information

MDPI

Article

Design of Distributed Discrete-Event Simulation Systems Using Deep Belief Networks

Edwin Cortes [1], Luis Rabelo [2], Alfonso T. Sarmiento [3,*] and Edgar Gutierrez [4]

[1] Institute of Simulation and Training, Orlando, FL 32816, USA; Edwin.Cortes@knights.ucf.edu
[2] Department of Industrial Engineering and Management Systems, University of Central Florida, Orlando, FL 32816, USA; Luis.Rabelo@ucf.edu
[3] Faculty of Engineering, Universidad de La Sabana, Chia 250001, Colombia
[4] Center for Latin America Logistics Innovation, Bogota 110111, Colombia; edgargutierrezfranco@gmail.com
* Correspondence: alfonsosava@unisabana.edu.co

Received: 2 September 2020; Accepted: 29 September 2020; Published: 1 October 2020

Abstract: In this research study, we investigate the ability of deep learning neural networks to provide a mapping between features of a parallel distributed discrete-event simulation (PDDES) system (software and hardware) to a time synchronization scheme to optimize speedup performance. We use deep belief networks (DBNs). DBNs, which due to their multiple layers with feature detectors at the lower layers and a supervised scheme at the higher layers, can provide nonlinear mappings. The mapping mechanism works by considering simulation constructs, hardware, and software intricacies such as simulation objects, concurrency, iterations, routines, and messaging rates with a particular importance level based on a cognitive approach. The result of the mapping is a synchronization scheme such as breathing time buckets, breathing time warp, and time warp to optimize speedup. The simulation-optimization technique outlined in this research study is unique. This new methodology could be realized within the current parallel and distributed simulation modeling systems to enhance performance.

Keywords: parallel distributed discrete-event simulation; deep learning; deep belief networks; breathing time buckets; breathing time warp; time warp

1. Introduction and Background

A fundamental paradigm in simulation is discrete-event simulation (DES) [1]. DES is characteristically involved with the modeling simulation of systems as a succession of events in a discrete fashion. These events arise at specific times and have the potential to change the state of the system. The execution of a single DES program on parallel and distributed computational systems is called parallel and distributed discrete-event simulation (PDDES) [2]. These systems can have characteristics from the high-performance computing systems and the hardware multi-threaded systems. These systems include schemes that communicate through shared memory modules. They also can include a more loosely coupled system where each processor has its local memory and a communication scheme based on messages. In addition, new initiatives, especially on platforms such as cloud-based virtualized arrangements and Internet-scale settings, can make PDDES more viable, easy to use, and cost-effective [3–12].

The widely used simulation schemes with distribution and parallelism at the event level aim to divide the global simulation tasks into a set of logical processes (LP) with communication capacities. These strategies exploit the inherent parallelism between the respective components of the problem with the concurrent execution of these LPs. This arrangement results in a simulation due to the collaboration between the set of LPs [13].

The simulations of systems with LPs have an architecture based on sets of LPs. The design of this arrangement of LPs to execute events synchronously or asynchronously in parallel has to have a communication system not only to exchange data but also to synchronize activities. Each LP is assigned to a specific region of the model to be simulated. The simulation engines can operate in an event-driven fashion and execute local events and the respective subset of state variables (and generate remote events—i.e., events in other LPs).

PDDES systems use synchronization techniques that fall "into two main categories: conservative approaches that avoid violating the constraint of local causality, and optimistic approaches that allow violations to occur but provide a mechanism" for recovery called rollback [14]. Rollback involves undoing incorrect modifications. The most effective implementation of the PDDES approach is the optimistic algorithm [15]. It is widely used for simulations in logistics, missile defense, and computational physics [16]. In this paper, we investigate the ability of deep learning neural networks to provide a mapping between features of a PDDES (software and hardware) to an optimistic time synchronization scheme to optimize speedup performance. We will explain the different synchronization techniques that this research implements below.

1.1. Conservative and Optimistic Schemes

Simulation objects must interact in a particular fashion to accomplish an efficient parallel and distributed execution with perfect integrity. Several innovative techniques have been developed to solve this challenging problem from conservative and optimistic viewpoints [2].

1.1.1. Conservative Viewpoint

The conservative viewpoint executes events for simulation objects (SOs) once it can be assured that an SO will get no other event with an earlier timestamp. Conservative approaches restrict how SOs may interact. SOs can only interact with other SOs as specified by connectivity rules established during the simulation's initialization.

The most general approach in the conservative domain is fixed time buckets [2]. Fixed time buckets (Figure 1) permit events to be scheduled and executed asynchronously by allowing an SO to schedule events in other simulation objects. This process only occurs tighter in time than the global lookahead (L) of the simulation. For instance, if an SO is at time T_A, then the speediest it can book an event for another SO is at $T_A + L_A$ (Figure 1), where L_A is the respective lookahead of the simulation.

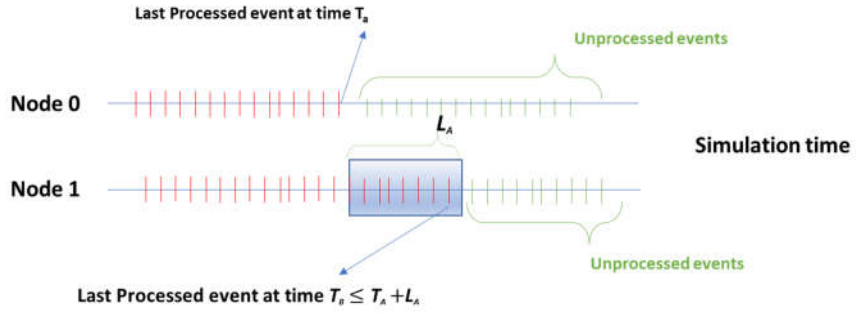

Figure 1. Fixed time buckets allow events to be scheduled and processed asynchronously using the concept of a global lookahead.

1.1.2. Optimistic Viewpoint

The optimistic viewpoint uses a unique approach for attaining parallelism by determinedly executing events but sometimes without considering causal accuracy. Rollback is employed to invalidate events that might have been executed when straggler event messages are accepted from other elements of the simulation system. Therefore, events are executed optimistically without the anticipation of rollback. This optimistic viewpoint has no limitations on how SOs intermingle; however, the disadvantage is that simulations must be built in a rollbackable style.

There are several schemes developed to implement an optimistic viewpoint. The most utilized ones are time warp (TW), breathing time buckets (BTB), and breathing time warp (BTW) [2,17–19].

Time Warp (TW)

The TW event management delivers a well-organized rollback procedure for each simulation object (SO). Each SO has a simulation clock that advances with the timestamp of its executed events. When a SO receives a straggler event, it rolls the SO back. The SO is rolled back until its last executed event before executing more events. If an event was rolled back, it needs to be reprocessed to continue the simulation.

TW only rollbacks the affected events when the SO receives a straggler message. The control structure must retract the events that were scheduled by rollbacked events. Each event must maintain a record of its created events until the event is consigned. Antimessages is the name given to messages used to withdraw wrongly scheduled event messages [2].

A fundamental concept in optimistic time management is the global virtual time (GVT). GVT approves when an event can be committed. Events with timestamps less than GVT are considered appropriately processed and will not be rolled back. The objective is to revise GVT across the simulation as frequently as possible without affecting the efficacy of the simulation due to extreme levels of synchronization. The best performance is on authentic parallel machines with shared memory and high-speed connections (Figure 2).

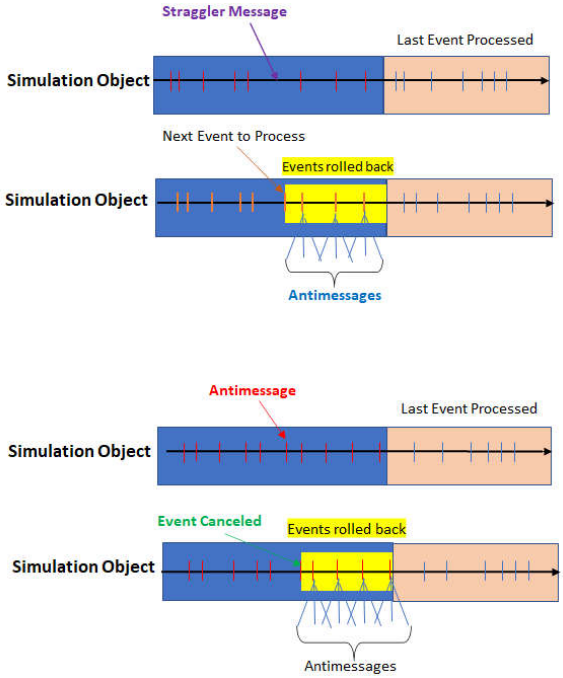

Figure 2. The implementation of rollback produced by straggler messages and antimessages in time warp (TW).

When a SO gets a straggler, message triggers rollbacks. TW rolls back each null event and deals with the straggler event. When an event is rolled back, this can cause antimessages to be generated for other events, which leads to more rollbacks and antimessages.

Breathing Time Buckets (BTB)

TW and fixed time buckets contribute to BTB [2]. The messages created while executing events are not sent pending, acknowledging that the event creating the messages will not be included in the rollback process. BTB is a mix, as explained below:

- BTB is TW without the scheme of using antimessages.
- BTB deals with events in the same style as fixed time buckets. The difference is that the size of the cycles is not predetermined.

The concept of the event horizon is essential in BTB [17–20]. The event horizon is defined as the specific time where events created turn back. All new events created at the last bucket are organized and combined into the event queue at the event horizon (Figure 3). This process is fundamental to exploit. The calculation of the global event horizon is essential to avoid problems with other SOs. The nodes are prepared to synchronize when they have executed events up to their local event horizon. Next, we can calculate GVT as the minimum local event horizon from all the nodes and commit events with timestamps less than or equal to GVT.

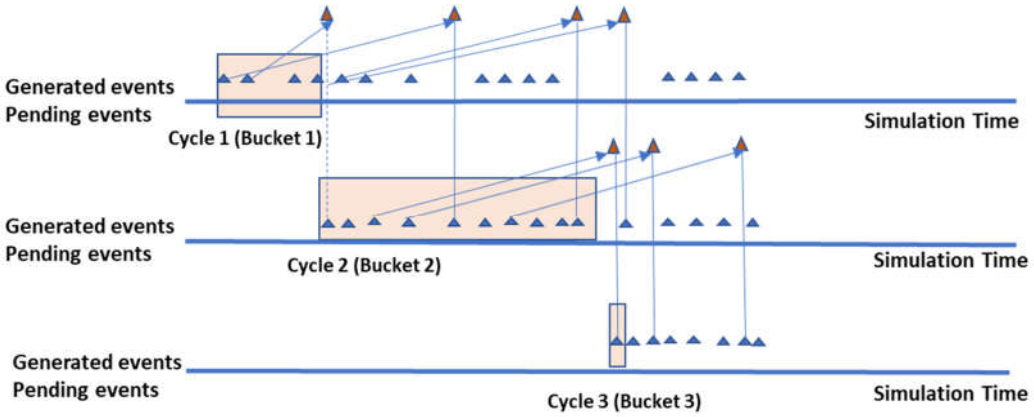

Figure 3. The event horizon for a single node and the insertion of events on the list.

A probable difficulty is that a number of nodes may have executed events that went further than GVT. Rollback, in this particular case, comprises removing messages that were produced but not sent by the specific event and subsequently returning the SO to the state before the event modified it.

Breathing Time Warp (BTW)

Breathing time warp is another optimistic hybrid scheme [17–20]. BTW attempts to fix the drawbacks with BTB and TW. TW has the possible problem of antimessage explosions and the corresponding increase in rollbacks. BTB has the possibility of a higher occurrence of synchronizations and reduced parallelism.

When events are close to the current GVT, cascading antimessage explosion can occur. The cause of this explosion is that events being executed far ahead in the simulation time of the rest will probably be rolled back. A potential solution is for those runaway events not to send their messages right away. Furthermore, using TW as the first step and then using BTB later reduces the occurrence of synchronizations and widens the bucket. The cycle is described below in five nodes (Figure 4):

1. TW phase: This phase starts with TW. There is a crucial flow parameter to fine-tune called N_{risk}. "N_{risk} is the number of events processed beyond GVT by each node" (locally) "that are allowed to send their messages with risk" [21].
2. BTB phase: At the end of the TW phase, messages are held back, and the BTB phase starts execution.
3. Computing GVT: At the end of the BTB phase, computing GVT is performed. There are two other crucial flow parameters to fine-tune called N_{gvt} and N_{opt}. "N_{gvt} is the number of messages received by each node before requesting a GVT update" [21]. On the other hand, "N_{opt} is the number of events allowed to be processed on each node beyond GVT" [21]. Therefore, Ngvt and N_{opt} control when GVT is calculated.
4. Committed Events: The events that are executed before GVT is committed.

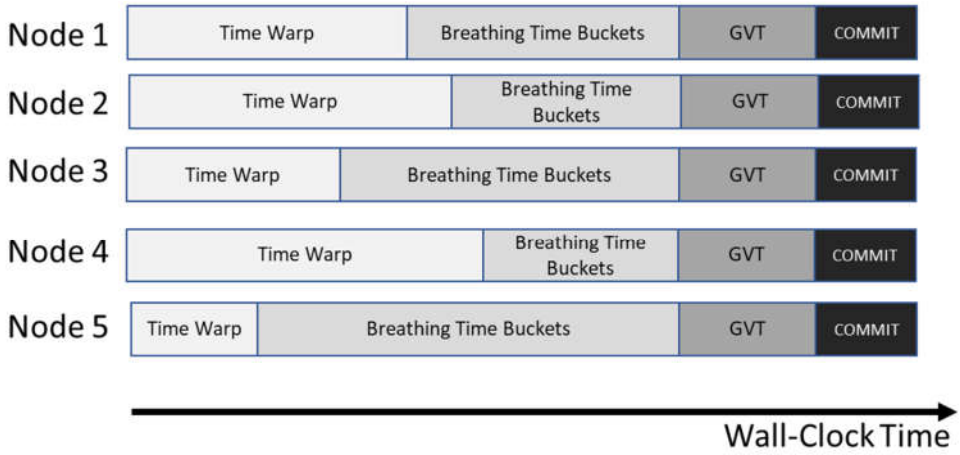

Figure 4. Example of the breathing time warp (BTW) event-processing cycle with a TW phase, a breathing time buckets (BTB) phase, computing of global virtual time (GVT), and the corresponding commitment of events in five nodes.

1.2. Problem Statement

Discrete-event simulation on parallel and distributed processors is very different from the single processor scheme, as realized in the traditional and commercial programs. As explained above, techniques such as BTB, BTW, and TW have been developed to implement optimistic time synchronization schemes, each with its respective strengths and weaknesses in PDDES [2]. However, there is no mechanism or efficient rules to decide a priori the best approach at a given simulation problem with the respective hardware, software, and network infrastructure in order to optimize a desired performance measure. Therefore, we introduce deep belief networks (DBNs) as a mechanism to decide a priori the best approach in Section 2. Section 3 describes the validation and variations of the DBN implementation built for this research. Section 4 illustrates the selection of the PDDES environment (WarpIV). Section 5 introduced programming in WarpIV using a case study and speedup (with its relationship to wall-clock time for these preliminary studies—other performance measures are possible, but this study just places emphasis on wall-clock time). Section 6 introduces the measure of complexity utilized to characterize a simulation computer program. The results of the DBN to map a PDDES environment to a synchronization scheme is explained in Section 7. Finally, we provide conclusions and further research in Section 8.

2. Deep Belief Networks

Hinton and Salakhutdinov [22] began deep learning in 2006 and contributed to a new movement in neural networks. Deep learning is self-learning by constructing a model with several layers and training it with data. This nature of multiple layers can improve the accuracy of the classification. These multiple levels of representation can provide complex mappings [23,24]. This paper studies the capabilities of deep belief network (DBN) for mapping the characteristics of the PDDES to an optimistic synchronization scheme in PDDES.

A deep belief network (DBN) is a machine learning assembly (deep) arranged of a stack of many restricted Boltzmann machines (RBMs) [25,26]. The visible layer of the DBN is the first visible layer of an RBM, while all other layers are hidden DBN layers. The hidden neurons are not connected between them; therefore, they are conditionally independent. To train a DBN, you must train a single RBM at a time. The "input layer is used to train the connection weights between the two layers", while the output layer is used to build the input of the next RBM [24]. The hidden layers of a DBN are unsupervised and act as feature detectors. These unsupervised layers can be useful by detecting features in the PDDES software and hardware and then with the supervised layer, creating the relationships between features and the synchronization schemes. DBNs have successfully created

mappings in challenging problems such as traffic flow prediction, electroencephalography, and natural language understanding [26–30]. The work presented in this paper is the first attempt to use DBNs to help design PDDES.

The learning mechanism in DBNs starts with the RBMs and their respective energy function. An energy function based on the connection weights and individual unit biases is used to define the probability distribution over the joint states of the neurons. For binary RBMs, the energy of the joint configuration of visible and hidden neurons is provided by:

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

where $\theta = (\boldsymbol{w}, \boldsymbol{b}, \boldsymbol{a})$ and $\mathbf{v} = (v_i)$ with $\mathbf{h} = (h_j)$ are the visible and hidden neurons. Variables b_i and a_j are the bias terms while w_{ij} is the weight between neurons *i* and *j* [31,32].

The following equation calculates the probability assigned to every possible pair of a visible vector **v**:

$$p(\mathbf{v}; \theta) = \frac{\sum_{\boldsymbol{h}} e^{-\mathrm{E}(\mathbf{v}, \mathbf{h}; \theta)}}{\sum_{\mathbf{v}} \sum_{\mathbf{h}} e^{-\mathrm{E}(\mathbf{v}, \mathbf{h}; \theta)}} \tag{2}$$

This vector is the partial derivative of the log-likelihood probability of a training vector for the neuron's weights

$$\frac{\partial \log [\mathrm{p}(\mathbf{v})]}{\partial \boldsymbol{w}} = \Delta w_{ij} = \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model} \tag{3}$$

Therefore, the learning rule (i.e., updating of the weights) for stochastic steepest ascent in the log probability of the training dataset is given by:

$$\Delta \boldsymbol{w_{ij}} = \boldsymbol{\varepsilon}\left(\langle \boldsymbol{v_i h_j} \rangle_{\boldsymbol{data}} - \langle \boldsymbol{v_i h_j} \rangle_{\boldsymbol{model}}\right) \tag{4}$$

where **ε** is the learning rate.

The individual activation probabilities are defined by:

$$p(v_i = 1 | \mathbf{h}; \theta) = \boldsymbol{\sigma}\left(\sum_{j=1}^{J} w_{ij}\, h_j + a_i\right) \tag{5}$$

where $\sigma(\lambda)$ = 1/(1 + e^(−λ)) is a sigmoid function [22,24]. Correspondingly, for training input **v** randomly selected, the binary state h_j of each hidden neuron *j* is set to 1 with a probability provided by:

$$p\left(h_j = 1 | \mathbf{v}; \theta\right) = \boldsymbol{\sigma}\left(\sum_{i=1}^{I} w_{ij}\, v_i + b_j\right) \tag{6}$$

Real-valued data are more naturally modeled by using a Gaussian–Bernoulli RBM (GRBM) with an energy function of the form:

$$\mathrm{E}(\mathbf{v}, \mathbf{h}; \theta) = -\sum_{i=1}^{I}\sum_{j=1}^{J} w_{ij}\, v_i h_j - \frac{1}{2}\sum_{i=1}^{I} (v_i - b_i)^2 - \sum_{j=1}^{J} a_j h_j \tag{7}$$

RBMs represent probability distributions after being trained. They assign a probability to every possible input-data vector using the energy function.

Real-valued GRBMs have a conditional probability for h_j = 1, a hidden variable turned on, given the evidence vector **v** of the form:

$$p\left(h_j = 1 | \mathbf{v}; \theta\right) = \boldsymbol{\sigma}\left(\sum_{i=1}^{I} w_{ij}\, v_i + b_j\right) \tag{8}$$

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

$$p(v_i|\mathbf{h};\theta) = \mathcal{N}\left(\sum_{j=1}^{J} w_{ij}\, h_j + a_i, 1\right) \quad (9)$$

where $\mathcal{N}(\mu_i, 1) = \frac{e^{-\frac{(v_i-\mu_i)^2}{2}}}{\sqrt{2\pi}}$ is a Gaussian distribution with the mean calculated by $\mu_i = \sum_{j=1}^{J} w_{ij}\, h_j + a_i$ and a variance unity [22,24].

3. Validation and Variations of the Implementation of Deep Belief Networks (DBNs)

The validation of the DBN software was built using MATLAB and was performed using standard benchmark pattern classification data from the MNIST handwritten digits database [33,34]. The MNIST database of handwritten digits has a training set of 60,000 examples and a test set of 10,000 examples. The digits were normalized in size and centered on a 28 × 28-pixel size image (Figure 5). The MNIST handwritten digits database has become a good database for researchers who want to test software that implements artificial intelligence algorithms on real-world data while spending minimal effort on pre-processing and formatting. Several architectures were developed using a different number of layers and neurons. The performance in the testing set achieved was more significant than 98% accuracy, which corresponds with the values reported by other researchers [35]. The software was also validated by sharing the source code and the results with the MNIST database with the research group of the creator of DBNs (Geoffrey Hinton) at the University of Toronto.

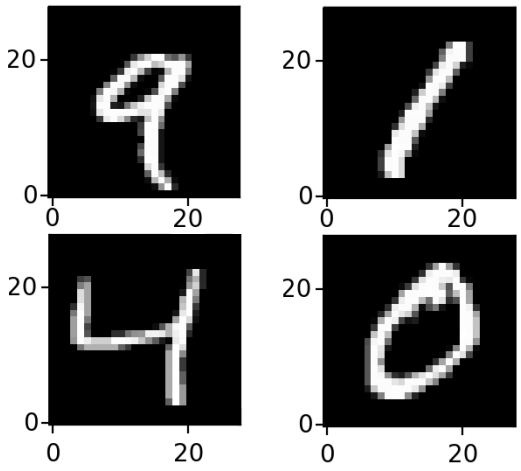

Figure 5. Example of handwritten digits from the MNIST handwritten digits database.

Additionally, the DBN software developed was modified to perform signal processing using NASA Space Shuttle data, a very well-known anomaly detection problem [36]. This DBN developed uses stochastic Bernoulli restricted Boltzmann machines in the implementation of DBN. Furthermore, this implementation 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.

The method devised for anomaly detection 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 6 shows the detection process.

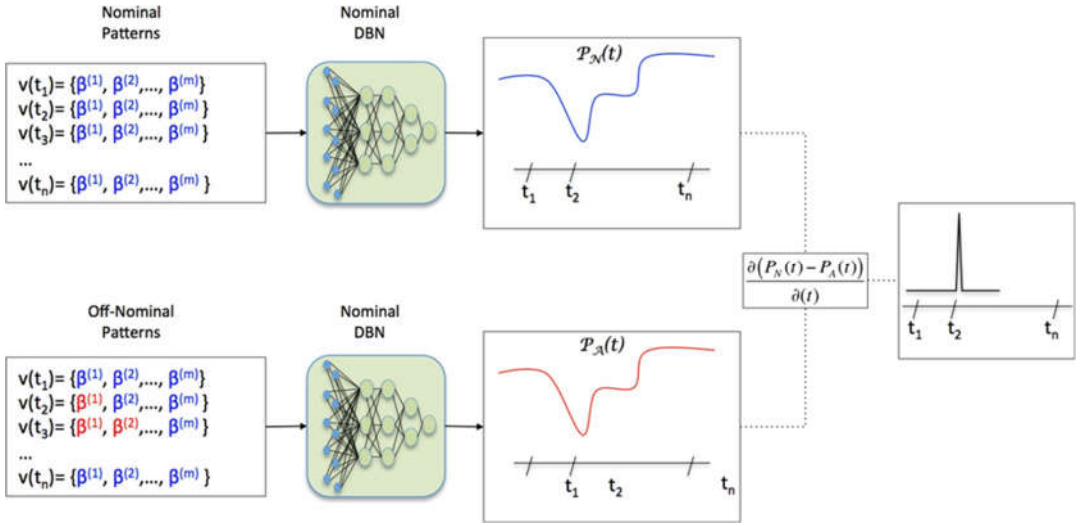

Figure 6. Detection by comparison of signals using nominal patterns as the basis to contrast with off-nominal patterns.

This case trained a deep belief neural network with six nominal temperature instrumentation signals, 2 transducers per the main engine. All data were normalized, therefore the data set had a magnitude range of [0, 1]. Six nominal instrumentation temperature signals collected from five independent space shuttle missions (i.e., flights) were used for training. Table 1 summarizes telemetry variables, missions, and timeframes used.

Table 1. Datasets of different shuttle flights (telemetry data from the three main engines) for training.

Flight Number	Flight Date	Start GMT	End GMT	TCID
133	24 February 2011	124700	150000	SA133B
132	14 May 2010	090000	111000	SA132B
131	4 May 2010	010627	045800	SA131A
128	28 August 2009	185712	210000	SA128B
126	14 October 2008	154210	190000	SA126A
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 ($\beta^{(5)}$, $\beta^{(6)}$).				

The structure of the DBN for this test case used three hidden layers, one input layer and one output layer. During the deep learning processes, the cross-entropy error was investigated at every epoch in each hidden layer to ensure a decreasing trend at every epoch. Many iterations of deep learning were executed by systematically changing combinations of hidden neurons at each layer and the epochs. This process employed ANOVA as a 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 produced an acceptable nominal DBN model. The final DBN parameters are listed in Table 2.

Table 2. Deep belief network (DBN) architecture and elements of the neurodynamics were built for the case study of the space shuttle.

Learning Rate	Hidden Layer 1 Neurons	Hidden Layer 2 Neurons	Hidden Layer 3 Neurons	Number Output Neurons	RBM Mini-Batch Size	RBM Epochs	DBN Mini-Batch Size	DBN Epochs	Weight Cost	Momentum
10^{-6}	30	20	10	1	50	50	50	1	0.01	0.5

The neuron activation probability at the output of the DBN was tested with data from a different shuttle flight (the STS-135). When the DBN was presented with the STS-135 data as its visible neurons, neuron activation propagated through the neural network until the output was reached. The prediction was performed with 100% accuracy. This case study illustrated the feasibility of using the output of a deep belief network as a possible detector of off-nominal patterns under certain specific restrictions and conditions described in this section. Additionally, it showed that our implementation was excellent.

4. Selection of a Parallel and Distributed Discrete-Event Simulation (PDDES) Platform

We selected a PDDES platform to implement the different distributed simulation designs and get the corresponding results for the time and synchronization management schemes. We studied several parallel and distributed discrete-event simulation (PDDES) engines. Several PDDES platforms were reviewed during our efforts, and they are listed as follows:

- Rensselaer's optimistic simulation system (ROSS) [37,38];
- Georgia tech time warp (GTW) [2];
- Synchronous parallel environment for emulation and discrete-event simulation (SPEEDES) [2];
- WarpIV engine [16].

The listed parallel processing computing engines can implement high-performance parallel simulation executives for discrete-event simulation applications with their own recommended compilers. Due to the superior features, we decided to use WarpIV in this research.

4.1. WarpIV Engine

WarpIV kernel can perform discrete-event simulations upon parallel and distributed settings [16,39,40]. The Warp engine can perform heterogeneous network applications using a high-speed arrangement, which mixes shared memory with standard protocols. This integration can also offer high bandwidth.

The modeling constructs and the time management schemes provided with the WarpIV engine kernels offer optimistic time mechanisms (e.g., TW, BTB, BTW). It also facilitates the component-based and interoperability modeling paradigm for simulation model reusability. WarpIV uses memory management caching techniques.

The simulation modeler can use scheduling methods. On the other hand, this simulation kernel allows for arbitrary arguments to be specified through the event interface construct. It uses C and C++ languages.

WarpIV engine has unique features. One of the main differences is the division between simulation objects and logical processes. Simulation objects inherit from the class logical process. Logical processes (LPs) are automatically distributed during startup to different nodes in several styles (e.g., block, scatter, user-defined).

4.2. Advantages of WarpIV

The Warp engine provides the resources for scheduling event processing in sequential, parallel, and distributed settings. These resources have the following advantages:

- It features state-of-the-art conservative, optimistic, and sequential time management modes.
- It distributes models and simulation objects automatically across multiple processors (even using the Internet) while handling event processing in logical time.
- It offers an excellent interface.
- It is updated to the latest operating systems, network connections, and extensions built to enhance functionality.
- It supports interoperability and reusability.
- There are training courses and support available.

5. Programming in Warp IV

We provide an example of the case studies used to develop the training database for this research, as depicted in Figure 7. This range detection implements a parallel distributed discrete-event simulation. It models the interactions of several aircraft and radars.

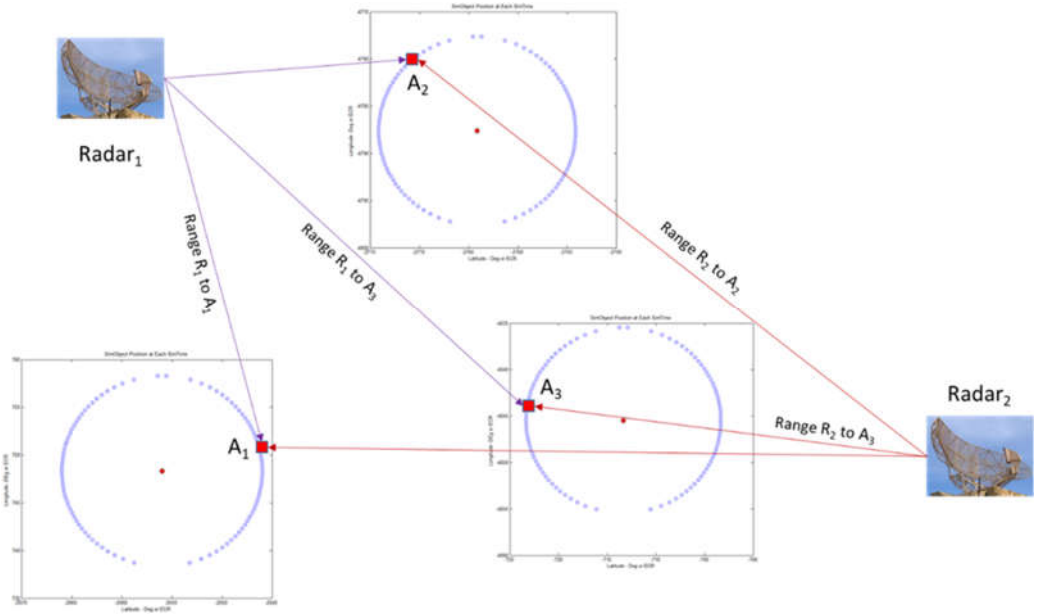

Figure 7. Simulation scenario (case study) using two classes of simulations objects (SOs) with their respective events and trajectories. These SOs are radars and aircraft.

These are the general features of its implementation:

1. It is a discrete-event simulation program (with capabilities to be executed in parallel/distributed computing environments). WarpIV provides a rollbackable version of the standard template library (STL) to accommodate mainstream C++ programmers [16,41]. Therefore, the programming is built using C/C++.
2. Time is in seconds for the simulation clock.
3. There are two (2) types of simulation objects (SOs):
    a. Aircraft.
    b. Ground Radars.

There is an event *TestUpdateAttribute* that updates the trajectory of the aircraft at specific times.

The event for the radars is *Scan*. At the initial simulation time, *Scan* is scheduled, and it happens at regular intervals depending on the technical specifications of the radars. WarpIV engine has the class logical process (LP) (Figure 8). Simulation object (SO) is a regular LP class and inherits from the LP Class. The logical process manager (LPM) can have several simulation objects (SOs), 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 (for this case study is *Aircraft* and *Radar*). 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. Events inherit from the event class (*Scan* and *TestUpdateAttribute* for this case study).

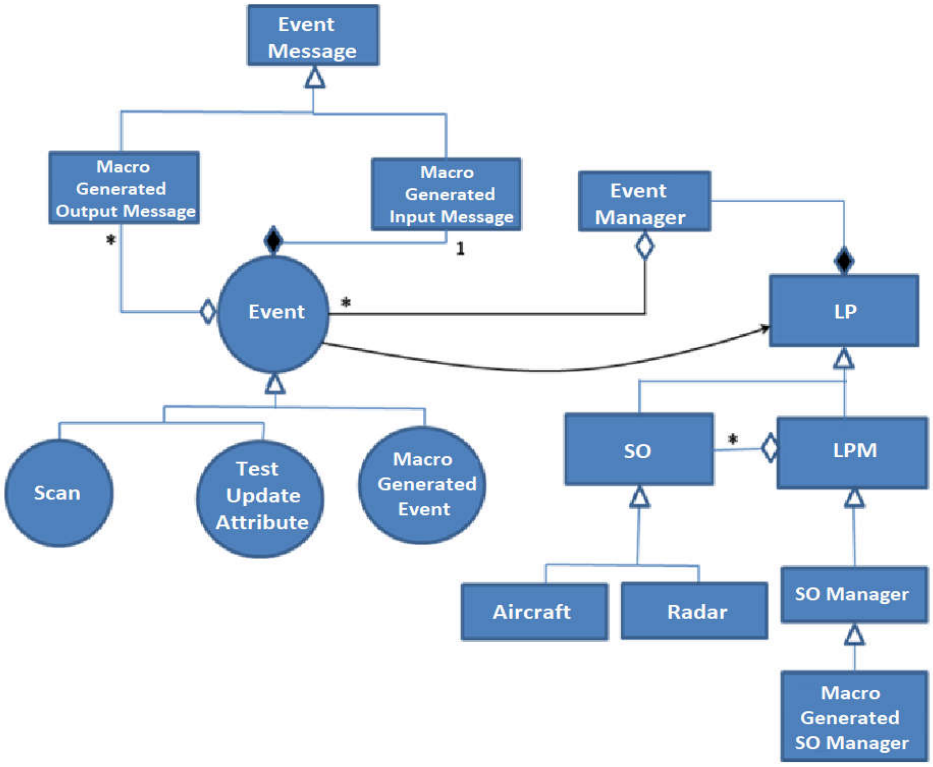

Figure 8. Unified modeling language (UML) schematics of the development with two types of simulation objects (*Aircraft* and *Radar*) and two events (i.e., *Scan* and *TestUpdateAttribute*). (The symbol * means: many).

The simulation randomly initializes the position of each aircraft and radar in the selected theater of operations. Earth-centered rotational Cartesian coordinates (ECR) represent positions (X, Y, Z). After initialization, the simulation begins with the aircraft flying in trajectories and the radars scanning the airspace to detect them using pre-established technical specifications.

4. The theater of operations is read from a file with the corresponding longitude and latitude. The speed (maximum and minimum) of the aircraft is read from a file (m/s). The range (scanning) of the radar can be read from a file or hardcoded in the program. See Figure 9 for an example of a theater of operations.
5. After the initialization routines, the simulation senses an aircraft's proximity to a radar utilizing the predefined technical specifications.
6. The *TestUpdateAttribute* event points to the method *TestUpdateAttribute*(). The event's framework scheduler kicks off this method at simulation time = zero. At each simulation time, each parallel instance (one for each aircraft) of the *TestUpdateAttribute*() method in C_RandomMotion.C (Figure 9) computes the path position of each aircraft.
7. The *Scan* event points to method *Scan()* for each radar. The event's framework scheduler kicks off this method 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 $Range = \sqrt{\Delta X^2 + \Delta Y^2 + \Delta Z^2}$, where Δ represents the difference between radar and aircraft positions (Δlatitude, Δlongitude, and Δaltitude) in earth-centered rotational coordinates (ECR).
8. The aircraft does not know the existence of the radars, but the radars can know their position.

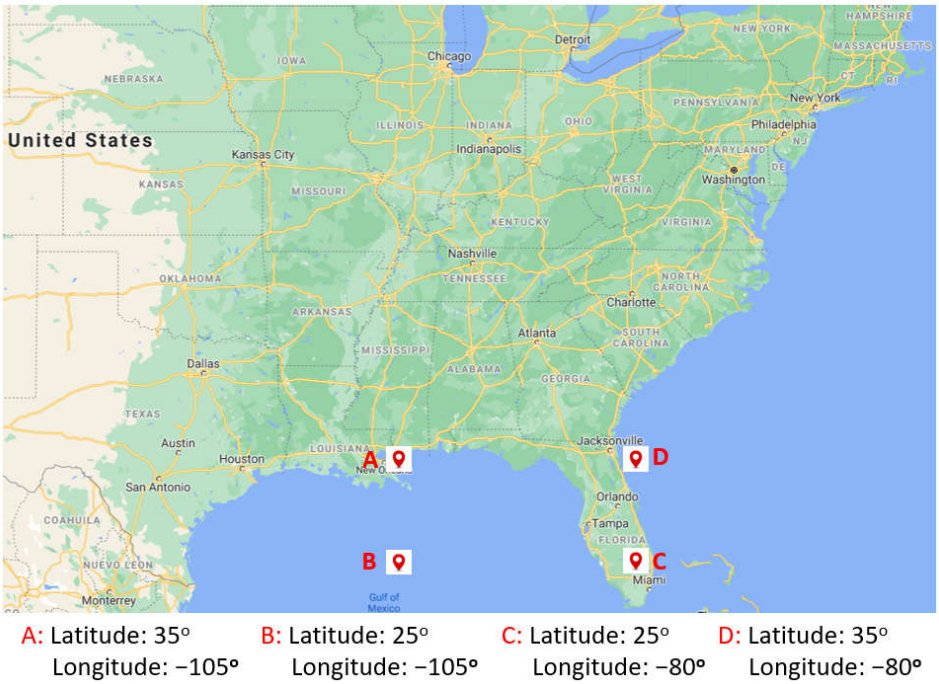

Figure 9. Example of a theater of operations as defined by the rectangle with vertices (A–D).

The aircraft detection simulation code implements each instance of aircraft 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. Figure 10 shows the different methods in the C programming language to implement the simulation model of Figures 7–9.

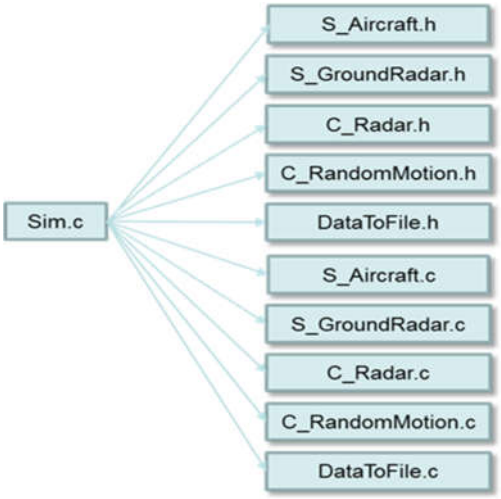

Figure 10. Different methods in the C programming language adapted to WarpIV to program the case study of Figure 7.

Now, we can execute this discrete-event simulation model in several nodes (local and global nodes, when the local and global nodes are more than 1, we have a parallel distributed discrete-event simulation system). Local nodes share the memory, and global nodes are distributed on the Internet

or a private network/cluster (Figure 11). A global node is a cluster, and a local node is a computational resource from a specific cluster. For instance, global four and local one involves four clusters. Of these clusters, each one will have a single computer (in total four nodes).

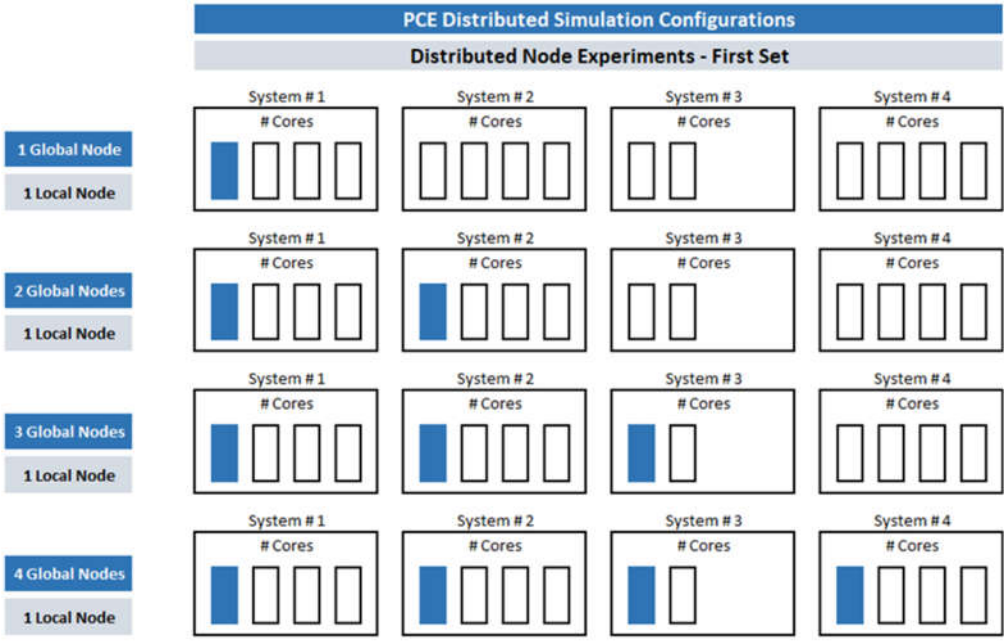

Figure 11. Examples of node configurations with cores and distributed computing elements for the experiments.

In addition, we can execute this model with the desired time synchronizations and management scheme. The following definitions are required to understand the experiments designed with the discrete-event simulation model and Warp IV:

- T (wall-clock time) is a measure of the actual time from start to finish, containing the time due to scheduled interruptions or waiting for computational assets.
- Speedup relative is the wall-clock time for a single node (sequential) divided by T (wall-clock time), considering all of the nodes used for that synchronization scheme (the wall-clock time of the node with the maximum value).

The speedup relative for the different time and synchronization management for these initial experiments is displayed in Table 3 and Figure 12. The best result of 2.9 was achieved by TW (the theoretical speedup for this problem is 3.0). BTW and TW are very comparable. BTB has better performance with multicore arrangements for this simulation case study.

Table 3. Example of simulation runs (variations) with different configurations and their respective wall-clock times for the case study depicted in Figures 7–10.

	# Nodes					
	Local	**Global**	**Wall Clock Time (s)**	**Speedup Rel**	**Speedup Theoretical**	**Server**
BTW	1	1	16.5	1	3	PC1
	1	2	14.1	1.2	3	PC1
	1	3	12.4	1.3	3	PC1
	1	4	11.4	1.4	3	PC1
	2 to 4	14	6.1	2.7	3	PC1
	4	8	6.5	2.6	3	PC1
	4	4	9.4	1.8	3	

	3	3	10.5	1.6	3	
BTB	1	1	16.1	1	3	PC1
	1	2	62.1	0.3	3	PC1
	1	3	148	0.1	3	PC1
	1	4	162.6	0.1	3	PC1
	2 to 4	14	7.7	2.1	3	PC1
	4	8	6.2	2.6	3	PC1
	4	4	9.4	1.7	3	
	3	3	10.2	1.6	3	
TW	1	1	17.2	1	3	PC1
	1	2	13.8	1.2	3	PC1
	1	3	12.6	1.4	3	PC1
	1	4	10.9	1.6	3	PC1
	2 to 4	14	5.9	2.9	3	PC1
	4	8	6.2	2.8	3	PC1
	4	4	10	1.7	3	
	3	3	11.4	1.5	3	

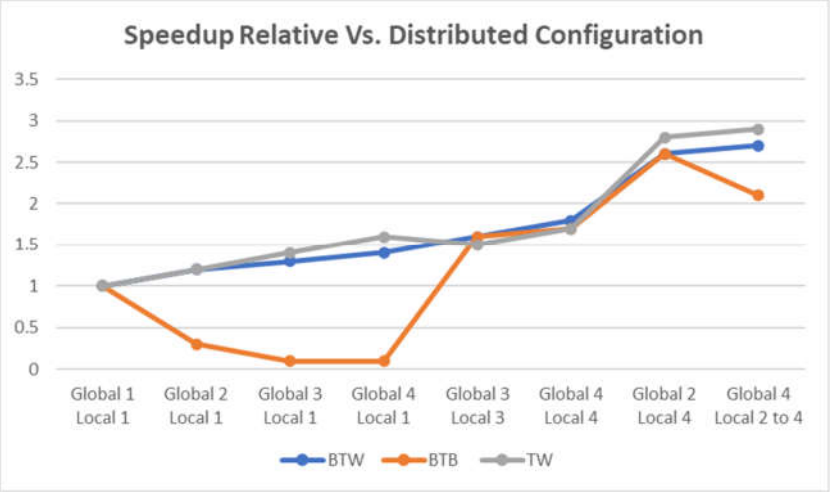

Figure 12. Speedup chart for different time and synchronization schemes (BTW, BTB, and TW) for the distributed configurations. It is essential to observe the differences in performance due to the configuration and the time and synchronization scheme for the case study—this graph will be different for other performance measures.

6. Measuring the Complexity of a Parallel Distributed Discrete-Event Simulation Implementation

Measuring simulation algorithm/software complexity is challenging. Shao and Wang [42] and Misra [43] investigated the complexity of software using the viewpoint of software due to creative activities. We are using the cognitive weights of basic control structures to measure simulation software complexity. Software constructs, such as loops, conditional statements, are assigned a weight value. The cognitive weights are as follows: a sequence is weighted with a factor of one, if-then-else with two, case statements with a three, a for-loop with three, repeat-until with three, a function call with a factor of two, parallel structures with a factor of four, and the interrupts for synchronization with a four. Figure 12 shows an example of the weight's calculations for a program in WarpIV.

The total cognitive weight of a computer program is calculated by applying the following equation where q is the total number of "main" constructs and m and n are the nested constructs with their specific cognitive weight (W_c):

$$W_c = \sum_{j=1}^{q} \left[\prod_{k=1}^{m} \sum_{i=1}^{n} W_c(j,k,i) \right] \quad (10)$$

Cognitive weights are one of the inputs to the DBN. Table 4 provides the cognitive weights for the implementation of the model represented in Figure 7. Additionally, we have captured other parameters that define the settings of the parallel distributed DES problems such as hardware, messaging, network, simulation objects, and classes, as explained as follows (21 inputs):

1. Cognitive weights of the software;
2. Simulation objects;
3. Classes of simulation objects;
4. Mean of events;
5. Standard deviation of events;
6. Mean of cognitive weights used by simulation objects;
7. Standard deviation of the cognitive weights;
8. Global nodes;
9. Mean of the local nodes per global node;
10. Standard deviation of the local nodes;
11. Mean number of cores/threads utilized;
12. Standard deviation of the number of cores/threads utilized;
13. Mean of the CPUs' speed;
14. Standard deviation of the CPUs' speed;
15. Mean of the memory size;
16. Standard deviation of the memory size;
17. Critical path;
18. Theoretical (maximum) speedup;
19. Ratio of local events divided by the local and global events ;
20. Ratio of the number of subscriber objects divided by publishers and subscribers;
21. Scatter or block distribution.

Table 4. Calculation of cognitive weights for the case study.

		Cognitive Weights	
C_Radar.C		3	
	C_Radar::Init()	41	
	C_Radar::Terminate()	13	
	C_Radar::DiscoverFo	5	
	C_Radar::RemoveFo	5	
	C_Radar::UpdateFoAttributes	5	
	C_Radar::ReflectFoAttributes	5	
	C_Radar::Scan()	2547	**<-Event**
C_RandomMotion.C		5	
	C_RandomMotion::Init	83	
	C_RandomMotion::Terminate	7	
	C_RandomMotion::TestUpdateAttribute	144	**<-Event**
	C_RandomMotion::RabeloCircle	16	
S_AirCraft.C		0	
	S_AirCraft::Init()	9	
	S_AirCraft::Terminate()	7	
S_GroundRadar.C		0	

	S_GroundRadar::Init()	0	
	S_GroundRadar::Terminate()	7	
Sim.c			
	main	17	
	Total Program Weights	**2919**	

The output vector has three components that correspond to the best scheme for that specific input vector. In our case, the one with the best speedup performance (as confirmed by executing the simulation model in WarpIV (Figure 13) and the best wall-clock time). For example, if the best performance is for TW, then, the output vector is 1 for TW, 0 for BTW, and 0 for BTB.

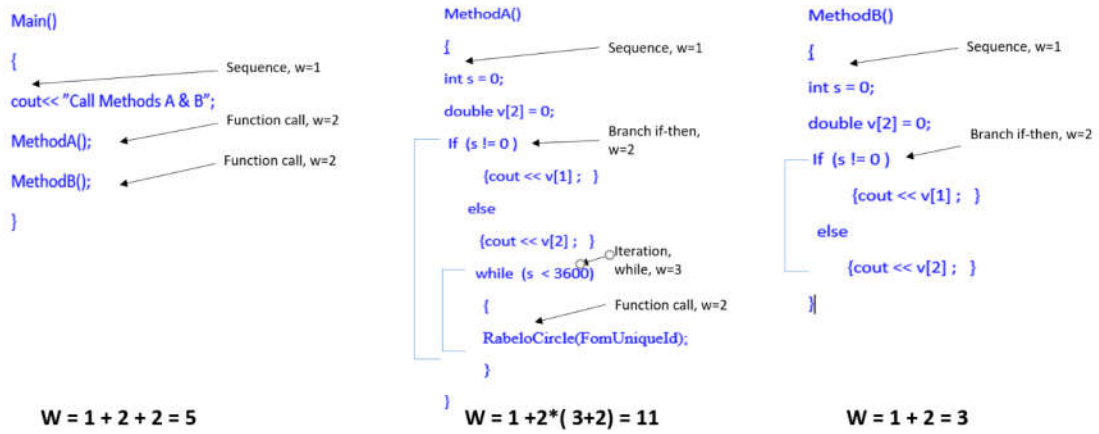

Figure 13. Calculation of the cognitive weights for a program.

7. Results

The performance criterion is the minimum wall-clock time, and this indicates the synchronization scheme with the best level of speedup achieved. The wall-clock time tells us the time it takes for the computer system to finish the simulation. It is the time to the solution: the number of seconds of wall-clock time to satisfy the termination criterion of detecting the aircraft in this case. Many research initiatives have used speedup and its relationship with the wall-clock time as a performance measure [44–47]. Table 5 indicates the training vector for the case study of Figure 7, with four global nodes and one local node using block as the distribution policy, with TW as the best performance (best wall-clock time). It is essential to say that if this case study is implemented using three global nodes and three local nodes using block as the distribution policy, then BTB is the synchronization scheme with the best performance (minimum wall-clock time).

Table 5. Example of a vector that defines the parallel distributed discrete-event simulation (PDDES) implementation for the aircraft detection model of Figure 7 with 4 global nodes and 1 local node using block as the distribution policy, with TW as the best performance (best wall-clock time).

Inputs (21 Input Neurons)	
Total Simulation Program Cognitive Weights	2919
Number of Sim objects	6
Types of Sim objects	3
Mean Events per Object	1
STD Events per Simulation Object	0
Mean Cog Weights of All objects	1345
STD Cog Weights of All objects	1317
Number of Global Nodes	4
Mean Local Nodes per Computer	1

STD Local Nodes per Computer	0
Mean number of cores	1
STD Number of cores	0
Mean processor Speed	2.1
STD processor Speed	0.5
Mean RAM	6.5
STD RAM	1.9
Critical Path%	0.32
Theoretical Speedup	3
Local Events/(Local Events + External Events)	1
Subscribers/(Publishers + Subscribers)	0.5
Block or Scatter?	1
Outputs (3 output neurons)	
BTB	0
BTW	0
TW	1

Another point is that a great deal of data is needed, so numerous problems were selected to generate case studies and variations of hardware/simulation objects/nodes to train the DBN. Two hundred and forty case studies and their variations were selected for training, sixty case studies and variations for validation, the right number of neurons and hidden layers, and one hundred for testing. The variations were produced with changes in the number of global and local nodes. The training session for a DBN was accomplished. Figure 14 shows the details of the training of the DBN. The best architecture had three hidden layers with 21 inputs, 50 neurons in each hidden layer (three-hidden layers), and one output layer with three neurons (one for each time and synchronization management scheme).

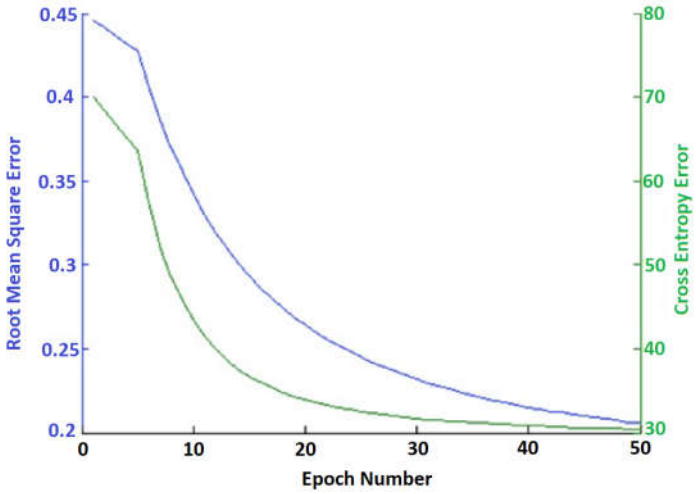

Figure 14. Root mean square error and cross-entropy error—training curve for the DBN developed with 21 inputs, 50 neurons in the first hidden layer, 50 neurons in the second hidden layer, 50 neurons in the third hidden layer, and 3 output neurons.

This DBN has the testing performance that is shown in Figure 15. Preliminary datasets were utilized with the multi-layer perceptron (backpropagation) [48]; however, the performance obtained was lower than 60%.

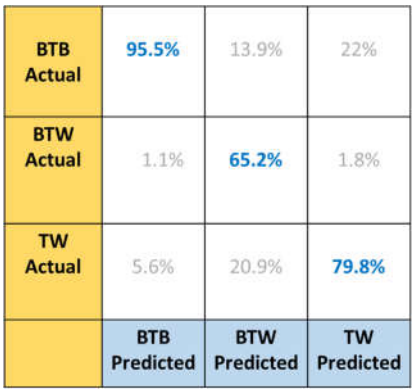

Figure 15. The testing performance of the DBNs built using 21 inputs, 50 neurons in the first hidden layer, 50 neurons in the second hidden layer, 50 neurons in the third hidden layer, and 3 output neurons.

The performance of the DBN can be increased using more case studies. The study demonstrated the feasibility of the new technique, which can be used to design parallel distributed discrete-event simulation configurations. This first effort places emphasis on speedup.

8. Conclusions and Further Research

The research work presented here implemented a decision-making scheme that, based on the simulation environment (software, hardware, and simulation logic), can identify the best synchronization and time management to perform a specific parallel and distributed DES. This new approach is original, pioneering, and uses deep learning. This development has the potential to save time on experimentation and provide better designs. The prototype developed in this research work can be improved and give a better performance with more case studies and even use recently developed deep learning algorithms that are more powerful. The method presented in this paper is straightforward and automatically selects the correct scheme (TW, BTW, BTB). Of course, it can be extended to more schemes, and it can continue learning with new case studies and using parallel distributed simulation repositories.

This study contributes to a new approach to an existing problem that is very complex. We recognize that PDDES is critical for the current trends in simulation and hardware/software developments. There were limitations to this research. This study was a preliminary effort; therefore, more case studies can be added to improve performance. Another point is that we focus on the DBN, and there is potential to use other types of deep learning, such as modified convolutional neural networks (CNNs) [49] and adversarial networks [50]. Modified CNNs and adversarial networks recently have been gaining attention as deep neural networks with the best performance. Another limitation was the utilization of only popular optimistic synchronization schemes. There is the potential to use other newer optimistic synchronization schemes and study load balancing among nodes [46,47]. The speedup based on the best wall-clock time was the only performance measure studied. It is essential to study more performance measures.

There are several issues that we will start exploring, and this approach may contribute. For example, cloud computing [4,6–9,11], the World Wide Web of simulation [3,5,7,10], and autonomic computing (AC) [6]. We can use our approach to design simulators/configurations for these platforms. Nevertheless, the input will have to be modified to characterize cloud computing, web-based elements and policies (e.g., on-demand service model of simulation resources, high-level architecture (HLA) support). This area of research is required due to cloud computing being recognized as the new dominant environment for enterprise IT. Across industries, cloud computing persists to be one of the fastest-growing areas.

Author Contributions: Conceptualization, E.C., L.R.; investigation, E.C., L.R., A.T.S., and E.G writing—review and editing E.C., L.R., A.T.S., and E.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Borshchev, A. *The Big Book of Simulation Modeling: Multimethod Modeling with AnyLogic 6*; AnyLogic North America: Chicago, IL, USA, 2013.
2. Fujimoto, R. *Parallel and Distributed Simulation*, 1st ed.; John Wiley & Sons: New York, NY, USA, 2000.
3. Page, E.; Buss, A.; Fishwick, P.; Healy, K.; Nance, R.; Paul, R. Web-based Simulation: Revolution or Evolution? *ACM Trans. Modeling Comput. Simul. (TOMACS)* **2000**, *10*, 3–17.
4. Fujimoto, R.; Malik, A.; Park, A. Parallel and distributed Simulation in the cloud. *SCS Modeling Simul. Mag.* **2010**, *3*, 1–10.
5. Jávor, A.; Fur, A. Simulation on the Web with distributed models and intelligent agents. *Simulation* **2012**, *88*, 1080–1092.
6. Amoretti, M.; Zanichelli, F.; Conte, G. Efficient autonomic cloud computing using online discrete event simulation. *J. Parallel Distrib. Comput.* **2013**, *73*, 767–776.
7. Jafer, S.; Liu, Q.; Wainer, G. Synchronization methods in Parallel and distributed discrete-event Simulation. *Simul. Model. Pract. Theory* **2013**, *30*, 54–73.
8. Padilla, J.; Diallo, S.; Barraco, A.; Lynch, C.; Kavak, H. Cloud-based simulators: Making simulations accessible to non-experts and experts alike. In Proceedings of the 2014 Winter Simulation Conference 2014, Savannah, GA, USA, 7–10 December 2014; pp. 3630–3639.
9. Yoginath, S.; Perumalla, K. Efficient Parallel Discrete Event Simulation on cloud/virtual machine platforms. *ACM Trans. Modeling Comput. Simul. (TOMACS)* **2015**, *26*, 1–26.
10. Padilla, J.; Lynch, C.; Diallo, S.; Gore, R.; Barraco, A.; Kavak, H.; Jenkins, B. Using simulation games for teaching and learning discrete-event simulation. In Proceedings of the 2016 Winter Simulation Conference (WSC), Washington, DC, USA, 11–14 December 2016; pp. 3375–3384.
11. Liu, D.; De Grande, R.; Boukerche, A. Towards the Design of an Interoperable Multi-cloud Distributed Simulation System. In Proceedings of the 2017 Spring Simulation Multi-Conference—Annual Simulation Symposium, Virginia Beach, VA, USA, 23–26 April 2017. pp. 1–12.
12. Diallo, S.; Gore, R.; Padilla, J.; Kavak, H.; Lynch, C. Towards a World Wide Web of Simulation. *J. Def. Modeling Simul. Appl. Methodol. Technol.* **2017**, *14*, 159–170.
13. Shchur, L.; Shchur, L. Parallel Discrete Event Simulation as a Paradigm for Large Scale Modeling Experiments. In Proceedings of the XVII International Conference "Data Analytics and Management in Data Intensive Domains" (DAMDID/RCDL'2015), Obninsk, Russia, 13–16 October 2015.
14. Tang, Y.; Perumalla, K.; Fujimoto, R.; Karimabadi, H.; Driscoll, J.; Omelchenko, Y. Optimistic parallel discrete event simulations of physical systems using reverse computation. In Proceedings of the Workshop on Principles of Advanced and Distributed Simulation (PADS'05), Monterey, CA, USA, 1–3 June 2005.
15. Ziganurova, L.; Novotny, M.; Shchur, L. Model for the evolution of the time profile in optimistic parallel discrete event simulations. In Proceedings of the International Conference on Computer Simulation in Physics and Beyond, Moscow, Russia, 6–10 September 2015.
16. Steinman, J. The WarpIV Simulation Kernel. In Proceedings of the Workshop on Principles of Advanced and Distributed Simulation (PADS 2005), Monterey, CA, USA, 1–3 June 2005.
17. Steinman, J. Breathing Time Warp. In Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS93), San Diego, CA, USA, 16–19 May 1993.
18. Cortes, E.; Rabelo, L.; Lee, G. Using Deep Learning to Configure Parallel Distributed Discrete-Event Simulators. In *Artificial Intelligence: Advances in Research and Applications*, 1st ed.; Rabelo, L., Bhide, S., Gutierrez, E., Eds.; Nova Science Publishers: Hauppauge, NY, USA, 2018.
19. Steinman, J. Discrete-Event Simulation, and the Event Horizon. *ACM SIGSIM Simul. Dig.* **1994**, *24*, 39–49.
20. Steinman, J. Discrete-Event Simulation and the Event Horizon Part 2: Event List Management. *ACM SIGSIM Simul. Dig.* **1996**, *26*, 170–178.

21. Steinman, J.; Nicol, D.; Wilson, L.; Lee, C. Global Virtual Time and Distributed Synchronization. In Proceedings of the 1995 Parallel and Distributed Simulation Conference, Lake Placid, NY, USA, 14–16 June 1995.
22. Hinton, G.; Salakhutdinov, R. Reducing the dimensionality of data with neural networks. *Science* **2006**, *313*, 504–507.
23. Yu, K.; Jia, L.; Chen, Y.; Xu, W. Deep learning: Yesterday, today, and tomorrow. *J. Comput. Res. Dev.* **2013**, *50*, 1799–1804.
24. Jiang, L.; Zhou, Z.; Leung, T.; Li, T.; Fei-Fei, L. Mentornet: Learning data-driven curriculum for very deep neural networks on corrupted labels. In Proceeding of the Thirty-Fifth International Conference on Machine Learning, Stockholmsmässan, Stockholm, Sweden, 10–15 July 2018.
25. Hinton, G. A practical guide to training restricted Boltzmann machines. *Momentum* **2010**, *9*, 926.
26. Mohamed, A.; Sainath, T.; Dahl, G.; Ramabhadran, B.; Hinton, G.; Picheny, M. Deep belief networks using discriminative features for phone recognition. In Proceedings of the Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference, Prague, Czech Republic, 22–27 May 2011.
27. Mohamed, A.; Dahl, G.; Hinton, G. Acoustic modeling using deep belief networks. *IEEE Trans. Audio Speech Lang. Process.* **2012**, *20*, 14–22.
28. Huang, W.; Song, G.; Hong, G. Deep architecture for traffic flow prediction: Deep belief networks with multitask learning. *IEEE Trans. Intell. Transp. Syst.* **2014**, *15*, 2191–2201.
29. Sarikaya, R.; Hinton, G.; Deoras, A. Application of deep belief networks for natural language understanding. *IEEE/ACM Trans. Audio Speech Lang. Process.* **2014**, *22*, 778–784.
30. Movahedi, F.; Coyle, J.; Sejdić, E. Deep belief networks for electroencephalography: A review of recent contributions and future outlooks. *IEEE J. Biomed. Health Inform.* **2018**, *22*, 642–652.
31. Hinton, G.; Osindero, S.; Yee-Whye, T. A Fast Learning Algorithm for Deep Belief Nets. *Neural Comput.* **2006**, *18*, 1527–1554.
32. Cho, K.; Ilin, A.; Raiko, T. Improved learning of Gaussian-Bernoulli restricted Boltzmann machines. In *Artificial Neural Networks and Machine Learning—ICANN 2011*; Springer: Berlin Heidelberg,: Berlin, Germany, 2011; Volume 6791, pp. 10–17.
33. LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **1998**, *86*, 2278–2324.
34. LeCun, Y.; Corinna, C. THE MNIST DATABASE of Handwritten Digits. Available online: http://yann.lecun.com/exdb/mnist/ (accessed on 22 September 2020).
35. Wu, M.; Chen, L. Image Recognition Based on Deep Learning. In Proceedings of the 2015 Chinese Automation Congress (CAC), Wuhan, China, 27–29 November 2015; pp. 542–546.
36. Cortes, E.; Rabelo, L. An architecture for monitoring and anomaly detection for space systems. *SAE Int. J. Aerosp.* **2013**, *6*, 81–86.
37. Carothers, C.; Bauer, D.; Pearce, S. ROSS: A high-performance, low memory modular time warp system. *J. of Parallel Distrib. Comput.* **2002**, *62*, 1648–1669.
38. Mubarak, M.; Carothers, C.; Ross, R.; Carns, P. Using massively parallel Simulation for MPI collective communication modeling in extreme-scale networks. In Proceedings of the 2014 Winter Simulation Conference, Savannah, GA, USA, 7–10, December 2014; pp. 3107–3118.
39. Steinman, J.; Lammers, C.; Valinski, M. A Proposed Open Cognitive Architecture Framework (OpenCAF). In Proceedings of the 2009 Winter Simulation Conference, Austin, TX, USA, 13–16 December 2009.
40. Steinman, J.; Lammers, C.; Valinski, M.; Steinman, W. External Modeling Framework and the OpenUTF. Report of WarpIV Technologies. Available online: http://www.warpiv.com/Documents/Papers/EMF.pdf (accessed on 30 September 2020).
41. Plauger, P.; Stepanov, A.; Lee, M.; Musser, D. *The C++ Standard Template Library*; Prentice-Hall PTR, Prentice-Hall Inc.: Upper Saddle River, NJ, USA, 2001.
42. Shao, J.; Wang, Y. A new measure of software complexity based on cognitive weights. *Can. J. Electr. Comput. Eng.* **2003**, *28*, 69–74.
43. Misra, S. A Complexity Measure based on Cognitive Weights. *Int. J. Theor. Appl. Comput. Sci.* **2006**, *1*, 1–10.
44. Kent, E.; Hoops, S.; Mendes, P. Condor-COPASI: High-throughput computing for biochemical networks. *BMC Syst. Biol.* **2012**, *6*, 91.

45. Wang, Y.; Jung, Y.; Supinie, T.; Xue, M. A Hybrid MPI–OpenMP Parallel Algorithm and Performance Analysis for an Ensemble Square Root Filter Designed for Multiscale Observations. *J. Atmos. Ocean. Technol.* **2013**, *30*, 1382–1397.
46. Zhan, D.; Qian, J.; Cheng, Y. Balancing global and local search in parallel efficient global optimization algorithms. *J. Glob. Optim.* **2017**, *67*, 873–892.
47. Grandison, A.; Cavanagh, Y.; Lawrence, P.; Galea, E. Increasing the Simulation Performance of Large-Scale Evacuations Using Parallel Computing Techniques Based on Domain Decomposition. *Fire Technol.* **2017**, *53*, 1399–1438.
48. Rumelhart, D.; Hinton, G.; Williams, R. Learning representations by back-propagating errors. *Nature 1986.* 323, 533–536.
49. Wang, X.; Zhao, Y.; Pourpanah, F. Recent advances in deep learning. *Int. J. Mach. Learn. Cybern.* **2020**, *11*, 747–750.
50. Ren, K.; Zheng, T.; Qin, Z.; Liu, X. Adversarial Attacks and Defenses in Deep Learning. *Engineering* **2020**, *6*, 346–360.