change state must inherit this class. This class provides the functionality of roll back. To declare any object as a simulation object “DEFINE_SIMOBJ” needs to be included in the class definition. Also, this object has to be plugged in the framework by calling the macro “PLUG_IN_SIMOBJ” in the main function. To initialize any object SPEEDES provides an Init method. This method is called at the start of the simulation. Similarly, to clean up the memory space before deleting the object SPEEDES provides the Terminate function.

The objects of each simulation class are managed by the simulation object manager. On each node, one simulation object manager is created per class. The
simulation manager is used to manage initialization and termination of objects, creating dynamic objects, managing external module subscription etc.

### 2.2.3.2 SPEEDES Architecture

SPEEDES is based on the concepts of Simulation Objects. SPEEDES has the architecture shown in Figure 6. SPEEDES is “highly object-oriented C++ environment and differs from other simulation environments in that its events are fully encapsulated objects, separate from the simulation objects.” An event is assigned to one (and only one) simulation object as shown in Figure 6.

![Figure 6: SPEEDES and its simulation and event objects (adapted and modified from Steinmann 2001)](image-url)
An event is created by a message. Events are separate objects in C++. They are different from simulation objects. User-defined events inherit capabilities from a base-class event. Events are initialized by data contained within the message. Each event is then assigned to its own simulation object.

2.2.3.3 SPEEDES Main Program

Figure 7 indicates the different phases of execution in SPEEDES and the different options of time management and synchronization algorithms.

Figure 7: SPEEDES Main Program (adapted and modified from Steinmann 2001)
Initialization Phase: During this phase, the simulation objects are created and initialized during the construction of the event queue object. The synchronization strategy is accomplished through the creation of the appropriate event queue object” (e.g., BTW event queue).

Process Phase 1: Phase 1 has two steps. In the first step, an event performs its calculations and creates several messages. A very important point is that the state (i.e., the state variables) of the simulation object must not have any alterations. In addition, messages that has the potential to modify/create other events are not straightaway released. Only variables affected by the event are stored within the event object. In the second step, the values calculated are swapped with the simulation object. After the swap, the event has the old state values, and the simulation object has the new values. This will facilitate rollbacks because only two consecutive exchanges will reestablish the original state of the simulation object.

Simulation Time Phase: During this Phase, GVT is updated. One problem in determining GVT is in “knowing whether there are messages still floating in the system” . “This problem is solved by each node keeping track of how many messages it has sent and received.” High speed synchronous “communications are used to determine when the
total number of messages sent is equal to the total number of messages received.” GVT can be determined when this condition is true.

*Process Phase 2:* In this Phase, memory clean up or sending external messages is performed. This Phase is performed only after the event is known to be valid (i.e., Rollback of the event is not possible). This Phase cannot change the state of the simulation object.

*External Blocking:* This Phase is performed if the simulation expects external entities to send messages to the simulation.

### 2.2.4 WarpIV Simulation Engine

The WarpIV is the next generation replacement for the SPEEDES parallel discrete-event simulation framework. This simulation kernel is able to host discrete-event simulations over parallel and distributed cluster computing environments (WarpIV, 2009). The Warp engine has very similar event management capabilities as the SPEEDES simulation kernel. However, it supports heterogeneous network applications through its portable high-speed communication infrastructure which integrates both
shared memory with standard network protocols to facilitate high bandwidth and low latency message passing services.

In general, the modeling constructs and time management schemes provided with the Warp engine kernels not only supports the implementation of optimistic time mechanisms but it also supports the component-based and interoperability modeling paradigm for simulation model reusability. With a full-featured rollback framework this kernel provides automatic rollback support when running optimistically and is able to maximize event-processing throughput by optimizing internal event-management data structures and by using sophisticated memory management caching techniques.

The simulation modeler is able to use scheduling methods to implement events based on user-define objects. On the other hand, this simulation kernel allows for arbitrary arguments to be specified through the event interface construct.

**2.2.4.1 WarpIV Unique Features**

There are some unique features of the Warp engine with respect to SPEEDES. One of the main differences with SPEEDES and is “the separation between logical processes and simulation objects. SPEEDES combines the functionality of these two
types into a single simulation object class, while WarpIV separates event management functionality from modeling framework functionality.” Simulation objects inherit from the Class Logical Process. The Logical Process (LP) (Figure 8) class plays a significant role in event management.

Figure 8: Logical Process with Aircraft and Radars as SOs

Logical process (LPs) are automatically distributed during startup to different nodes. There are three kinds of logical processes (that inherits from the Class LP):
1. Master LP manager (MPLM) - provided on every node during the Initialize Phase. The MPLM can have several LP managers and inherits from the LP class. The MLPM on each node stores its logical process managers.

2. LP manager (LPM) - provided on each node for each type of simulation object. The LPM belongs to only one master logical process manager. They manage logical processes (such as Simulation Objects) as part of its state and participate in the creation and deletion of scheduled events. Thus LPMs store Simulation Objects.

3. The Simulation Object (SO) Class – provide the base class for simulation objects. The SOs are the entities in the simulation.

LPs are related to the events in different ways such as:

- LPs manage pending and processed events
- LPs manage uncommitted events during TW, BTW, and BTB schemes (i.e., the optimistic viewpoint).

Distributing model components across processors to achieve parallel processing, represents a scalable high-performance operation in networked multicore computing environments. In the case of multiple computer systems being used for a simulation run would require a change on the structure to reflect the total number of nodes participating
in the simulation run. As shown in Figure 9, when using 2 computer systems and having the main system with 2 nodes and another system number 3 nodes.

![Figure 9: PDDES 5 Node Distributed Simulation Example](image)

2.2.4.2 WarpIV Architecture

This simulation engine has the Class LP. Simulation Object (SO) is a regular logical process class and inherits from the LP Class. A LPM is able to have several Simulation Objects and a Simulation Object can belong to only one LPM. A SO manager class (that inherits from the LPM Class) for each user-defined simulation object type is automatically generated by a macro. With regards to events: events always have one
input message and zero or more outgoing messages that are generated and sent to create new events – see Figure 10.

![Diagram of events and their relationships](image)

**Figure 10:** Examples of events and their relationships – two events: Check and Status Update

It is important to mention that there is a type of event called “process.” A process is an event that passes time before exiting. This process type is very important in discrete-event simulations.

### 2.3 Simulation Optimization

According to Reklaitis (1983,) optimization is thought of as a way of finding the best solution from a series of alternatives without having to decidedly enumerate and evaluate all possible solutions. In essence, the objective is to find a set of input
parameters that generate optimal output solutions to either maximize or minimize specified objectives. Optimization is modeled through the use of a mathematical formulation that produce mathematical results and numerical techniques which guide an algorithm/method to identify an optimum solution.

Optimization practitioners have difficulties while dealing with some real-world problems that usually involve: uncertainty, non-linear constraints, objective functions, or combinatorial relationships. Although such problems well suited to be addressed using simulation, they are definitely too complex for mathematical formulations, therefore it becomes desirable to combine simulation and optimization to address those (April et al, 2004).

Simulation optimization essentially combines simulation with an optimization technique. Olafsson and Kim (2002) referred to it, as “the process of finding the best values of some decision variables for a system where the performance is evaluated based on the output of a simulation model of this system.” In 1997, Carson and Maria referred to it as “the process of finding the best input variable values from among all possibilities without explicitly evaluating each possibility.”
In terms of finding the best solution to a problem using optimization techniques, simulation optimization can be used to determine the system configuration (input parameters) to “provide a near optimal if not an optimal solution” (Law and McComas, 2002). This technique is notably complicated, due to the stochastic nature of the systems, evaluation of certain designs can only be estimated (Banks et al, 2000).

Simulation optimization searches for more than just simply the best configuration from a number of pre-selected scenarios (Law and Kelton, 2000) it provides advantages such as:

- no restriction on the number of scenarios
- not restricted to only evaluating certain levels of specific input parameters
- it requires less operator interaction and decision making
- it may not be necessary to evaluate all levels of input parameters to determine the near-optimal solution

According to Law and (Kelton, 2000; Kuriger, 2006), the optimization method interacts with the simulation. It runs through the system up to the point when a stopping rule is satisfied. The basic procedure can be defined in the following phases:

- Phase 1: determine the initial levels of the decision variables, $X_0$. 

33
• Phase 2: run the simulation for the current input levels and get the corresponding output.

• Phase 3: feed the output into the optimization technique and generate a new X.

• Phase 4: check if the stopping criteria been satisfied, if:
  - YES, then stop and report the optimal solution obtained
  - NO, then update the decision variables and repeat phase 2.

It is believed that the complexity of simulation optimization has prompted different approaches. (Medaglia, 2001). This author strongly believes that “bringing soft computing methodologies into the area of simulation optimization will lead to the solution of real world system problem in an efficient manner”. The use of genetic algorithms in simulation optimization to tackle complexity, by Boesel et al. (1998,1999) provided statistical guarantees on the quality of the resulting solution. Additionally, Glover et al (1999) used neural networks in their simulation optimization research to give the user the option of engaging a neural network accelerator to aid in the screening of values of the different input parameters.
2.4 Deep Belief Neural Networks

Deep neural architectures with multiple hidden layers were difficult to train and unstable with the backpropagation algorithm. Empirical results show that using backpropagation alone for neural networks with 3 or more hidden layers produces poor solutions (Larochelle, H., Bengio, Y., & Louradour, J., 2009).

In 2006, Hinton, G. E., & Osindero, S., 2006 provided novel training algorithms that trained multi-hidden layer deep belief neural networks. His work introduced the greedy learning algorithm to train a stack of restricted Boltzmann machines, which compose a DBN, one layer at a time. The central concept of accurately training a DBN, that extract complex patterns in data, is to find the matrix of synaptic neuron connection weights that produce the smallest error for the training (input-data) vectors.

The fundamental learning blocks of a DBN are stacked restricted Boltzmann machines (RBM). The greedy algorithm proposed by Hinton, G. E., Osindero, S., & Yee-Whye, T., 2006, focuses on allowing each RBM model in the stack to process a different representation of the data. Then, each model transforms its input-vectors non-linearly and generates output-vectors that are then used as input for the next RBM in the sequence.
When RBMs are stacked, they form a composite generative model. RBMs are generative probabilistic models between input units (visible) and latent (hidden) units (Längkvist, M. & Karlsson, 2014). An RBM is also defined by (Zhang, C. X. & Zhang, J. S., 2014) as a parameterized generative model representing a probability distribution. Figure 11 (Hinton, G. E., 2007) shows an RBM (at lower level) with binary variables in the visible layer and stochastic binary variables in the hidden layer. Visible units have not synaptic connections between them. Similarly, hidden units are not interconnected. No hidden-hidden or visible-visible connectivity makes the Boltzmann machines restricted.

During learning, the RBM at higher-level (Fig. 11) uses the data generated by the hidden activities of the lower RBM.

![Figure 11: Two RBMs.](image)
Zhang, C. X. & Zhang, J. S., (2014) states that learning in an RBM is accomplished by using training data and “adjusting the RBM parameters such that the probability distribution represented by the RBM fits the training data as well as possible.” RBMs are energy-based models. As such, a scalar energy is associated to each variable configuration. Per Bengio (2009,) learning from data corresponds to performing a modification of the energy function until its shape represents the properties needed. This energy function has different forms depending on the type of RBM it represents. Binary RBMs, also known as Bernoulli (visible)-Bernoulli (hidden) have an energy (energy of a joint configuration between visible and hidden units) function of the form

$$E(v, h; \theta) = - \sum_{i=1}^{I} \sum_{j=1}^{J} w_{ij} v_i h_j - \sum_{i=1}^{I} b_i v_i - \sum_{j=1}^{J} a_j h_j$$  \hspace{1cm} (2.1)$$

GRBM, Gaussian (visible)-Bernoulli (hidden), have an energy function of the form

$$E(v, h; \theta) = - \sum_{i=1}^{I} \sum_{j=1}^{J} w_{ij} v_i h_j - \frac{1}{2} \sum_{i=1}^{I} (v_i - b_i)^2 - \sum_{j=1}^{J} a_j h_j$$  \hspace{1cm} (2.2)$$

The variable $w_{ij}$ represent the weight (strength) of neuron connection between visible ($v_i$) and hidden units ($h_j$). Variables $b_i$ and $a_j$ are the visible units biases and the hidden units biases, respectively. I and J are the number of visible and hidden units,
respectively. The set $\theta$ represents the vector variables $\mathbf{w}$, $\mathbf{b}$, and $\mathbf{a}$, (Hinton, G., 2010), (Mohamed, A. R. & Sainath, T. N., & Hinton, G. E., 2011), (Mohamed, A. R., Dahl, G. E., & Hinton, G., 2012).

RBM$s$ represent probability distributions after being trained. They assign a probability to every possible input-data vector using the energy function. Per (Mohamed, A. R., Dahl, G. E., & Hinton, G., 2012), the probability that the model assigns to a visible vector $\mathbf{v}$ is

$$p(\mathbf{v}; \theta) = \frac{\sum_h e^{-E(\mathbf{v}, \mathbf{h}; \theta)}}{\sum_v \sum_h e^{-E(\mathbf{v}, \mathbf{h}; \theta)}}$$  \hspace{1cm} (2.3)

For binary RBMs, the conditional probability distributions are sigmoidal in nature and are defined by

$$p(h_j = 1|\mathbf{v}; \theta) = \sigma \left( \sum_{i=1}^{l} w_{ij} v_i + b_j \right)$$ \hspace{1cm} (2.4)

$$p(v_i = 1|\mathbf{h}; \theta) = \sigma \left( \sum_{j=1}^{f} w_{ij} h_j + a_i \right)$$ \hspace{1cm} (2.5)

where $\sigma(\lambda) = \frac{1}{1+e^{-\lambda}}$ is a sigmoid function (Le Roux, N. & Bengio, Y., 2008), (Hinton, G. E. & Osindero, S., 2006), (Mohamed, A. R., Dahl, G. E., & Hinton, G., 2012),
Real-valued GRBMs have a conditional probability for $h_j = 1$, a hidden variable turned on, given the evidence vector $v$ of the form

$$p(h_j = 1|v; \theta) = \sigma \left( \sum_{i=1}^{l} w_{ij} v_i + b_j \right)$$  \hspace{1cm} (2.6)

The GRBM conditional probability for $v_i = 1$, given the evidence vector $h$, is continuous-normal in nature and has the form

$$p(v_i|h; \theta) = \mathcal{N} \left( \sum_{j=1}^{l} w_{ij} h_j + a_i, 1 \right)$$  \hspace{1cm} (2.7)

where $\mathcal{N}(\mu, 1) = \frac{e^{-(v_i-\mu)^2}}{\sqrt{2\pi}}$ is a Gaussian distribution with mean $\mu_i = \sum_{j=1}^{l} w_{ij} h_j + a_i$ and variance of unity. (Mohamed, A. R., Dahl, G. E., & Hinton, G. , 2012), (Cho, K., Ilin, A., & Raiko, T. ,2011) and (Tran, V. T., AlThobiani, F., & Ball, A. ,2014).

Learning from input-data in an RBM can be summarized as calculating a good set of neuron connection weight vectors, $w$, that produce the smallest error for the training (input-data) vectors. This also implies that a good set of bias ($b$ and $a$) vectors must be
determined. Because learning the weights and biases is done iteratively, the weight update rule is given by $\Delta w_{ij}$ (equation 2.8).

Using equation 2.3, the learning rule for an RBM (weight update rule) is the partial derivative of the log-likelihood probability of a training vector with respect of the weights,

$$
\frac{\partial \log[p(v)]}{\partial w} = \Delta w_{ij} = \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}
$$

(Salakhutdinov, R. & Murray, I., 2008), (Hinton, G., 2010), and (Zhang, C. X., Zhang, J. S., Ji, N. N., & Guo, G., 2014). However, this exact computation is intractable because the calculation of $\langle v_i h_j \rangle_{model}$ takes exponential time to calculate exactly (Mohamed, A. R. & Sainath, T. N., & Hinton, G. E., 2011). In practice, the gradient of the log-likelihood is approximated.

Contrastive divergence learning rule is used to approximate the gradient of the log-likelihood probability of a training vector with respect of the neuron connection weights. The simplified learning rule for an RBM has the form

$$
\Delta w_{ij} \propto \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{reconstruction}
$$

(2.9)
(Längkvist, M., Karlsson, L., & Loutfi, A., 2014), (Hinton, G. E., & Salakhutdinov, R. R., 2006), (Wulsin, D. & Gupta, J., 2011), and (Mohamed, A. R., Dahl, G. E., & Hinton, G., 2012). The reconstruction values for $v_i$ and $h_j$ are generated by applying equations 2.4 and 2.5, or 2.7 for GRBM, (Mohamed, A. R., Dahl, G. E., & Hinton, G., 2012) in a Markov Chain using Gibbs sampling. Post Gibbs sampling, the contrastive divergence-learning rule for an RBM can be calculated and the weights of the neuron connections updated based on $\Delta w$. The literature also show that RBM learning rule (equation 2.9) may be modified with constants such as learning rate, weight-cost, momentum, and mini-batch size for a more precise calculation of neuron weights during learning. Per, Hinton, G. E. & Osindero, S., 2006, the contrastive divergence learning in an RBM is efficient enough to be practical.

In RBM neuron learning, a gage of the error between visible unit probabilities and their reconstruction probabilities computed after Gibbs sampling is accomplished by cross-entropy. The cross-entropy, between the Bernoulli probability distributions of each element of the visible units $v_{data}$ and its reconstruction probabilities $v_{recon}$, is defined by (Erhan, D., Bengio, Y., Courville, 2010) as

$$CEE = -\sum_i [v_{data_i} \log(v_{recon_i}) + (1 - v_{data_i}) \log(1 - v_{recon_i})]$$

(2.10)
For the final DBN learning phase, after each stack of RBMs in the DBN pre-training via greedy layer-wise unsupervised, the complete DBN is fine-tuned in a supervised way. The supervised learning via the backpropagation algorithm uses label data (classification data) to calculate neuron weights for the complete deep belief neural network. Hinton, G. E., & Osindero, S., 2006 used the wake-sleep algorithm for fine-tuning a DBN. However, recent research has demonstrated the backpropagation algorithm is faster and has lower classification error (Wulsin, D. & Gupta, J., 2011). In backpropagation, the derivative of the log probability distribution over class labels is propagated to fine-tune all neuron weights in the lower levels of a DBN.

In summary, The Greedy Layer-Wise algorithm proposed by Hinton pre-trains the DBN one layer at a time using contrastive divergence and Gibbs sampling, starting from the bottom first layer of visible variables to the top of the network – one RBM at a time. After pre-train, the final DBN is fine-tuned in a top-down mode using several algorithms such as the supervised backpropagation (Hinton, G. E., & Salakhutdinov, R. R.,2006), (Hinton, G., Deng, L., & Yu, D., 2012), (Larochelle, H., Bengio, Y., Louradour, J., & Lamblin, P.,2009) or the wake-sleep (Hinton, G. E., Osindero, S., & Yee-Who, T.,2006) and (Bengio, Y.,2009) – among others.
2.5 Summary and Gaps

This literature review analyzed the concepts, applications, and challenges related to the parallelism, multiprocessing, and distributed nature inherent in different Parallel and Distributed Discrete Event Simulation (PDDES) engines. In addition, applications of deep belief neural networks for pattern recognition using classification as well as simulation optimization were also analyzed by this research.

In PDDES applications, simulation objects must interact in certain fashion in order to accomplish an efficient parallel and distributed execution with perfect integrity. Several innovative techniques have been developed in order to solve this challenging problem from two viewpoints: Conservative and Optimistic. Can a pattern classification mechanism using deep belief neural network and measures of complexity be designed and optimized so that it can be used as a detector of the best optimistic scheme for a Parallel Distributed Discrete Event Simulation Program?

This literature review studied the parallel optimistic techniques to identify applications or mechanism which allow the PDDES simulation analyst to select a particular optimistic PDDES technique during simulation conceptualization. The optimistic techniques studied included the Time Warp, Breathing Time Buckets, and Breathing Time Warp techniques which have been developed in order to implement
optimistic time synchronization schemes, each with its respective strengths and weaknesses. However, there is no mechanism or efficient rules to decide a priori the best approach at a given PDDES simulation problem. This is an important problem to be solved and simulation optimization using DBNs is potentially a good solution.

Industrial engineering and simulation professionals have concentrated in single processor and sequential systems. In contrast to sequential systems, PDDES allows for other simulation problems and solutions to be developed which can apply to important applications in defense and industry. However, the complexities associated with PDDES has been an impediment.
CHAPTER THREE: RESEARCH METHODOLOGY

3.1 Methodology Introduction

The research methodology proposed in this doctoral dissertation is structured in three phases. This chapter discusses the phases and their individual components. Figure 12 shows the research methodology utilized in this research.

![Figure 12: Dissertation Research Methodology](image)

Phase one covers the research question, the review of literature, and the research gap.

Phase two addresses the refined research question, deep belief neural network development, complexity measures development, and the selection of the parallel
distributed DES environment. Phase three covers the methodology, testing and validation, analysis of results and summary of research findings.

3.2 Methodology

3.2.1 Proposed Research Question

The research question initiates the research methodology process. This investigation starts by asking: Is there a mechanism to accurately model and predict what the best time management and synchronization scheme is for a parallel discrete event simulation environment (program and hardware)? If so, what are the methodologies that facilitate the implementation of this mechanism?

3.2.2 Summary of Literature Review

The review of literature demonstrates that potential mechanism can be based on a deep belief neural networks. Deep belief neural networks can effectively learn linear and highly non-linear characteristics from different types of input data. The literature shows deep neural networks can perform classification on binary or continuous real-valued
input vectors. Different probabilistic mathematics are used to learn specific patterns of information embedded in the input vectors. For example, GRBMs and RBMs are probabilistic stochastic learning models typically applied to real-value units and binary units, respectively. Literature shows that learning non-linear functions is proportional to the numbers of hidden layers as well as the number of hidden neurons at each layer. Also, the number of epochs for optimizing neuron connection weights, either in the contrastive divergence leaning (pre-training) phase or the backpropagation learning (fine-tuning) phase, directly affects the performance of the neural network. The literature demonstrated that the effectiveness of a DBN model to learn is highly influenced by the neuron connection weights computed during the pre-training phase (for RBMs/GRBMs), performed by the contrastive divergence algorithms.

In addition, we studied different approaches for simulation optimization and parallel distributed DES environments available.

### 3.3 Gap Analysis

The literature reveals that there is not such approach for the prediction of the best time management and synchronization mechanism for a Parallel Distributed Discrete Event Simulation.
3.4 Refined Research/Preliminary Methodology and Development Tools

To address this research’s goals based on information from the literature review, the following is considered: A new methodology that will be using several mechanisms to implement specific steps such as pattern recognition (e.g., DBN) and the capture of the structure of the PDDES problem (e.g., software complexity and hardware details).

Then, it is important to achieve the following developments:

1. Design and implement a probabilistic model of a DBN using RBM and associated deep learning algorithms: contrastive divergence, Gibbs sampling, backpropagation. The respective validation of the software developed will be done by:
   a. Using a standard benchmark pattern classification data from MNIST (LeCun, Y., 2012) hand-written digits database to test constructed DBN and its effectiveness as a classifier.
   b. Obtain Space Shuttle Main Engine instrumentation data from nominal and off-nominal missions. This is a well know problem in pattern recognition and it will help us to validate the algorithm. In addition, we have contacted the University of Toronto (Geoffrey Hinton’s Research Group) for them to revise the contrastive deference neuron learning
logic implemented by this research and provide their expertise. Geoffrey
Hinton’s research group is the leading researchers in the world for
DBNs (http://www.cs.toronto.edu/~hinton/).
c. Train and validate DBN by selecting initial algorithm parameters: DBN
number of hidden layers, number of hidden neurons per layer, number
of output neurons for DBN, learning rate, epoch iterations for RBM
learning via contrastive divergence, number of epoch iterations during
overall neural network learning fine-tuning phase via backpropagation.
2. Obtain the measures of complexity to represent parallel distributed discrete-
event simulation programs and computing configurations.
3. Select and study a Parallel Distributed Discrete-Event Simulator environment
suitable for this research. Analyze their programming environment and learn
the different issues of the software constructs of the environment.

3.5 Methodology, Testing, and Validation, and Conclusions

The methodology described in Figure 13 will be applied to different types of
software parallel distributed DES Software and hardware configurations. These case
studies will then be used to implement the methodology and the processes to design,
optimize, and test the capabilities of deep belief neural networks in the context of pattern detectors for time management and synchronization schemes to be used.

Figure 13: Proposed methodology (processes/message passing and the process/shared memory schematics are adapted from Jeff Steinman (warpiv.com))

As this work provides a reference methodology to predict the best time management and synchronization scheme based on patterns in the hardware and programing constructs of a parallel distributed discrete event simulation environment, the summary of this research contribution is intended to be a starting point for others to expand upon.
CHAPTER FOUR: DEVELOPMENT OF DBN FOR PATTERN RECOGNITION

This chapter discusses the development and validation of deep belief neural networks for detecting patterns in parallel discrete event simulation. Validation of pattern detection and classification is performed using two known space shuttle anomalies as well as standard benchmark pattern classification data from MNIST (LeCun, Y., 2012) hand-written digits database. Figure 14 shows the implemented high level processes for optimizing deep learning and performing pattern detection and classification.
The deep learning section establishes the fundamental processes for training a deep belief network. It summarizes the pertinent variables that impact the performance of neuron learning and how variables are implemented from the governing equations (Eq. 2.4 - 2.7.) The core logic of neuron learning in restricted Boltzmann Machines is also outlined. This is of most importance because DBNs are constructed by stacking RBMs on top of each other to form the hidden layers of deep belief networks. Finally, the deep learning section discusses DBN parameters that are fixed in value and others which require varying to insure the trained DBN models data accurately.

The validation (detection analysis) section investigates the reaction of a trained DBN to input data that contains nominal and off-nominal signals. Using a DBN that is trained with nominal signals exclusively, its output neuron activation probability is examined to determine if it can detect off-nominal signal behavior embedded in input data.

4.1 Pre-Processing Telemetry

Pre-processing of input signals depends on signal characteristic. This research uses stochastic Bernoulli Restricted Boltzmann Machines in the implementation of DBN algorithms. RBMs require vector elements, of data modeled by a DBN, to have a
magnitude range from zero to unity. As a result, all training data is normalized. For an input matrix \( v \), normalization constant \( \max(|v|) \) is used to compute \( v \)-normalization as:

\[
v \leftarrow \frac{v}{\max(|v|)}.
\]

### 4.2 Training Data: Matrix \( v \)

Learning from data in a deep belief neural network implies that the magnitudes of neuron connections throughout the network are computed, layer-by-layer, from an input matrix. The structure of the input matrix is important because neuron activation probabilities (eq. 2.4-2.7) are propagated throughout the network as vectors, where every element in the activation probability vector is associated with the format from input data. This work trains a DBN from a collection of instrumentation signals identified by the variable \( \beta \) in equation \( x \), which describes the shape of input matrix \( v \).

\[
v = \begin{bmatrix}
\beta_1^1 & \beta_1^2 & \cdots & \beta_1^m \\
\vdots & \vdots & \ddots & \vdots \\
\beta_n^1 & \beta_n^2 & \cdots & \beta_n^m
\end{bmatrix}
\]

(4.1)

Input matrix \( v \) has elements consisting of \( m \) telemetry signals of magnitude \( \beta^1, \beta^2, \ldots, \beta^m \). Each row in \( v \) represents a sample of measurements and columns are measurement signals. Data time sample \( n = 2 \), for example, contains signals \( v = \{\beta_2^1, \beta_2^2, \ldots, \beta_2^m\} \) where superscript \( m \) represents a unique telemetry signal and the
subscript a data sample number. \( \forall \beta \in \mathbf{v} \), input matrix \( \mathbf{v} \) is strictly composed of nominal data during DBN training. However, during the detection analysis phase, \( \mathbf{v} \) will contain nominal and off-nominal data.

4.3 Deep Belief Neural Networks: Algorithms

As previously discussed, DBNs are made of stacked RBMs. Contrastive divergence for RBMs approximates the partial derivative of the log-likelihood probability (Eq. 2.8) of a training vector in a Markov Chain using Gibbs sampling. This approximation, \( \Delta w \) (Eq. 2.9 and Fig. 15,) is the fundamental quantity needed to iteratively update the weight of neuron connections at each epoch in an RBM. Deep learning is done by learning the neuron connections between visible and hidden units at each RBM in the stack in an unsupervised fashion via contrastive divergence - described in figure fifteen.
Figure 15: RBM Neuron Learning: Gibbs Sampling and Weight Update

A primer of symbols in figure 15 is listed in Table 1. The learning rate, weight cost, and momentum constants are used at every epoch of RBM neuron learning during contrastive divergence to calculate the change in neuron weight $\mathbf{w}$-matrix, visible neuron bias $\mathbf{b}$-matrix, and hidden neuron bias $\mathbf{a}$-matrix.
Fine-tuning of neuron weights is done via backpropagation after all neuron connection weights and their biases are learned from contrastive divergence. In backpropagation, the derivative of the log probability distribution over class labels is propagated from top to bottom in a DBN to update all neuron weights at lower layers. Supervised backpropagation, along with conjugate gradient optimization, fine-tunes neuron weights and finalizes deep learning in a deep belief neural network.

### 4.4 DBN Training Parameters

There are eleven parameters relevant for neuron learning in a DBN. Table three shows the parameters to be used for calculating neuron connection weights during deep learning.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>λ</td>
<td>Learning Rate</td>
</tr>
<tr>
<td>Ω</td>
<td>Mini-batch Size</td>
</tr>
<tr>
<td>ψ</td>
<td>Weight Cost</td>
</tr>
<tr>
<td>m</td>
<td>Momentum</td>
</tr>
<tr>
<td>µ</td>
<td>Mean of Vector</td>
</tr>
<tr>
<td>w</td>
<td>Neuron Weight Matrix</td>
</tr>
<tr>
<td>d- subscript</td>
<td>Data</td>
</tr>
<tr>
<td>r- subscript</td>
<td>Reconstruction</td>
</tr>
</tbody>
</table>
Table 2: Deep Belief Neural Network Parameters for Learning

<table>
<thead>
<tr>
<th>Learning Rate</th>
<th>Hidden Layer 1 Neurons</th>
<th>Hidden Layer 2 Neurons</th>
<th>Hidden Layer 3 Neurons</th>
<th>Number Output Neurons</th>
<th>RBM Mini-Batch Size</th>
<th>RBM Epochs</th>
<th>DBN Mini-Batch Size</th>
<th>DBN Epochs</th>
<th>Weight Cost</th>
<th>Momentum</th>
</tr>
</thead>
</table>

Figure 16 depicts a representative DBN with three hidden layers, 10 input units to be modeled, and one output neuron. Hidden layers 1, 2, and 3 each contain three, three, and two hidden neurons – respectively.

In practice the number of input units and hidden neurons, also called hidden or latent units, may be large. In Figure 16, the bottom yellow arrow represents RBM epochs that are iterations in learning RBM neuron connections - $w$. The top yellow arrow represents
DBN epochs, which are iterations of top-down fine-tuning of all DBN neuron weights via supervised backpropagation.

The number of hidden neurons at each layer required to successfully learn from a particular training set varies depending on the amount of visible units (input data to model) and complexity of the data. For this research, several parameters in Table 3 are fixed and are not considered for optimization. Table 3 shows the fixed values selected by experimentation, but with a baseline from (Mohamed, A. R., Dahl, G. E., & Hinton, G., 2012).

Table 3: Fixed DBN Parameters for Learning

<table>
<thead>
<tr>
<th>Learning Rate</th>
<th>Number Output Neurons</th>
<th>RBM Mini-Batch Size</th>
<th>DBN Mini-Batch Size</th>
<th>Weight Cost</th>
<th>Momentum</th>
<th>DBN Epochs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-6}$</td>
<td>1</td>
<td>50</td>
<td>50</td>
<td>0.01</td>
<td>0.5</td>
<td>1</td>
</tr>
</tbody>
</table>

The learning rate, weight cost, and momentum constants are used at every epoch of RBM neuron learning during contrastive divergence to calculate the change in neuron weight \( w \)-matrix, visible neuron bias \( b \)-matrix, and hidden neuron bias \( a \)-matrix.

This research constructs a DBN with only one output neuron for the anomaly detection cases. For the classification cases, typically the number of output neurons is directly proportional to the number of classification classes (or labels) of training data. For DBN
validation of anomaly detection, this work trained DBNs with nominal data exclusively and all data was given a label of unity.

4.5 DBN Optimization Parameters

Optimization of DBN parameters in neuron learning could benefit from sophisticated optimization processes for selecting parameter combinations that best model input data during learning. Table four shows DBN parameters what will be changed during the process of learning until a successful model is achieved.

<table>
<thead>
<tr>
<th>Number of Neurons Hidden Layer 1 (multi-level)</th>
<th>Number of Neurons Hidden Layer 2 (multi-level)</th>
<th>Number of Neurons Hidden Layer 3 (multi-level)</th>
<th>RBM Epochs (multi-level)</th>
</tr>
</thead>
</table>

This research employs analysis of variances (ANOVA) as “brute force” optimization to help identify DBN parameters, factors, that influence the cross entropy (CE) or the root mean square (RMS) minimum errors during stochastic DBN training. Having small values of CE and RMS during deep learning insures DBN neuron connections have magnitudes that produce good models of the input data. Using
engineering judgment and ANOVA iterations, a good combination of parameters can be chosen to train a particular DBN to “best” model input signals. Cross-entropy is used in RBM neuron stochastic learning as a gage of the error between visible unit probabilities and their reconstruction probabilities. From Figure 15 variables, cross entropy is calculated by

\[
CEE = -\sum_l [v_d \log(v_r) + (1 - v_d) \log(1 - v_r)]
\] (4.2)

Cross-entropy error is examined by analysis of variances after a large number of DBN learning runs. The analysis treats neurons at the different layers (1, 2, and 3) and RBM epochs as factors. The different values of factors are then considered levels. For every layer in the DBN stochastic training process, CE is calculated at every epoch of that layer. Since there are many epochs per layer, CE is averaged over all epochs in that layer.

The “brute-force” ANOVA attempts to examine the factors that may influence the mean value of CE. The statistical hypothesis for ANOVA is described as

\( H_0: \text{Different levels of a factor have the same effect on the mean of CE} \)

\( H_1: \text{Different levels of a factor influence the mean of CE differently.} \)
Using the output of ANOVA, with a significance level of $\alpha = 0.05$, any resulting factor with a p-value $\leq \alpha$ merits rejection on the null hypothesis. This information is then used, along with engineering judgment, to determine the effect on cross-entropy error by changing factor levels. The final selections of factor levels that produce small values of CE are then chosen as the best combination of DBN parameters that model input data.

### 4.6 Pattern Detection

The detection analysis of the deep belief neural network investigates the reaction of a trained DBN to input data that contains nominal and off-nominal signals. Using a DBN that is trained with nominal signals exclusively, its output neuron activation probability is examined to determine if it can detect off-nominal signal behavior embedded in input data.

When an input matrix $v$ containing signals is used as the input units of an already-trained DBN, neuron activation probabilities propagate through the net layer-by-layer until the DBN output is reached—Figure 17.
The output neuron of a nominally trained DBN in response to inputs containing nominal and off-nominal signals is examined by two proposed methods. Method one attempts to establish a limit value, $\varepsilon$, of the output neuron probability $p()$ that can be used
as the threshold for detection of anomalous signals. Figure 18 illustrates the epsilon concept.

Method two uses the output neuron probability $p()$ of the nominally trained DBN as a detector of small differences. The rest of this chapter discusses the two methods used to evaluate the output probabilities as detectors of patterns.

### 4.6.1 Method One

This method examines the derivative of $p()$ for the detection of behaviors in DBN output responding to input data containing nominal and off-nominal signal. Figure 19 shows the detection process.
In Figure 19, input matrix $v$ is populated with nominal and off-nominal vehicle hardware signals and is inputted to the nominally trained DBN. Then, neuron activation probabilities propagate through the net layer-by-layer in reaction to the input until the DBN output is reached. The derivative of output $|P(t)|$ is then computed and the location $t_d$ where $\frac{\partial[P(t)]}{\partial t}$ is a maximum is recorded. Mean and standard deviation are calculated from the complete nominal data set $v_{\text{nominal}}$ used in training the DBN. If the value of $v_{\text{nominal}}(t_d)$ falls within the interval bounded by mean $\pm$ std and mean $\pm k*\text{std}$, then the DBN output probability $p(t)$ can be used as an indicator of off-nominal signal behavior.
when \( p(t) \geq \epsilon \). The variable \( k \) represents the number of standard deviations away from mean. Changing \( k \) allows for resizing the detection interval as needed.

### 4.6.2 Method Two

This method examines the difference between two DBN output probabilities. The output probability of a DBN in reaction to its own nominally trained telemetry signal set is compared with the output probability of the same DBN in reaction to its own nominally trained data set – but with a small change. This method allows for the use of a DBN to detect slight changes in telemetry signals. Figure 20 shows the proposed detection process.

![Diagram of Method Two: Detection by Comparison](image)

Figure 20: Method Two: Detection by Comparison
Method two is superior in comparison with method one because it does not require any knowledge of the training data. In Figure 20, a nominal set of telemetry signals is inputted to the nominally trained DBN. The exact sets of signals \( v \) used as input are the same as the ones used to train the DBN. By doing this, the output activation probability \( P_N \) acts as a “fingerprint” of what nominal means for the DBN in reaction to an input. In principle, \( P_N \) represents a model of the nominal signal structures from learning data. Inputting a telemetry matrix \( v \) containing both nominal and anomalous signals to the nominal DBN generates the second output activation probability \( P_A \). This method detects off-nominal behavior in signals by comparing output probabilities \( P_N \) and \( P_A \) as illustrated in Figure 20.

### 4.7 Analysis of Neuron Activation during Pattern Detection

This section discusses validation findings on the activation activities of neurons at the output layer of a deep belief neural network. Sections 4.1 to 4.6 methods and processes are applied to different types of data, which include data sets from seven space shuttle missions with telemetry signals from the space shuttle main engines. Results are presented here.
Deep belief neural networks were trained from case study nominal signals exclusively. Therefore, the created DBNs can be seeing as representative models of the nominal instrumentation signals used during deep learning. The outputs of these nominal models were examined by inputting off-nominal and nominal signals to the trained DBN. The resulting neuron activations at the output of the DBN were analyzed to determine if neurons reacted to off nominal patterns at locations where off-nominal patterns were present. Ultimately, the analysis of the neuron activation probability curve of the single output neuron from the output layer of a DBN was used as the basis for off-nominal pattern detection.

DBN learning at RBM layers is unsupervised. Therefore, neuron activations through the neural network strongly represent features from the unlabeled training data. Although the fine-tuning stage of DBN learning uses labeled data, the effect on the learning of RBM neuron connection weights is small. All test cases presented in this chapter trained DBNs using many epochs of unsupervised neuron learning, via RBM, and only one epoch of fine-tuning using labels (supervised) at the last phase of learning (see Figure 16.) Table 5 shows a summary of case studies implemented in this chapter. These cases include both real and simulated patterns and are classified in three categories.
Table 5: Case Studies by Pattern

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Title / Pattern</th>
<th>Cat</th>
<th>DBN Learning Data</th>
<th>DBN Test Nominal+ Off-Nominal</th>
<th>Nominal DBN Learn Matrix Sizes</th>
<th>Off-Nominal + Nominal Matrix Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>STS-135 Space Shuttle MFV Leak</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
<td>1166&lt;sup&gt;nom123&lt;/sup&gt; x 6</td>
<td>1424&lt;sup&gt;nom133&lt;/sup&gt; x 6</td>
</tr>
<tr>
<td></td>
<td>Pattern: Leak (Temperature Decay)</td>
<td></td>
<td></td>
<td></td>
<td>756&lt;sup&gt;nom123&lt;/sup&gt; x 6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1190&lt;sup&gt;nom133&lt;/sup&gt; x 6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6194&lt;sup&gt;nom133&lt;/sup&gt; x 6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1842&lt;sup&gt;nom133&lt;/sup&gt; x 6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(6 signals)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>STS-100 Space Shuttle Main Engine Start Transients</td>
<td>2</td>
<td>Real</td>
<td>Real, Sim</td>
<td>483&lt;sup&gt;nom15&lt;/sup&gt; x 15</td>
<td>483 x 15</td>
</tr>
<tr>
<td></td>
<td>Pattern: Step Function</td>
<td></td>
<td></td>
<td>(15 signals)</td>
<td>(6 signals)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>One Signal, One Pattern</td>
<td>3</td>
<td>Sim</td>
<td>Sim</td>
<td>2500 x 6 (500 x 6) x five</td>
<td>500 x 6</td>
</tr>
<tr>
<td></td>
<td>Pattern: Bessel Function</td>
<td></td>
<td></td>
<td>(6 sinusoidal signals)</td>
<td>(5 nominal)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>One Signal, Two Patterns</td>
<td>3</td>
<td>Sim</td>
<td>Sim</td>
<td>10,000 x 6 (2000 x 6) x five</td>
<td>2000 x 6</td>
</tr>
<tr>
<td></td>
<td>Pattern: Step Function at Two Locations</td>
<td></td>
<td></td>
<td>(6 linear signals)</td>
<td>(5 nominal)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Two Signals, Two Patterns</td>
<td>3</td>
<td>Sim</td>
<td>Sim</td>
<td>10,000 x 6 (2000 x 6) x five</td>
<td>2000 x 6</td>
</tr>
<tr>
<td></td>
<td>Pattern: Bessel and Step Function</td>
<td></td>
<td></td>
<td>(6 linear signals)</td>
<td>(4 nominal)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Three Signals, Three Patterns</td>
<td>3</td>
<td>Sim</td>
<td>Sim</td>
<td>10,000 x 6 (2000 x 6) x five</td>
<td>2000 x 6</td>
</tr>
<tr>
<td></td>
<td>Pattern: Step Function, Sloped Line, and Bessel</td>
<td></td>
<td></td>
<td>(6 linear signals)</td>
<td>(3 nominal)</td>
<td></td>
</tr>
</tbody>
</table>

Category one consists of training a DBN with real nominal data and testing neuron activation using real off-nominal data. Category two trains a DBN with real nominal data and tests neuron activation using real and simulated off-nominal data. Category three uses simulated data to train a DBN as well as to test neuron activation with simulated off-nominal data.
4.7.1 Case 1: Space Shuttle Main Engine MFV Leak Pattern

STS-135 was the last shuttle flight of the space shuttle program. During countdown operations, liquid hydrogen and liquid oxygen propellants are loaded into the shuttle external tank. This operation is critical and complex and requires the three main engines to be thermally conditioned with cryogenic propellants. Ground systems supply the propellants, which flow through the vehicle’s main propulsion system and into the external tank. Cold propellants are re-circulated upstream of each engine main fuel valve (MFV) during loading operations. Since the MFV is closed before engine start, temperature transducers located downstream of the MFV (see Figs. 21 and 22) monitor temperatures for liquid hydrogen leaks. Hydrogen leaks can potentially cause loss of vehicle due to the rapid diffusivity of hydrogen that can ignite in the presence of an energy source. Any leaks pass the valve ball seals are registered by the temperature transducers located at the outlet of the main fuel valve.

During STS-135 taking test, one of the three shuttle main engines (main engine #3) detected a hydrogen leak that was registered by two temperature signals on that engine. The other two engine temperature profiles were nominal. Figure 21 shows a diagram of the main fuel valve and location of temperature transducers $\beta^{(1)}$ and $\beta^{(2)}$ at the valve outlet.
4.7.1.1 Telemetry pre-processing

This case trained a deep belief neural network with six nominal temperature instrumentation signals - 2 transducers per main engine (Fig. 22.) All data were normalized so the data set of matrix $v$ (Eq. 4.1) elements had a magnitude range of $\{0,1\}$. 
4.7.1.2 Training Matrix

Six nominal instrumentation temperature signals collected from five independent space shuttle missions trained a DBN. Table 6 summarizes telemetry variables, missions, and timeframes used.

Table 6: MFV Telemetry - Nominal

<table>
<thead>
<tr>
<th>STS</th>
<th>Flight Date</th>
<th>Start GMT</th>
<th>End GMT</th>
<th>TCID</th>
</tr>
</thead>
<tbody>
<tr>
<td>133</td>
<td>02/24/2011</td>
<td>124700</td>
<td>150000</td>
<td>SA133B</td>
</tr>
<tr>
<td>132</td>
<td>05/14/2010</td>
<td>090000</td>
<td>111000</td>
<td>SA132B</td>
</tr>
<tr>
<td>131</td>
<td>04/05/2010</td>
<td>010627</td>
<td>045800</td>
<td>SA131A</td>
</tr>
<tr>
<td>128</td>
<td>08/28/2009</td>
<td>185712</td>
<td>210000</td>
<td>SA128B</td>
</tr>
<tr>
<td>126</td>
<td>11/14/2008</td>
<td>154210</td>
<td>190000</td>
<td>SA126A</td>
</tr>
</tbody>
</table>

Space Shuttle Main Engine Main Fuel Valve Telemetry Retrieved
Engine 3: E41T3153A1, E41T3154A1 ($\beta^3$, $\beta^4$), Engine 2: E41T2153A1, E41T2154A1 ($\beta^5$, $\beta^6$), Engine 1: E41T1153A1, E41T1154A1 ($\beta^7$, $\beta^8$).
Input matrix $v$ consisted of temperature signals $\beta^1$ to $\beta^6$. Each column in $v$ represented each temperature telemetry signal. Rows in $v$ represented data point instances of all missions combined. The resulting $v$ matrix size is $\mathbb{R}^{5573 \times 6}$. Figure 23 shows a plot of the nominal data used for deep learning.

![Plot of nominal MFV Temperatures (5 Missions)](image)

Figure 23: Nominal MFV Temperatures (5 Missions)

### 4.7.1.3 Neural Net Training Parameters

The structure of the deep belief net for this test case used three hidden layers, one input layer, and one output layer. The size of the input units was 5573 data points. The number of hidden neurons in the first, second, and third hidden layers was initially set to 300, 300, 300 – respectively. Table 7 shows the initial set of DBN parameters selected for deep learning.
Table 7: DBN Initial Parameters for Learning

<table>
<thead>
<tr>
<th>Learning Rate</th>
<th>Hidden Layer 1 Neurons</th>
<th>Hidden Layer 2 Neurons</th>
<th>Hidden Layer 3 Neurons</th>
<th>Number Output Neurons</th>
<th>RBM Mini-Batch Size</th>
<th>RBM Epochs</th>
<th>DBN Mini-Batch Size</th>
<th>DBN Epochs</th>
<th>Weight Cost</th>
<th>Momentum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-6}$</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>1</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>1</td>
<td>0.01</td>
<td>0.5</td>
</tr>
</tbody>
</table>

4.7.1.4 Optimization

During the deep learning processes, the cross entropy error was investigated at every epoch in every hidden later to insure a decreasing trend at every epoch. Many iterations of deep learning were executed by randomly changing combinations of the number of hidden neurons at each layer and the number of epochs. This process employed ANOVA as a “brute-force” technique to help identify if a particular factor affected cross entropy. Results from the iterative process of changing the number of hidden neurons at the three hidden layers of the DBN as well as the number of RBM epochs that produced an acceptable nominal DBN model. Final DBN parameters are listed in Table 8.

Table 8: Final DBN Parameters

<table>
<thead>
<tr>
<th>Learning Rate</th>
<th>Hidden Layer 1 Neurons</th>
<th>Hidden Layer 2 Neurons</th>
<th>Hidden Layer 3 Neurons</th>
<th>Number Output Neurons</th>
<th>RBM Mini-Batch Size</th>
<th>RBM Epochs</th>
<th>DBN Mini-Batch Size</th>
<th>DBN Epochs</th>
<th>Weight Cost</th>
<th>Momentum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-6}$</td>
<td>30</td>
<td>20</td>
<td>10</td>
<td>1</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>1</td>
<td>0.01</td>
<td>0.5</td>
</tr>
</tbody>
</table>
4.7.1.5 Detection Analysis

Five nominal missions were used for neuron deep learning in the DBN discussed in previous sections. In this section, mission STS-135 data is used to test the previously trained nominal DBN and examine the activities of its output neuron probability in reaction to data that contains nominal and off-nominal temperature telemetry.

STS-135 main fuel valve data from the three main engines show a leak pattern in two of the six MFV temperature signals. The input matrix $v$, as a result, contains two columns of data with off-nominal patterns (leak) and four nominal columns of data. The resulting $v$ matrix size is $\mathbb{R}^{1424 \times 6}$. Table 9 shows telemetry source.

<table>
<thead>
<tr>
<th>STS</th>
<th>Flight Date</th>
<th>Start GMT</th>
<th>End GMT</th>
<th>TCID</th>
</tr>
</thead>
<tbody>
<tr>
<td>135</td>
<td>06/15/2011</td>
<td>163000</td>
<td>175400</td>
<td>SAA135A1</td>
</tr>
</tbody>
</table>

Space Shuttle Main Engine Main Fuel Valve Telemetry Retrieved

Engine 3: E41T3153A1, E41T3154A1 ($\beta^{(1)}$, $\beta^{(2)}$). Engine 2: E41T2153A1, E41T2154A1 ($\beta^{(3)}$, $\beta^{(4)}$). Engine 1: E41T1153A1, E41T1154A1 ($\beta^{(5)}$, $\beta^{(6)}$).

Figure 24 shows two temperature transducers from main engine #3 displaying a hydrogen leak detected by $\beta^1$ and $\beta^2$ (see Figs. 21 and 22)
The neuron activation probability at the output of the DBN was tested with data from STS-135, which includes both nominal and off-nominal signals. When the DBN was presented with STS-135 data as its visible units, neuron activation probabilities propagate through the net layer-by-layer until the DBN output is reached. The activation probabilities at each layer are governed by Eqs. 2.4 and 2.5 where the values of \( \mathbf{w}, \mathbf{b}, \) and \( \mathbf{a} \) matrices were fixed after DBN learning. After neuron activation reached the DBN output neuron, a probability curve \( p() \) of dimensions \( \mathbb{R}^{1424 \times 1} \) is created. This curve represents the reaction of the nominal DBN to signals not previously seen during training.

Process method one was applied to DBN output function \( p() \) and its derivative magnitude, \( |\partial[P(t)]/\partial t| \), was computed using several \( \Delta t \) intervals. Figure 24 shows
representative $\Delta t$ for two points only. Figure 26 illustrates $p()$ and its derivative using different $\Delta t$ intervals.

$$\Delta t = \left| \frac{\Delta p}{\Delta t} \right| = \frac{\Delta p}{\Delta t}$$

Figure 25: Output Neuron $\Delta p/\Delta t$

Figure 26: Output Neuron $p()$ and Its Derivatives
After examining the location in time where the maximum derivative was found (Fig. 26 red square,) its location was compared to the location where nominal data was bounded by a temperature of $\mu - 1\sigma$ and $\mu - 3\sigma$. For this test case, the anomaly detection limit, epsilon, was found. Figure 27 shows a plot of STS-135 main fuel valve leak along with the DBN output neuron $p()$ reacting to leak. The three dotted vertical lines are locations in time where the MFV temperature deviated from normal. First, second, and third line represent $1\sigma$, $2\sigma$, and $3\sigma$ sigma deviations from normal, respectively.

![Figure 27: Neuron Activation $p()$ vs. STS-135 Leak](image)

Using the epsilon threshold value of the output $P()$ as a criterion allows for an objective indicator of off-nominal behavior when the output probability $P()$ exceeds the threshold.
This case study illustrated the feasibility of using the output of a deep belief neural network as a possible detector of off-nominal patterns under certain specific restrictions and conditions described in this section.
4.7.2 Case 2: SSME Start Transient Pattern

The space shuttle main engine is commanded to start by the space shuttle onboard computers at a countdown time of T - 6.6. This timeframe is necessary so that engine start sequences transition all engine hardware (valves, pumps, turbines, etc.) from static to transient and eventually to a steady state mode before light off. The main engines are one of the most complex rocket engines ever developed. As a result, transient behavior of engine hardware during start is highly non-linear.

This case study attempts to model fifteen highly non-linear main engine telemetry signals (Fig. 28) from one engine by training a deep belief neural network with nominal instrumentation signals from the STS-100 mission.

Figure 28: SSME Simplified Flow Diagram
The nominal engine DBN model was tested by this case study to determine if the output of the DBN reacting to nominal telemetry can be compared to the same DBN reacting to off-nominal telemetry. The change in profile of both DBN neuron outputs was then examined to identify if the change could be used as a detector of patterns different from nominal.

### 4.7.2.1 Telemetry pre-processing

All fifteen telemetry signals were normalized to insure matrix $v$ elements had a magnitude range of $\{0,1\}$.

### 4.7.2.2 Training Matrix

DBN training matrix $v$ contains five seconds of telemetry signals $\beta^1$ to $\beta^{15}$ recoded during main engine start transient, resulting in $v$ matrix size of $\mathbb{R}^{483 \times 15}$ containing discrete and continuous nominal patterns listed in table 10.
Table 10: Engine Start Transient Telemetry Vectors for DBN Input

<table>
<thead>
<tr>
<th>Telemetry Vector</th>
<th>Engine Telemetry Description</th>
<th>Telemetry Variable</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>β₁</td>
<td>Phase in Effect</td>
<td>E41J1512D3</td>
<td>Discrete</td>
</tr>
<tr>
<td>β₂</td>
<td>Operating Mode</td>
<td>E41J1513D3</td>
<td>Discrete</td>
</tr>
<tr>
<td>β₃</td>
<td>Main Combustion Chamber (MCC) Ave. Pressure</td>
<td>E41P1023D3</td>
<td>Continuous</td>
</tr>
<tr>
<td>β₄</td>
<td>High Pressure Fuel Turbine Discharge Temperature</td>
<td>E41T1173D3</td>
<td>Continuous</td>
</tr>
<tr>
<td>β₅</td>
<td>High Pressure Oxidizer Turbine Discharge Temp</td>
<td>E41T1175D3</td>
<td>Continuous</td>
</tr>
<tr>
<td>β₆</td>
<td>Fuel (LH₂) Flowrate</td>
<td>E41R1021D3</td>
<td>Continuous</td>
</tr>
<tr>
<td>β₇</td>
<td>Oxidizer Preburner Oxidizer Valve Actuator Position</td>
<td>E41H1028D3</td>
<td>Continuous</td>
</tr>
<tr>
<td>β₈</td>
<td>Fuel Preburner Oxidizer Valve Actuator Position</td>
<td>E41H1027D3</td>
<td>Continuous</td>
</tr>
<tr>
<td>β₉</td>
<td>MCC Coolant Chamber Valve Actuator Position</td>
<td>E41H1026D3</td>
<td>Continuous</td>
</tr>
<tr>
<td>β₁₀</td>
<td>High Pressure Oxidizer Boost Pump Discharge P</td>
<td>E41P1033D3</td>
<td>Continuous</td>
</tr>
<tr>
<td>β₁₁</td>
<td>Main Fuel Valve Actuator Position</td>
<td>E41H1024D3</td>
<td>Continuous</td>
</tr>
<tr>
<td>β₁₂</td>
<td>Main Oxidizer Valve Actuator Position</td>
<td>E41H1025D3</td>
<td>Continuous</td>
</tr>
<tr>
<td>β₁₃</td>
<td>High Pressure Oxidizer (LO₂) Turbopump Discharge P</td>
<td>E41P1030D3</td>
<td>Continuous</td>
</tr>
<tr>
<td>β₁₄</td>
<td>High Pressure Fuel (LH₂) Turbopump Discharge P</td>
<td>E41P1029D3</td>
<td>Continuous</td>
</tr>
<tr>
<td>β₁₅</td>
<td>Oxidizer (LO₂) Flowrate</td>
<td>E41R1022D3</td>
<td>Continuous</td>
</tr>
</tbody>
</table>

Engine hardware instrumentation profiles are highly non-linear during engine start sequences. Figure 29 shows nominal engine telemetry during the first five seconds of engine start transient - used for DBN deep learning. The plot is zoomed-in to illustrate transient behavior, since the scales of different telemetry signals have a large range.
4.7.2.3 Neural Net Training Parameters

The structure of the deep belief net for this test case used three hidden layers, one input layer, and one output layer. The size of the input units was 483 time data points with fifteen discrete and continuous signals. The number of hidden neurons at all hidden layers was initially 500. Table 11 shows the initial set of DBN parameters selected for deep learning.

Figure 29: Engine Start Transient – 5 seconds
Table 11: DBN Initial Parameters for Learning

<table>
<thead>
<tr>
<th>Learning Rate</th>
<th>Hidden Layer 1 Neurons</th>
<th>Hidden Layer 2 Neurons</th>
<th>Hidden Layer 3 Neurons</th>
<th>Number Output Neurons</th>
<th>RBM Mini-Batch Size</th>
<th>RBM Epochs</th>
<th>DBN Mini-Batch Size</th>
<th>DBN Epochs</th>
<th>Weight Cost</th>
<th>Momentum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-6}$</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>1</td>
<td>50</td>
<td>200</td>
<td>50</td>
<td>1</td>
<td>0.01</td>
<td>0.5</td>
</tr>
</tbody>
</table>

4.7.2.4 Optimization

During the deep learning processes, the cross entropy error was investigated at every epoch in every hidden later to insure a decreasing trend at every epoch. Many iterations of deep learning were executed by randomly changing combinations of the number of hidden neurons at each layer and the number of epochs. Analysis of variances was not employed to help identify factors affecting cross entropy error. Instead, trial-and-error was used for the iterative process of changing the number of hidden neurons at the three hidden layers of the DBN and the number of RBM epochs that produced an acceptable nominal DBN model based average minimum cross-entropy error at every hidden layer of neurons. Final DBN parameters are listed in Table 12.

Table 12: Final DBN Parameters

<table>
<thead>
<tr>
<th>Learning Rate</th>
<th>Hidden Layer 1 Neurons</th>
<th>Hidden Layer 2 Neurons</th>
<th>Hidden Layer 3 Neurons</th>
<th>Number Output Neurons</th>
<th>RBM Mini-Batch Size</th>
<th>RBM Epochs</th>
<th>DBN Mini-Batch Size</th>
<th>DBN Epochs</th>
<th>Weight Cost</th>
<th>Momentum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-6}$</td>
<td>50</td>
<td>100</td>
<td>150</td>
<td>1</td>
<td>50</td>
<td>200</td>
<td>50</td>
<td>1</td>
<td>0.01</td>
<td>0.5</td>
</tr>
</tbody>
</table>
4.7.2.5 Detection Analysis

One nominal mission was used for neuron deep learning in the DBN discussed by this case study. Nominal STS-100 telemetry used to train the DBN was used in two ways. First, the nominal data was fed back to the nominal DBN and its neuron output probability function $P_N$ was analyzed. By doing this, the output activation probability $P_N$ acted as a “fingerprint” of what nominal means for the DBN. Second, the same nominal data was perturbed by changing the value of telemetry $\beta^9$ (main combustion chamber coolant valve position.) The value of $\beta^9$ was incremented by $10^k$ at the time location 200 to 202, where $k = \{2^1, 2^2, 2^3, 2^4\}$. Then, the complete nominal + small change data was inputted to the nominal DBN and neuron output probability function $P_A$ was compared with the reaction to nominal-only. Figure 30 shows one anomalous pattern introduced - expanded for visibility.
Process method two was applied to this test case. Row one of Figure 31 illustrates nominal data and DBN neuron activation $P_N$. Row two illustrates anomalous data and DBN neuron activation $P_A$. 

Figure 30: STS-100 Simulated Anomalous Pattern in $\beta^9$
By comparing the difference between $P_N$ and $P_A$ neuron output probabilities, an off-nominal pattern was detected for a small perturbation in signal $\beta^9$. The small-introduced anomaly was detected by the output neuron probability difference for perturbation of $10^{-2}$, $10^{-4}$, and $10^{-8}$. Smaller perturbations were not detected by the
difference between $P_N$ and $P_A$ neuron output probabilities. Figure 32 displays a detection spike at the location where simulated anomaly was introduced.

![Figure 32: Anomaly Detected by $P_N$-$P_A$](image)

### 4.8 Summary

In summary, this chapter discussed the processes for the design of deep belief neural networks, neuron learning in restricted Boltzmann machines, deep learning implementation techniques, and validation test cases with respect of anomaly detection and classification. The DBN used the output of a DBN as detector of patterns in telemetry signals using two methods. The first method attempted to find a limit value of the neuron output probability curve at the output of a DBN. This limit value, epsilon, could then be used as a threshold for detection of anomalies in telemetry data. The second method compared the activation probabilities obtained by feeding a DBN with nominal signals and by feeding the same DBN with a mixture of nominal and off-nominal signals. The
difference of the output activation probabilities between both cases can then be used as a possible detector of anomalous patterns.

Using detection method two for analyzing DBN outputs shows that even when training signals are highly non-linear, the DBN model output is capable of detecting small changes in signals. This case study also showed that non-linear processes can be modeled by a single curve produced by the DBN output probability curve. This finding suggests that within the constraints of this case study, the output probability curve essentially compresses the information it modeled from a matrix of size $\mathbb{R}^{483\times15}$ to one of size $\mathbb{R}^{483\times1}$ - which represents a 93% decrease in size. This case study illustrated the feasibility of using the output of a deep belief neural network as a possible detector of off-nominal patterns under certain specific restrictions and conditions as described in this section.
CHAPTER FIVE: PROGRAMS IN PARALLEL DISTRIBUTED DISCRETE EVENT SIMULATION AND THEIR CHARACTERIZATION

This chapter presents the parallel distributed discrete event simulators investigated and the rationale for WarpIV selection. This chapter discusses the features of programming parallel distributed discrete-event simulators and their characterization using measures of complexity. In addition, it provides the classification structure between these measures and the different time management and synchronization schemes.

5.1 Parallel Distributed Discrete-Event Simulators

We studied several Parallel and Distributed Discrete Event Simulation (PDDES) engines. A number of PDDES were reviewed during our efforts and they are listed as follows:

- Georgia Tech Time Warp (GTW)
- Rensselaer’s Optimistic Simulation System (ROSS)
- Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES)
- WarpIV Simulation Engine

The listed parallel processing computing engines have the capabilities to implement high performance parallel simulation executives for discrete-event simulation applications.
The parallel computations are performed through the implementation of optimistic synchronization techniques for time advance or speed-up of simulation programs. Figure 33 depicts the several simulators that were investigated.

![Simulation Engines Investigated](image)

Figure 33: Simulation Engines Investigated

5.1.1 Georgia Tech Time Warp (GTW)

It was found that the GTW parallel computation kernel is currently not supported and was deemed not suitable for our project efforts. Other parallel and distributed discrete-event simulations has been developed and inspired from the initial developments
of the Georgia Tech implementation of GTW, for example, the Rensselaer’s Optimistic Simulation System (ROSS).

5.1.2 Rensselaer’s Optimistic Simulation System (ROSS)

The Rensselaer’s Optimistic Simulation System (ROSS) is a parallel discrete-event simulator that executes on shared-memory multiprocessor systems (Carothers 2000). However, during our research efforts it has determined that other simulators have better API for building simulation model complexity such as WarpIV and SPEEDES.

5.1.3 Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES)

The Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES) is a general purpose parallel and distributed discrete-event simulation framework (speedes.com). This simulation framework was developed to serve as the core infrastructure for several DoD simulation systems. It was developed in the early 1990’s by NASA AMES Research Center engineers. The framework uses the Standard Simulation Architecture (SSA), which is defined by the government, for the encapsulation of critical functionality and extending capability through higher-level abstraction (Steinmann 2001).
SPEEDES was evaluated and the mechanisms are very similar to WarpIV but with less efficiency and a higher overhead.

5.1.4 WarpIV Engine

The WarpIV is the next generation replacement for the SPEEDES parallel discrete-event simulation framework. This simulation kernel is able to host discrete-event simulations over parallel and distributed cluster computing environments (WarpIV, 2009). The Warp engine has very similar event management capabilities as the SPEEDES simulation kernel. However, it supports heterogeneous network applications through its portable high-speed communication infrastructure which integrates both shared memory with standard network protocols to facilitate high bandwidth and low latency message passing services.

In general, the modeling constructs and time management schemes provided with the Warp engine kernels not only supports the implementation of optimistic time mechanisms but it also supports the component-based and interoperability modeling paradigm for simulation model reusability. With a full-featured rollback framework this kernel provides automatic rollback support when running optimistically and is able to
maximize event-processing throughput by optimizing internal event-management data structures and by using sophisticated memory management caching techniques. The simulation modeler is able to use scheduling methods to implement events based on user-define objects. On the other hand, this simulation kernel allows for arbitrary arguments to be specified through the event interface construct.

In addition, we acquired, installed and tested the WarpIV simulation engine at UCF. This is a simulation environment that implements PDDES. It has a communication system optimized to support tightly coupled computer systems (e.g., shared memory multiprocessors). In addition, it was designed to allow construction of optimized implementations for different networked systems. It is an enhanced version of SPEEDES

5.1.4.1 WarpIV Engine Advantages

The Warp engine provides the infrastructure for scheduling event processing to occur in a sequential, parallel, and/or distributed environment. The advantages of this type of PDPES are the followings:

- Features state-of-the-art sequential, parallel conservative, and parallel optimistic time management modes to ensure the highest possible
performance for a wide variety of problems across different computer and network systems.

- In parallel, the simulation engine automatically distributes models across multiple processors while managing event processing between the models in logical time.
- Offers easy-to-use interfaces for scheduling events, waiting for user-specified conditions to occur, and dynamically creating or deleting models.
- Enjoys flexible polymorphic component infrastructure to promote interoperability and reuse for complex systems.

For this research, WarpIV was selected as the engine due to its superior features,

5.2 Programming in WarpIV

We provide an example of programming in WarpIV in this sections to illustrate this simulator and PDDES paradigm as depicted in Figure 34.
The aircraft range detection simulation program implements a parallel distribute discrete event simulation with interaction of multiple aircrafts and multiple radars. These are the general simulation problem features:

1. A discrete-event simulation program (with capabilities for execution in parallel/distributed computing environments)

2. Simulation clock time is in seconds
3. Total simulation time: 100 seconds (changeable).

4. There are Two (2) types of simulation objects:
   a. Aircraft
   b. Ground Radars

5. The program will emphasize three aircraft SimObject and three radars SimObject; however, simulation configuration can accommodate additional instances of entities - for comparisons:
   a. Three (3) Aircrafts
   b. Three (3) radars

6. The theater of operations is read from a file with the corresponding longitude and latitude. The maximum and minimum speed of the aircrafts is read from a file (meters/second). The range of the radar can be read from a file or hardcoded in the program.

7. The simulation randomly initializes the position of each aircraft object and each ground radar object. Their position (X, Y, and Z) is represented by the earth centered rotational Cartesian coordinates (ECR). Therefore, the information of the theater of operations (latitude and longitude) is changed to ECR (we use a routine/method for that conversion. There are some rules about the locations of the radars and aircrafts (in order to have avoid detection and flying trajectories outside of the theater of operations).
8. After initialization, the simulation detects an aircraft’s proximity to a ground radar using a pre-established range value for detection (as explained above). Proximity (range) is calculated in parallel using radar position and moving entity position vectors via \( \text{Range} = \sqrt{\Delta X^2 + \Delta Y^2 + \Delta Z^2} \), where \( \Delta \) represents the difference between radar and aircraft positions (\( \Delta \)latitude, \( \Delta \)longitude, and \( \Delta \)altitude) in Earth Centered Rotational Coordinates (ECR). The simulation reads additional parameters, but it only uses range.

9. The trajectory of the aircraft is a circle. Therefore, an event \( \text{TestUpdateAttribute} \) that each 0.2 seconds updates the trajectory to keep the circle. It is executed five times during a simulated time of one second 5 times (0.2, 0.4, 0.6, 0.8, 1.0, etc.) Therefore, during 100 seconds, the system will have this event occurring 500 times at simulated time = 0, simulated time = 0.2, simulated time = 0.4, etc. for each aircraft.

10. The \( \text{TestUpdateAttribute} \) event points to method \( \text{TestUpdateAttribute}() \). This method is kicked off by the event’s framework scheduler at Simulation Time = zero. At each Simulation Time, each parallel instance (one for each aircraft) of the \( \text{TestUpdateAttribute}() \) method in C_RandomMotion.C, computes the circular path position of each aircraft. The center of the circular path for each object is established randomly. Using these ECR coordinates, each aircraft’s
circular path trajectory is established by keeping the Z-coordinate (“Altitude”) fixed, but changing the Y(“Longitude”) coordinate at every simulation time.

11. The event for the radars is “Scan”. At the initial Simulation Time, Scan is schedule and it happens at each simulated second.

12. The Scan event points to method Scan() for each radar. This method is kicked off by the event’s framework scheduler at Simulation Time = zero. At each discrete Simulation Time, each parallel instance of the Scan() method computes the proximity of an aircraft to each ground radar. Proximity (range) is calculated in parallel using radar position and moving entity position vectors via 

\[ nge = \sqrt{\Delta X^2 + \Delta Y^2 + \Delta Z^2} \]

, where \( \Delta \) represents the difference between radar and aircraft positions (\( \Delta \)latitude, \( \Delta \)longitude, and \( \Delta \)altitude) in Earth Centered Rotational Coordinates (ECR).

13. In each simulated second, a hardware delay of 50 milliseconds occurs in each radar object (we have three instances of radar)... therefore for each radar a total of 5 Seconds (Wall Clock Time) for each 100 Seconds of Simulated Time is expected.

14. The aircrafts do not know the existence of the radars but the radars can know their position.
The simulation randomly initializes the position of each aircraft object and each ground radar object. Their position \((X, Y, Z)\) is represented by the earth centered rotational Cartesian coordinates (ECR). After initialization, the simulation detects an aircraft’s proximity to a ground radar using a pre-established range value for detection. At initial sim execution time, discrete events *Scan* and *TestUpdateAttribute* are implemented.

![UML schematics of the development with two types of Simulation Objects (Aircraft and Radar) and two events (i.e., Scan and Trajectory Update).](image)

Figure 35: UML schematics of the development with two types of Simulation Objects (Aircraft and Radar) and two events (i.e., Scan and Trajectory Update).
Figure 36: The implementation of the problem with different components

After initialization, the Scan and TestUpdateAttribute events are scheduled by the parallel framework for the beginning of the simulation execution at Simulatio Time = zero using constructs such as SCHEDULE_Scan(0.0, this) and SCHEDULE_TestUpdateAttribute(0.0, this). For this aircraft detection simulation program, the Scan event points to method Scan() defined in the C_Radar class as a WpEntityComponent method. This method is kicked off by the event’s framework scheduler at Simulation Time = zero. At each discrete Simulation Time (of 1 second), each parallel instance of the Scan() method cycles through a while-loop to compute the proximity of an aircraft to each radar.
For this aircraft detection simulation program, the Scan event points to method Scan() defined in the C_Radar class as a \textit{WpEntityComponent} method. This method is kicked off by the event’s framework scheduler at Simulation Time = zero. At each discrete Simulation Time (of 1 second), each parallel instance of the Scan() method cycles through a while-loop to compute the proximity of an aircraft to each radar.

Figure 37: Area selected for the verification & validation simulation

The \textit{TestUpdateAttribute} event points to method \textit{TestUpdateAttribute()}. This method is kicked off by the event’s framework scheduler at Simulation Time = zero. At each simulated time, each parallel instance of the \textit{TestUpdateAttribute()} method, cycles through a while-loop that computes the circular path position of each aircraft. The center of the circular path for each object is established randomly. Using these ECR coordinates,

\begin{itemize}
  \item Latitude = 0 degrees at equator line
  \item Above equator \hspace{0.5cm} 0 ° \leq \text{Lat} \leq 90 °
  \item Below equator \hspace{0.5cm} -90 ° \leq \text{Lat} \leq 0 °
  \item Longitude = 0 degrees at Prime Meridian Line (PML)
  \item West of PML \hspace{0.5cm} -180 ° \leq \text{Lon} \leq 0 °
  \item East of PML \hspace{0.5cm} 0 ° \leq \text{Lon} \leq 180 °
\end{itemize}
each aircraft’s circular path trajectory is established by keeping the Z-coordinate ("Altitude") fixed, but changing the Y("Longitude") coordinate at every simulation time. For each Y-coordinate change, the X-coordinate is incremented using the circular path governing equation. Using a pre-established radial distance and initial center coordinates \((X_0, Y_0, Z_0)\), the minimum and maximum longitude for each entity circular flight is determined by \(Y_{\text{max}} = Y_0 + \text{radius}\) and \(Y_{\text{min}} = Y_0 - \text{radius}\), respectively. Once the longitude boundaries are established for each object’s circular path, the longitude is incremented from minimum to maximum at each discrete sim time by

\[
Y = Y_{\text{min}} + \frac{(Y_{\text{max}} - Y_{\text{min}})}{C} \times \text{SIM\_TIME}.\text{GetTime}(), \text{where SIM\_TIME ranges from 0 to C.}
\]

The corresponding latitude is then computed by \(X_0 \pm \sqrt{r^2 - Y^2 - Y_0^2 + 2Y_0Y} \).

Once the circular positions are determined, the moving entity position defined by \((X, Y, Z)\) is mapped for each corresponding aircraft and published. As a result, any subscribers to the simulation federation aircraft object can receive new dynamic aircraft positions.

The aircraft detection simulation code implements each instance of aircrafts as federation objects and initializes their subscription. Federation objects (Fo) are used to facilitate the grouping of entity and entity components with related attributes. The grouped attributes can then be distributed and published to other entity components and entities that are subscribers. During simulation execution, object attributes such as dynamic position (latitude, longitude, and altitude) and Aircraft identification are published.
The benchmark for the different time management and synchronization schemes (TW, BTB, and BTW) is explained in the following experiment:

These are the specifications for the computer systems used in the experiment:

Table 13: Computer systems specification

<table>
<thead>
<tr>
<th>PC Specs</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Processor</td>
<td>i5</td>
</tr>
<tr>
<td>Speed</td>
<td>2.60 GHz</td>
</tr>
<tr>
<td>Ram</td>
<td>6 GB</td>
</tr>
<tr>
<td>System Type</td>
<td>64 bit</td>
</tr>
</tbody>
</table>

The experiment executed several runs (24 in total = 8 for each time management and synchronization scheme) with specific computing configurations (Figures 46 and 47).

Table 14: The utilization of the different systems during the experiments

<table>
<thead>
<tr>
<th># Nodes</th>
<th># of PCs - PC #</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Global</td>
<td>PC 1</td>
</tr>
<tr>
<td>BTW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>BTB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>TW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
Table 15: Experiment results for each computing configuration and time management and synchronization scheme (BTW, BTB, and TW)

<table>
<thead>
<tr>
<th># Nodes</th>
<th>Wall Clock Time</th>
<th>Speedup Rel</th>
<th>Speedup Theoretical</th>
<th>PT</th>
<th>Min Committed PT per node</th>
<th>Max Committed PT per node</th>
<th>Mean Committed PT per node</th>
<th>Sigma</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>16.5</td>
<td>1.0</td>
<td>3.0</td>
<td>15.6</td>
<td>15.6</td>
<td>15.6</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>14.1</td>
<td>1.2</td>
<td>3.0</td>
<td>15.6</td>
<td>5.3</td>
<td>10.3</td>
<td>7.8</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>12.4</td>
<td>1.3</td>
<td>3.0</td>
<td>15.7</td>
<td>5.2</td>
<td>5.3</td>
<td>5.2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>11.4</td>
<td>1.4</td>
<td>3.0</td>
<td>15.6</td>
<td>0.0</td>
<td>5.3</td>
<td>3.9</td>
</tr>
<tr>
<td>2 to 4</td>
<td>14</td>
<td>6.1</td>
<td>2.7</td>
<td>3.0</td>
<td>15.4</td>
<td>0.0</td>
<td>5.2</td>
<td>1.1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>6.5</td>
<td>2.6</td>
<td>3.0</td>
<td>15.5</td>
<td>0.0</td>
<td>5.2</td>
<td>1.9</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>9.4</td>
<td>1.8</td>
<td>3.0</td>
<td>15.5</td>
<td>0.0</td>
<td>5.2</td>
<td>3.9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>10.5</td>
<td>1.6</td>
<td>3.0</td>
<td>15.8</td>
<td>5.3</td>
<td>5.3</td>
<td>5.3</td>
</tr>
<tr>
<td>BTB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>16.1</td>
<td>1.0</td>
<td>3.0</td>
<td>15.6</td>
<td>5.7</td>
<td>5.7</td>
<td>5.7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>62.1</td>
<td>0.3</td>
<td>3.0</td>
<td>15.6</td>
<td>13.0</td>
<td>7.8</td>
<td>2.5</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>148.0</td>
<td>0.1</td>
<td>3.0</td>
<td>15.6</td>
<td>5.1</td>
<td>5.2</td>
<td>3.9</td>
</tr>
<tr>
<td>2 to 4</td>
<td>14</td>
<td>7.7</td>
<td>2.1</td>
<td>3.0</td>
<td>15.4</td>
<td>0.0</td>
<td>5.2</td>
<td>1.1</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>6.2</td>
<td>2.6</td>
<td>3.0</td>
<td>15.3</td>
<td>0.0</td>
<td>5.2</td>
<td>1.2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>9.4</td>
<td>1.7</td>
<td>3.0</td>
<td>15.5</td>
<td>0.0</td>
<td>5.2</td>
<td>3.9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>10.2</td>
<td>1.6</td>
<td>3.0</td>
<td>15.6</td>
<td>5.2</td>
<td>5.2</td>
<td>5.2</td>
</tr>
<tr>
<td>TW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>17.2</td>
<td>1.0</td>
<td>3.0</td>
<td>15.6</td>
<td>15.6</td>
<td>15.6</td>
<td>15.6</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>13.8</td>
<td>1.2</td>
<td>3.0</td>
<td>15.6</td>
<td>5.3</td>
<td>10.3</td>
<td>7.8</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>12.6</td>
<td>1.4</td>
<td>3.0</td>
<td>15.6</td>
<td>5.2</td>
<td>5.3</td>
<td>5.2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>10.9</td>
<td>1.6</td>
<td>3.0</td>
<td>15.5</td>
<td>0.0</td>
<td>5.2</td>
<td>3.9</td>
</tr>
<tr>
<td>2 to 4</td>
<td>14</td>
<td>5.9</td>
<td>2.9</td>
<td>3.0</td>
<td>15.4</td>
<td>0.0</td>
<td>5.2</td>
<td>1.1</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>6.2</td>
<td>2.8</td>
<td>3.0</td>
<td>15.3</td>
<td>0.0</td>
<td>5.2</td>
<td>1.9</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>10.0</td>
<td>1.7</td>
<td>3.0</td>
<td>15.5</td>
<td>0.0</td>
<td>5.2</td>
<td>3.9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>11.4</td>
<td>1.5</td>
<td>3.0</td>
<td>15.8</td>
<td>5.2</td>
<td>5.3</td>
<td>5.3</td>
</tr>
</tbody>
</table>

Table 16 shows the Wall Clock Time (in seconds) for each experiment. These are the definitions of the columns of Figure 48:

**Wall Clock Time** (elapsed wall time) is a measure of the real time that elapses from start to end, including time that passes due to programmed (artificial) delays or waiting for resources to become available. In other words, it is the difference between the time at which a simulation finishes and the time at which the simulation started. It is given in seconds.
**Speedup Theoretical** is based on the Simulation Object with the longest processing time. It is the maximum (approximated) Speedup expected using an excellent parallelized scheme (taking advantage of the programming features, computer configuration of the system, and partitions of the problem).

**PT** (processing time) is the total CPU time required to process committed events, in seconds. The processing time does not include the time required to process events that are rolled back, nor does it include additional overheads such as event queue management and messages.

**Min Committed PT per Node** is the Minimum Committed Processing Time per Node of the computing system configuration utilized.

**Max Committed PT per Node** is the Maximum Committed Processing Time per node of the computing system configuration utilized.

**Mean Committed PT per Node** is the Mean Committed Processing Time per node of the Computing system configuration utilized.

**Sigma** is the standard deviation of the processing times of the different nodes utilized in the experiment.

The benchmark for the different time management and synchronization schemes (TW, BTB, and BTW) is depicted in Figure 42. TW has the best result of 2.9 (close to the theoretical speedup of 3.0). BTW and TW are very comparable. BTW does not perform
well with this type of task for distributed systems. However, BTW has better performance with the utilization of multicore configurations (tightly coupled) for this specific problem.

Figure 38: Combined Speedup chart for BTW, BTB, and TW

5.3 Complexity

Measuring simulation algorithm complexity is challenging. Researchers have proposed measures that categorized complexity by measures such as number of codes lines, code internal structures, and interfaces. Shao, J & Wang, Y. ,(2003) and Misra, S. (2006) examine software complexity with the perspective of software being a product of
the human creative process. As such, they explore complexity measures based on cognitive weights, which takes into account the complexity of cognitive and psychological components of software. In this paradigm, cognitive weights represent the effort and relative time required to comprehend a software piece. The approach suggests that software complexity is directly propositional to the complexity of understanding the information contained in it. This measure is the most recognized by the research community; as a result it was selected.

Using cognitive weights of basic control structures to measure complexity addresses the cognitive and architectural aspects of software complexity. Basic fundamental logic blocks of software constructs such as conditional if-then statements, method calls, for-loops, etc. are assigned a weight value. Table 17 shows the cognitive weights of each type of basic software control structure (BCS).

<table>
<thead>
<tr>
<th>Category</th>
<th>Basic Control Structure</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Sequence</td>
<td>1</td>
</tr>
<tr>
<td>Branch</td>
<td>If-Then-Else</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Case</td>
<td>3</td>
</tr>
<tr>
<td>Iteration</td>
<td>For Loop</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Repeat-until</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>While-do</td>
<td>3</td>
</tr>
<tr>
<td>Embedded Component</td>
<td>Function Call</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Recursion</td>
<td></td>
</tr>
<tr>
<td>Concurrency</td>
<td>Parallel</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Interrupt</td>
<td>4</td>
</tr>
</tbody>
</table>
The total cognitive weight of a piece of software is computed by applying equation 5.1

\[ W_c = \sum_{j=1}^{q} \left[ \prod_{k=1}^{m} \sum_{i=1}^{n} W_c(j,k,i) \right] \]

For the simplified case where basic software control structure are not embedded within other BCS (m=1), the total cognitive weight simplifies to equation 5.2.

\[ W_c = \sum_{j=1}^{q} \sum_{i=1}^{n} W_c(j,i). \]

Cognitive weight scores for a particular block of software contributes more to total weigh if multiple basic control structures are encompassed within nested sections. For example, methodA() in Figure 39 achieves a larger cognitive weight than methodB() due to nested while-loop inside the if-then construct.

Figure 39: Cognitive Weights Sample Calculation
This research implements cognitive weights to measure the complexity of a parallel discrete event simulation with respect of implemented algorithms. Because each simulation object in a simulation implements discrete events defined as code functions, the complexity of each object is also computed by applying equations 5.1 and 5.2 for all event/methods mapped to each simulation object. As a result, several parameters that gage simulation complexity are then used as inputs to the deep belief neural network for deep learning. These are: Total Simulation program cognitive weights, maximum cognitive weights of all sim objects, minimum cognitive weights of all objects, mean cognitive weights of all objects.

In addition, Table 17 captures other parameters that define the hardware, flow processing, potential messaging and other important characteristics that define a parallel distributed discrete-event simulator.
Table 17: Vector that defines a PDDES problem

<table>
<thead>
<tr>
<th>Complexity Parameters that Capture the hardware/software Structure of a Parallel Distributed Discrete-Event Simulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Simulation Program Cognitive Weights</td>
</tr>
<tr>
<td>Number of Simulation objects</td>
</tr>
<tr>
<td>Types of Simulation objects</td>
</tr>
<tr>
<td>Mean Events per Simulation Object</td>
</tr>
<tr>
<td>STD Events per Simulation Object</td>
</tr>
<tr>
<td>Mean Cog Weights of All objects</td>
</tr>
<tr>
<td>STD Cog Weights of All objects</td>
</tr>
<tr>
<td>Number of Global Nodes</td>
</tr>
<tr>
<td>Mean Local Nodes per Computer</td>
</tr>
<tr>
<td>STD Local Nodes per Computer</td>
</tr>
<tr>
<td>Mean Number of cores/threads</td>
</tr>
<tr>
<td>STD Number of cores/threads</td>
</tr>
<tr>
<td>Mean processor Speed</td>
</tr>
<tr>
<td>STD processor Speed</td>
</tr>
<tr>
<td>Mean RAM</td>
</tr>
<tr>
<td>STD RAM</td>
</tr>
<tr>
<td>Critical Path%</td>
</tr>
<tr>
<td>Theoretical Speedup</td>
</tr>
<tr>
<td>Locals Event/(Local Events + External Events)</td>
</tr>
<tr>
<td>Subscribers/(Publishers + Subscribers)</td>
</tr>
<tr>
<td>Block or Scatter?</td>
</tr>
</tbody>
</table>

- **Total Simulation Program Cognitive Weights:** It is the total number of cognitive weights of the simulation program.

- **Number of Simulation objects:** It is the total number of simulation objects in the simulation.

- **Types of Simulation objects:** It is the number of classes of Simulation Objects utilized in the simulation.
• **Mean Events per Simulation Object**: It is the mean of the events per simulation object.

• **STD Events per Simulation Object**: It is the standard deviation of the events per simulation object.

• **Mean Cog Weights of All objects**: It is the mean of the number of cognitive weights used by the simulation objects in the simulation.

• **STD Cog Weights of All objects**: It is the standard deviation of the number of cognitive weights used by the simulation objects in the simulation.

• **Number of Global Nodes**: It is the total number of Global Nodes in the simulation.

• **Mean Local Nodes per Computer**: It is the mean of the local nodes per global node utilized in the simulation.

• **STD Local Nodes per Computer**: It is the standard deviation of the local nodes per global node utilized in the simulation.

• **Mean Number of cores**: It is the mean number of cores/threads utilized by each global node in the simulation.

• **STD Number of cores**: It is the standard deviation of number of cores/threads utilized by each global node in the simulation.

• **Mean processor Speed**: It is the mean processor speed of the CPUs used in the simulation.
- **STD processor Speed**: It is the standard deviation of the speed of the CPUs used in the simulation.

- **Mean RAM**: It is the mean of the RAM memory used by the CPUs in the system.

- **STD RAM**: It is the standard deviation of the RAM memory used by the CPUs in the system.

- **Critical Path%**: It is the Critical Path taking into consideration the sequential estimated processing time.

- **Theoretical Speedup**: It is the theoretical (maximum) speedup to be achieved with perfect parallelism in the simulation.

- **Local Events/(Local Events + External Events)**: It is the ratio of the total local events divided by the summation of the total local events and the total external events during a specific unit of Simulation Time (estimated).

- **Subscribers/(Publishers + Subscribers)**: It is the ratio of the total number of objects subscribing to a particular object divided by the summation of the total number of publishers and subscribers.

- **Block or Scatter?**: Block and scatter are decomposition algorithms being used to distribute the simulation objects in the parallel/distributed system - If Block is being selected then this value is 1, and if Scatter is selected then this value is 0.

For example, for the program discussed in Section 5.2, this is the following input vector using the hardware and complexity specifications from Figure 39 and Tables.
15 and 16 for a configuration of 4 Global Nodes and 1 Local Node (a loosely coupled system) using “Block” as the distribution scheme for the Simulation Objects is:

Table 18: Vector that defines the PDDES problem of Section 5.2 with 4 Global Nodes and 1 Local Node using Block

<table>
<thead>
<tr>
<th>Complexity Parameters that Capture the hardware/software Structure of a Parallel Distributed Discrete-Event Simulator</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Simulation Program Cognitive Weights</td>
<td>2919</td>
</tr>
<tr>
<td>Number of Sim objects</td>
<td>6</td>
</tr>
<tr>
<td>Types of Sim objects</td>
<td>3</td>
</tr>
<tr>
<td>Mean Events per Object</td>
<td>1</td>
</tr>
<tr>
<td>STD Events per Simulation Object</td>
<td>0</td>
</tr>
<tr>
<td>Mean Cog Weights of All objects</td>
<td>1345</td>
</tr>
<tr>
<td>STD Cog Weights of All objects</td>
<td>1317</td>
</tr>
<tr>
<td>Number of Global Nodes</td>
<td>4</td>
</tr>
<tr>
<td>Mean Local Nodes per Computer</td>
<td>1</td>
</tr>
<tr>
<td>STD Local Nodes per Computer</td>
<td>0</td>
</tr>
<tr>
<td>Mean Number of cores</td>
<td>1</td>
</tr>
<tr>
<td>STD Number of cores</td>
<td>0</td>
</tr>
<tr>
<td>Mean processor Speed</td>
<td>2.1</td>
</tr>
<tr>
<td>STD processor Speed</td>
<td>0.5</td>
</tr>
<tr>
<td>Mean RAM</td>
<td>6.5</td>
</tr>
<tr>
<td>STD RAM</td>
<td>1.9</td>
</tr>
<tr>
<td>Critical Path%</td>
<td>0.32</td>
</tr>
<tr>
<td>Theoretical Speedup</td>
<td>3</td>
</tr>
<tr>
<td>Local Events/(Local Events + External Events)</td>
<td>1</td>
</tr>
<tr>
<td>Subscribers/(Publishers + Subscribers)</td>
<td>0.5</td>
</tr>
<tr>
<td>Block or Scatter?</td>
<td>1</td>
</tr>
</tbody>
</table>
And the output for a DBN will be based on Table 16 where the Wall Clock Time for BTW is 11.4 seconds, for BTB is 162.6 seconds, and for TW is 10.9 seconds. Table 19 displays the output vector of the respective case study.

Table 19: TW has the minimum wall clock time

<table>
<thead>
<tr>
<th>Time Management and Synchronization Scheme</th>
<th>Best (Minimum Wall Clock Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTW</td>
<td>0</td>
</tr>
<tr>
<td>BTB</td>
<td>0</td>
</tr>
<tr>
<td>TW</td>
<td>1</td>
</tr>
</tbody>
</table>

5.4 New Methodology

This is the methodology devised in order to recognize the best time management and synchronization scheme for a PDDES problem. The input vector is define based on the complexity and features of the software, hardware, and messaging of the PDDES problem (as depicted in Table 19). The output vector defines the best time management and synchronization scheme (TW, BTW, BTB) as shown in Table 20 for the respective case study. This pattern classification is achieved using a DBN trained with case studies performed a Parallel Distributed Discrete-Event Simulator.
Figure 40: Classification of optimistic synchronization scheme with DBN
CHAPTER SIX: TESTING

6.1 Introduction

This chapter deals with the testing of our proposed idea of using deep belief networks as pattern-matching (classification) mechanisms for time management and synchronization of parallel distributed discrete-event simulations. The performance criterion and the knowledge acquisition scheme will be presented. This discussion includes an analysis of the results.

6.2 Performance Criterion Used

For these studies the performance criterion which will be used is the minimum wall-clock time. Wall-clock time stands for the actual time taken by the computer system to complete a simulation. Wall-clock time is different from CPU time. CPU time measures the time during which the processor (s) is (are) actively working on a certain task (s). Wall-clock time calculates the total time for the process (es) to complete.
6.3 Selected Problems (Rationale)

Several PDDES problems were selected to generate the case studies in order to train the DBN. A total of 400 case studies were used. Two hundred case studies were selected for training (i.e., to obtain the learning parameters), one hundred case studies for validation (i.e., to obtain the right architecture, and one hundred to test the developed DBN for classification.

6.4 Training Session

Deep learning of the highly non-linear patterns associated with the selected PDDES problems was performed using several steps. Prior to training the DBN all input data were normalized so the data set of matrix \( v \) elements had a magnitude range of \( \{0,1\} \). The structure of the deep belief net used three hidden layers, one input layer, and one output layer. The output layer consisted of three neurons corresponding to the classification labels for the parallel simulation optimization schemas TW, BTW, and BTB. The size of the input units were 400 x 21 data points, where there were 400 case studies representative of various hardware and simulation complexity configurations. Twenty one columns on data represented the simulation complexity parameters as described PDDES complexity vector (Table 18.)
The number of hidden neurons in the first, second, and third hidden layers were set to 100, 100, 100 – respectively. Other DBN parameter values used for deep learning are shown in Table 20.

Table 20: PDDES DBN Parameters

<table>
<thead>
<tr>
<th>Learning Rate</th>
<th>Hidden Layer 1 Neurons</th>
<th>Hidden Layer 2 Neurons</th>
<th>Hidden Layer 3 Neurons</th>
<th>Number Output Neurons</th>
<th>RBM Mini-Batch Size</th>
<th>RBM Epochs</th>
<th>DBN Mini-Batch Size</th>
<th>DBN Epochs</th>
<th>Weight Cost</th>
<th>Momentum</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-6}$</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>3</td>
<td>10</td>
<td>50</td>
<td>10</td>
<td>50</td>
<td>0.01</td>
<td>0.5</td>
</tr>
</tbody>
</table>

During learning, the DBN is pre-trained one layer at a time with 50 epochs using contrastive divergence and Gibbs sampling, starting from the bottom first layer of visible variables to the top of the network – one RBM at a time. After pre-train, the final DBN is fine-tuned (with 50 epochs) in a top-down mode using supervised backpropagation. The next three figures show the cross entropy and root mean square errors at each epoch of training during each hidden layer.
Figure 41: Root Mean Square error and Cross Entropy error for hidden layer 1

Figure 42: Root Mean Square error and Cross Entropy error for hidden layer 2
6.5 Testing Performance

Once the DBN was trained, its classification performance was tested by applying one hundred PDDES input test vectors to the visible units of the trained DBN. In reaction to the input vectors, the DBN neuron activation probabilities propagate through the net layer-by-layer until the DBN output is reached. Then, the pattern detection and classification is done by selecting the resulting maximum probability of the three output neurons corresponding to either TW, BTW, or BTB. For example, for each test input data vector (Table 18) corresponding to our particular PDDES case, the DBN classified the input data and predicted that the hardware and software complexity associated with the test data vector will execute the parallel simulation most efficiently if it is run with
either TW, BTW, or BTB time synchronization schema. As a result, the DBN predicted which schema is best for the given associated PDDES input data.

Figure 44 shows the classification performance of this research’s trained DBN. The training set was of 200 case studies selected, the validation set with 100 case studies, and the testing set was composed of 100 case studies.

<table>
<thead>
<tr>
<th></th>
<th>Using 3 Hidden Layers</th>
<th>Using 2 Hidden Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BTB</strong> Actual</td>
<td>98.9%</td>
<td>13.9%</td>
</tr>
<tr>
<td><strong>BTW</strong> Actual</td>
<td>1.1%</td>
<td>99.3%</td>
</tr>
<tr>
<td><strong>TW</strong> Actual</td>
<td>5.6%</td>
<td>6.9%</td>
</tr>
</tbody>
</table>

Figure 44: Confusion matrix for two DBNs.

### 6.6 Analysis

Stating the research question initiated the research methodology process. This investigation started by asking: Is there a mechanism to accurately model and predict the best time management and synchronization scheme for a parallel discrete event
simulation environment (program and hardware)? Based on the results, this was accomplished in spite of the limited number of case studies.
CHAPTER SEVEN: CONCLUSIONS AND FUTURE RESEARCH

This research proposed a methodology for pattern recognition and detection using deep belief neural networks and complexity measures of parallel distributed discrete event environments. The pattern recognition engine selected the best time management and synchronization scheme in order to have the best performance with respect of the minimization of the simulation wall clock time.

7.1 Conclusions

Discrete event simulation can be viewed as a sequence of event computations where each event computation contains a simulation-time time stamp indicating the occurrence of an event. These event computations can modify state variables, and/or schedule new events into the simulated future. Discrete-event simulation in single processing systems is not a difficult computational problem. However, the implementation of discrete-event simulation in parallel and/or distributed systems is a very complex problem. Time management and synchronization schemes are critical in order to accomplish a discrete-event simulation that takes advantage of parallelism. It is very well recognized that to take advantage offered by parallel and distributed systems, the time management and synchronization mechanism must be optimistic in nature. Optimistic methods do not avoid causality errors because events are processed
optimistically. When a causality error is determined to have occurred, then simulation system is rolled back to a correct point in simulated time.

A very important situation in favor of PDDES is the utilization of Simultaneous Multi-threading (SMT) and Chip Multi-Processor (CMP) technologies. Therefore, hardware components to execute parallel distributed DES must be taken into consideration. In addition, the features of the parallel distributed DES software model must be described. Thus, the dimensions must include the features of the software, the hardware, and the messaging/sharing infrastructure.

This research implemented a pattern recognition scheme to identify the best optimistic time management and synchronization scheme to execute a particular parallel distributed discrete even simulation problem. This innovative pattern recognition approach utilizes Deep Belief Neural Networks and measures of complexity to quantify and capture the highly non-linear structure and variable dependencies of the parallel distributed discrete even simulation problem. The implementation of this approach was successful and eliminated trial and error or utilizing “inconsistent” and/or “fuzzy” rules in order to select the optimal time management and synchronization scheme. This method is direct (i.e., fast execution) and automatically selects the right simulation optimization scheme (TW, BTW, BTB).

The deep belief network model created can be used as a detector of patterns that were learned during training by inputting a mixture of diverse data from different
problems in PDDES. In reaction to an input vector containing previously established variable dependencies of the parallel distributed discrete even simulation problem, the ingested vector then triggers neuron activation probabilities that propagate through the DBN layer-by-layer until the DBN output is reached. The DBN output probabilities are then examined for classification to select the best optimistic time management and synchronization scheme to be used.

Figure 45: Pattern recognition methodology for the identification of an optimistic time management and synchronization scheme for PDDES
This research’s very important contribution was the analysis and selection of features to capture the software and hardware structure of a PDDES problem. The analysis found the following input vector:

Table 21: Vector that defines a PDDES problem

<table>
<thead>
<tr>
<th>Complexity Parameters that Capture the hardware/software Structure of a Parallel Distributed Discrete-Event Simulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Simulation Program Cognitive Weights</td>
</tr>
<tr>
<td>Number of Simulation objects</td>
</tr>
<tr>
<td>Types of Simulation objects</td>
</tr>
<tr>
<td>Mean Events per Simulation Object</td>
</tr>
<tr>
<td>STD Events per Simulation Object</td>
</tr>
<tr>
<td>Mean Cog Weights of All objects</td>
</tr>
<tr>
<td>STD Cog Weights of All objects</td>
</tr>
<tr>
<td>Number of Global Nodes</td>
</tr>
<tr>
<td>Mean Local Nodes per Computer</td>
</tr>
<tr>
<td>STD Local Nodes per Computer</td>
</tr>
<tr>
<td>Mean Number of cores/threads</td>
</tr>
<tr>
<td>STD Number of cores/threads</td>
</tr>
<tr>
<td>Mean processor Speed</td>
</tr>
<tr>
<td>STD processor Speed</td>
</tr>
<tr>
<td>Mean RAM</td>
</tr>
<tr>
<td>STD RAM</td>
</tr>
<tr>
<td>Critical Path%</td>
</tr>
<tr>
<td>Theoretical Speedup</td>
</tr>
<tr>
<td>Locals Event/(Local Events + External Events)</td>
</tr>
<tr>
<td>Subscribers/(Publishers + Subscribers)</td>
</tr>
<tr>
<td>Block or Scatter?</td>
</tr>
</tbody>
</table>
7.2 Contributions to the Body of Knowledge

There are two important contributions to the Body of Knowledge:

1. This research’s approach is the first approach to select an appropriate optimistic time management and synchronization scheme for a Parallel Distributed Discrete-Event Simulation problem. This approach is very efficient and fast in execution.

2. The utilization of software measures of complexity in combination with hardware features of CMP/SMT and loosely coupled systems to capture the structure of a PDDES problem. This approach to describe structures of PDDES problems is unique.

7.3 Future Research

This dissertation contributes with a unique idea. A new approach to an existing problem that is exceedingly complex in nature was developed and tested. We recognize that parallel distributed discrete event simulation is very important and the trends in CMP and SMT, and the multi-enterprise (system of systems) simulations will make it more relevant and dominant in the future. These trends and our approach can be utilized for more research endeavors such as:
1. Compare between DBNs and other machine learning paradigms such as other deep neural network architectures, classification trees, and support vector machines. Since only one type of neural networks (deep belief neural networks) were used in this study due to the highly nonlinear nature of the problem and the potential for thousands of data samples, it is important to consider other machine learning techniques in future research.

2. Perform an intensive/systematic sensitivity analysis of the input vector and its components. Some components may need to be combined and/or eliminated in order to increase simulation performance. This can be done with different statistical procedures developed in recent years to fine-tune neural networks. In addition, the utilization of other machine learning paradigms such as genetic programming have the potential to help in the creation of synthetic variables.

3. Select other measures in complexity of software. For example, the utilization of chaos theory. We know that computer software and the software development process belong to the class of complex systems. Chaos theory has been used by several researchers to measure software complexity. This could potentially be a very important study.

4. Utilize PDDES in order to support the development of Big Data Analytics schemes. PDDES can use different partitions/clusters of the new storage systems such as Hadoop in order to develop complex search and predictive
mechanisms that take advantage of the distributed nature of PDDES and Big Data technologies.

5. Determine, very importantly, how to use the mechanism provided in order to develop new algorithms and monitoring rules to combine TW and BTB in a more efficient way (rather than static parameters). Therefore, we have the potential to calculate GVT more effectively.

6. Perform a more formal design of experiments (DOE) in order to select the different PDDES problems to obtain the training, validation, and testing data. This is essential in order to capture the multidimensional space of possible patterns.

We hope this research can potentially generate an awareness in the simulation community (in particular Industrial Engineers) that DES encompass other aspects of simulation, in addition to Arena and/or Simio, since PDDES can be used not only to accelerate the simulations (i.e., reduce execution time) but to also create unique solutions in multi-enterprise systems of systems and provide a better mapping to the developing of DES software and web-based/mobile applications.
APPENDIX A: SPACE SHUTTLE MAIN ENGINE
Figure 46: The Space Shuttle Main Engine
APPENDIX B: DEEP BELIEF NEURAL NETWORK NEURON MODEL
Idea behind deep belief

- Two or more hidden layers of neurons
- Stacking RBMs
- Pre-train W's each layer independently unsupervised via Contrastive Divergence
- Layer above uses transpose of W as initial value
- Fine tune W's with supervised backpropagation wake-sleep algorithms

\[
p(h|v; w, a) = \frac{1}{1 + e^{-\sum v_{i=1}^a}}
\]

\[
p(v| h; w, b) = \frac{1}{1 + e^{-\sum h_{i=1}^b}}
\]

\(p()\) is Neuron Activation Conditional Probability

Figure 47: DBN Neuron Model
LIST OF REFERENCES


Ning, L. (2013, June). Network Intrusion Classification Based on Probabilistic Neural Network. In Computational and Information Sciences (ICCIS), 2013 Fifth International Conference on (pp. 57-59). IEEE.


Prediction. In Neural Information Processing (pp. 766-773). Springer Berlin Heidelberg. doi:10.1007/978-3-642-24965-5_86


Tran, V. T., AlThobiani, F., & Ball, A. (2014). An approach to fault diagnosis of


