A DEEP LEARNING AND PARALLEL SIMULATION METHODOLOGY FOR AIR TRAFFIC MANAGEMENT

A Dissertation
Presented to
The Academic Faculty

By

Young Jin Kim

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
Computational Science and Engineering

Georgia Institute of Technology

December 2017

Copyright © Young Jin Kim 2017

A DEEP LEARNING AND PARALLEL SIMULATION METHODOLOGY FOR AIR TRAFFIC MANAGEMENT

Approved by:

Dr. Dimitri Mavris, Advisor
School of Aerospace Engineering
Georgia Institute of Technology

Dr. Richard Fujimoto, Co-advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Daniel Schrage
School of Aerospace Engineering
Georgia Institute of Technology

Dr. Duen Horng Chau
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Andres Rodriguez
Artificial Intelligence Products Group
Intel Corporation

Date Approved: July 28, 2017

To my family

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Dr. Dimitri Mavris, for giving me the great opportunity to work in Aerospace Systems Design Lab where I could learn a lot of invaluable lessens. Especially, Dr. Mavris always emphasized logical thinking, reasoning and communications which will be great guidances for my entire career. It has been an honor to do my research under supervision of one of the leading researchers in this field.

I would like to thank Dr. Richard Fujimoto, for guiding me to find a path for my research. I have been deeply impressed by his passion for the research. Whenever I have a hard problem to solve, Dr. Fujimoto always kindly provided great advices and opinions on my research topic. It has been an honor to do my research under supervision of one of the leading researchers and pioneers of the distributed simulation field.

I would like to thank Dr. Daniel Schrage and Dr. Polo Chau for giving me invaluable comments and guidances for my thesis. They helped so much to enhance my insights to my research. Thanks to those guidances, the contents of the thesis have improved a lot.

I would like to thank Dr. Andres Rodriguez for giving me great advices and guidances to my research and works. It has been always my great pleasure to discuss with him regarding challenging research topics. I was really fortunate enough to finish my thesis with him and work with him.

I would like to thank people who I worked and interacted with; Dr. Simon Briceno, Dr. Woong Je Sung, Dr. Dongwook Lim, Dr. Jongki Moon, Sun Choi, Sanggyu Min, Youngchul Park. I could have a really great time in the lab thanks to them.

Finally, I would like to thank my family. Their belief and support always give me a great motivation. I would like to specially thank my wife, Ayoung. Without her supports, nothing could have been achieved. I am so lucky to have her in my life.

TABLE OF CONTENTS

Acknowledgments iv

List of Tables ix

List of Figures x

Chapter 1: Introduction 1

1.1 Background 1

1.2 Air Traffic Management 2

1.2.1 Prescriptive models 3

1.2.2 Descriptive models 4

1.3 Parallel Simulation 6

1.3.1 Computer Simulation 6

1.3.2 Parallel Discrete Event Simulation 6

1.3.3 Spatial Parallel Simulation 9

1.3.4 Time-Parallel Simulation 12

1.4 Artificial Neural Networks 13

1.4.1 Recurrent Neural Networks 14

1.5 Research Contributions 16

1.5.1 Time-Parallel Simulation of Air Traffic Networks 17

1.5.2 Combining Time and Spatial Parallelism for Air Traffic Networks . 18
1.5.3 Recurrent Neural Network Model for Airspace Applications 18
1.6 Thesis Organization 19
Chapter 2: Time-parallel simulation 20
2.1 Overview 20
2.2 Related Work 21
2.3 Modeling Techniques 22
2.3.1 Queueing network based discrete event simulation 22
2.3.2 Fluid flow modeling 23
2.3.3 System Dynamics Modeling 24
2.3.4 Agent-Based Modeling 25
2.4 Simulation Model 27
2.4.1 Simulation Model Resources 28
2.4.2 Ground Delay Programs 28
2.4.3 Event Definitions 29
2.4.4 Delay Model 30
2.4.5 Simulation Execution 32
2.5 Algorithm 33
2.5.1 Time-Parallel Simulation 33
2.5.2 Simulation Executive 35
2.5.3 Fix Up Computation 35
2.5.4 Workload Distribution 41

2.6 Experimental Results 41
2.6.1 Air Traffic Scenario and Data 41
2.6.2 Experimentation Environment 43
2.6.3 Fix Up Computation Comparison 44
2.6.4 Workload Distribution Comparison 48
2.6.5 Limited Airport Capacity Comparison 50
2.7 Conclusion 52

Chapter 3: Exploiting Spatial Parallelism in Air Traffic Network Simulation . . 53

3.1 Overview 53
3.2 Combining Time and Spatial Parallelism 54
3.2.1 Time Warp Parallel Simulation of Air Traffic Networks 54
3.2.2 Time and Spatial Parallel Simulation of the Air Traffic Network . . 64
3.3 Experimental Results 66
3.3.1 Experimentation Environment 66
3.3.2 Parallelism Analysis 67
3.4 Conclusion 78

Chapter 4: Recurrent Neural Networks for Flight Delay Prediction 79

4.1 Overview 79
4.2 Deep Recurrent Neural Networks 80
4.2.1 Vanilla Recurrent Neural Networks 80
4.2.2 LSTM 82
4.2.3 GRU 82

4.2.4 Deep architecture of RNN 84
4.3 Network Training 85
4.3.1 Day-to-day delay status model 86
4.3.2 Deep architecture for the day-to-day delay status RNN model 88
4.3.3 Individual flight delay model 88
4.3.4 Regularization 89
4.3.5 Training methods 90
4.4 Experimental Results 91
4.4.1 Accuracy measurement 92
4.4.2 Day-to-day delay status model 92
4.4.3 Individual flight delay model 99
4.4.4 Generalization of day-to-day model for different airports 100
4.5 Conclusion 101

Chapter 5: Conclusions and Future Work 102
5.1 Contributions 102
5.2 Future Research 104

Chapter A: Application Programming Interface of Simulation Software 108
A.1 Time and Space Parallel Simulation 108

References 225

LIST OF TABLES

2.1 The number of simulation rounds for the time-parallel simulation. 51

3.1 Example of domain decomposition of the NAS - 5 LPs. 57

3.2 Speed up by using single parallel algorithms. 72

3.3 Speed up by using dual parallel algorithms. 77

4.1 Characteristics of three kinds of activation functions. 81

4.2 Inputs and outputs of the day-to-day delay status model. 87

4.3 Inputs and outputs of the individual flight delay model. 90

4.4 Accuracy of day-to-day model - vanilla RNN. 93

4.5 Accuracy of day-to-day model - LSTM. 93

4.6 Accuracy of day-to-day model - GRU. 94

4.7 Accuracy of day-to-day model - different epochs. 96

4.8 Accuracy of day-to-day model - number of stacked layers. 97

4.9 Accuracy of individual flight delay models. 100

4.10 Accuracy of day-to-day model for different airports. 101

4.11 Additional training with a model from Atlanta airport - JFK airport. 101

LIST OF FIGURES

1.1 Two different parallel simulations of air traffic network. 8

1.2 Notional example of state-matching problem. 13

1.3 A neuron of artificial neural networks. 14

1.4 Three different activation functions. 15

1.5 Feed forward artificial neural networks and Recurrent neural networks. . . . 16

2.1 Air traffic environment in terms of control volumes and merge and diverge models [73]. 24

2.2 Summary of the model of the airline agent [63]. 26

2.3 Delay model of the departure. 32

2.4 Single airport event flow. 33

2.5 Two different parallel simulations of air traffic network. 34

2.6 Communication of fix up messages. 37

2.7 Evaluation of update for the fix up computation. 38

2.8 High level view of the fix up computations - sequential operation. 39

2.9 High level view of the fix up computations - gather operation. 39

2.10 *Sequential fix-up* computation vs. *Collective fix-up* computation. 45

2.11 *Sequential fix-up* computation vs. *Collective fix-up* computation with *No fix-up* computation. 45

2.12 *Sequential fix-up* computation vs. *Collective fix-up* computation. 46

2.13 *Sequential fix-up* computation vs. *Collective fix-up* computation with *No fix-up* computation. 47

2.14 Same time intervals vs. Same amount of traffic. 48

2.15 Same time intervals vs. Same amount of traffic with No fix up computation. 49

2.16 Capacity limits vs. No capacity limits with No fix up computation. 51

3.1 Air traffic flow chart [82]. 56

3.2 ARTCC map [83]. 56

3.3 Time and spatial allocation of LPs. 66

3.4 Xeon Phi 7250 architecture [84]. 68

3.5 Speed up of Time Warp simulation - small workload. 70

3.6 Speed up of Time Warp simulation - large workload. 70

3.7 Speed up of time parallel simulation- large workload. 71

3.8 Speed up of dual parallel simulation - time division: 2. 74

3.9 Efficiency of dual parallel simulation - time division: 2. 74

3.10 Expected speed up vs actual speed up - time division: 2. 75

3.11 Speed up of dual parallel simulation - time division: 4. 76

3.12 Efficiency of dual parallel simulation - time division: 4. 76

3.13 Expected speed up vs actual speed up - time division: 4. 77

4.1 Long Short-Term Memory Cell [96]. 83

4.2 Gated Recurrent Unit Cell [97]. 84

4.3 Day-to-Day departure delay status model. 87

4.4 Deep architecture for the RNN model. 88

4.5 Individual flight delay model. 89

4.6 Dropout Neural Net Model [101]. 91

4.7 Accuracy comparison for different recurrence units. 94

4.8 Accuracy changes with increasing number of epochs (until 200 epochs). . . . 95

4.9 Accuracy changes with increasing number of epochs (until 1000 epochs). . 96

4.10 Accuracy changes with increasing number of layers. 98

4.11 Actual delay vs Predicted delay (ATL). 98

4.12 Actual delay vs Predicted delay (JFK). 99

SUMMARY

Air traffic management is widely studied in several different fields because of its complexity and criticality to a variety of stakeholders including passengers, airlines, regulatory agencies, air traffic controllers, etc. However, the exploding amount of air traffic in recent years has created new challenges to ensure effective management of the airspace. A fast time simulation capability with high accuracy is essential to effectively explore the consequences of decisions from the airspace design phase to the air traffic management phase. In this thesis, two key components for enabling intelligent decision support are proposed and studied.

To accelerate fast time simulations, a time-parallel simulation approach has been studied and applied to air traffic network simulation in addition to exploitation of spatial parallel simulation. This approach splits the simulation time axis into time intervals and simulates the intervals concurrently potentially achieving a high level of parallelism. This approach requires a way to ensure that the distributed simulation takes into account dependencies across time periods. A methodology to address this issue is proposed. The proposed time-parallel algorithm works seamlessly with the spatial parallel approach. In particular, the synchronization algorithm used for the spatial parallel simulation is integrated with the time-parallel simulation algorithm. In this thesis, an efficient algorithm spanning these aspects of the distributed simulations is proposed and implemented. The implemented simulation is tested in a variety of scenarios and balances time and spatial parallelism to improve speed up.

As another aspect, to predict the future scenarios more accurate, it is necessary to feed the appropriate input vales to the simulation program. This input can be acquired by learning the previous patterns in data, statistically. Recent improvements in machine learning and artificial intelligence research enable an accurate prediction of the future state variables in the air traffic network system. Recurrent neural network is one type of algorithm which

can effectively model sequential state variables. In that sense, a recurrent neural network approach is proposed for modeling the input of each simulation scenario. By utilizing a large amount of historical flight and weather data, the proposed recurrent neural network model learns the best parameters in the model to predict the future status of the airports in the National Airspace System (NAS). In particular, airports daily capacity in the future is a key input variable for the NAS simulation model. The proposed model is trained to accurately predict the airports daily capacity.

Based on real world air traffic data, the improvements in the performance and the accuracy of both techniques have been investigated and presented. The proposed approaches show significant improvements for supporting air traffic management decision making.

CHAPTER 1
INTRODUCTION

1.1 Background

According to the Federal Aviation Administration (FAA)'s latest forecast, domestic enplanement of commercial airlines will increase by 1.5 times over the next twenty years, from 696 million to 1,052 million. Over the same time period, it is estimated that the number of passengers taking international flights into or leaving the U.S. will more than double from 206 million to 452 million [1]. This increased traffic could result in significant delays in the National Airspace System (NAS). According to one study [2], air traveler delays accounted for approximately $33 billion in direct or indirect costs to passengers, airlines and other parts of the NAS in 2007.

In addition to this increase of traditional air traffic, new types of air traffic are also expected to be exploding including Unmanned Aircraft Systems (UAS). In 2020, the forecast shows that 7 million unmanned fleets will be flown in the NAS whereas the current number of unmanned fleets is 2.5 million [1]. Furthermore, there is an emerging literature for the new concept of Personal Air Vehicle (PAV) and air taxi [3], [4], [5], [6]. This will also increase the burden of air traffic management. Athenes, Averty *et al.* has shown that the complexity of airspace is increasing the workload of air traffic controllers as measured by the Traffic Load Index (TLI) [7]. The TLI consists of weight factors based on the time urgency and uncertainty of each aircraft controlled by air traffic controllers.

As a result of the increased and diversified air traffic, air traffic controllers need to make decisions more precisely while handling a much larger amount of air traffic. This increased workload will exceed the capability of air traffic controllers. If there is a decision support system which can provide possible solutions quickly, it should alleviate the problem of this

situation. The system also can be utilized as a component of an automated decision making system. To achieve the goal of better decision support, the system should have the ability to explore the decision space quickly. Also, it should be able to generate reliable and accurate evaluations of target state variables of the system.

1.2 Air Traffic Management

Air traffic management (ATM) encompasses all the activities required to manage the NAS safely and efficiently. This usually consists of two components: air traffic control (ATC) and air traffic flow management (ATFM) [8]. ATC is mainly dealing with tactical decisions including real-time separation procedures for collision detection and avoidance. In order to provide the ATC services, the NAS is divided into several different sectors and human air traffic controllers are controlling the traffic's procedures assisted by air traffic control systems. On the other hand, ATFM handles more strategic decisions, globally. ATFM procedures detect and resolve demand-capacity imbalance problems across the NAS. Thus, ATFM focuses on the flow of air traffic considering a global view of the entire airspace.

Much of the previous ATM has been done with ATC level decision making. This approach was sufficient with a relatively small amount of traffic. However, when the amount of traffic increases, it becomes more important to handle the problem with a more scalable approach [8]. Furthermore, there exist many opportunities to gain more efficient ATM by using the flow management scheme. For that reason, there have been a significant number of studies to model the ATFM problem. They can be categorized into two types of models which are prescriptive models and descriptive models depending on the model's goal to address the problem. Prescriptive models usually define an optimization problem with a given situation, then solve the model with various numerical methods. This can be used for decision making processes. On the other hand, descriptive models try to regenerate or mimic the behaviors occurring in the NAS, then find and analyze patterns and key characteristics of the system. For this purpose, computer simulations are widely utilized.

1.2.1 Prescriptive models

In the following, the use of demand and capacity planning to create prescriptive models is described.

Demand management models

Initially, administrative and economic policies were used for demand management [9]. This approach has the problem of correctly setting appropriate capacity limits and resulted in inefficient air traffic management. In order to mitigate the problem, auctions and congestion pricing models have been incorporated into the demand management models [8]. In the case of the auction model, a bid mechanism is utilized for allocating the landing and departure slots to airlines [10], [11], [12]. On the other hand, congestion pricing models try to manage demands by imposing a certain congestion fee for departures and landings [9]. Daniel [13] models equilibrium congestion prices at hub airports. There have been several research studies related to the efficiency of the congestion pricing models [14], [15].

Capacity models

The capacity of an airport is usually determined by multiple factors [8]. They include weather conditions, aircraft types and types of operations-whether they are landings, takeoffs, and the sequence of those [16]. Barnhart *et al.* [9] proposed an approach using a capacity envelope with a convex shape. Based on the capacity envelope, the capacity management models try to optimize the allocation of the possible time slots for landing and departure. This can be solved using integer programming methods [17]. Based on this optimization problem, there have been approaches that add uncertainty to the model [18], [19].

1.2.2 Descriptive models

As explained before, descriptive models are usually realized as simulation models. They can be divided into two different categories based on the scope of the model [8]. The first category is airspace models which analyzes global behavior of the airspace and deal with interconnection among the entities inside the system. The other category is airport models which focus on local characteristics of the facilities such as runways and taxiways.

Airspace models

The Airport and Airspace Simulation Model (known as SIMMOD) [20] and the Total Airport and Airspace Modeler (TAAM) [21] are considered the first airspace models used widely. They have capabilities to model a high level of detail such as gates, terminals, runways and en route flight schedules. These detailed models provide a variety of functionalities and model at a microscopic level. In order to support more aggregate level analysis, there have been several different models. The Logistics Management Institute (LMI) developed LMINET using a queueing network model of the NAS [22]. LMINET can be used for analyzing delays at airports and en route sectors. However, it does not model the interdependencies within a flight schedule, and cannot capture the characteristics of delay propagation [8]. Another queueing network based model, NASPAC (National Airspace System Performance Analysis Capability) [23], was developed by the MITRE corporation and adopted by the FAA. It uses a daily flight plan as an input and simulates flight times and delays. The MITRE corporation also improved it and developed another queueing network based model called the Detailed Policy Assessment Tool (DPAT) [24]. It has more detailed en route sectors and weather models. It also includes a parallel simulation capability.

There have been some models that add special functionalities for a specific purpose of the analyses. The Aviation Environmental Design Tool (AEDT) was developed for the analysis of environmental impacts across the airspace and airports and delivered to the FAA [25]. It generates all the flight trajectories under investigation and computes all

noise, emissions and emission dispersions made by flights. For the aircraft models, it utilizes the Base of Aircraft Data (BADA) from EUROCONTROL. For the evaluation of new future air traffic management concepts, the National Aeronautics and Space Administration (NASA) developed two airspace models. The Airspace Concept Evaluation System (ACES) is an agent-based model to examine the costs and benefits of novel ATM operational paradigms [26], [27]. It models all the major components of the ATM system as agents and emulates their activities across the NAS. The Future ATM Concepts Evaluation Tool (FACET) was also developed by NASA Ames Research Center and includes detailed dynamics model of each individual aircraft [28], [29]. It models all the en route navigations and weather conditions in the NAS. It computes detailed dynamic models of aircraft and weather models, and has computationally heavy workloads. In addition, NASA continues to improve and modernize the airspace model and proposed the Shadow Mode Assessment using Realistic Technologies for the National Airspace System (SMART NAS) [30]. In the newly designed model, NASA includes state-of-the-art modeling and simulation techniques and data analytics methods.

Airport models

On the other hand, a significant amount of research has focused on the airport facilities [8], [31]. An analytical model to determine the capacity of a single runway was firstly proposed by Blumstein [32]. The Airfield Capacity Model (ACM) which calculates the maximum throughput capacity of multiple runways was released by the FAA [33]. Queueing network based airport models have also been proposed. The difficulty in modeling an airport as a queueing network is that the process is not stationary. Koopman proposed a time-varying Poisson distribution for the arrival sequences [34]. Kivestu extended this research by applying the second order methods to model a time dependent queue [35]. Malone analyzed the characteristics of the queues modeled by Koopman and Kivestu, then developed some rules of thumb that could help planners and operators of airport facilities [36]. The Inter-

national Air Transport Association (IATA) introduced a fast time airport simulation model called Total AirportSim [37]. It includes runways, gates and terminals and is compatible with SIMMOD input data.

1.3 Parallel Simulation

1.3.1 Computer Simulation

Computer simulations are used in a variety of applications including entertainment, training, scientific research, weather forecasting, etc. By modeling the rules and properties of the target systems properly, one can assess different scenarios in a way that is safer and less expensive than experimentation with the actual system. In cases where experimentation is difficult or impossible such as exploring nuclear reactions or the creation of the universe, computer simulation can provide a way to examine the system. If one does not have enough data for future planning, computer simulations can generate data. Computer simulation is also used in early design phase explorations. It can also help to explore the decision space for system management.

Computer simulation has also been used for the air traffic management domain as discussed earlier. In the designing of a new airport or adding a new runway to an existing airport, the impacts of the new airport can be evaluated using computer simulations during the design phase. It can evaluate the noise impact near the airport area, emissions for the entire NAS, and economic impacts caused by the addition of a new runway. It is also actively utilized for the training of the air traffic controllers before a new air traffic control system is added. Computer simulation can support decision makings by air traffic controllers in managing the airspace.

1.3.2 Parallel Discrete Event Simulation

When a simulation is realized as a computer simulation, it consists of three major components. The first component is a collection of variables which characterize the state of the

system under investigation. The second component is a set of state transition rules which modify the state variables of the model over time. The last component is the time management scheme during the execution of the simulation program. The state variables depend on simulation time.

Computer simulations can be classified into two categories: continuous simulations and discrete simulations. Continuous simulations assume that the changes of state variables occur continuously over time. Usually, the behavior of the system is modeled by differential equations. By solving a set of numerical equations, the state variables of the system are computed over simulation time. This type of simulation is widely used for the simulation of fluids, materials and structural deformations.

On the other hand, discrete simulations assume that the changes in state of the system occur at certain points in the time domain. These state changes can be encapsulated in "events" and such simulations are also called discrete event simulations. For instance, a departure event in an air traffic simulation program can encapsulate the decreased number of the aircraft in the airport and the increased number of the aircraft in the airspace. This thesis focuses on discrete event simulations.

In terms of the speed of computer simulations that are of interest here, they can be categorized into two different types: fast time computer simulation and real-time computer simulation. Fast time simulations execute multiple different scenarios as rapidly as possible. They then provide evaluations of those scenarios to users. On the other hand, real-time computer simulations do not need to be faster than real-time. Rather, they need to provide on-time evaluations of the state. This thesis focuses on the fast time simulation capability to support decisions of air traffic controllers. For decision support in real world situations, the planning of future schedules, the possible design space explorations, etc., fast simulation execution is often important. As an example, when an air traffic controller is managing the airspace, the performance of the system could be enhanced by completing many simulation runs in a short period of time to provide evaluations of the results of different decisions.

There have been several approaches proposed to speed up discrete event simulations. They have mainly focused on the parallel execution of part or all of the simulation in order to efficiently utilize parallelized hardware. These techniques are often referred to as Parallel Discrete Event Simulation (PDES) methods. Various different studies have been conducted to distribute one discrete event simulation program into multiple computing units to achieve parallel execution. They can be categorized into two approaches termed spatial (or state) parallel simulation and time-parallel simulation.

Spatial parallel simulations divide the state variables into a certain number of buckets and execute the simulations of those buckets in parallel. Each bucket is called as Logical Process (LP). LPs execute concurrently. On the other hand, time-parallel simulations divide the simulation time axis into a number of intervals and simulate these intervals concurrently. Here, each time interval simulator is called an LP. These two concepts are illustrated in Figure 1.1.

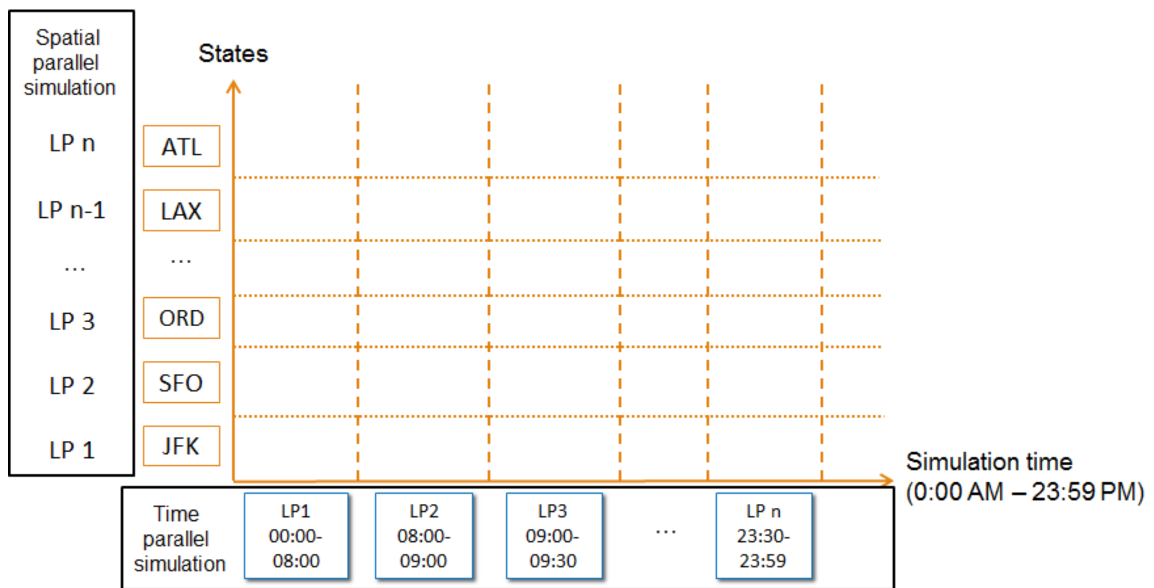

Figure 1.1: Two different parallel simulations of air traffic network.

Both approaches have non-trivial problems associated with splitting the simulation into LPs and distributing them across different computing units. In the case of the spatial parallel simulation, dependencies among the states distributed across different LPs need to be

linked properly. A synchronization algorithm is needed for this purpose. For the time-parallel simulation, each LP is dependent on the previous time interval LP and this dependency needs to be addressed appropriately. This is called the “state-matching” problem. The remainder of this section describes approaches to address these problems in greater detail.

1.3.3 Spatial Parallel Simulation

In spatial parallel simulation, each LP may advance simulation time separately from the other LPs, so the current time can be different in different LPs. If the states residing in different LPs are independent, this would be fine. However, there are usually dependencies between LP states and they need to be synchronized appropriately. For instance, the number of arriving aircraft in one airport at a certain time will depend on departures from other airports. The time stamps of these events are determined by the LPs simulating those airports. If interactions between LPs are not synchronized properly, the simulation will produce incorrect results.

To avoid this problem, several different approaches to synchronize LPs have been proposed. They can be divided into two categories: conservative and optimistic synchronization algorithms.

Conservative Synchronization Algorithms

As the name implies, conservative synchronization algorithms prevent out-of-order event processing. One way to avoid out-of-order execution of events is to ensure each LP only processes events with time stamp less than the minimum time stamp of events that will later be produced by other LPs. In order to find this lower bound on the time stamp of future events, one can determine the minimum time stamp of future messages other LPs might send. Then, the earliest time stamp among these is the safe time to which each LP can proceed [38]. It can be shown that this procedure ensures correct simulation results if

all the LPs are following this rule.

This approach can lead to deadlock situations. This means that every LP is waiting for a message from another LP and no LP can actually send a message to the others. To resolve this deadlock problem, Chandy *et al.* proposed a "null message" approach [39]. The algorithm is also called the 'Chandy/Misra/Bryant' algorithm after the inventors. Every LP sends null messages to the other LPs when it has no events to process. The extra messages, avoid deadlock situations.

One disadvantage of the null message approach is the additional communication overhead. The frequency of null messages depends on lookahead values which derives from the distances between airports in an air traffic simulation. With a small lookahead value, there might be many null messages sent. There have been several proposals to address this problem. Thomas *et al.* reduce the number of null messages by grouping some LPs together [40]. Inside the same group, there is no need to send null messages. Wang *et al.* proposed an approach to add a requested time stamp for the null message in a discrete and continuous hybrid simulation model [41].

Another class of conservative synchronization algorithms use a synchronous execution approach. The LPs find a certain time stamp to which they can proceed without causing synchronization errors [38]. Then, all the LPs proceed to this simulation time and stop, using a barrier communication mechanism. This algorithm is presented in Nicol's paper where the Yet Another Windowing Network Simulator (YAWNS) algorithm is proposed [42]. The basic idea of the algorithm is to calculate a lower bound of time stamp (LBTS) of events each LP may later send and compute a global minimum or LBTS value. Then, by processing all events before the LBTS, global synchronization is achieved. Nicol also analyzed the efficiency of conservative synchronization in parallel discrete event simulations [43]. In order to improve the performance of synchronous algorithms, different approaches have been developed including a tree and butterfly topology [38].

To gain more efficient communications, hardware supported synchronization on multi-

core architecture was proposed by Lynch and Riley [44]. The Global Synchronization Unit (GSU) is inter-connected with all the CPU cores in the system. Liu and Rong presented a hierarchical composite synchronization algorithm which combines the null message and synchronous execution algorithms [45].

Optimistic Synchronization Algorithms

Optimistic synchronization algorithms allow out-of-order execution of the events, but recover using a rollback mechanism. Jefferson initially proposed this approach to discrete event simulations [46]. After the rollback operation, LPs resume the simulation at the time stamp of the rollback. In order to rollback message sent, each LP sends anti-messages to cancel these messages. In the Time Warp algorithm, the states of each LP are saved (checkpoint) prior to processing each event to allow state variable to be restored when a rollback occurs. This incurs a certain amount of overhead. Also, the rollback itself requires a certain amount of overhead for both computation and communication. There has been a significant amount of research related to reducing these overheads. Lin *et al.* proposed that checkpoints do not need to be taken so frequently [47]. Instead, he proposed an algorithm that finds an optimal interval to take checkpoints. Rönngren *et al.* presented an adaptive approach to change checkpoint intervals dynamically based on the rollback behavior [48]. In terms of efficient resource management, Lin *et al.* investigated how to optimally manage memory in time warp simulations [49]. Fujimoto and Panesar studied the importance of buffer management strategies in shared-memory time warp simulations [50]. There have been proposals to reduce the number of rollbacks. Wang *et al.* proposed an approach to limit optimism by using a reinforcement learning algorithm [51]. A reward for the reinforcement learning is defined using the Event Commit Rate (ECR) and the model is trained by the Q-learning algorithm.

1.3.4 Time-Parallel Simulation

Time-parallel simulation involves dividing the simulation time axis into non-overlapping segments T_1, T_2, ... T_N where the end of time segment T_i coincides with the beginning of time segment T_{i+1}. Then, the simulation of each time segment is assigned to a different logical process LP_1, LP_2, ... LP_N, with LP_i responsible for simulating time interval T_i. Each such LP may then execute concurrently on a different processor. This would be a very straightforward approach to parallelization if the simulations of the different time segments were independent of each other. This is rarely the case, however, because (1) the initial state of LP_i depends on (is identical to) the final state of LP_{i-1} — the so-called *state matching problem* — and (2) the events occurring in LP_i may depend on the simulation computations performed in earlier time segments. One notional example of the state-matching problem is illustrated in Figure 1.2.

To address this problem, time-parallel simulations typically require multiple rounds of execution, and utilize a *fix up* computation to correct errors resulting from data dependencies between LPs that were not accounted for in the initial execution. Time-parallel simulations can become inefficient if an error propagates through multiple time segments because an additional round of fix up computations is needed to propagate the correction through each successive time segment. There have been several different approaches to address this problem. Some use repeated fix up computations to correct errors in guessed initial states [38]. Fujimoto, Nikolaidis *et al.* presented an algorithm using regeneration points to simulate ATM multiplexers [52]. Time-parallel simulation of queues using parallel prefix computation algorithms were introduced by Greenberg, Lubachevsky et al [53]. In order to improve parallel efficiency, several approximate state matching algorithms have been developed. Wang and Abrams proposed an approximate time-parallel simulation algorithm of queueing systems with losses [54]. Kiesling, Tobias *et al.* proposed a time-parallel simulation algorithm with approximate state matching and analyzed its efficiency and accuracy [55]. More recently, Thi, Fourneau *et al.* applied time-parallel simulation to

stochastic automata networks [56]. Grasedyck, Lobbert el al. use space and time-parallel simulation for multigrid PDE simulations [57]. An approach to simulate large-scale dynamic transportation systems using spatial and time-parallel simulation has been proposed by Qu and Zhou [58].

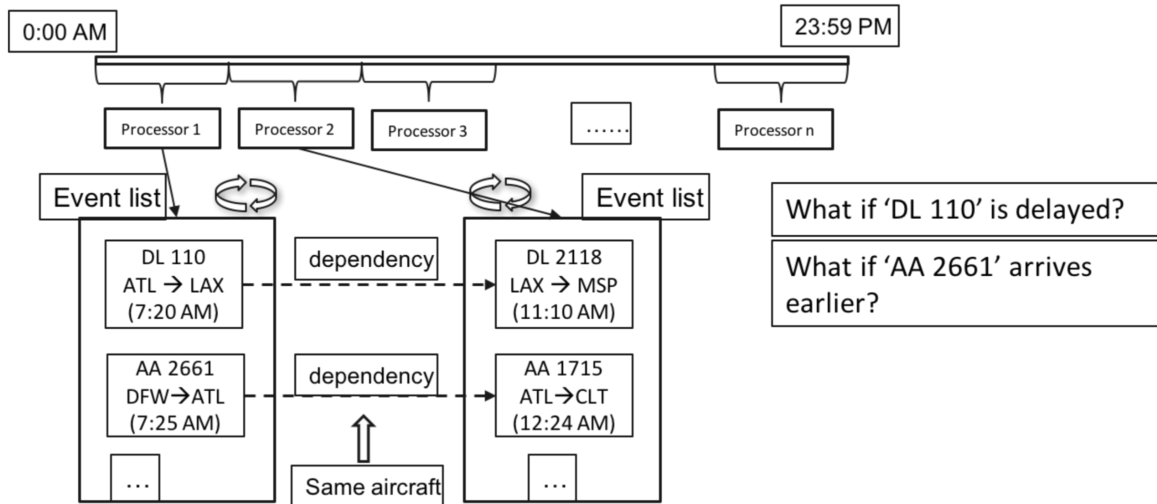

Figure 1.2: Notional example of state-matching problem.

1.4 Artificial Neural Networks

Artificial Neural Networks (NN) is a non-linear function estimation model for a given system which has multiple inputs and outputs. By loosely imitating the activities happening in a human brain, it tries to find the best mapping function between input and output. The basic unit of the ANN is called a neuron and it consists of a lot of connecting links between input variables and output variables. By going through a neuron, input variables are multiplied by some constant weight values, then summed into a specific output variable. After that, all the output variables go into the non-linear activation functions such as a logistic sigmoid function (*sigmoid*), a hyperbolic tangent function (*tanh*) and rectified linear unit (*ReLU*). Figure 1.3 shows the basic neuron of the ANN. Figure 1.4 shows three different kinds of activation functions. By stacking multiple neurons hierarchically, a complex system can be modeled mathematically. Activation functions enable the ANN models to

express non-linear patterns in the data. The constant weight values for each neuron are found using the pre-existing data set which is available when the model is constructed. By utilizing optimization methods such as the gradient descent optimization, some locally minimal constant weight values which minimize the errors of the model's prediction or estimation are calculated [59].

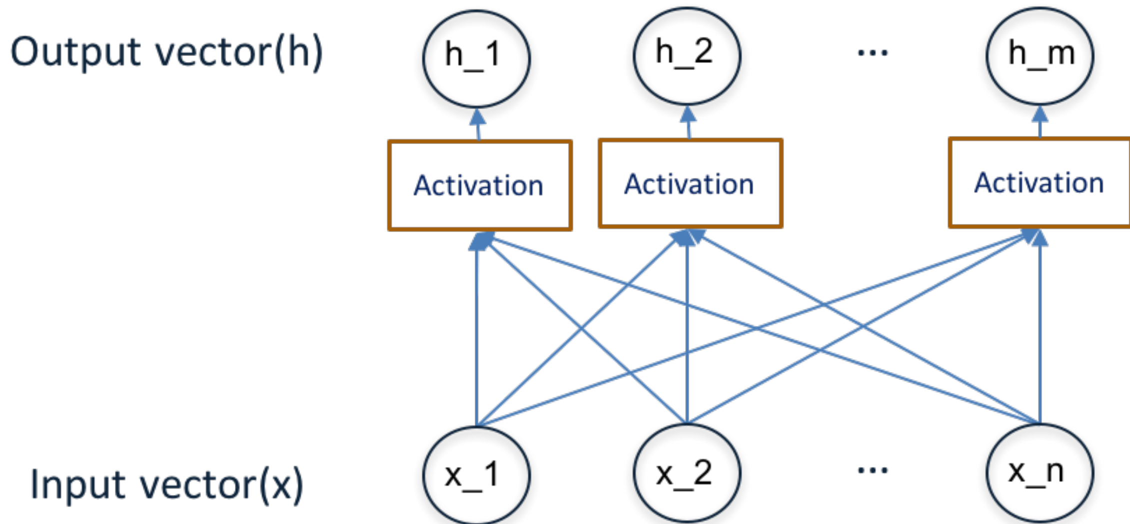

Figure 1.3: A neuron of artificial neural networks.

1.4.1 Recurrent Neural Networks

Recurrent Neural Networks (RNN) is a class of the artificial neural networks that models the behaviors of dynamic systems using hidden states. The main characteristic of RNN compared to the standard ANN is that RNN utilizes outputs from previous time as an input for the model. The difference is illustrated in Figure 1.5. On the left side, the standard ANN architecture is illustrated and RNN is shown on the right side. The output of the model is fed back into the model. Given an input sequence $x = (x_1, x_2, \ldots, x_k, \ldots, x_T)$, RNN computes the evolution of hidden states $h = (h_1, h_2, \ldots, h_k, \ldots, h_T)$ and output sequence $y = (y_1, y_2, \ldots, y_k, \ldots, y_T)$. This computation is performed iteratively solving the following equations for the time span from $t = 1$ to T. Here, x_k, h_k and y_k can be any arbitrary sized vectors which are represented as the dimension of input space, hidden space

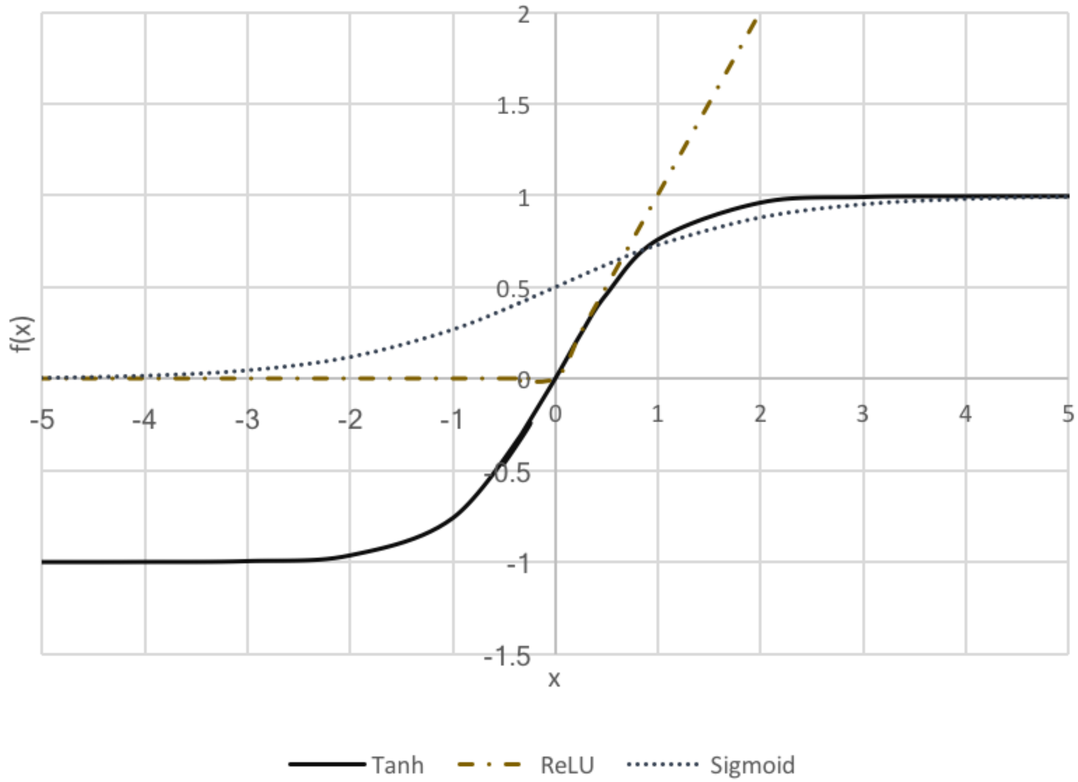

Figure 1.4: Three different activation functions.

and output space. Here, h_0 which is used for the time step 1 is given as an initial condition and can be set as an arbitrary vector at the initial time.

$$h_t = \phi_h \left(W_{hh} h_{t-1} + W_{xh} x_t + b_h \right) \tag{1.1}$$

$$y_t = \phi_0 \left(W_{hy} h_t + b_y \right) \tag{1.2}$$

W_{hh} denotes the weight matrix for the transition of hidden states from the previous time step to the current time step, W_{xh} denotes the weight matrix for the input to hidden layer and W_{hy} denotes the weight matrix for the hidden layer to output. b_h and b_y are biases for each equation. ϕ_h and ϕ_o are activation functions for hidden states and output, respectively [60]. As stated before, for these activation functions, *rectified linear unit* or a saturating nonlinear function such as *sigmoid* and *tanh* is applied element-wisely to the given vector

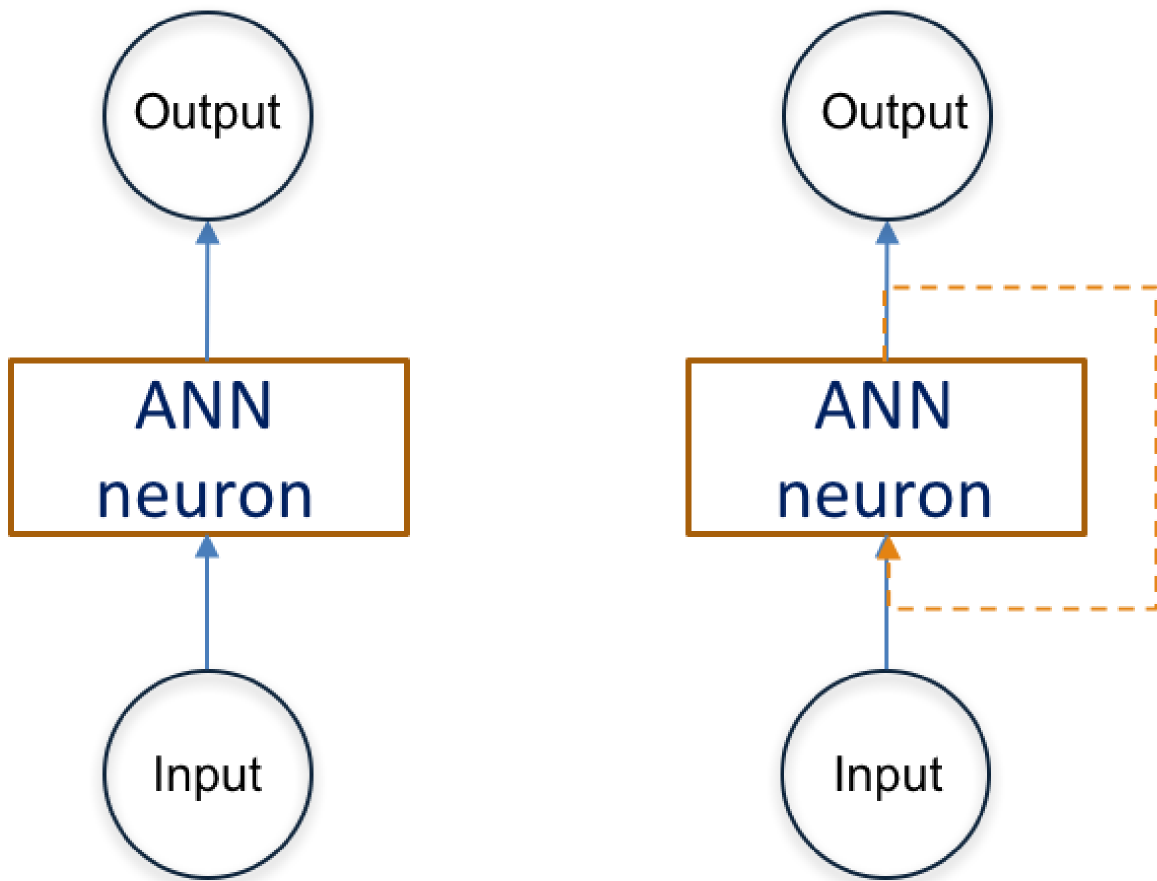

Figure 1.5: Feed forward artificial neural networks and Recurrent neural networks.

usually.

1.5 Research Contributions

Returning to the problem of managing air traffic, an accurate fast time simulation methodology can evaluate possible decision spaces and suggest different options. Time and spatial parallel simulation provides a way to speed up simulations. Recurrent neural network model provides a way to model necessary input variables for the simulations such as the capacity of airports precisely. The two approaches developed here are novel concepts to model the NAS. Neither has been utilized for the air traffic network system applications previously. The value of those components is presented next.

1.5.1 Time-Parallel Simulation of Air Traffic Networks

A new time-parallel simulation algorithm has been proposed and implemented in this research. In particular, this is the first application of time-parallel simulation to the air traffic simulation problem. The developed algorithm shows a sub-linear scalability up to 44 LPs while achieving 27 times faster simulation compared to the sequential simulation for the ideal air traffic schedule. The detailed descriptions and analyses are presented in Chapter 2.

As discussed earlier, the main issue for the time-parallel simulation is the state matching problem. Three properties of air traffic network system simulation mitigate the problem of fix up computations and their propagation through multiple time segments [61]. At first, the simulations are driven by statically defined *flight schedules* that indicate the scheduled departure and arrival times of each aircraft at each airport it visits. These schedules are known prior to the execution of the simulation and can be used by each LP to initially simulate the activity within its time segment. Secondly, so long as the computed arrival time of a simulated aircraft leads to an on-time departure, an error in the computed arrival time will not impact the aircraft's departure, i.e., errors in computing the arrival time will not propagate further in the future trajectory of the aircraft. This greatly lessens the propagation of errors through the time segment. Error propagation resulting from fix up computations largely occurs when the aircraft arrives sufficiently late to affect its subsequent departure time from the airport.

A third factor that facilitates the use of time-parallel simulation concerns the fact that airlines are often organized around a "hub" model where many flights travel through certain *hub* airports where passengers change flights to reach their eventual destination. A typical airline schedule will include many flights arriving within a small time window, a short period of time where passengers change flights, followed by the departure of many flights from the hub. This cycle repeats throughout the day. This results in bursts of high activity at the airport followed by periods of light activity before the next round of flights arrive. This

is advantageous for time-parallel simulations because the effects of congestion-induced delays occurring in one busy period are mitigated by periods of light traffic, meaning computation errors due to delayed aircraft are less likely to propagate throughout the day in a typical airport in the absence of severe events that impact the entire airport for a prolonged period of time. Moreover, these factors suggest that a time-parallel simulation approach may be well suited for air traffic simulation.

1.5.2 Combining Time and Spatial Parallelism for Air Traffic Networks

A new algorithm to combine both time and spatial parallelism into one simulation program for air traffic networks is proposed and implemented. The developed algorithm offers more performance than the stand-alone time-parallel simulation or spatial parallel simulation. For a realistic air traffic scenario including air traffic delays, the combined algorithm achieves 10 times faster simulation which is better than each single parallelism. The detailed descriptions and analyses are presented in Chapter 3.

Spatial parallel simulations require their own synchronization algorithms to synchronize multiple LPs. Therefore, it is also important to carefully design a spatial parallel algorithm to obtain an efficient implementation for the air traffic network simulation. Furthermore, the spatial parallel algorithm must operate together with the time-parallel simulation algorithm. The different communication channels for the time and spatial parallel simulation algorithms must be coordinated properly. Events and workloads also need to be distributed well across the parallel computer. This research addresses these issues and develops an approach to adjust the degree of time and spatial parallelism.

1.5.3 Recurrent Neural Network Model for Airspace Applications

A new RNN based prediction model is proposed and implemented. This is the first application of RNN model to air traffic data analysis. The proposed model presents a highly accurate prediction of the future delay of airports. Several different variants of RNN mod-

els are tested with different control variable settings. The detailed descriptions and analyses are presented in Chapter 4.

According to the hierarchical nature of the model and the very high degree of freedom, RNN is expected to model a certain sequential system very precisely. However, because of the complexity of the model, RNN was hard to train and actually not a viable solution. By leveraging recent advances in computing power and numerical optimization methods, RNN now enables various tasks such as automatic image captioning, machine translation, autonomous driving cars, etc.

Air traffic network system exhibits highly sequential nature. If some delays occur across the NAS at a certain time, these delays will affect the future status of air traffic network and increase the possibility of delays in the next time period. This sequential nature can be naturally modeled in RNN. Also, weather impacts might move across the NAS which causes sequential propagation in the space domain. Therefore, RNN models are expected to capture those key components of the NAS simulation and provide the simulation program with an accurate input. As a result of this relevance, an RNN based flight delay prediction was introduced by Kim *et al.* in 2016 [62].

1.6 Thesis Organization

In Chapter 2, the details of the time-parallel simulation algorithm will be presented. The model for the NAS simulation will be also described. The performance evaluation of the time-parallel simulation is discussed. Chapter 3 describes the approach combining time and spatial parallelisms together. The integration of the two parallel algorithms is discussed. Also, an approach to balance the two parallelization approaches is explored using various experiments. In Chapter 4, the details of RNN models are presented with the theoretical background. Based on the model, an actual model for air traffic application has been built and evaluated with the real world air traffic data and weather data. In Chapter 5, conclusions from this thesis are presented.

CHAPTER 2

TIME-PARALLEL SIMULATION

2.1 Overview

As stated in the first chapter, the amount of air traffic is rapidly increasing and will cause various problems such as delay in the NAS. Fast time simulation was suggested as a tool to help address this issue. Rapid execution of simulation models is important in order to explore a wide variety of scenarios quickly. When used for operations, fast execution is important in order to make decisions in a timely fashion. Parallel processing offers an approach to accelerate simulation executions, and several different parallel simulation algorithms have been explored. These approaches use spatial parallelism where one divides the NAS into distinct regions, and one distributes the state and associated computations to transform this state across different processors so that they can be performed concurrently. It is necessary to properly synchronize these computations. In an air traffic network simulation, the components of the system such as airports, air traffic control centers, flights etc. for a region are typically mapped to a single logical process (LP). Each LP computes its internal states during the simulation run and communicates necessary events with other LPs. Challenges to achieving significant speed up of these simulations include the realization of efficient synchronization among the LPs and managing communication overheads.

Another approach to parallel simulation is time-parallel simulation. As introduced in Chapter 1, it divides the simulation time axis into multiple time segments. Then, each time segment is running on a different processor. Algorithms to handle the state-matching problem is a key component of the time parallel simulation. In particular, a fix up computation is a good solution to this problem because it avoids re-execution of the simulation. Fix up computation algorithms for the air traffic network system are discussed in this chapter.

As discussed in Chapter 1, there are three promising characteristics of air traffic simulations with respect to time-parallel simulation. First, they can utilize pre-acquired flight schedules in the simulation. Second is the errors will not propagate if arrival delays are not large enough to result in delaying the subsequent departure. The third characteristic is the hub model which also reduces the probability that errors will propagate.

This chapter describe a new time-parallel simulation algorithm and evaluates it ability to speed up the simulation of the NAS. Details of time-parallel simulation of the NAS are described and the algorithm to resolve the state matching problem is presented. The algorithm is verified using real world air traffic data.

The remainder of this chapter is organized as follows. The next section describes related work. The discrete event simulation model utilized here is then described. The time-parallel simulation algorithm is described, followed by presentation of experimental results based on actual historical data for the NAS. The concluding remarks follow.

2.2 Related Work

There is a significant literature concerning modeling approaches for simulating air traffic network systems including queueing network based models, agent-based models and system dynamics models, e.g., see [22], [63], [64], [65]. Several studies have focused on the use of parallel simulation algorithms to speed up the simulation models. Wieland reports results concerning the use of parallel simulation algorithms for aviation applications [66]. Lee, Pritchett *et al.* presented several different parallel simulation algorithms for the analysis of the NAS [67]. Hybinette and Fujimoto proposed a new method to maximize the parallel efficiency of parallel aviation simulation by using a simulation cloning technique [68].

There have been several studies using the time-parallel simulation approach for a variety of applications. It was used for trace-driven cache simulation [69], ATM multiplexers simulation [52], stochastic automata networks [56], multigrid PDE simulation [57], and

other applications. Several approaches attack the state matching problem. Some use repeated fix up computations to correct errors in guessed initial states [38]. Time-parallel simulation of queues using parallel prefix computation algorithms were introduced by Greenberg, Lubachevsky *et al.* [53]. In order to improve parallel efficiency, there have been several approximate state matching algorithms developed. Wang and Abrams proposed an approximate time-parallel simulation algorithm of queueing systems with losses [54]. Kiesling, Tobias *et al.* proposed a time-parallel simulation algorithm with approximate state matching and analyzed its efficiency and accuracy [55].

Although there is a growing literature in time-parallel simulation, there has not been an attempt to apply it to the problem of air traffic network simulations. The effort described here addresses this issue.

2.3 Modeling Techniques

In order to determine the most appropriate technique to model the air traffic network system, historical modeling techniques of the system were investigated. Modeling techniques include discrete event simulation using a queueing network model, fluid flow models, system dynamics models, and agent-based models. Characteristics, strengths, and weaknesses for each of these modeling techniques are discussed next [65].

2.3.1 Queueing network based discrete event simulation

In the mid 1980s, air traffic simulation modeling mainly relied on discrete event simulation (DES) techniques based on queueing networks. Queueing network based simulation is the simulation technique which consists of state variables, an event list and a global clock. Time-stamped events are generated and consumed as the simulation proceeds through time. At the consumption of each event, state variables are changed, and new events may be scheduled. The event list, which is implemented by a priority queue data structure, stores and removes events in time stamp order [38]. For example, in the air traffic network sim-

ulation, arrivals and departures of airplanes can be defined as events, and the occupancy of runway and gates can be modeled as state variables. One example using this queueing network approach is the National Airspace System Performance Analysis Capability (NASPAC) simulation model that was first developed in 1989 by the Federal Aviation Administration (FAA) and Mitre Corporation. In 2008, it was modernized to improve its performance [70]. Another example of a queueing network model of the air traffic network system is LMINET developed by Logistics Management Institute (LMI) for the National Aeronautics and Space Administration (NASA). It uses the airport delay model, which defines a system's components as a queue with a delay time (service time for each component). For example, a taxi-queue stores arrival events of the airplane and models the taxiing of the airplane as a delay in the taxi-queue [22]. Queueing network modeling of the air traffic network is increasing in popular with improvements in discrete event simulation techniques [71, 72]. One advantage of the queuing network based discrete event simulation is that it is straight forward to design a spatial parallel simulation. Each individual queue can be mapped to an LP. Depending on the number of parallel hardware units, multiple queues can be assigned to one LP.

2.3.2 Fluid flow modeling

The fluid flow modeling approach models the air traffic network as a set of fluid flows. It is called the Eulerian approach because it uses Euler equations of fluid dynamics to model air traffic flow. This model spatially aggregates air traffic to generate models of air traffic flow in a network of interconnected control volumes. It focuses more on the aggregated aspects of the network system such as arrival rate at a certain airport rather than on the detailed arrival and departure times of individual airplanes [73].

Figure 2.1 shows a basic example of this flow model. An approach proposed by P.K. Menon and NASA in 2004 is a good example of the fluid flow modeling of the air traffic

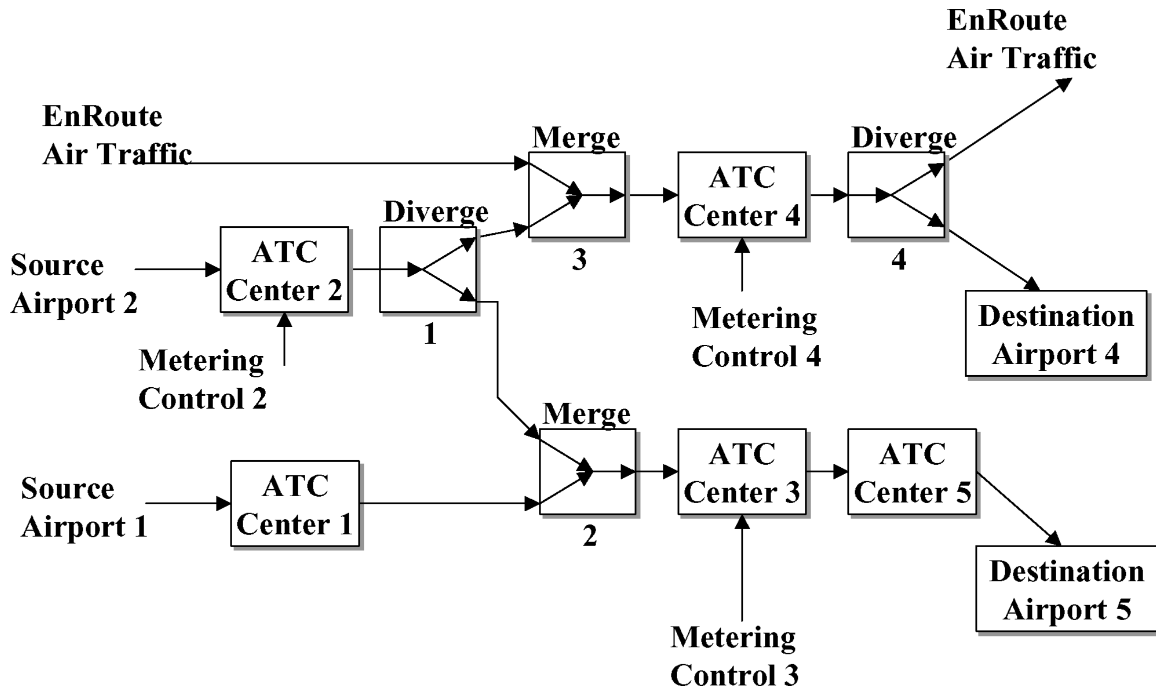

Figure 2.1: Air traffic environment in terms of control volumes and merge and diverge models [73].

network. Under a quasi-steady-state assumption, this research applied linear control theory to the controlling of air traffic [73]. This model offers an advantage that classical control theories can be easily adopted because of the continuous characteristic of this model. However, detailed departure and arrival times, which are not available in this model, can be very helpful to decision makers. Also, this model cannot show the complex collective behaviors of the air traffic network system such as the propagation of delay times of flights throughout the system.

2.3.3 System Dynamics Modeling

System dynamics modeling of the air traffic network system has been suggested since the 2010's as another modeling approach. This modeling technique uses feedback loops along with stock and flow diagrams. Interrelationship among the entities of the system is the main focus of this model. Equations for the stock and flow model are found to quantify the relationship of the components using historical data or data from a physics-based model.

The simulation involves processing these equations. Because of the continuous nature of this model, it also focuses on the macro level of the system rather than the micro or operational level [74]. Dr. Pinon developed one simulation model using system dynamics to valuate and select the technology portfolios for small and medium airports [64]. This modeling technique captures the collective complex behavior of the air traffic network system, but it can be unsuitable in some scenarios. It shows good estimation of the system's states in a macro level, but real state values are smoothed in a micro level. It is difficult to evaluate trade-offs between entities' policies and local rule changes because it is a more deductive approach. The deductive approach means that governing equations of relationships are extracted from the existing data and applied to the simulation model using a top-down approach.

2.3.4 Agent-Based Modeling

Agent-based modeling (ABM) is widely used in logistics, economics and civil engineering simulations [75]. ABM has also become widely used in air traffic network simulation. An ABM consists of numerous agents which represent individual components of the system. The simulation models the agents' local activities based on their local rules for interacting with each other. More detailed characteristics of the agents are described below [76]:

- Agents are entities with well-defined boundaries and interfaces
- Agents are situated in a particular environment
- Agents strive for specific objectives
- Agents are autonomous
- Agents can be both reactive and proactive in order to achieve their objectives

When modeling the air traffic network system, entities of the system such as airlines, government, and air traffic controllers can be modeled as agents. Figure 2.2 shows one example

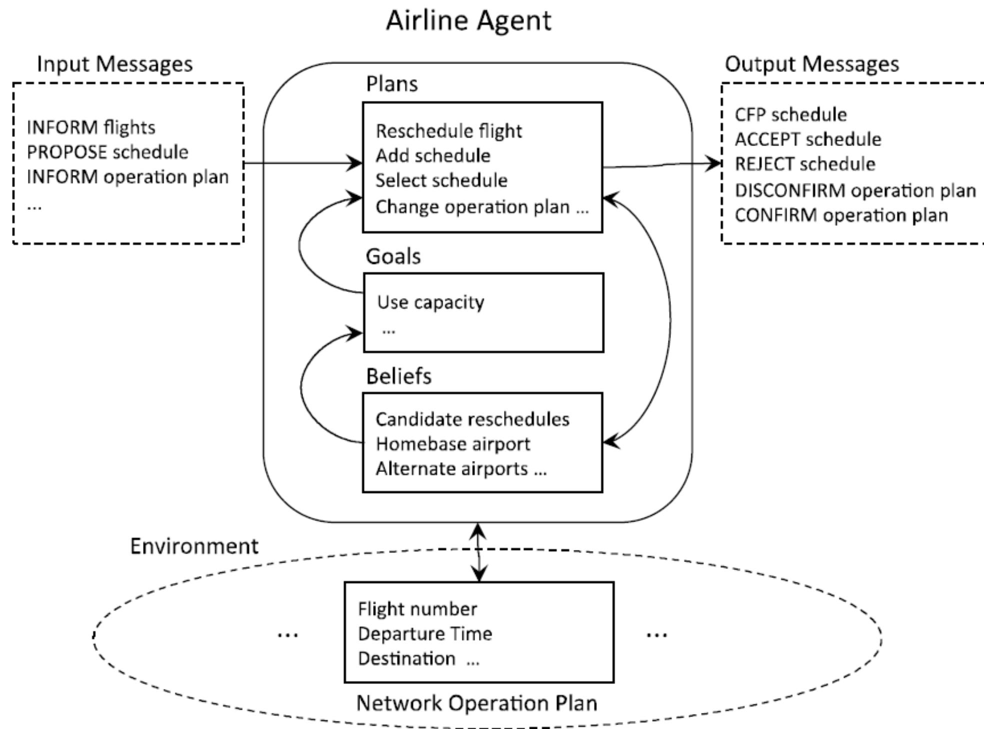

Figure 2.2: Summary of the model of the airline agent [63].

of an agent design. An airline agent determines flight schedules based on the interaction between other environment agents. Because the agents' characteristics are very analogous to the characteristics of the complex system, ABM is well-fitted to model the air traffic network in order to capture its emergent behavior. The agent-based modeling technique is built using a bottom-up approach and can capture not only a detailed level of information but also macro level information. It can simulate complex collective behaviors of the system. The Airspace Concepts Evaluation System (ACES) developed by NASA in 2003 was the first example of an agent-based model of the air traffic network system. It models Aeronautical/airline Operational Control (AOC), Air Route Traffic Control Center (ARTCC), Terminal Radar Approach Control (TRACON), Airport Traffic Control Tower (ATCT), flight and other agents. It simulates the air traffic network accurately but requires a very long time to implement and run. It was estimated that a simulation of 25,000 flights through 250 airports and 20 centers required five to six hours in 2003 [77]. In Europe, the

Single European Sky ATM Research (SESAR) program's Complex Adaptive Systems for Optimization of Performance in ATM (CASSIOPEIA) project attempted to develop a new agent-based model of the air traffic network system. It uses a higher level of abstraction than previous ABM research and models different ATM stakeholders and new collaborative decision processes for flow traffic management [63].

Considering the characteristics of the different modeling techniques, the queuing network based discrete event simulation was selected as a model in this research. It is able to model each individual aircraft. Also, different level of abstractions can be easily implemented by just replacing event definitions. The simulation executive can be reused for different applications. ABM also can model different abstraction levels, but it is typically slower than queueing network based DES. Thus, DES is a better choice to build a decision support tool. Also, the developed parallel simulation algorithm can also be extended to use the other modeling techniques.

2.4 Simulation Model

Here, a discrete event simulation model is used to model the NAS for a single day in the continental U.S. Specifically, we model the domestic airports in the U.S. with the commercial flights using a queuing network-like model. Each flight departs from an airport based on the flight schedule for the day and the availability of resources for the flights. These resources could include runways, gates, taxi ways and the air traffic controllers in the airport. In addition, the airspace and air traffic controllers en route can be viewed as resources. Here, we focus on modeling airport runways and delays associated with waiting to utilize the runways. Each airport models arriving and departing flights based on the schedule for the day. The sequence of arriving and departing flights is modeled as events within the simulation. The main goal of the simulation is to track delays encountered by each flight in order to improve management of the NAS by minimizing delays and the number of diverted flights. Further, each aircraft is used for a sequence of flights throughout

the day resulting in correlations and interactions among those flights. Details of the event definitions, resources and the sequence of events in the system are described next.

2.4.1 Simulation Model Resources

Runways are the main resource modeled within this simulation system. A principal task of the simulation is to compute queuing delays resulting from congestion as many flights compete for the available runways. The hub model described earlier that is used by major airlines can exasperate runway delays because arrivals and departures are scheduled to cluster into certain, busy time periods. All incoming and outgoing flights must wait until a runway is available to land or take off.

2.4.2 Ground Delay Programs

One important aspect associated with modern air traffic systems concerns the management of large volumes traffic arriving at certain congested air traffic spaces such as the north-eastern United States. Due to the limited amount of fuel an aircraft can carry as well as the expense of burning fuel in flight, the time that an arriving aircraft can wait to land is limited. If the aircraft cannot be assigned for landing, it may need to be diverted to an alternate airport. Not surprisingly, this can lead to many other subsequent problems and issues. Ground delay programs (GDP) are used to avoid this situation [78]. The main idea of a GDP is to add delay to an aircraft while it is still on the ground prior to departure if the destination airport is expecting a severe shortage of capacity for arriving flights. Such limits in capacity can be caused by severe weather conditions or high traffic volumes. Modeling a GDP requires definition of the airport's capacity as measured by the number of flights an airport can handle over a specific time period. This value is usually determined by weather conditions near the airport. Delays depend on the volume of air traffic arriving at that airport.

2.4.3 Event Definitions

The simulation utilizes five different types of events:

- Arrived

  The arrived event represents the arrival of an aircraft at either its original destination or the alternate airport in the case of a diversion. This event is typically scheduled when a departed event is processed at the airport originating the flight. Alternatively, it can be also scheduled when a diverted event is processed. When this event is processed a runway is assigned as a resource that is used by the flight. In the time-parallel simulation, the time stamp of the arrived event could be changed by the fix up computation. This is discussed in greater detail later.

- Diverted

  The diverted event represents the diversion of an aircraft to land at an alternate airport because of limited fuel and/or inability of the destination airport to handle the incoming flight due to limited capacity. The processing of this event will schedule another arrived event at the alternate airport.

- Landed

  The landed event represents the completion of the landing sequence for an aircraft. It represents the exit of the aircraft from the runway it used in the landing sequence. This event is scheduled when an arrived event is processed with some amount of time to model the landing of the aircraft on its assigned runway.

- Taxi-in

  The taxi-in event represents the arrival of an aircraft to a gate in order to unload and load passengers. In this simulation, the traffic on the ground at the airport is modeled by a fixed time delay. Therefore, the event is scheduled when a landed event is processed with some amount of time delay determined by an analysis of historical

data for that airport.

- Departed

  The departed event represents the departure of an aircraft from an airport based on the schedule and availability of resources at the destination airport. The event is scheduled when a taxi-in event is processed using certain delay computations. This computation utilizes two different models related to flight delays. The first model uses the delay of the previous flight to compute the delay of the next flight. This model is based on research in delay analysis described in [79], and is described in greater detail later. A second model utilizes ground delay programs in use for air traffic management. As described earlier, ground delays may be inserted before each departure if significant delays are anticipated at the destination airport.

An event handler procedure is defined for each event type. The pseudo code for each event is presented in Algorithm 1.

2.4.4 Delay Model

In the simulation of one day of the NAS, an aircraft can have multiple flights. When it arrives at a destination airport, it will prepare for the next flight to be flown. Therefore, the delay from the previous flight might affect the delay of subsequent flights by that aircraft. To accommodate possible delays, schedules are defined that include buffer times are used to absorb delays in both ground operations and flight delays en route. Turn-around buffer time is used to mitigate possible departure delays by adding extra time for ground operations. On the other hand, block buffer time is used for mitigating possible arrival delays by adding extra time in the sky. When these buffer times are able to absorb delays on the ground and in the sky, there will be no delays relative to the published flight schedule. However, if the actual delays exceed the buffer times, there will be delays that can propagate through to other flights.

Algorithm 1 Simulation event handler.

```
1:  procedure ARRIVED
2:      if runway.isBusy() then
3:          if runway.waitTime() > threshold then
4:              executive.scheduleEvent(new Diverted)
5:          else
6:              runway.insert(this)
7:      else
8:          executive.scheduleEvent(new Landed)

9:  procedure DIVERTED
10:     new_dst ← findNearAirport()
11:     sendMessage(new_dst)

12: procedure LANDED
13:     executive.scheduleEvent(new Taxi-in)
14:     if runway.isWaiting() then
15:         runway.pop_event()
16:         executive.scheduleEvent(new Landed)

17: procedure TAXI-IN
18:     dep_time ← max(dep_schedule, sim_time + G^d − b^sg)
19:     executive.scheduleEvent(new Departed)

20: procedure DEPARTED
21:     if destination.isBusy() then
22:         executive.scheduleEvent(new Departed)
23:     else if
24:         then sendMessage()
```

The formulas to compute the departure and arrival delays are shown below. D^d and A^d represent the actual departure delay and the actual arrival delays, respectively. G^d represents the ground handling time and b^{sg} represents the scheduled turn around buffer time on the ground. Similarly, R^d represents the block time in the air and b^{sr} represents the scheduled block buffer time.

$$D^d = \mathbf{max}\{0, A^d_{-1} + G^d - b^{sg}\} \tag{2.1}$$

$$A^d = \mathbf{max}\{0, D^d + R^d - b^{sr}\} \tag{2.2}$$

As stated earlier, new departed events and arrived events can be scheduled based on the computation in equations (2.1) and (2.2). The delay model for the departure is illustrated in Figure 2.3. It shows the case where a delayed taxi-in event doesn't affect the next departure schedule because there is sufficient buffer time on the ground.

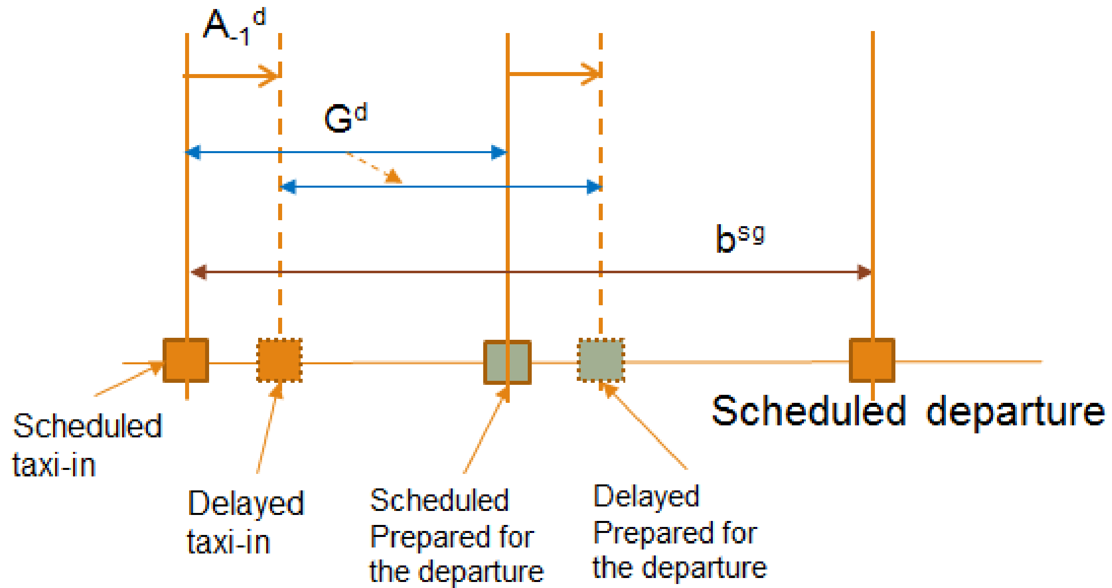

Figure 2.3: Delay model of the departure.

2.4.5 Simulation Execution

When the initial arrival and departure schedules are given for an airport, the events described above are scheduled according to the schedule. The arrived events are scheduled based on the anticipated arrival times. Then the arrived events are processed with queuing delays in the runway resource. As a result of processing arrived events, the landed and taxi-in events are then scheduled. If the airport could not process the arrived event because of the limited capacity in the airport, then a new diverted event will be scheduled. Then, the delay model and the ground delay program model are utilized for the computation of the next time for departure while processing the taxi-in event. As a result of the combined model computation, a new departed event is scheduled. The sequence of event processing

is illustrated in Figure 2.4.

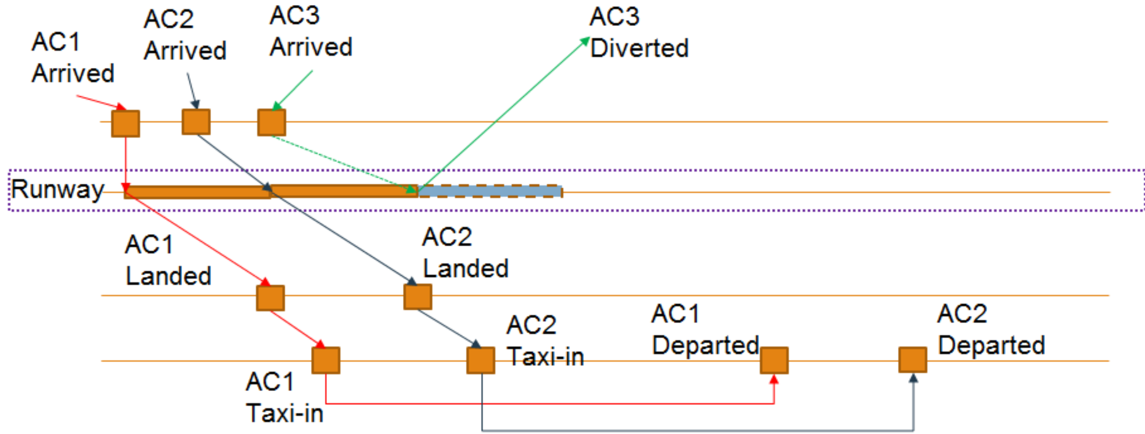

Figure 2.4: Single airport event flow.

2.5 Algorithm

This section explains the algorithms used to execute the air traffic model described in the previous section. In particular, the time-parallel simulation algorithm is described as well as the fix up computation.

2.5.1 Time-Parallel Simulation

The goal of the simulation is to compute the state of the entire system, in this case the NAS, over a single day. Time-parallel simulation and space-parallel simulation are illustrated conceptually in Figure 2.5 with a space-time diagram. While the space-parallel simulation assigns a set of nodes (airports) in the air traffic network to a single LP, the time-parallel simulation assigns a segment of simulation time to each LP. Then, the results from each logical process are merged to produce the final result. One could use both time and space parallelism to simulate the NAS, however, this chapter focuses exclusively on the time-parallel simulation approach. A combined approach is investigated in Chapter 3.

Each LP simulates the NAS over a time segment, say $[T_y, T_z]$. In other words, the simulator must compute the trajectory of each aircraft over $[T_y, T_z]$, as it visits different

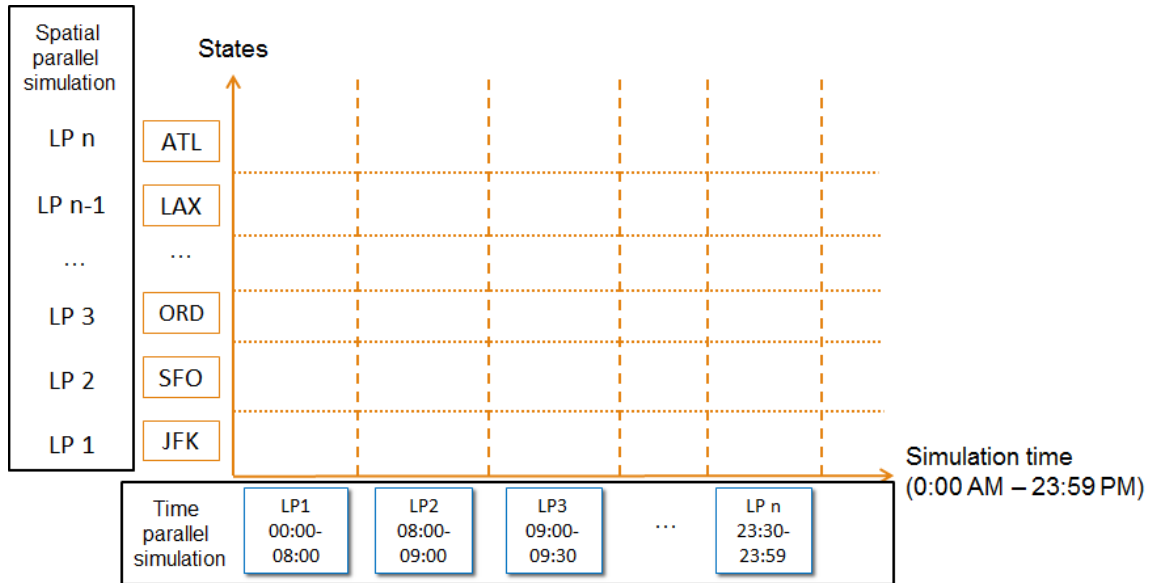

Figure 2.5: Two different parallel simulations of air traffic network.

airports during that time segment. The simulator has the scheduled arrival (and departure) times of aircraft. But initially, the location of each aircraft A_i at the start of the time interval T_y is unknown. However, the simulator does have the scheduled departure times. So, in the initial simulation, the simulator assume A_i leaves the gate on time from the first airport it visits during [T_y, T_z], and it then simulates A_i's trajectory based on this assumption. This information, the *departure time of the aircraft from the gate of the first airport it visits during [T_y, T_z]*, is the key piece of information needed for the time parallel algorithm to work correctly.

After the first round, i.e., all time segment simulations complete their execution, the simulator for the previous time segment [T_x, T_y] will have computed the actual time A_i left the gate at the first airport it visited during [T_y, T_z]. Now, this time was computed based on the assumption of an on-time departure at the first airport A_i visited during [T_x, T_y] which may not be correct. Nevertheless, the simulator for [T_y, T_z] compares the time computed by the simulator for [T_x, T_y] that A_i departed from the first airport it visited in [T_y, T_z] with the on-time departure it had assumed. If A_i did in fact depart on time, A_i was simulated correctly in the first round so nothing more needs to be done. If A_i departed

late, then the trajectory of T_i (and possibly other aircraft) during $[T_y, T_z]$ needs to be fixed up or re-simulated. If there are no differences in event ordering, the fix up computation can correct the simulation, however, if the correction involves reordering events, the initial simulation must be discarded and repeated using the corrected event ordering.

At the end of round 2, the simulator for the previous time segment $[T_x, T_y]$ will have re-computed a new time A_i left the gate for the first airport it visited during $[T_y, T_z]$. Here, another round of fix up computation may be needed. If this time matches what it had reported to the simulator of $[T_y, T_z]$ after the first round, nothing needs to be done. However, if it does not match, the trajectory of A_i during $[T_y, T_z]$ needs to be re-computed again. The above process repeats until no mismatches occur. Once this becomes correct for each aircraft, for each time segment, the time parallel simulation is done.

2.5.2 Simulation Executive

The simulation executive manages event processing and advancements in simulation time. Like the executive in a sequential simulation, it processes events from an event list in timestamp order, and updates the current simulation time of each LP with each event it processes. The executive also determines when the simulation is completed. The pseudo code for the executive is shown in Algorithm 2.

2.5.3 Fix Up Computation

A more detailed description of the fix up computation, that occurs at the end of each round of simulation is described next.

Update and Communication Strategies

At the end of the simulation run, each LP constructs a fix up message. This fix up message consists of two components. The first component is the changed schedules for the last flights flown in the LP for each aircraft. All of the LPs maintain a list of the aircraft flown

Algorithm 2 Simulation executive.

```
1: procedure IsRunning
2:     if sim_time = end_time then
3:         return TRUE
4:     else
5:         return FALSE

6: procedure StartExecutive
7:     ExecutiveThread()

8: procedure ScheduleEvent(evt)
9:     event_list.insert(evt)

10: procedure ExecutiveThread()
11:     while event_list.size() > 0 do
12:         cur_evt ← event_list.pop()
13:         cur_evt.process()
14:         sim_time ← cur_evt.time()
15:     return
```

throughout the day, so this is simply an array of delay times (T_{delay}). More specifically, each delay time id defined as the time difference ($T_{delay} = T_{act} - T_{org}$) between the original scheduled arrival time (T_{org}) for the flights and the actual arrival time (T_{act}) after the simulation run. The other component is the queueing status of each airport at every time stamp. This means the wait time when a new flight arrives to the airport at that specific time point. In other words, based on the queueing status, we know how much time each aircraft needs to wait. These two fix up messages are shown in Figure 2.6. In this specific notional example, there are three different aircraft in the system and each LP is simulating 9 minutes each. They are used for several different flights and the fix up message holds the last flights' delay for each LP. In the figure, each box indicates *'Aircraft (Last flight number in the LP) : delay'*. For instance, aircraft A's last flight in LP 1 was delayed 5 minutes and aircraft B's last flight in LP 1 has no additional delay. In case of the airport queueing status, LP 1 has a sequence of numbers represents the delay time in the runway queue at a specific time for each airport. In this example, the aircraft A is supposed to arrive five minutes

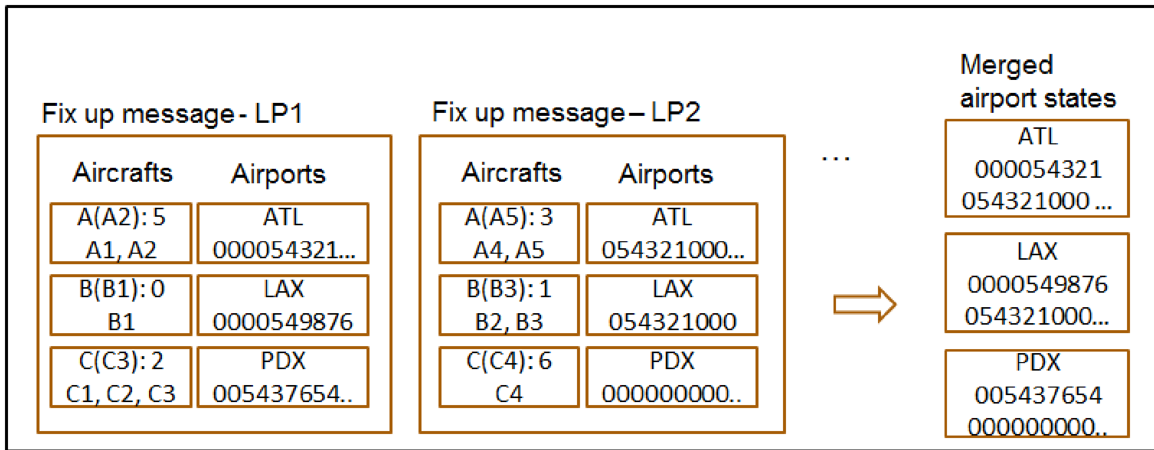

Figure 2.6: Communication of fix up messages.

later from the beginning time of LP 1. If we assume that the landing takes 5 minutes, the expected waiting time will be five, four, three, two and one for the next five minutes. Here, the next step is iterating through the updated aircraft in the LP which have different arrival times from the previous LP. When it goes through the aircraft's flights, it removes flights from the queue at the original schedule spot and adds it to the new spot in the queue. As a result of the evaluation, one can determine whether a new simulation run is necessary in the specific LP or not. If all the changes are simply updating the arrival times and do not propagate to the next flight, the LP does not need to re-compute the simulation. On the other hand, if the evaluation results in a change to the upcoming flights, another round of simulation is run based on the updated flight times. Depending on the algorithms, the evaluation of updates can be done sequentially or collectively. This means that the fix up messages are either sent to the next LP or collected into the first LP.

This modification of queue status is illustrated in Figure 2.7. The aircraft A's update message from LP 1 to LP 2 was that 5 minutes delay was added to the original scheduled arrival. Therefore, LP 2 does the evaluation step by firstly removing the flights at the original place which is the second time spot in LP 2's status. This removes all expected delays for five minutes. Then, it adds the flight into the delayed time slot which is five minutes later. The expected delay was zero for that time slot but it has 5 minutes expected

delay at that time point after evaluation.

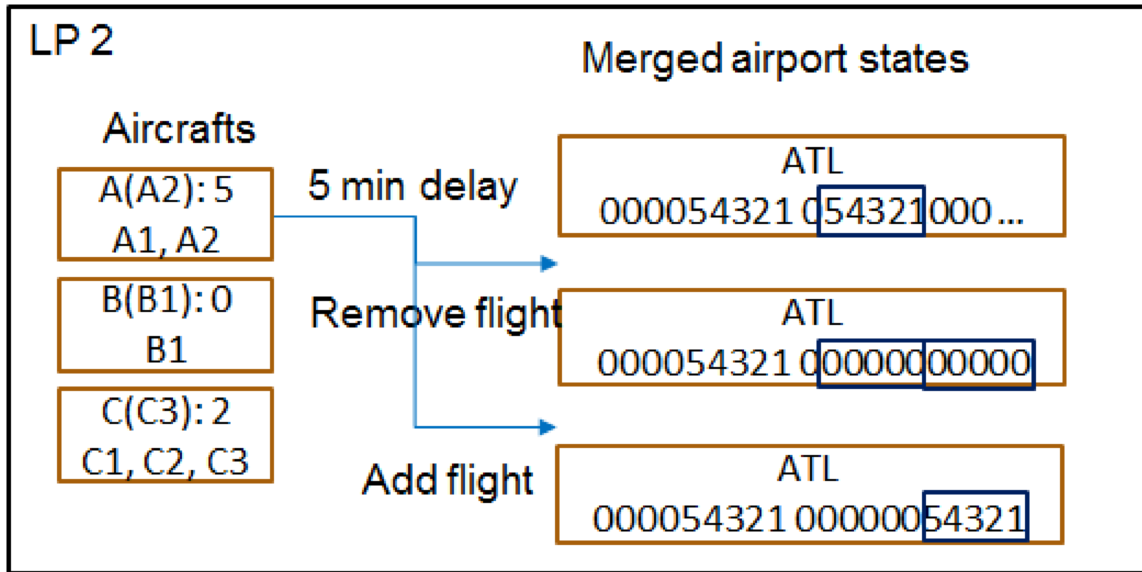

Figure 2.7: Evaluation of update for the fix up computation.

A high level view of the two different communication strategies is illustrated in Figure 2.8 and Figure 2.9. Figure 2.8 shows a fix up computation that is accomplished separately by a sequential communication that propagates corrections in earlier time intervals (LP_i) to later ones (LP_{i+1}). This approach ensures that all communications among the LPs happen only once. Based on the fix up message received from the previous LP, it might or might not need to do the simulation run again. However, the simulation results should be final at every LP, so it avoids unnecessary rounds of fix up computations. On the other hand, because this fix up computation is sequential, there could be performance degradation especially if there are a large number of LPs.

Figure 2.9 shows the collective computation of the output variable by gathering all the results into the first logical process. This utilizes the collective API from MPI such as *MPI_Gather* and *MPI_Scatter*. By utilizing this collective approach, we can reduce the number of communications to a constant number instead of the number proportional to the number of LPs. Here, two rounds of communications are required for one fix up computation. Because general implementations of the collective APIs of MPI roughly has

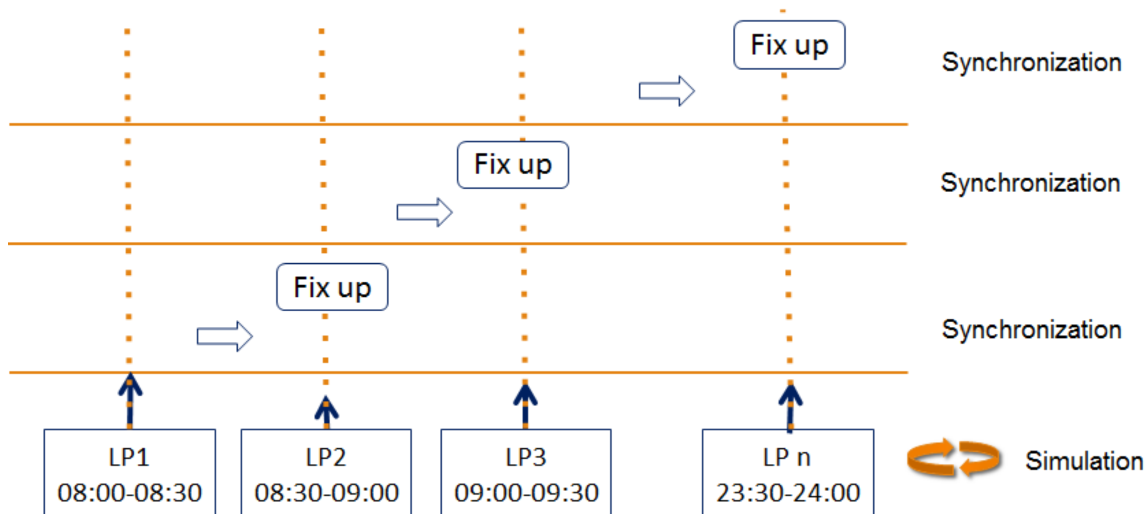

Figure 2.8: High level view of the fix up computations - sequential operation.

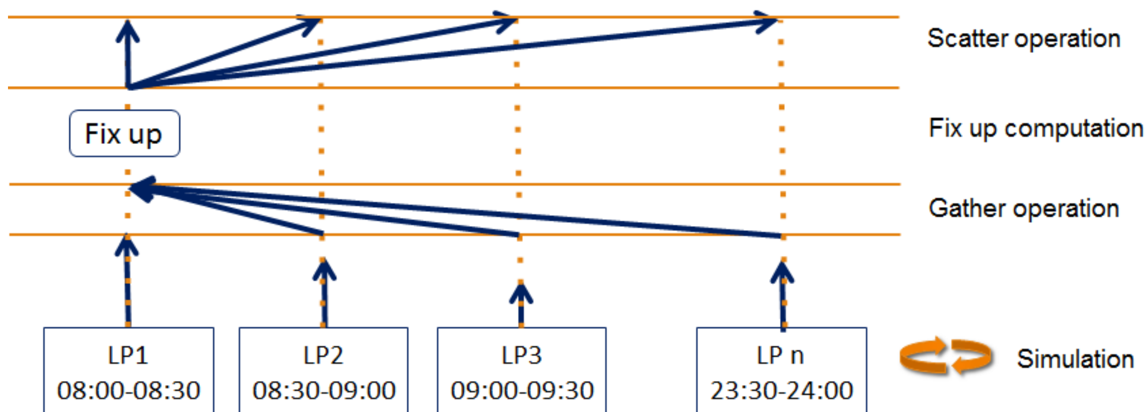

Figure 2.9: High level view of the fix up computations - gather operation.

a complexity of $O(\log n)$, it could reduce the burden of communications significantly. After collecting all the updated information into one LP, it can quickly evaluate all the states need to be fixed. In the next part of the section, the criteria for rerunning the simulation after the fix up computations are described.

Rerunning the Simulation

When the updated events from the LP processing the previous time segment, i.e., the *preceding LP*, have no impact on the subsequent event scheduled by the updated event, we can

simply update state variables affected by the changed event in order to ensure computed statistics are correct. This is when the updated next departure time based on the queue evaluation is still the same as the original scheduled departure time. This is computed by Equation 2.1. In this case the fix up computation will not propagate beyond the LP and the update will remain within this LP.

If the updated events affect the scheduling of the subsequent departed events within an LP, another round of the simulation is needed to correct the differences. Based on the updated arrival time, all the affected time pieces are corrected by running the simulation again. At the end of the simulation, another round of fix up computation is performed for the evaluation. The explained procedure is presented in Algorithm 3.

Algorithm 3 Fix up computation.

```
 1: procedure CommunicateFixupMsg
 2:    Msg.flights ← flight_result
 3:    Msg.queue_state ← airport_state
 4:    SendAndReceive(Msg)

 5: procedure EvaluateUpdate
 6:    queue_states ← Msg.queue_state
 7:    for flight_update ∈ Msg.flights do
 8:       UpdateQueue(flight_update)
 9:       if IsDepChanged(flight_update) then
10:          isRerunNeeded ← TRUE

11: procedure IsDepChanged(offset)
12:    if offset + G^d − b^sg > 0 then
13:       return TRUE

14: procedure PostFixup
15:    isRerunNeeded ← FALSE
16:    CommunicateFixupMsg()
17:    EvaluateUpdate()
18:    return isRerunNeeded
```

2.5.4 Workload Distribution

It is important to balance the computational workload across the different LPs because the slowest LP will dominate the execution time. In order to achieve this, we may partition the time domain so that all the LPs have a similar amount of traffic to model. In order to measure how this balancing affects the efficiency of the time-parallel simulation, two workload distribution algorithms were tested. The initialization portion of the algorithm is presented in Algorithm 4. Based on the options which are "SAME_TIME" and "SAME_TRAFFIC", the workload is distributed across the LPs.

2.6 Experimental Results

In order to investigate performance and various algorithm alternatives, the time-parallel simulator algorithm was implemented. The main focus of these experiments was to evaluate the initial workload distribution approaches as well as the variations on the fix up computation algorithms in addition to evaluating the overall speed up obtained by the time-parallel simulation of the NAS.

2.6.1 Air Traffic Scenario and Data

To evaluate the simulation, two simulation scenarios were utilized. At first, computed results by the simulation without any capacity restrictions were compared with a perfect schedule of historical data. When the departure schedules of all the flights from one specific date are given, the simulation model runs all the traffic during the day in the absence of airport capacity limitations. This scenario should produce simulation results that exactly match scheduled arrival times. In the validation test, the developed simulation model yielded the correct results except in some instances where inconsistencies in the historical data led to differences in model predictions.

The other scenario used for the verification is a real traffic data with a capacity limita-

Algorithm 4 Initialization.

```
 1: procedure DISTRIBUTE(option)
 2:   if option = SAME_TIME then
 3:     duration ← total_time/num_lps
 4:     start_time ← lpID × duration
 5:     end_time ← start_time + duration
 6:   else if option = SAME_TRAFFIC then
 7:     lp_traffic ← total_traffic/num_lps
 8:     time ← 0
 9:     while traffic < lpID × lp_traffic do
10:       traffic ← traffic + traffic_at_time
11:       time ← time + 1
12:     start_time ← time
13:     while traffic < (lpID + 1) × lp_traffic do
14:       traffic ← traffic + traffic_at_time
15:       time ← time + 1
16:     end_time ← time

17: procedure DOINITIALIZE()
18:   schedule ← LoadData(start_time, end_time)

19: procedure STARTSIMULATION()
20:   Distribute(option)
21:   DoInitialize()
22:   isRerunSimulation ← TRUE
23:   while isRerunSimulation do
24:     Barrier()
25:     executiveHandle ← StartExecutive()
26:     while executiveHandle.IsRunning() do
27:       Wait()
28:     Barrier()
29:     isRerunSimulation ← PostFixup()
30:   return
```

tion in the airport because of the weather conditions and congestions of the NAS. It might cause some additional round of computations. Because the assumption that the NAS has unlimited capacity for the air traffic is no longer valid, there exist air traffic delays. Depending on the weather condition and the congestion of the NAS, the flight times will be different from the expected flight times used for the initial simulation. This will cause a discrepancy between the original flight schedules and the updated schedules generated by

the simulation. As a result, there are more severe situations for the state-matching problem.

TranStats from the *Bureau of Transportation Statistics* is a large transportation database maintained by the U.S. Department of Transportation. [80] It collects various historical data related to the transportation systems in the United States including aviation, maritime, and highway data. Among these, the airline on-time performance database stores all the airline information, original departure/arrival flight schedules, actual departure/arrival times, causes of delays, etc. of all domestic flights in the U.S. The database is publicly available and can be downloaded by specifying a specific time including year and month.

For this experiment, air traffic data for August 19, 2016 was used. This date included inclement weather conditions in the continental U.S. The air traffic on that day has sufficiently complex patterns and includes flight delays which are providing a realistic, challenging test case to evaluate the parallel simulation algorithm. Across the 297 airports within the NAS, 16,614 flights for that day were simulated.

2.6.2 Experimentation Environment

Hardware configuration

For the experimentation, a parallel machine using Intel's Xeon® CPU (E5-2699 v4, 2.20 GHz, Broadwell micro-architecture) was utilized. It has two sockets and each socket has 22 processing cores. Therefore, up to 44 physical cores were utilized in the experiments. Also, each socket has 64 GB DDR4 memory installed so in total 128 GB of memory. Because all the experiments are performed in a single node, there is no network connectivity for the experiments. Each thread corresponds to an LP and executes on one physical core. To maximize cache hit rates, each physical core executes only one LP during the simulation run.

Software Configuration

C++11 standards and libraries were utilized for the implementation on top of CentOS v7.2. Even though this experiment was performed in a single machine configuration, it has been implemented using *MPICH* to enable later extension to a multi-machine clustering environment. The version of the *MPICH* used in this implementation is v3.2 and the version of *gcc* used is v4.8.5.

2.6.3 Fix Up Computation Comparison

Two different fix up computation algorithms described in Section 2.5.3 were implemented and tested. In these experiments, the wall clock time required for each LP was measured and the slowest time is used in computing speed ups. Figure 2.10 shows the speed up as the number of LPs was varied for the two different fix up computation algorithms. As seen from the graph, the collective fix up computation algorithm executes faster than the sequential fix up computation. This can be explained by the communication overheads in the fix up computation. There should be $n-1$ communications among all LPs which results in $O(n)$ communication complexity. On the other hand, all the collective operations used in this implementation have $O(\log n)$ communication complexity.

One further observation is that the parallel efficiency drops significantly after a certain number of LPs are reached. These results are compared with the speed up when no fix up computation is performed, as shown in Figure 2.11. With no fix up computation, the simulation, of course, produces incorrect results, however, this simulation provides an upper bound on performance and illustrates the performance degradation that results from the fix up computation. As expected, the speed up without fix up computations yields almost ideal, linear speed up, and sometimes even shows super linear speed up. We believe this is because more cache memory is available as the number of processors (LPs) increases.

Returning to the time-parallel simulation, we hypothesize that the performance degradation as the number of processors is increased is because the simulation computation

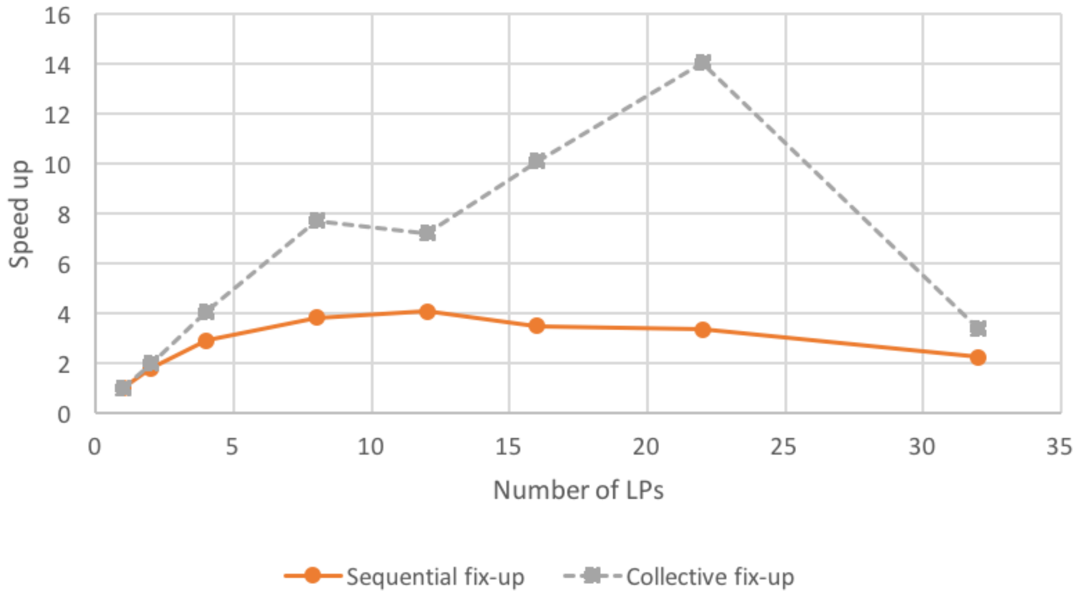

Figure 2.10: *Sequential fix-up* computation vs. *Collective fix-up* computation.

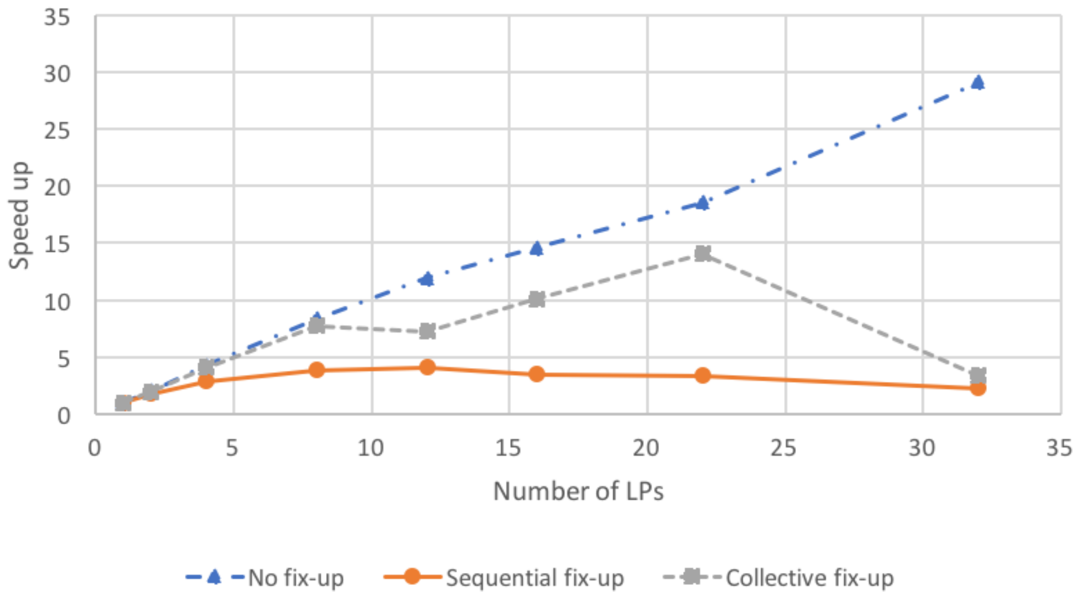

Figure 2.11: *Sequential fix-up* computation vs. *Collective fix-up* computation with *No fix-up* computation.

models a single day of traffic, so the amount of computation in each LP decreases as more LPs are added. For a large number of LPs the fix up computation requires additional communication, incurs a more significant overhead. This explains the performance degradation

in the time-parallel simulation as the number of LPs becomes large. In this experiment, the policy which distributes the same amount of traffic is used.

In order to verify that the degradation of parallel efficiency is because of the small size of the original computations, another experiment was performed that included artificially enlarged event computations. This is done by adding a spin-loop which computes 1,000,000 times integer additions in each event computation. Figure 2.12 shows the result of this experiment. In this experiment, both the sequential fix up and collective fix up computations show good speed ups. The collective fix up computation shows slightly better performance. These speed ups can also be compared with the speed up without fix up computation in Figure 2.13. All of these show significant speed ups demonstrating the potential performance improvement of the time-parallel algorithm.

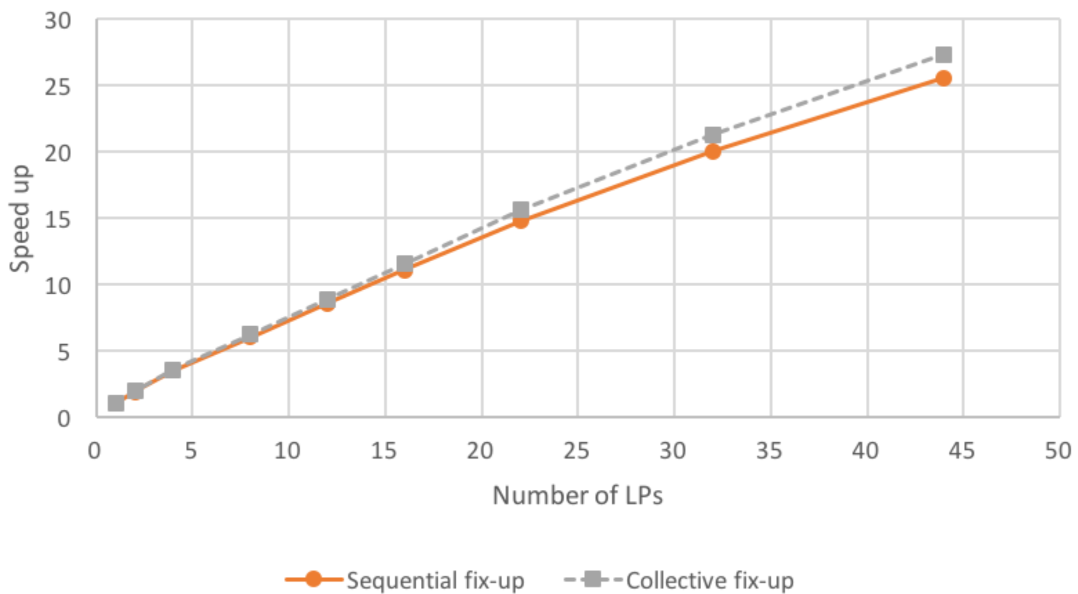

Figure 2.12: *Sequential fix-up* computation vs. *Collective fix-up* computation.

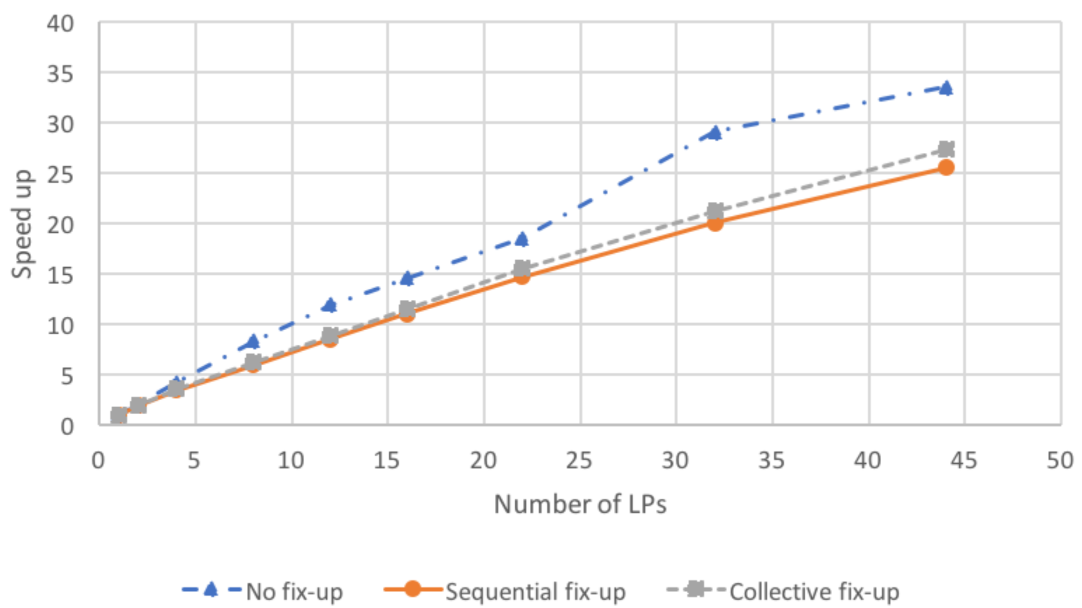

Figure 2.13: *Sequential fix-up* computation vs. *Collective fix-up* computation with *No fix-up* computation.

2.6.4 Workload Distribution Comparison

For another experiment, the different initial workload distribution policies described in 2.5.4 were used in the simulation. In both cases, they scale well as shown in Figure 2.14. And, the case with the same amount of traffic shows better performance than the case with the same time distribution. This result shows that it is important to evenly distribute workloads across the LPs. In the case of 44 LPs, 1.65 times more speed up can be acquired by distributing workload uniformly. For the comparison with the speed up without fix up computation, Figure 2.15 shows all three graphs. For this experiment, the collective fix up computation is used.

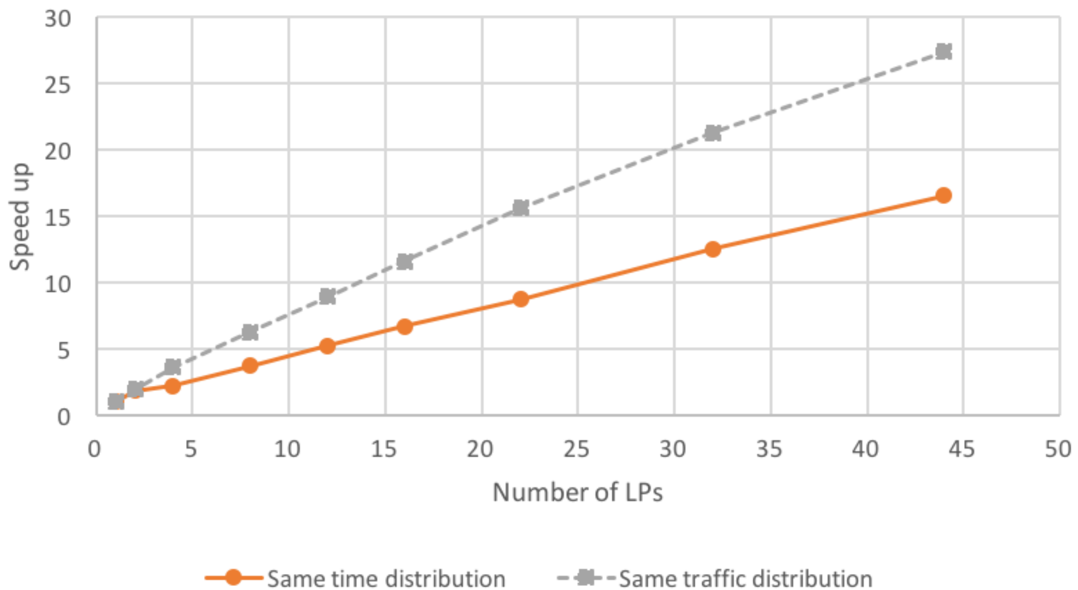

Figure 2.14: Same time intervals vs. Same amount of traffic.

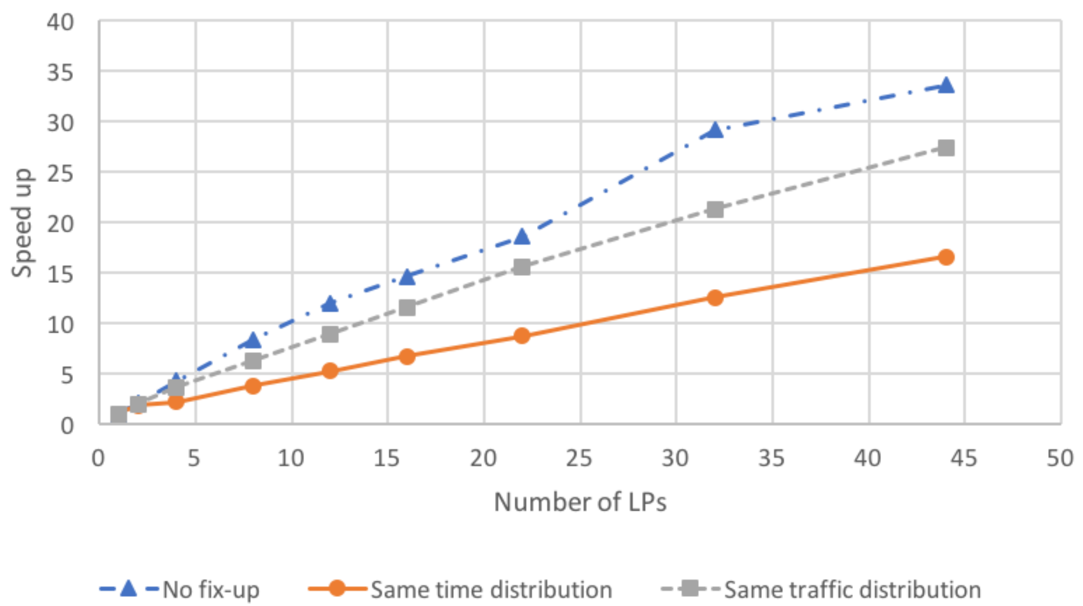

Figure 2.15: Same time intervals vs. Same amount of traffic with No fix up computation.

2.6.5 Limited Airport Capacity Comparison

For the last experiment, the actual flight times including traffic delays of the real world data were utilized. With this setting, the same number of LPs were tested to measure the performance. Figure 2.16 shows speed ups with different numbers of LPs. The lowest line shows the speed up for this specific traffic scenario and the middle line shows the speed up for the ideal schedule scenario. From the figure, it is seen that the time-parallel simulation exhibits a performance degradation in the scenario compared to the ideal traffic situation. It shows a good scalability until 4 LPs, but the efficiency drops for 8 LPs. The reason is that the simulation runs require multiple rounds because of the state-matching problem. As can be seen in Table 2.1, there are multiple rounds of the simulation to fix the incorrect simulation results which are computed based on the original flight schedules. In the case of 44 LPs, the number of simulation rounds go up to 9 rounds. Another observation from this scenario is that the number of simulation rounds does not increase linearly in proportion to the number of LPs. Up to 4 LPs, only one round is needed, but the number increases to 4 rounds at 8 LPs. 12 LPs also shows 4 rounds, but increases to 6 rounds at 16 LPs. This is an expected characteristic of the time-parallel simulation of the NAS. Because of the hub-network model, some LPs can absorb the propagation of delays well. For example, when we divided the entire time into 4 LPs, those 4 LPs can absorb all the delays within themselves. So, no additional simulation rounds are needed. But, the division of 8 LPs results in many state mismatches. In this case, a time period containing concentrated traffic is split across different LPs.

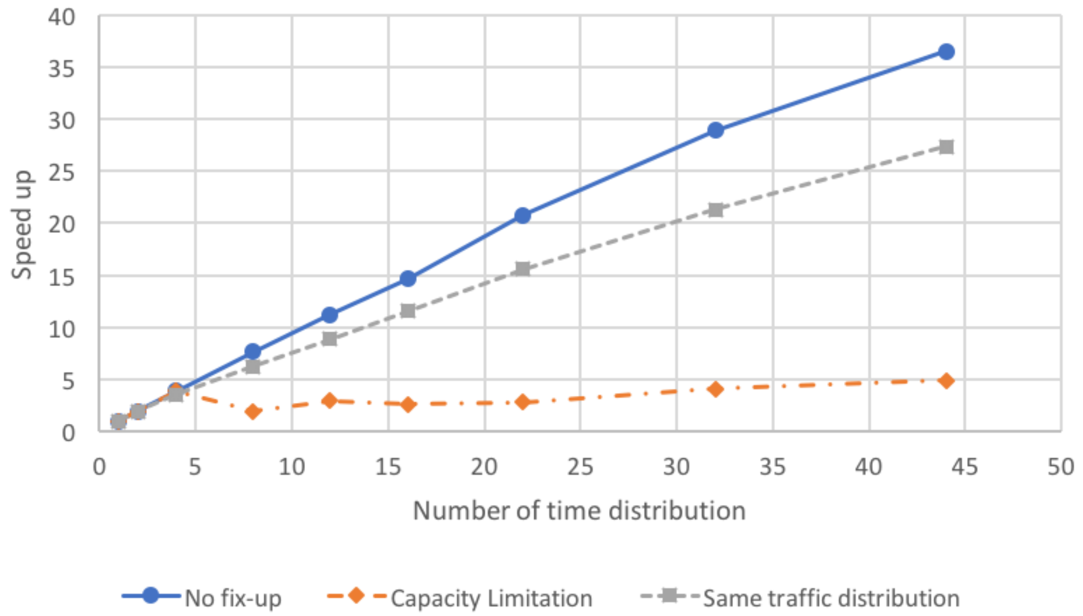

Figure 2.16: Capacity limits vs. No capacity limits with No fix up computation.

Table 2.1: The number of simulation rounds for the time-parallel simulation.

Number of LPs	Number of simulation rounds
1	1
2	1
4	1
8	4
12	4
16	6
22	8
32	8
44	9

2.7 Conclusion

Time-parallel simulation offers a new approach to accelerating air traffic simulations of the NAS. The schedule-driven nature and other aspects of the air transportation schedules such as the inclusion of buffer times makes it an application that appears to be well suited for the time-parallel simulation approach. Therefore, a time-parallel simulation algorithm for simulating the NAS was proposed. Experimental results illustrate that this algorithm can achieve high level of parallelism and speed up for this application. Preliminary measurements indicate that the collective fix up computation yields better performance than the sequential fix up computation. It is also seen that even distribution of workloads across the LPs is necessary for efficient parallel simulation. These results suggest that time-parallel simulation offers a viable approach to accelerating certain air traffic simulations. At the same time, it also shows some limitations for a scenario which involves some air traffic delays. It causes multiple rounds of simulations and reduces the efficiency of the simulation.

There are several open avenues for future research. In order to mitigate the performance degradation problem, an improved algorithm such as dynamic time distribution needs to be added. Also, an algorithm to find optimal splitting points in the simulation is needed. Certain algorithmic improvements of the simulation method may yield additional performance enhancements. Approaches using both spatial and time-parallel simulation is described next. Realization exploiting SIMD architectures is another area of future research. Further, a shared memory model such as OpenMP can be integrated with the MPI based simulator and may achieve better speed up in many core systems; there are many physical cores in a single machine so it will be more efficient to exploit shared memory among LPs where possible, and rely on MPI for communication between LPs mapped to different machines. Finally, the simulations could be adopted for use in real-time symbiotic simulation applications.

CHAPTER 3
EXPLOITING SPATIAL PARALLELISM IN AIR TRAFFIC NETWORK SIMULATION

3.1 Overview

As explained in the earlier chapters, there are two different types of parallel simulation algorithms for air traffic network analysis. By distributing state variables across multiple LPs, a spatial parallel simulation can be implemented. On the other hand, by distributing time segments across multiple LPs, a time-parallel simulation can be created. In Chapter 2, the main focus was on the methodology to implement a time-parallel simulation. In this chapter, a novel approach to the simulation of the air traffic network system is proposed by combining spatial and time parallel simulation algorithms. By simultaneously using two parallel algorithms, a higher level of parallelism can be achieved. In particular, it is expected that spatial parallel simulation algorithms can supplement time-parallel simulation algorithms to improve scalability. Although, there has been a significant amount of research in spatial parallelism, combining both spatial and time parallelism has not been widely studied. In doing so, it is important to coordinate the different LPs residing in different time and spatial zones efficiently. To the author's knowledge, this is the first attempt to apply a combined parallel simulation approach to NAS applications.

This chapter consists of the following sections. In Section 3.2, the proposed algorithm to combine spatial and time parallel simulations is described as well as the spatial parallel algorithm is realized in this research. In Section 3.3, the experimental environment and results are presented using the developed spatial and time parallel simulation program. Finally, the conclusion for this chapter is presented in Section 3.4.

3.2 Combining Time and Spatial Parallelism

3.2.1 Time Warp Parallel Simulation of Air Traffic Networks

There are two main approaches to partition a simulation for parallel execution known as task parallelization and domain decomposition [81]:

- **Task parallelization** - The different functional modules of a traffic simulation model (flight, airport gates, airport controllers, en route traffic controllers) are mapped to different LPs. The advantage of this approach is that it is straightforward to implement. Each LP need only implement its specific functionality. The disadvantage of this approach is that the slowest module will dominate the execution speed. For example, if one certain airport controller has many more computations compared to other components, it will become a bottleneck.

- **Domain decomposition** - In this approach, each LP covers a specific geographic region. Inside the region, different kinds of functional modules reside. Each LP performs all of the individual activities contained with the region. In this approach, all LPs have the same logic but use different data.

Here, the domain decomposition is used for two reasons. First, parallel efficiency is a concern in the task parallelization approach as discussed above. Second, the domain decomposition reduces the amount of communications. With task parallelization, all modules must communicate with each other. This may result in much communication between LPs. On the other hand, the domain decomposition approach only requires communications among spatially neighboring LPs. This is closer to the actual communications in the air traffic system. Here, the simulation is constructed as in Figure 2.5.

Domain Decomposition of the NAS

For the domain decomposition of the NAS, we need to understand the sequence of traffic control operates in order to determine how the domains of the NAS are constructed. The sequence of air traffic management operations across the NAS for a flight is shown in Figure 3.1. When a flight is ready to depart, the Airport Traffic Control Tower (ATCT) gives a departure clearance. Then, ATCT controls the flight's ground route so that it can reach a runway. Once, the runway is ready, ATCT allows the flight to take off. After taking off, it is controlled by ATCT within a 5 mile range. Then, it is handed over to the Terminal Radar Approach Control (TRACON). TRACON manages the flight until it reaches the altitude of 10,000 feet or a distance of 40 miles from the airport. From that point, the flight is managed by the Air Route Traffic Control Centers (ARTCC) which control the regions through which the flight travels. Depending on the flight route, it can pass through several different ARTCCs. When the flight approaches the destination airport, the reverse sequence takes place. When the flight is within a range of 40 miles or its altitude is less than 10,000 feet, TRACON takes control of the flight. ATCT gives a clearance for landing and the flight lands at a runway following the control of ATCT. Finally, the flight travels to a gate to unload passengers.

There are 22 ARTCCs in the U.S. NAS. Each ARTCC manages its assigned region. 20 ARTCCs are in the continental US and the other two ARTCCs are controlling the remote regions. One remote ARTCC is controlling the Alaska region and the other controls Hawaii region. Inside the area controlled by each ARTCC, there are many airports. The flights flying near the airport, especially for taking off and landing, are managed by ATCT as explained earlier. The actual ARTCC regions of the NAS is shown in Figure 3.2.

One way to decompose the NAS across multiple LPs is to assign each ARTCC zone to one LP. In that case, the maximum number of spatial regions is only 22. Another approach

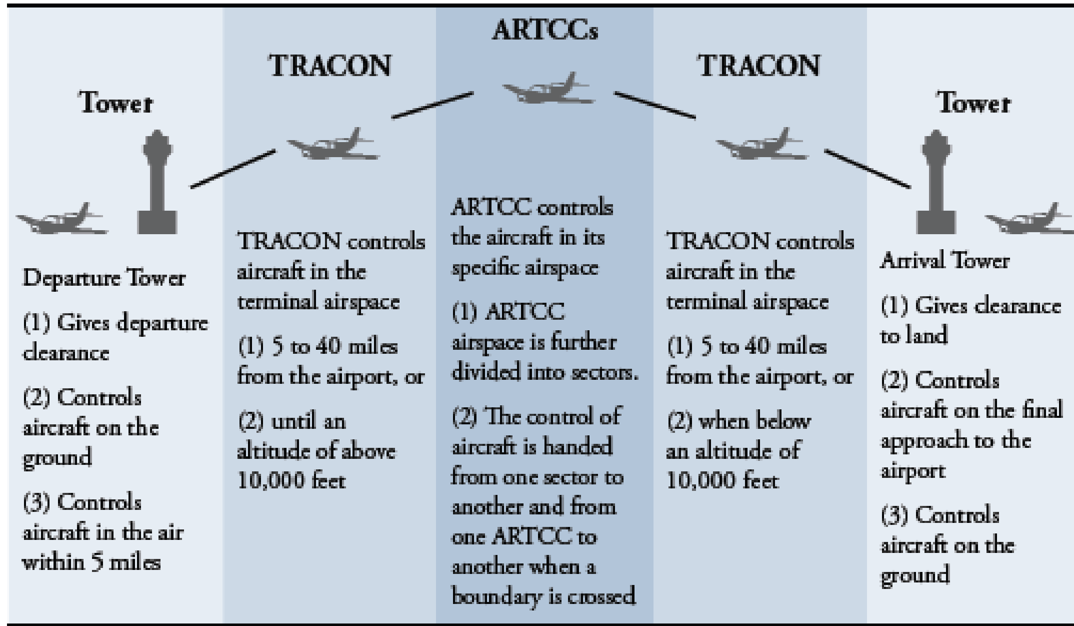

Figure 3.1: Air traffic flow chart [82].

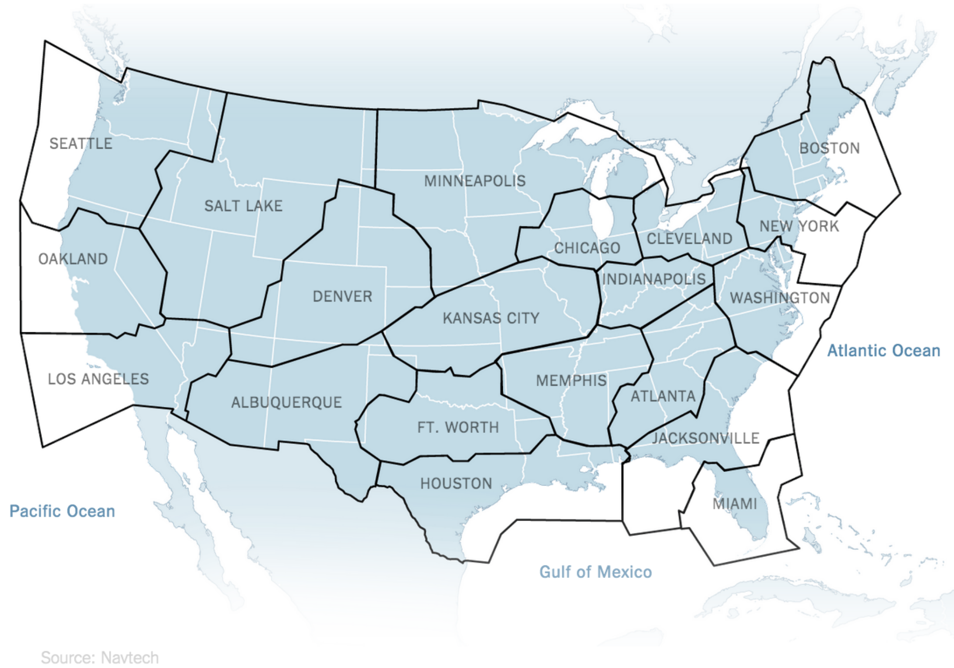

Figure 3.2: ARTCC map [83].

is to place each U.S. state into a separate LP, in this case, resulting in 50 LPs. Based on the flight schedule data used here, American Samoa (AS), U.S. Virgin Island (VI) and Puerto Rico (PR) are also included increasing the maximum number of LPs to 53. In addition to this pre-defined division of the NAS, we also need to define a flexible way to divide the NAS in order to measure how well the parallel simulation scales. As a first step to implement that, all the 53 possible states are lined up and split into the LPs by prorating them. One empirical fact is that the horizontal grouping of the states is better than the vertical grouping of the states. Because of the time difference between the eastern side of the NAS and the western side of the NAS, the vertical grouping may cause the LPs handling the western side of the NAS to be idle during the early morning time period. Similarly, the vertical grouping makes the LPs handling the eastern side of the NAS to be idle during the night time. Therefore, the horizontal grouping can distribute the flights more evenly. Table 3.1 shows one example of divisions of states across LPs. In the table, each LP spans across the horizontal regions.

Table 3.1: Example of domain decomposition of the NAS - 5 LPs.

LP ID	States
LP0	AS, VI, HI, PR, LF, TX, LA
LP1	CA, AZ, NM, OK, AR, MS, AL, GA, SC, TN, NC
LP2	NV, UT, CO, KS, MO, KY, WV, VA, DC, MD, DE
LP3	OR, ID, WY, NE, IA, IL, IN, OH, PA, NJ, CT
LP4	SD, MT, ND, MN, WI, MI, NY, MA, NH, ME, RI, VT, AK

Components of Time Warp Simulation

The time-parallel simulation algorithm proposed in the previous chapter divides time axis and assign each time intervals to different LPs. Time Warp simulation is well fitted to this time-parallel simulation because too much optimism which might cause too many rollback operations is naturally prevented by the time splitting. The sequential simulation algorithm

Algorithm 5 GVT computation.

```
1: procedure INITIATEGLOBALCONTROL
2:     if history_queue.size() > threshold then
3:         SendMessage(global_sync)
4:         GlobalControl()

5: procedure GLOBALCONTROL
6:     while sentMessage ≠ receivedMessage do
7:         ReceiveAllMessage()
8:         AllReduceSum([sentMessage, receivedMessage])
9:     localMinimumTime ← event_list.head().time()
10:    globalMinimum ← AllReduceMin(localMinimumTime)
11:    CommitGVT(globalMinimum)

12: procedure COMMITGVT(GVT)
13:    globalVirtualTime ← GVT
14:    stateHistory.free(GVT)
15:    while processedMessage.size() ≠ 0 && processedMessage.begin().time() < GVT
   do
16:        processMessage.pop()
17:    logger.commitLog(GVT)
```

discussed in Section 2.5.2 can be modified to enable execution with Time Warp. First, a rollback mechanism is needed when the LP detects any out-of-order execution of event. This includes an anti-message mechanism to roll messages back. Second, there needs to be a global time management mechanism added. These additions are described below.

- Global Virtual Time

  Global virtual time enables LPs to commit I/O related operations and reclaim memory. Here, a synchronous GVT algorithm is used. The pseudo-code for the algorithm is presented in Algorithm 5:

  1. (Line 1 - 4): Each time, an LP processes an event, it checks if any memory must be reclaimed. If memory needs to be reclaimed, the LP send messages to the other LPs to initiate a global control operation.
  2. (Line 5 - 11): Once a global control operation is initiated, all LPs synchronize

to determine GVT value. All LPs wait until all messages they have sent have been received at the destination LP. After confirming that all messages have been delivered, each LP computes its local minimum time value. Then, all LPs use an all reduce operation to compute a GVT value.

3. (Line 12 - 17): Finally, the LPs free any memory with time value earlier than GVT. Also, they delete all saved messages with time stamp less than GVT. Log messages are also saved into files.

- Rollback

  In addition to the GVT algorithm, LPs need to have a mechanism to correct any out-of-order event execution. This is accomplished using a rollback operation. The operation rolls back the state of the LP to a most recent simulation time before that of the message causing the rollback. Then, the LP execution can resume from that time. The Time Warp simulator saves each LP's state periodically to enable rollback. It is also necessary to cancel messages sent by rolled back computations. This is done using the anti-message message cancellation mechanism. An anti-message is same as the original message, but contains a flag indicating it is an anti-message. Whenever an LP needs to cancel a message, it sends an anti-message to the original destination LP. In the receiving LP, if it has been processed, the LP is rolled back to the most recent simulation time earlier than the canceled message time stamp. The LP cancels the original message if it has not been processed. The detailed algorithm is presented in Algorithm 6:

  1. (Line 1 - 7): Rollback operations are initiated by a message received from another LP. If an LP receives an anti-message, it calls the message annihilation routine. On the other hand, if it receives a positive message, it checks the time stamp of the message. If the time stamp is lower than the current simulation time of the LP, it executes the rollback routine.

Algorithm 6 Rollback algorithm.

```
1: procedure SCHEDULEEVENTFROMREMOTE(evt)
2:     if evt ⟹ ANTI_MSG then
3:         Annihilate(evt)
4:     else
5:         if evt.time < simTime then
6:             Rollback(evt.time)
7:         event_list.add(evt)

8: procedure ANNIHILATE(evt)
9:     if evt.time < simTime then
10:         Rollback(evt.time)
11:     else
12:         event_list.delete(evt)

13: procedure ROLLBACK(time)
14:     states ← statesHistory(time)
15:     while processedMessage.size() ≠ 0 && processedMessage.end().time() > time do
16:         message ← processedMessage.pop()
17:         event_list.add(message)
18:         sendAntiMessage(message)
19:     simTime ← message.time()
```

2. (Line 8 - 12): The annihilation routine first compares the time stamp of the anti-message with the current simulation time at the LP. If it is lower than the simulation time, it executes the rollback routine. On the other hand, if the original positive message has not been processed yet, it simply cancels the original message. In this case, the annihilation does not cause additional rollbacks.

3. (Line 13 - 19): The rollback routine iterates through the list of processed events from the most recent to the least recent event. Any events with time stamp less than the rollback time are removed from the processed event list and added back to the scheduled event list. At the same time, the LP also sends anti-messages for any sent messages. After finishing the iteration loop, the simulation time of the LP is set to the time stamp of the last message added to the event list. Then, the LP resumes its execution from that event.

Algorithm 7 Time warp simulation executive.

```
 1: procedure SENDTERMINATEMESSAGE
 2:     localState ← Terminate
 3:     idx ← 0
 4:     while idx < NumOfLPs do
 5:         SendMessage(localState)
 6:         idx ← idx + 1

 7: procedure RUNTIMEWARPSIMULATION
 8:     while CheckTermination() = FALSE do
 9:         SendAllMessages()
10:         ReceiveAllMessages()
11:         SaveStateData()
12:         event ← event_list.pop()
13:         simTime ← event.time()
14:         event.process()
15:         processedMessage.add(event)
16:         InitiateGlobalControl()
17:         GlobalControl()
18:     SendTerminateMessage()
19:     CommitGVT(simTime)

20: procedure CHECKTERMINATION
21:     if event_list.size() > 0 then
22:         return FALSE
23:     else if checkOtherLPDone() = TRUE then
24:         return TRUE
25:     else
26:         SendTerminateMeaage()
27:         return FALSE
```

- Time Warp Simulation Executive

  Based on the GVT and rollback algorithms, the Time Warp simulation was constructed. Each LP processes scheduled events and communicates with other LPs to manage the execution until the simulation reaches to the termination condition. The pseudo code is shown in Algorithm 7.

  1. (Line 1 - 6): When an LP process all of its events, it sends a termination messages to all the other LPs.

2. (Line 7 - 19): In the main simulation routine, each LP iterates over all the events it has received. At every time, it starts with checking the termination status using the 'CheckTermination()' routine. If the termination status is satisfied, it stops the iteration and sends a final termination message to all the other LPs using the 'SendTerminateMessage()' routine. If the LP still has more events to process or receives messages from the other LPs, the LP starts the event processing routine. First, it sends and receives all the pending communication messages. Then, it saves the current status of the LP. The LP then removes the smallest time stamped event from the 'event_list' and processes that event. At the same time, the time stamp of the LP is also updated. After the event is processed, it is stored in the 'processedMessage' list. Finally, the LP initiates the global control routine as discussed earlier to update the GVT. Once the iteration has completed, the LP sends termination confirm messages if the execution has completed and commits I/O operation and release memory.

3. (Line 20 - 27): In the termination condition check routine, the LP checks if the 'event_list' is empty. If there are any events to be processed, the LP is not ready to finish. If there is no more event to be processed, it checks if it has received the termination messages from all the other LPs. If it received termination messages from all the other LPs, it can finish. Otherwise, it sends the termination messages to the other LPs and waits until the other LPs complete their jobs.

Integrated Simulator

By combining all the elements discussed in this chapter and Chapter 2, a time and spatial parallel simulation can be created. After initializing the LPs for each time interval, they execute the simulation as if they were executing a spatial parallel simulation. They synchronize with the other LPs simulating the same time interval. Once, they complete the spatial parallel simulations of their assigned time interval, the time parallel simulation algorithm

Algorithm 8 Integrated time and spatial simulation.

```
1: procedure RunSimulation
2:     skipThisLP ← FALSE
3:     completed ← FALSE
4:     while completed = FALSE do
5:         [skipThisLP, completed] ← CheckCompleted()
6:         if skipThisLP ≠ TRUE then
7:             GenerateTraffic(time_interval, domain)
8:             RunTimeWarpSimulation()

9: procedure CheckCompleted
10:    global_delay_data ← []
11:    delay_data ← localStatus[delay]
12:    AllReduceSum(delay_data, spaceComm)
13:    AllGather(delay_data, global_delay_data, timeComm)
14:    if LP_ID ≠ 0 And global_delay_data[LP_ID - 1] ≠ 0̄ then
15:        completed ← FALSE
16:        UpdateScheduleglobal_delay_data
17:    else if time_round > time_order then
18:        skipThisLP ← TRUE
19:    return [skipThisLP, completed]

20: procedure GenerateTraffic(time_interval, domain)
21:    traffic_data ← ReadTrafficData()
22:    idx ← 0
23:    while idx < traffic_data.size() do
24:        if traffic_data[idx] ∈ time_interval And traffic_data[idx] ∈ domain then
25:            event_list.add(traffic_data[idx])
26:        idx ← idx + 1
```

executes the fix up computations. They evaluate if an additional round of the simulation is needed. If there is need for another round of the simulation, the LPs update initial conditions using an updated flight schedule, and start another round of the simulation. Then, each time interval initiates a new spatial parallel simulation with the updated flight schedules. These routines are repeated until all the simulation resolve the state-matching problem.

1. (Line 1 - 8): When an LP starts a simulation, it initializes two boolean variables which are used for checking completion of the Time Warp simulation for an assigned time interval. Then, the LP starts to execute the loop containing the checking

for global state-matching of the time parallel algorithm and executes a Time Warp simulation. First the 'CheckCompleted' routine is called to check for global state-matching. Depending on the results, it may execute another round of Time Warp simulation. When another round of the Time Warp simulation is executed, the flight events are populated again into the 'event_list' for the simulation.

2. (Line 9 - 18): The completion check routine first initializes the 'global_delay_data' container for collecting all the delay data across time and space. Then, every LP executing the same time interval performs a 'reduce sum' operation to compute a global delay for the time interval. Finally, all the LPs across the different time intervals perform a 'gather' operation for the LPs to obtain the global delay status. Finally, all the LPs read the data and check if there are any delays cause an additional round of time parallel simulation. This result is returned to the caller of this routine.

3. (Line 19 - 25): In the traffic generation routine, the traffic data is populated by data files which contain the flight schedules. Then, the LP iterates through the traffic data which has entire flight schedules for the day and determines if every flight schedule is assigned to itself regionally and timely. If the data corresponds to this LP, the flight schedule is added into the 'event_list'.

3.2.2 Time and Spatial Parallel Simulation of the Air Traffic Network

An open question concerns how much time parallelism and how much spatial parallelism should be utilized in the simulation. In some parallel simulation applications, the larger amount of time parallelism can result in a greater speed up. In other cases, more spatial parallelism will result in greater speed up. It is also possible that a balance of time and spatial parallelism can yield the best speed up. To investigate the best possible combination of these two forms of parallelism in the simulation of the air traffic network system, the simulation program is constructed in a configurable manner. Based on configuration settings,

the workload is automatically distributed across different LPs. The following explains how the distribution algorithm is implemented.

Job Allocation for Logical Processes

The configuration of the two different parallelisms is specified by two variables called 'space_division' and 'time_division'. The total number of the LPs is determined by multiplying those two values.

$$LP_NUM_{total} = space_division \times time_division$$

Once the number of LPs are specified, the simulation splits the entire communications according to the number of time pieces. The first $\frac{LP_NUM}{space_division}$ LPs are assigned to simulate the first time interval. Then, the next $\frac{LP_NUM}{space_division}$ LPs are assigned for the next time interval, and so on.

Once, the time division is completed, all LPs inside the same time interval initialize a communication channel for their Time Warp synchronization while running the simulation. Based on the geographical regions assigned to each LP, the event list is populated to execute the simulation. The allocation of LPs based on this explained algorithm is presented in Figure 3.3. In addition to the spatial communication channels, another communication channel is created for the LPs simulating the same spatial domains across the time intervals. This communication channel is used for the time parallel simulation algorithm including fix up computations. The pseudo code for the explained initialization of the communication is presented in Algorithm 9.

1. (Line 1 - 9): At first, the initialization of the global communication is performed. Then, each LP gets the global size of the parallel simulation and its global identification number. By division and modulo operations, it can compute the orders in time and space. Finally, based on the orders and the ID number, it creates communication

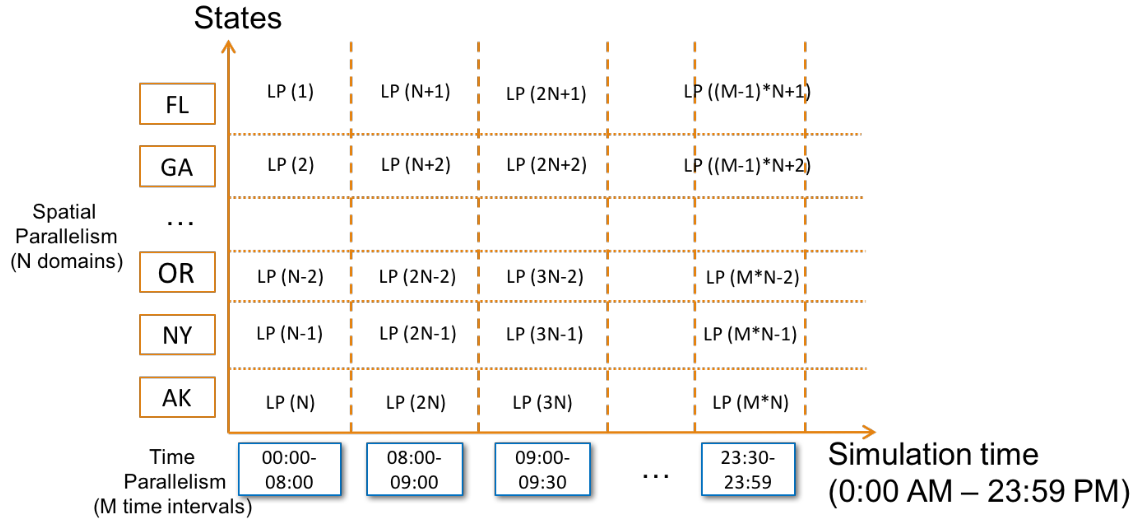

Figure 3.3: Time and spatial allocation of LPs.

Algorithm 9 Initialization of time and spatial communications.

1: **procedure** INITIALIZECOMMUNICATIONS($space_division$, $time_division$)
2: CommInit()
3: LP_NUM ← CommSize()
4: LP_ID ← CommRank()

5: time_order ← LP_ID / space_division
6: space_order ← LP_ID mod space_division

7: spaceComm ← CommSplit(time_order, LP_ID)
8: timeComm ← CommSplit(space_order, LP_ID)

9: Barrier()

channels for the spatial parallel algorithm and time-parallel algorithm.

3.3 Experimental Results

3.3.1 Experimentation Environment

Hardware configuration

A parallel machine using Intel's Xeon Phi® CPU (7250, 1.40 GHz, Knights Landing micro-architecture) was mainly utilized for these experiments. Each chip has 36 tiles of CPU

cores with a 2 dimensional mesh interconnect. Each tile has up to 2 CPU cores. The machine utilized for these experiments has a total of 68 CPU cores. The architecture of the CPU is shown in Figure 3.4. One special characteristic of the Knights Landing CPU is that it utilizes a Multi Channel DRAM (MCDRAM) on the package. MCDRAM is a high bandwidth memory; better performance can be obtained by placing all of the data within the MCDRAM. The machine has 16 GB of MCDRAM and 192 GB of DDR4 memory. For some initial experiments of the stand-alone parallelism case, Intel's Xeon® CPU (E5-2699 v4, 2.20 GHz, Broadwell micro-architecture) was utilized to compare the results with the previous chapter's results. Because all the experiments are performed in a single node, there is no network connectivity for the experiments. Each thread corresponds to an LP and executes on one physical core. To maximize cache hit rates, each physical core executes only one LP during the simulation run.

Software Configuration

C++11 standard libraries were utilized for the implementation. The system runs on the Ubuntu 14.04.5 LTS operating system. Even though this experiment is performed in a single machine configuration, it has been implemented using *MPICH* to enable later extension to a multi-machine clustering environment. The version of the *MPICH* used in this implementation is v3.2 and the version of *gcc* used is v4.8.4.

3.3.2 Parallelism Analysis

Based on the developed time and spatial parallel simulation program, several experiments were conducted. First, the speed up of the spatial parallel simulation algorithm was measured by setting the degree of time parallelism to one. This experiment measures the efficiency of the Time Warp implementation. Also, the speed up of the time parallel simulation algorithm was measured by setting the degree of spatial parallelism to one. In addition, the performance of the both time and spatial parallel algorithms working together was also

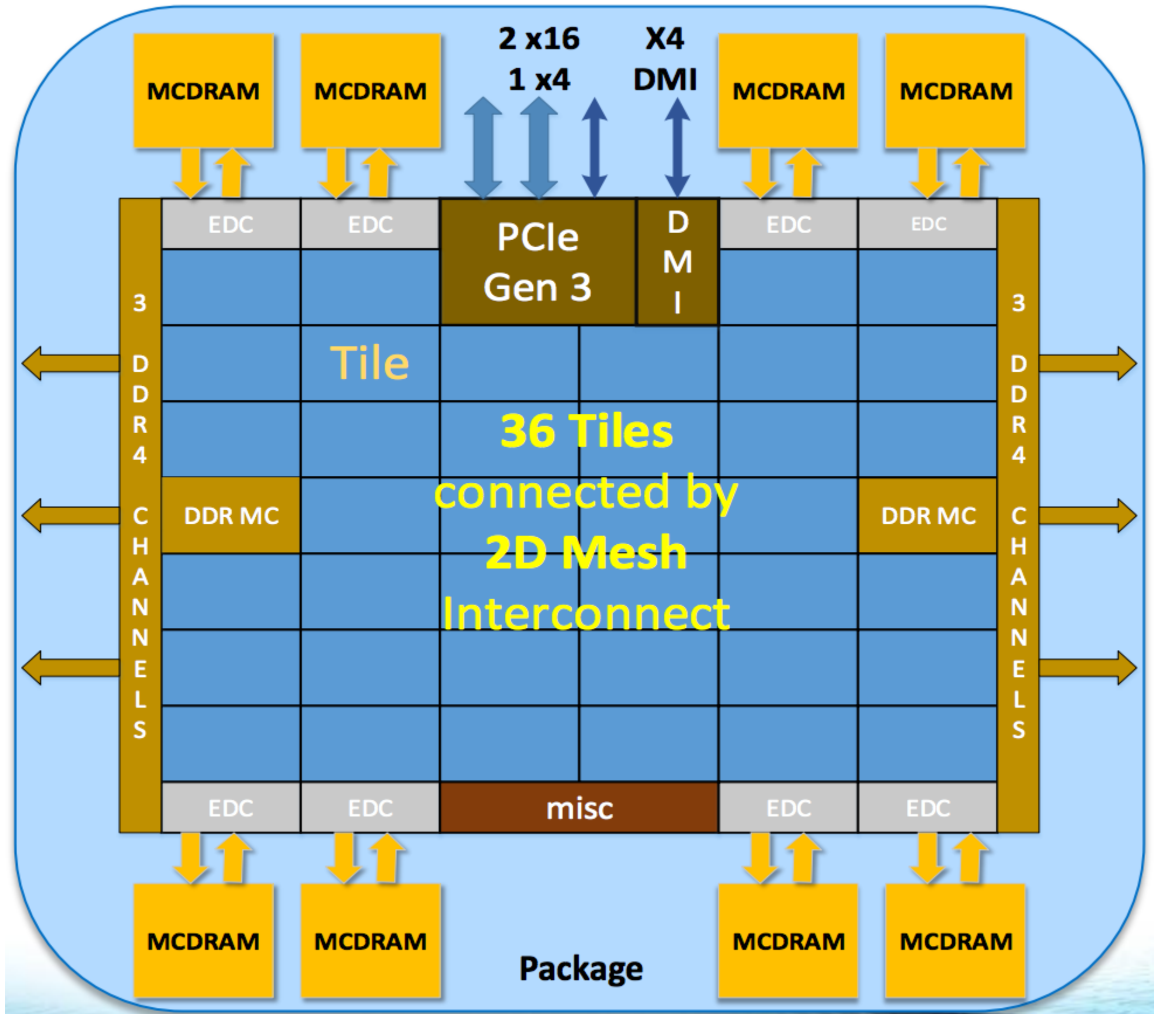

Figure 3.4: Xeon Phi 7250 architecture [84].

measured. Various combinations of time and spatial simulations were measured and analyzed. From the analysis of these cases, one can characterize the performance of different combinations of time and spatial parallelism.

Performance Using Spatial or Time Parallelism in Isolation

Figure 3.5 shows the speed up of the Time Warp simulation with a small workload which has no additional arithmetic computation. In this test, all parallel versions are slower than the sequential simulation. This is because the communication overheads dominate because there is relatively little computation between communications. To investigate the parallel

efficiency under a heavy workload, the amount of computation was artificially increased by adding 1,000,000 integer arithmetic computations per each event. Figure 3.6 shows the resulting speed up. Up to 8 LPs, the speed of the simulation increases with the increase of the number of LPs. With 5 LPs, the simulation executes almost 2.6 times faster than the sequential version. However, when the number of LPs increases beyond 8 LPs, it shows a significant performance degradation. This is because of increased communication and excessive rollbacks.

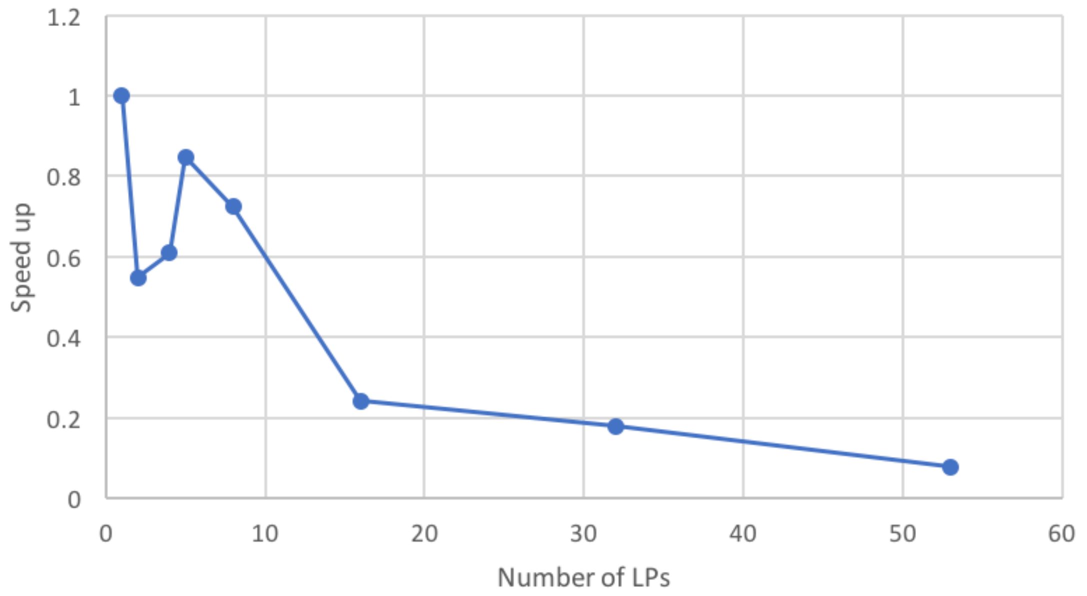

Figure 3.5: Speed up of Time Warp simulation - small workload.

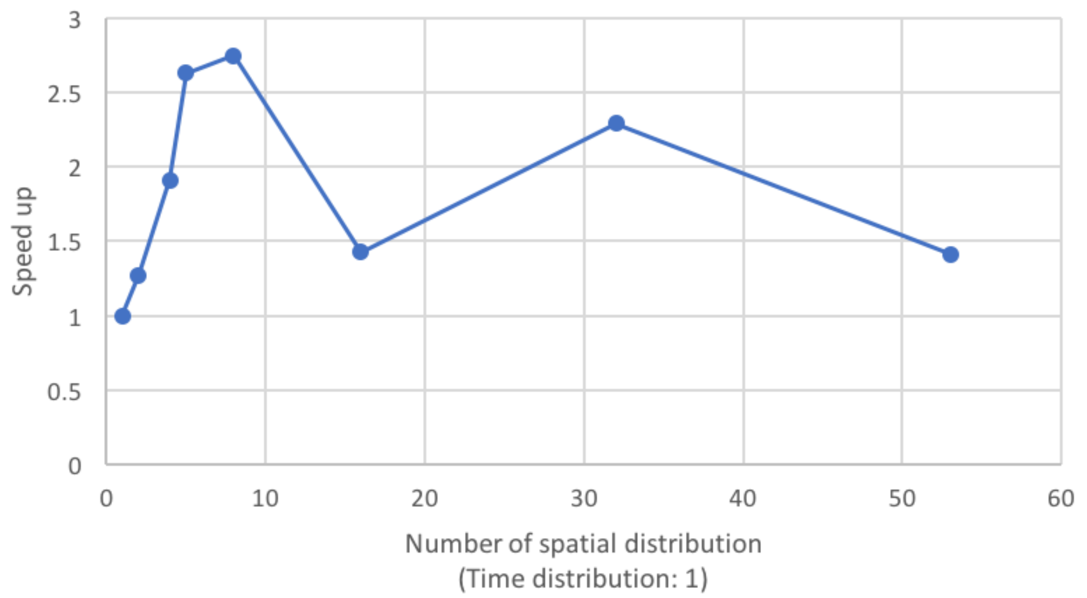

Figure 3.6: Speed up of Time Warp simulation - large workload.

The speed up of the integrated simulation using time parallel simulation only has also been measured to verify that the integration of the spatial parallel simulation algorithm does not degrade the performance of the time parallel algorithm. Figure 3.7 shows the speed up of the time parallel simulation algorithm by setting the degree of spatial parallelism to one. The results are consistent with the implementation of the time parallel algorithm shown in Figure 2.16. It shows a good speed-up up to 4 LPs. However, the efficiency declines when more LPs are utilized. It may be noted that the time parallel simulation shows better scalability than the spatial parallel simulation which suffered from communication overheads as the number of LPs increased.

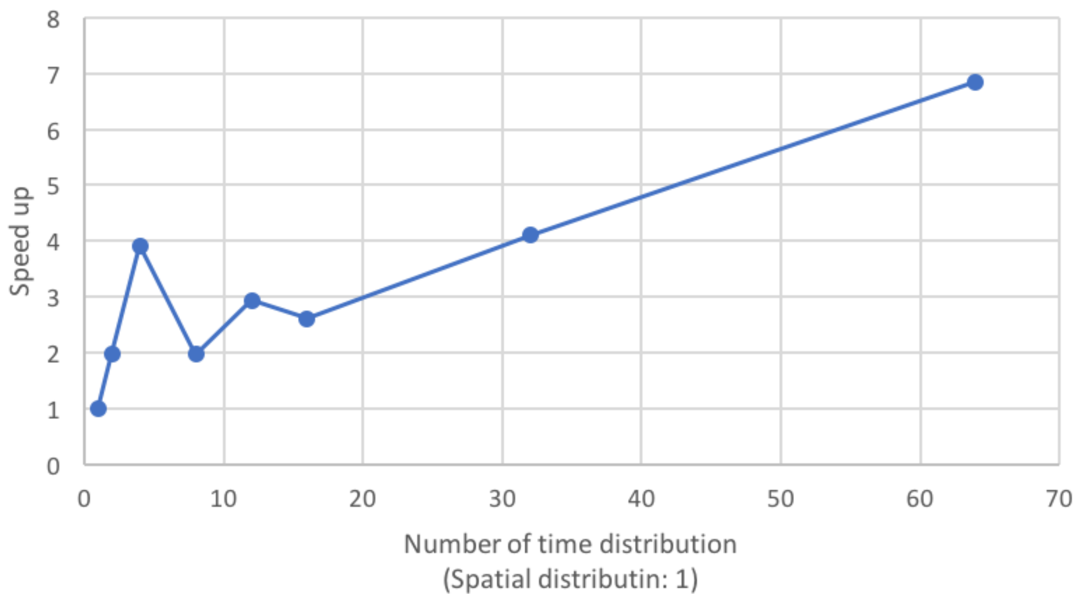

Figure 3.7: Speed up of time parallel simulation- large workload.

The speed up results from the different parallel simulation algorithms operating in isolation are shown Table 3.2. One may note that the division of 5 spatial regions are included in these experiments. The NAS is categorized into five regions based on the latitude of the states. In the simulation result, the case of 5 spatial divisions shows the best efficiency. This is because the workload is better distributed for parallel execution. The maximum number of spatial division is 53 because there are 53 regions including 50 states and 3 independent

regions.

Table 3.2: Speed up by using single parallel algorithms.

Time division	Space division	Total number of LPs	Speed up
1	1	1	1
2	1	2	1.983723732
4	1	4	3.918442948
8	1	8	1.973017747
12	1	12	2.931938218
16	1	16	2.610305943
32	1	32	4.101205776
64	1	64	6.845090425
1	1	1	1
1	2	2	1.271560034
1	4	4	1.911319084
1	5	5	2.628851916
1	8	8	2.748376714
1	16	16	1.424655224
1	32	32	2.290080683
1	53	53	1.414010307

Performance Using Both Spatial and Time Parallelism

These experiments examine using both time and spatial parallelisms, to understand interactions between the two. First, the degree of time parallelism is set to 2, and the degree of spatial parallelism is varied. Considering the number of available cores, the total number of LPs which is the product of the degree of time and spatial parallelism is at most 64. When the time axis is divided into two intervals, the degree of spatial parallelism is set in the range from 2 to 32. When the time axis is divided into four intervals, the degree of spatial parallelism is set in the range from 2 to 16. Figure 3.8 shows the speed up for the case where the degree of time parallelism is fixed at 2 and the amount of spatial parallelism is varied. In this case, the dual parallelism works best when the degree of spatial parallelism is 8, or 16 total LPs. In that case, a speed up of 5.47 is obtained. In terms of efficiency, setting the degree of spatial parallelism to 5 results in a speed up of 5.01 using 6 fewer cores. Figure 3.9 shows the efficiency of these cases. Another observation is that the dual parallel simulation shows no performance degradation due to the interaction between the two different parallel algorithms. Figure 3.10 shows a comparison between the expected speed up and the actual speed up. The expected speed up is computed by multiplying the speed up achieved using time or spatial parallelism in isolation as discussed earlier.

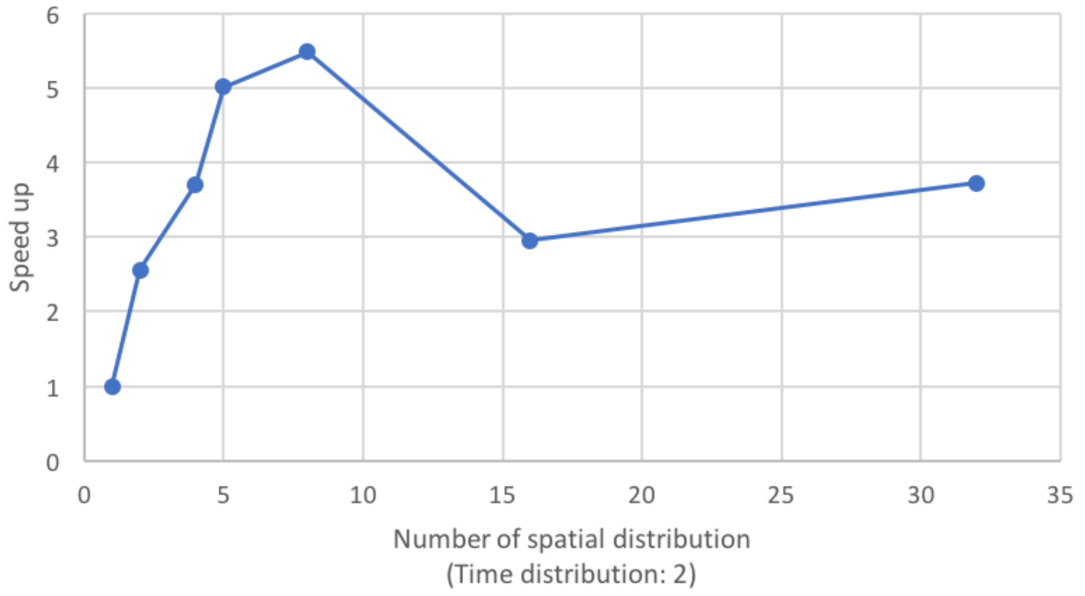

Figure 3.8: Speed up of dual parallel simulation - time division: 2.

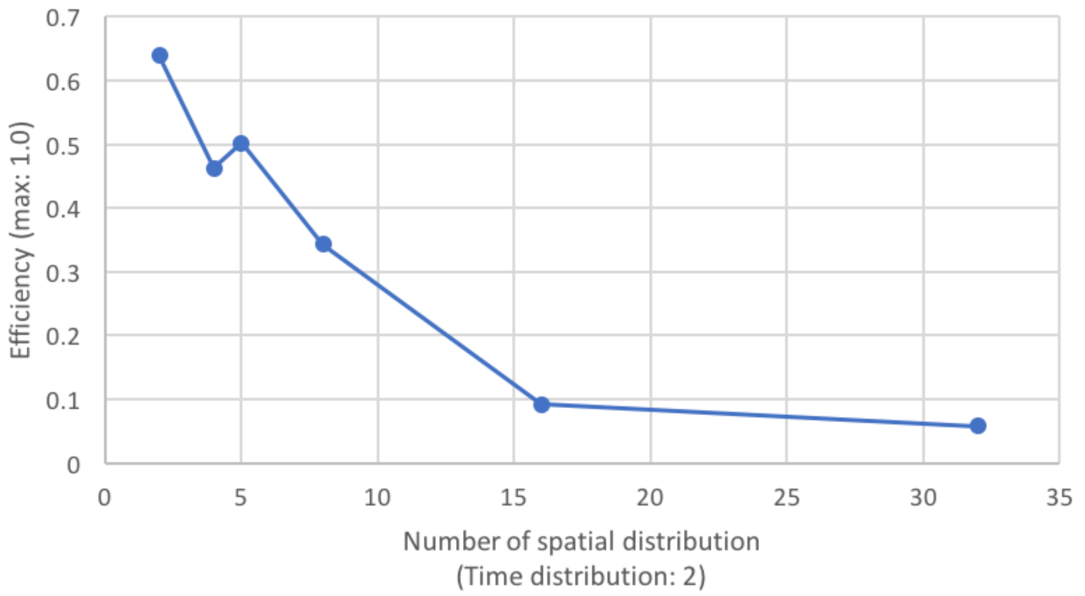

Figure 3.9: Efficiency of dual parallel simulation - time division: 2.

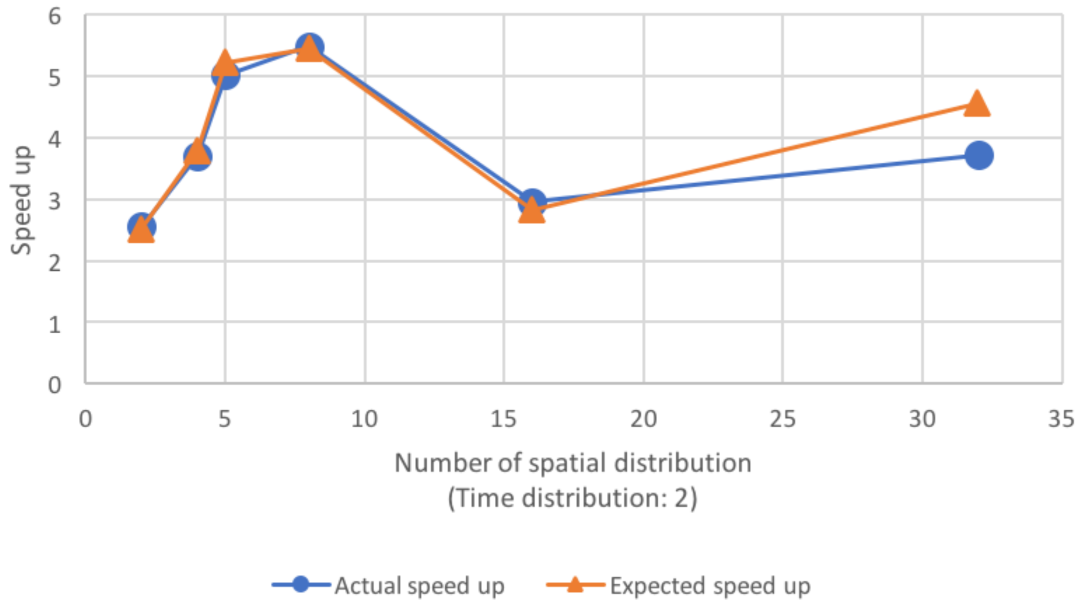

Figure 3.10: Expected speed up vs actual speed up - time division: 2.

The next set of experiments set the degree of time parallelism to 4 and the degree of spatial parallelism is varied from 2 to 16. Figure 3.11 shows the speed up with different degrees of spatial parallelism. The best speed up is $10.9X$ with 8-fold spatial parallelisms which means a total of 32 LPs. However, in terms of efficiency, a spatial parallelism of 5 yields better efficiency. An efficiency of 50% can be seen in Figure 3.12. In that case, 10-fold speed up is achieved using 20 LPs. Finally, the actual speed up is compared with the expected speed up which is the product of the time and spatial parallelisms in isolation. Figure 3.13 shows the result. It shows a consistent pattern for most of the test cases. However, when the number of total LPs become 64, the actual speed up is almost half of the expected speed up. From this result, it can be concluded that there are some interactions between the two different parallel simulation algorithms when the number of LPs is increased. All the speed up results from the dual parallel simulation algorithm are shown Table 3.3.

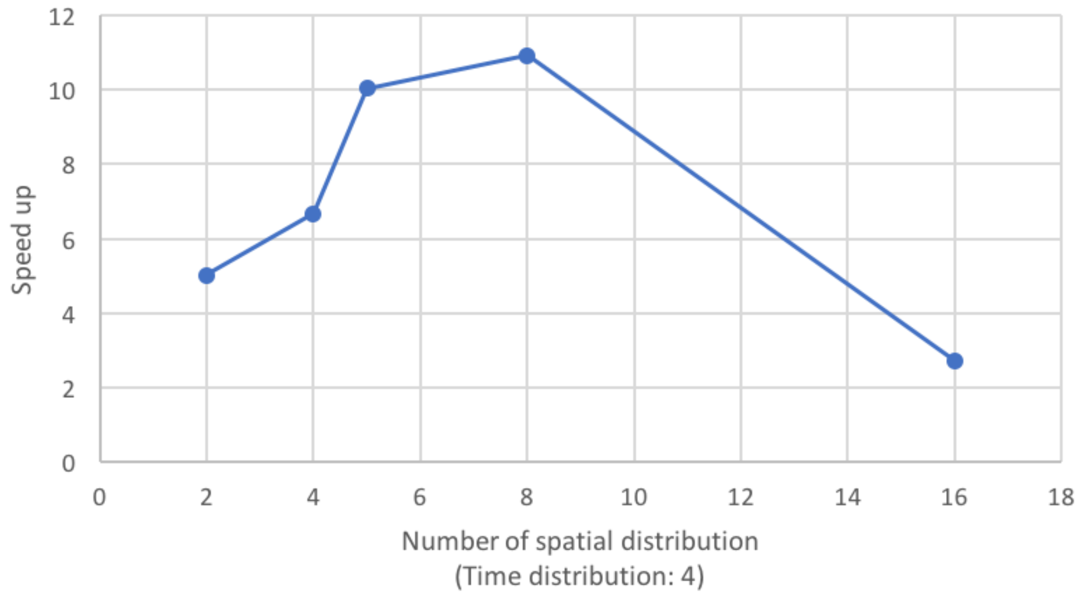

Figure 3.11: Speed up of dual parallel simulation - time division: 4.

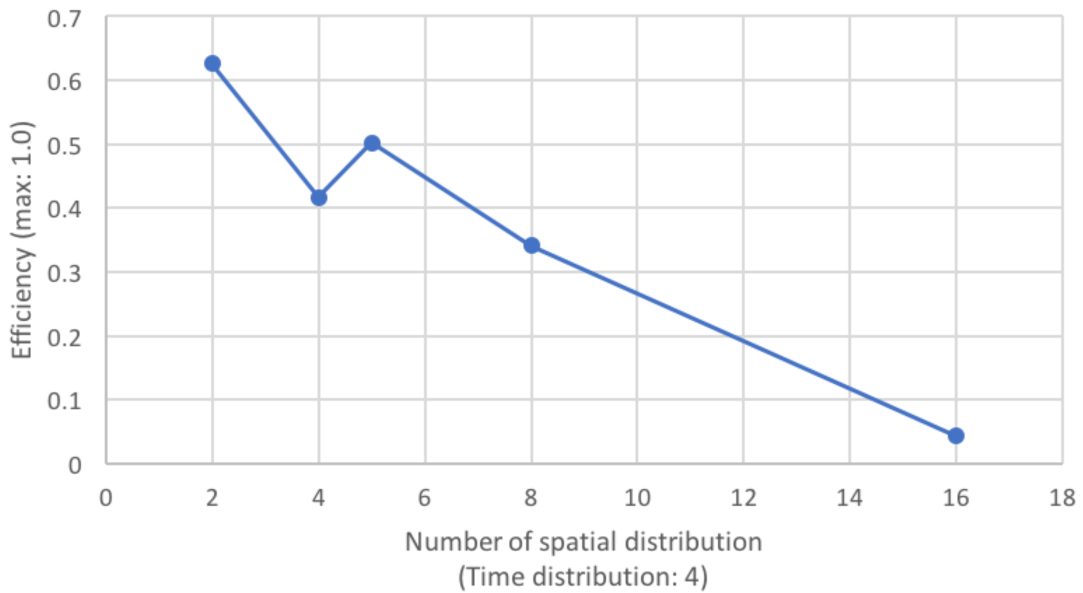

Figure 3.12: Efficiency of dual parallel simulation - time division: 4.

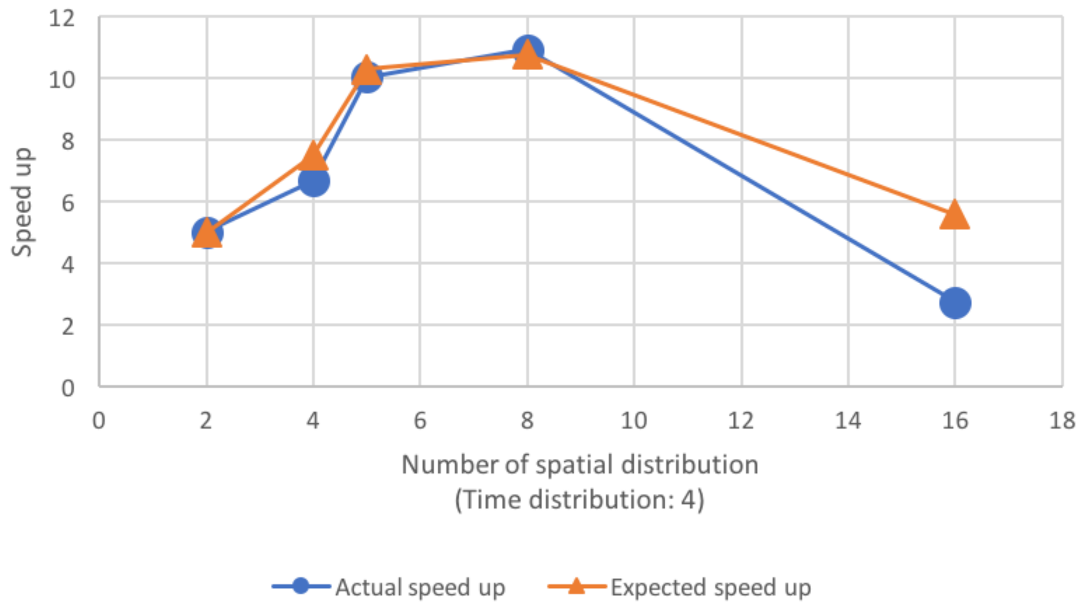

Figure 3.13: Expected speed up vs actual speed up - time division: 4.

Table 3.3: Speed up by using dual parallel algorithms.

Time division	Space division	Total number of LPs	Speed up
2	2	4	2.55640261
2	4	8	3.700238162
2	5	10	5.011837615
2	8	16	5.477427135
2	16	32	2.951136693
2	32	64	3.717920321
4	2	8	5.011222847
4	4	16	6.662033964
4	5	20	10.04040167
4	8	32	10.92419857
4	16	64	2.73136388

3.4 Conclusion

Two different paradigms for parallel simulation are time parallel simulation and spatial parallel simulation. Each has its own strengths and weaknesses. Time parallel simulations can provide a significant speed up in the case where the initial conditions of each time interval is well-known and errors do not propagate through entire time segments. However, if this is not the case, the performance of the time parallel simulation can be degraded significantly. In that situation, multiple rounds of simulations will be needed. On the other hand, spatial parallel simulations can speed up the simulation largely independent of the different initial conditions. However, the spatial parallel simulation algorithm suffers from communication costs when the amount of computation between message in an LP is modest.

In order to utilize the advantages of both parallelisms, both time and spatial simulation algorithms are merged into one simulation program for the air traffic network system application. By making them work together efficiently without interference between them, a new combined parallel simulation algorithm has been proposed. The combined algorithm achieves good performance by exploiting the advantages offered by each algorithm. A variety of combinations of both types of parallelism has been examined to explore the best achievable performance of the combined algorithm for the NAS application. As a result, over ten-fold speed up was achieved.

For future research, there will be an opportunity to improve the parallel efficiency by dynamically distributing the workload across different LPs. Also, other different spatial parallel algorithms can be applied and tested to find the best approach for the NAS simulation application.

CHAPTER 4
RECURRENT NEURAL NETWORKS FOR FLIGHT DELAY PREDICTION

4.1 Overview

Flight delays in the National Airspace System (NAS) lead to a significant amount of costs according to a previous study [2]. In 2007, this accounted for approximately $33 billion as direct or indirect cost to passengers, airlines and other parts of the NAS. In order to reduce the wasted costs, various studies have been performed for the analysis and prediction of air traffic delays [85], [86], [87]. Based on the analysis and the prediction, more efficient and mitigating air traffic management strategies could be established. Furthermore, this prediction can be utilized as an input to a simulation model which enable to realize a reliable simulation model.

To achieve this, there have been a group of analyses using data analytics and statistical machine learning, inspired by the success of their techniques in many fields. Tu *et al.* [88] analyzed long-term and short-term patterns in air traffic delays using statistical methods. Xu *et al.* [89] proposed a Bayesian network approach to estimate delay propagation. Rebollo *et al.* [90] analyzed air traffic network characteristics and predicted air traffic delays using machine learning techniques. Choi *et al.* [91] proposed a machine learning model combined with weather data. However, there is still room for improvement in the accuracy.

In the meantime, artificial neural networks (ANN) based deep learning paradigm which was inspired by the hierarchical structure of human perception has been widespread. Deep ANN improves the accuracy of the classification and regression dramatically in many machine learning tasks such as image recognition, speech recognition, machine translation and etc [92], [93]. Furthermore, it is now utilized for the ground traffic flow prediction [94].

Especially, considering the current improvements of deep ANN algorithms, it is meaningful to evaluate the applicability and the performance of a deep ANN architecture for the flight delay prediction.

There exist a lot of different deep learning architectures including stacked autoencoders, convolutional neural networks and recurrent neural networks. In this research, recurrent neural networks was selected as the architecture for the day-to-day delay status prediction task because it can capture sequential and temporal relationships existing in the data. Intuitively, delay states of previous days' flights affect subsequent days' flight delays. Section 4.2 explains the deep learning algorithms used in this study and Section 4.3 explains the architecture of the networks trained in the study. Section 4.4 presents the experiment results using the deep learning model and the conclusion is given in Section 4.5.

4.2 Deep Recurrent Neural Networks

Recurrent Neural Networks (RNN) is an artificial neural networks that models the behaviors of dynamic systems using hidden states. There are three commonly used RNN architectures: vanilla RNN, Long Short-Term Memory (LSTM) networks and Gated Recurrent Unit (GRU). Vanilla RNN uses the standard RNN equations. LSTM and GRU are proposed to improve the accuracy of vanilla RNN architecture. In this section, general architectures of vanilla RNN, LSTM and GRU networks are explained. Then, the benefits of stacking these networks are discussed and the ways to make an architecture deeper using RNN are also discussed.

4.2.1 Vanilla Recurrent Neural Networks

Vanilla RNN is a standard format of the recurrent architecture. As explained earlier, with an input sequence $x = (x_1, x_2, \ldots, x_k, \ldots, x_T)$, hidden states $h = (h_1, h_2, \ldots, h_k, \ldots, h_T)$ and output sequence $y = (y_1, y_2, \ldots, y_k, \ldots, y_T)$, they are computed by Equation 1.1 and Equation 1.2. This computation is performed iteratively solving the following equations

for the time span from $t = 1$ to T. W_{hh} denotes the weight matrix for the transition of hidden states from the previous time step to the current time step, W_{xh} denotes the weight matrix for the input to hidden layer and W_{hy} denotes the weight matrix for the hidden layer to output. b_h and b_y are capturing biases for each equation. ϕ_h and ϕ_o are activation functions for hidden states and output, respectively. As discussed in Chapter 1, three widely used activation functions are a logistic sigmoid function, a hyperbolic tangent function and Rectified Linear Unit (ReLU). The characteristics of these activation functions are listed in Table 4.1.

Table 4.1: Characteristics of three kinds of activation functions.

	TanH	ReLU	Sigmoid
Equation	$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f(x) = \frac{1}{1+e^{-x}}$
Derivative	$f'(x) = 1 - f(x)^2$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = f(x)(1 - f(x))$
Range	$(-1, 1)$	$[0, \infty)$	$(0, 1)$

One problem with this vanilla RNN architecture is that it is difficult to optimize mathematically. The problem is called as 'vanishing gradients' and caused by a long sequence of the networks. When a network is trained, a numerical gradient method is widely used. However, in a long sequence of the networks, the cumulated gradient multiplication makes the gradient value zero. This results in the RNN model not updated any more even with more training with other data [95]. The model's optimization process might not successfully completed in this situation. Therefore, the accuracy of the model might not be good depending on the applications.

4.2.2 LSTM

The LSTM architecture uses memory cells which will replace ϕ_h and ϕ_o of standard RNN architecture to store hidden layer information and it shows better performance for the sequence of long range than vanilla RNN architectures. By utilizing four different kinds of gate functions, it minimizes the 'vanishing gradients' problem. In this research, the LSTM memory cell proposed by Alex Graves *et al.* [96] was used. This single memory cell is repeated across the sequences. It has an input gate(i), a forget gate(f), an output gate(o) and a cell activation vectors(c), all of which are the same size as the hidden vector h. The following equations represent the computations of the model:

$$i_t = \phi\left(W_{xi}x_t + W_{ht}h_{t-1} + W_{ci}c_{t-1} + b_i\right) \tag{4.1}$$

$$f_t = \phi\left(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\right) \tag{4.2}$$

$$c_t = f_t c_{t-1} + i_t \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) \tag{4.3}$$

$$o_t = \phi\left(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o\right) \tag{4.4}$$

$$h_t = o_t \tanh\left(c_t\right) \tag{4.5}$$

where ϕ is the logistic sigmoid function. The cell architecture of LSTM module is illustrated in Figure 4.1.

4.2.3 GRU

The GRU architecture is a variant of LSTM architecture. Similar to the LSTM, a GRU cell replaces ϕ_h and ϕ_o of standard RNN architecture to store hidden layer information. The main difference between GRU and LSTM is that GRU does not need to carry an additional state vector for the next recurrent layer. In the case of LSTM, cell activation vectors(c) are passed to the next time step to keep more memory. On the other hand, GRU merges the cell

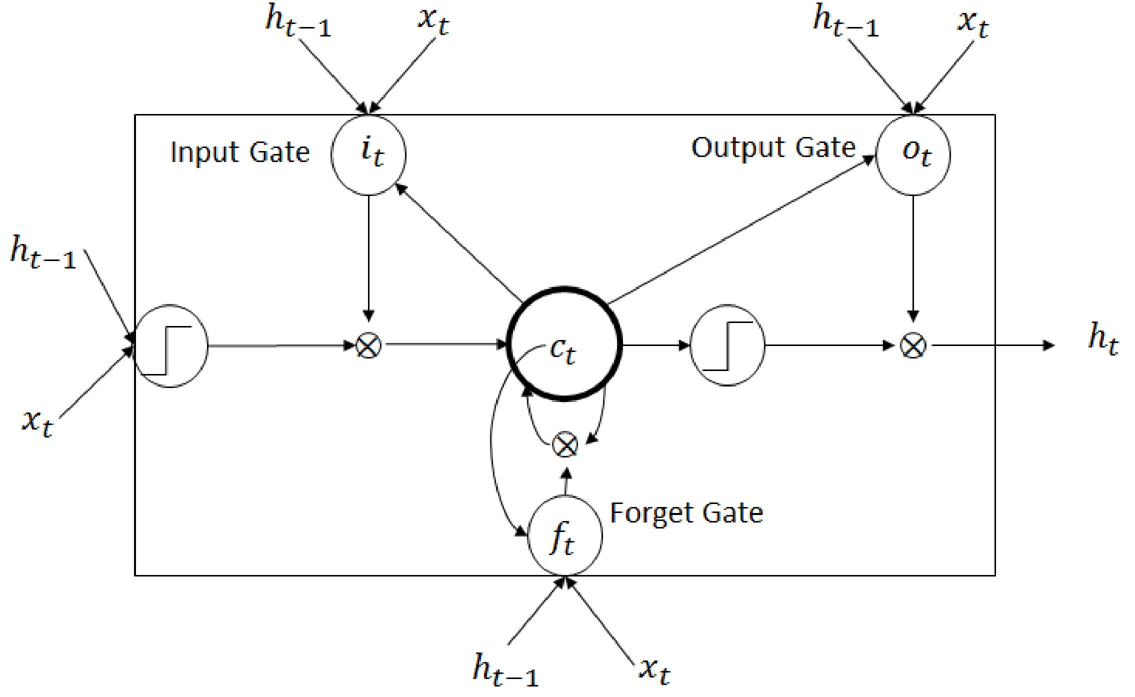

Figure 4.1: Long Short-Term Memory Cell [96].

activation vectors(c) into the hidden state vector h. Also, GRU combines the input gate(i) and the forget gate(f) into a single update unit. This single memory cell is repeated across the sequence. By combining those, the model can be simplified and shows similar performance with LSTM for various tasks. The following equations represent the computations of the model:

$$z_t = \phi \left(W_{xz} x_t + W_{hz} h_{t-1} \right) \tag{4.6}$$

$$r_t = \phi \left(W_{xr} x_t + W_{hr} h_{t-1} \right) \tag{4.7}$$

$$\tilde{h_t} = \tanh \left(W_{xh} x_t + W_{hh} \left(r_t \bigodot h_{t-1} \right) \right) \tag{4.8}$$

$$h_t = (1 - z_t) \bigodot h_{t-1} + z_t \bigodot \tilde{h_t} \tag{4.9}$$

where ϕ is the logistic sigmoid function. $\odot$ denotes an element-wise multiplication of two vectors. The cell architecture of GRU module is illustrated in Figure 4.2.

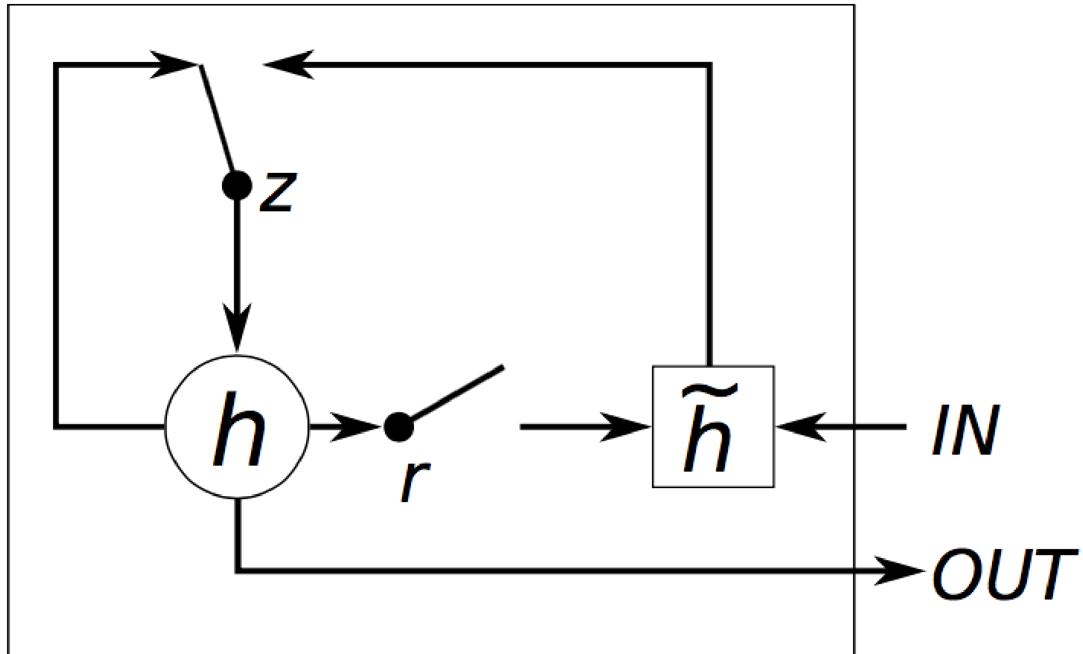

Figure 4.2: Gated Recurrent Unit Cell [97].

4.2.4 Deep architecture of RNN

From the past studies [96], [98], it has been shown that a deep and hierarchical model can be more efficient and accurate at representing some functions than a shallow one. Inspired by this hypothesis, a deep architecture of model is designed for the task of flight delay prediction. There are four different ways to deepen the RNN model: deep input-to-hidden, deep hidden-to-output, deep hidden-to-hidden transition and stacks of hidden states. Each one strengthens the model in a different manner. First, the deep input-to-hidden architecture has the effect of non-linear dimensionality reduction. And, it will discover the underlying factors of variation from the original input. The deep hidden-to-output architecture may be useful to disentangle the factors of variation in the hidden state, making it easier to predict the output. The deep hidden-to-hidden transition architecture allows the RNN to learn a highly nonlinear and non-trivial transition between the consecutive hidden states.

Lastly, the stack of hidden states enables a model to capture state transitions of different timescales.

For this research, the deep input-to-hidden function, the deep hidden-to-output function and stacked RNN are applied. They are also illustrated in Figure 4.4. The non-linearity of the state transition is already covered by RNN architectures. The equations for deep input-to-hidden and deep hidden-to-output transitions just add more affine layers and non-linear transformation layers, thus the formulations are left out of this thesis. The mathematical formulation of the stacked RNN is as follows:

$$h_t^{(l)} = f_h^{(l)}\left(h_t^{(l-1)}, h_{t-1}^{(l)}\right) = \phi_h\left(W_l h_{t-1}^{(l)} + U_l h_t^{(l-1)}\right) \tag{4.10}$$

where $h_t^{(l)}$ is the hidden state of the l-th level at time t. When $l = 1$, the state is computed using x_t instead of $h_t^{(l-1)}$. The hidden states of all the levels are recursively computed from the bottom level $l = 1$.

4.3 Network Training

The proposed model has a two-stage approach. The first stage is to predict daily delay status using deep RNN. The next stage is to predict delays of individual flights using daily delay status from the first stage. For the training of the model, historical on-time performance data of the commercial airline flights and historical weather data for the ten major airports in the U.S. have been collected. Then, the historical data was grouped by airports so that the day-to-day sequence of arriving and departing flights at a specific airport can be fed into the first stage of the model. By computing hidden states sequentially, the delay status of subsequent days is predicted as an output. For the second stage, the daily from the first stage, is used as the model input to predict the delays of individual flights. The details of the actual network configuration and the methods used for the network training are described in this section.

4.3.1 Day-to-day delay status model

The purpose of the first stage is to get a day-to-day delay status model. From the Transtats database of U.S. Department of Transportation [80], on-time performance data of commercial airline flights is collected. Including the flight schedule, origin airport and destination airport, all available data attributes are collected from the database. Table 4.2 shows the details of the flight data used. All of the departure delay times and all of the arrival delay times for each single day are averaged, respectively. The averaged values are used for representing the delay status of one single day. The binary status which is either not-delayed or delayed is acquired by applying threshold value to the averaged delay value. Several different threshold values are tested to analyze the most effective threshold value.

It is expected that the weather conditions at the origin and the destination airports were important factors for the prediction task. Therefore, all the weather data related to the flight data was gathered from the Integrated Surface Database (ISD) of National Oceanic and Atmospheric Administration (NOAA) [99]. Similar to the historical flight data, all the available data attributes are collected from the weather database. Then, the weather data for a day is averaged. For both flight data and weather data, there is no pre-filtering for the available data attributes. By using the deep input-to-hidden architecture, it is expected that the most important features of the model are extracted automatically. The list of the weather attributes selected for the prediction task is shown in Table 4.2. Based on the flight and weather input data, the binary class of delay is computed as an output. Then, this classification is repeated for the consequent days at an airport. In the sequence of the flight status, the delay of the previous days will affect the delay of the following days. This sequential characteristic makes the recurrent relationships at the airport. Using an example of the departure delay sequence of Atlanta airport, the concept for the model is illustrated in Figure 4.3.

Table 4.2: Inputs and outputs of the day-to-day delay status model.

Airports	ATL, LAX, ORD, DFW, DEN, JFK, SFO, CLT, LAS, PHX
Time period	Jan. 2010 - Aug. 2015
Attributes of (Input variables) Flight data	Day of week, Season, Month, Date
Attributes of (Input variables) Weather data	Wind direction, Wind speed, Cloud height, Visibility, Precipitation, Snow Accumulation, Intensity, Descriptor, Observation Code (Daily average)
Classification (Output variable)	Class of delay with different threshold values (10 minutes, 15 minutes, 30 minutes) from averaged departure and arrival delay data

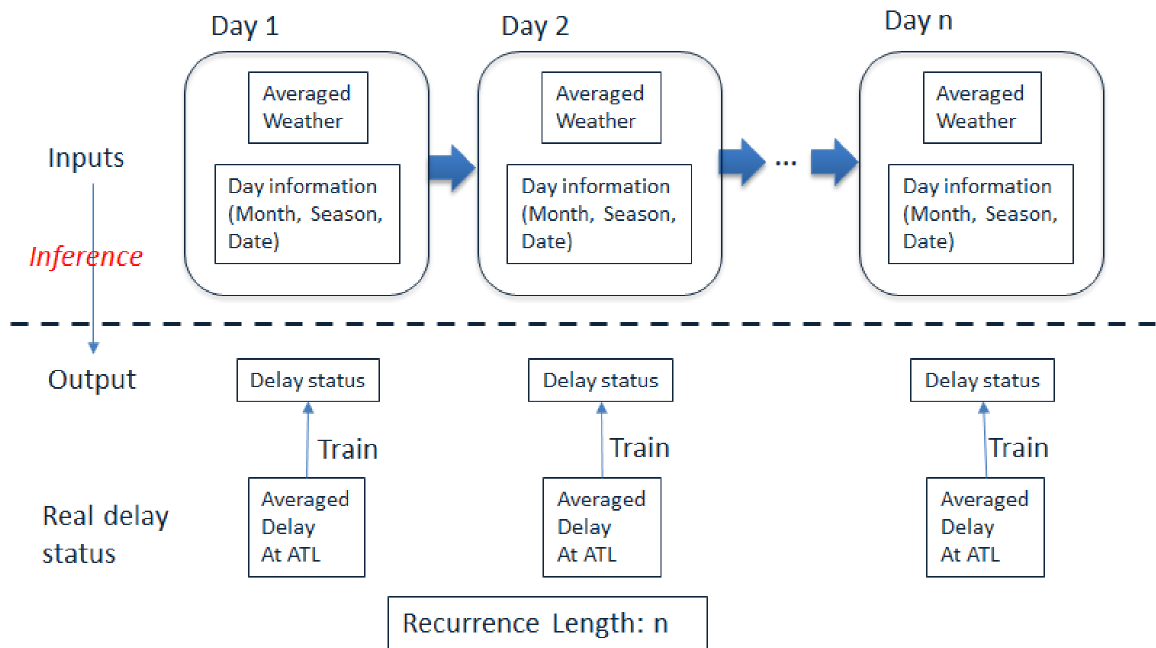

Figure 4.3: Day-to-Day departure delay status model.

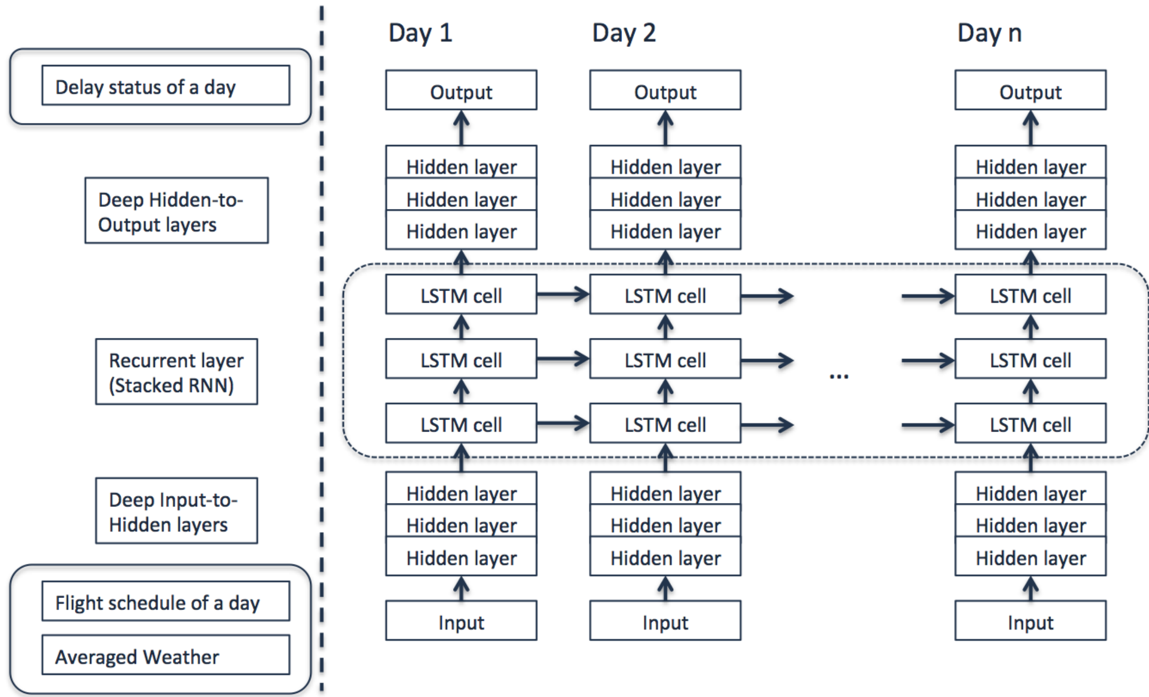

Figure 4.4: Deep architecture for the RNN model.

4.3.2 Deep architecture for the day-to-day delay status RNN model

In order to learn the sequential nature of the air traffic flight delays correctly, deep architectures described in the previous section are utilized. The deep input-to-hidden functions, the deep hidden-to-output functions and stacked RNN architectures are merged into the designed model. The architecture of the network is illustrated in Figure 4.4.

4.3.3 Individual flight delay model

Once, a delay status of one day is acquired, it is fed into the second stage model. The second stage consists of the layered neural networks (NN) model. It computes a delay class of one specific flight using a given delay status of flight date and a historical delay class with historical weather data. For each depth, the hyperbolic tangent function ($Tanh$) is followed by a fully connected linear layer. At the final depth, the logistic sigmoid function is used instead of $Tanh$ because the final output should be a binary class which is 0 and 1. The inputs and outputs of the model are summarized in Table 4.3. The networks built for

this stage is also illustrated in Figure 4.5. The number of layers and the number of nodes in each layer of the NN model can vary. Impacts of those numbers will be discussed in the next section.

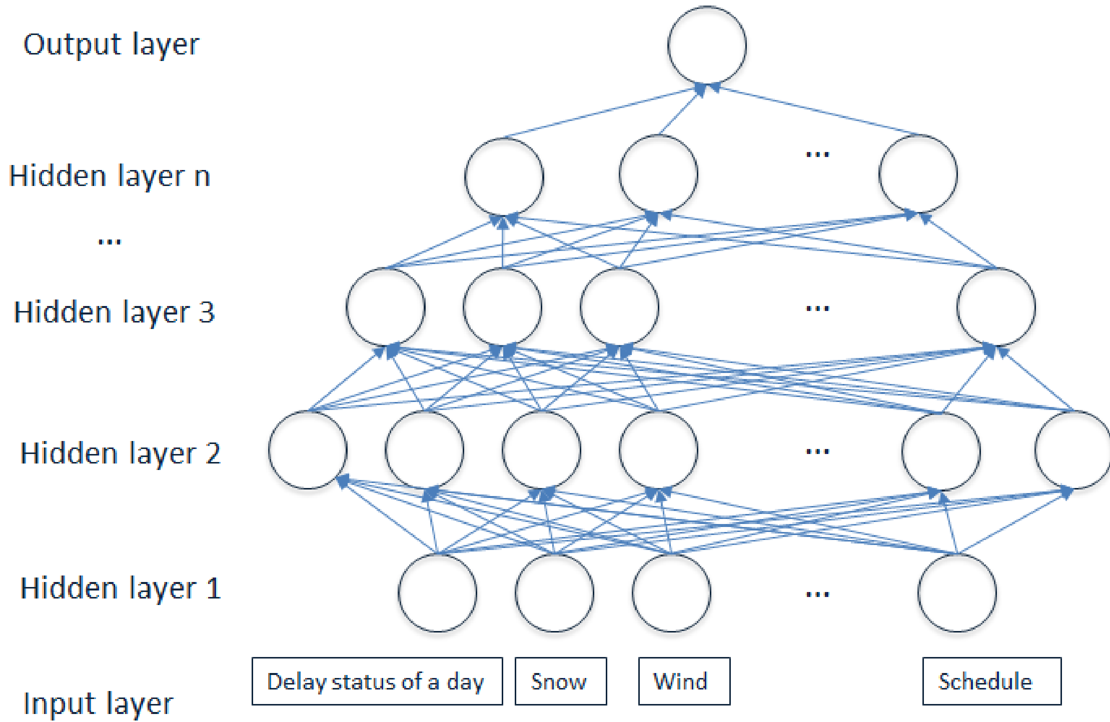

Figure 4.5: Individual flight delay model.

4.3.4 Regularization

One of the most important issues that needs to be handled appropriately is how to prevent over-fitting of the both day-to-day delay status model and individual delay prediction model. As the complexity of the model increases with depth, the model is more prone to over-fitting. It results in a serious degradation of the accuracy of the model. The dropout technique proposed by Hinton *el al.* was used in this study [100]. It has been proved that dropout enhances the accuracy of the deep learning model by randomly dropping units (along with their connections) from the neural networks during training [101]. As a result of the random drop, a dropout updates a randomly picked subset of entire network during

Table 4.3: Inputs and outputs of the individual flight delay model.

Attributes of (Input variables) Flight data	Day of week, Season, Month, Date Origin airport, Destination airport Scheduled departure time Scheduled arrival time Delay status of origin airport Delay status of destination airport
Attributes of (Input variables) Weather data	Wind direction, Wind speed, Cloud height, Visibility, Precipitation, Snow Accumulation, Intensity, Descriptor, Observation Code
Classification (Output variable)	Class of delay with different threshold values (15 minutes, 30 minutes)

the training phase. This increases a randomness in the model and results in a more generalized model. During the prediction phase, this random sampling does not happen. As a result, this has an equivalent effect to an ensemble model which has an exponential number of different "thinned" networks. This significantly reduces over-fitting and gives major improvements over other regularization methods. The procedure of the dropout is shown in Figure 4.6. The left side shows the original networks model and the right side shows one thinned networks using a dropout.

4.3.5 Training methods

For the training of the designed model, the stochastic gradient descent (SGD) algorithm is utilized. In contrast to the conventional gradient descent algorithm which is called batch gradient descent, it uses only one sample data at every iteration step of the training optimization. By using only one random sample at a time, it reduces computation time and memory space used for the training, significantly [102]. The algorithm may not converge to the direct descent direction of a local optimum because of the noise in a single data point. However, it is not a problem when a large amount of data is available. Furthermore,

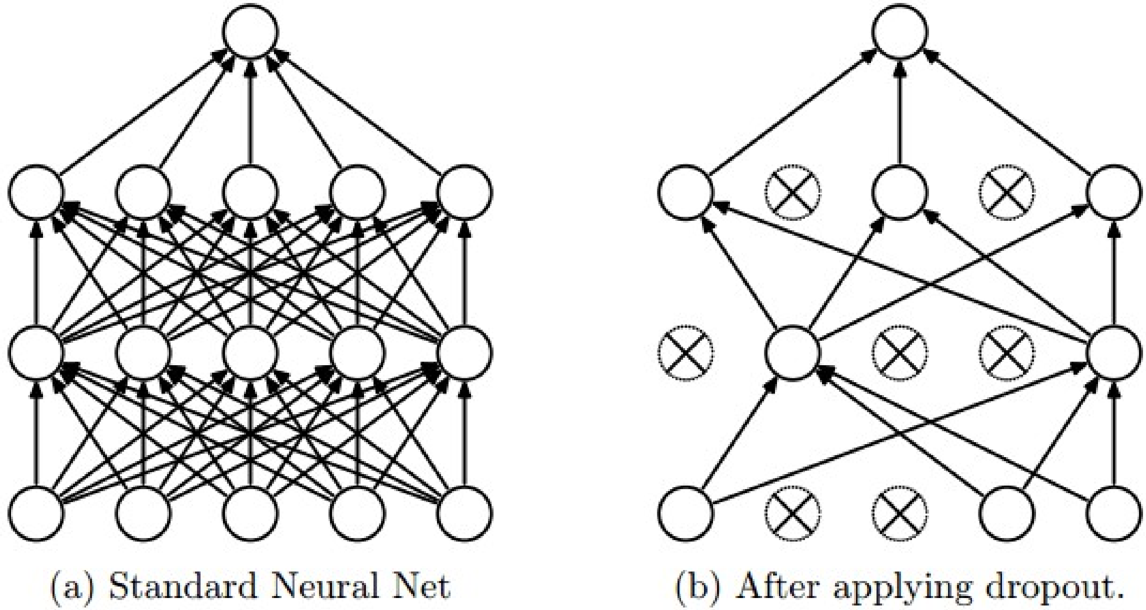

Figure 4.6: Dropout Neural Net Model [101].

by adding random sampling procedure at every iteration step, SGD is another effective method to prevent over-fitting and increase general performance. Mini-batch gradient descent algorithm is in between batch gradient descent and SGD. It uses a subset of data for each iteration so reduces the time to converge. For some models in the study, mini-batch gradient descent algorithm was also utilized.

4.4 Experimental Results

Using the implemented day-to-day RNN model and the individual flight NN model, experiments have been performed to analyze the effectiveness of the RNN models. At first, day-to-day delay status model was trained with different RNN settings. To investigate the efficiencies of different recurrent units, three recurrent units studied in previous chapters are applied to ten airports respectively. Also, to investigate the impact of different epochs and the number of stacked layers, one airport is tested with those different settings. Here, an epoch means one full pass through the entire dataset for a training.

Then, the individual flight delay model was trained and tested by varying parameters as

an example usage of the trained day-to-day delay model. In the case of the individual flight delay model, historical data for Atlanta airport was utilized. Finally, for evaluating the generalization performance of the model, one setting acquired from the day-to-day model experiment of Atlanta airport was applied to other major airports and the accuracies were analyzed.

4.4.1 Accuracy measurement

In order to measure the accuracy of the model, one tenths of the air traffic data are held out while the model is trained. Then, the validation of the model accuracy is done with unseen data for the model. At every 10 iterations, the accuracy using one batch of validation set is measured. And, those accuracies are averaged all together.

4.4.2 Day-to-day delay status model

Three different types of recurrence architectures are used for measuring the accuracy of the model for 10 major airports in the NAS. For all the cases, only single stack of the recurrent layer is utilized. And, the threshold value to determine if the airport is delayed or not is set to 10 minutes. The length of the sequence used here is set to 7. With these setting, 25 epochs of trainings have been performed. Table 4.4, Table 4.5 and Table 4.6 are showing the accuracy measured for the validation data set for vanilla RNN, LSTM and GRU respectively. From the result, it is consistently observed that the model works better for some specific airports. For instance, all three models show high accuracies for Atlanta airport, Charlotte airport, Chicago airport and Dallas airport. On the other hand, they show relatively lower accuracies for New York JFK airport, Los Angeles airport and Las Vegas airport. From the observation, it can be assumed that the recurrence architecture is not enough to capture all the dynamic characteristics of some airports. Especially, because they are using only one stack of recurrent layer, all the states dependent on different time frames might not be captured correctly. Figure 4.7 shows the accuracies from

the three different recurrent architectures. Considering the variance from the randomness of the stochastic gradient descent method, LSTM and GRU architectures show the similar performance. However, vanilla RNN shows worse accuracy across different airports. This is consistent with the expectation that the vanilla RNN has a limitation to model the sequential characteristics fully, because of its relatively simple architecture.

Table 4.4: Accuracy of day-to-day model - vanilla RNN.

Airport	Accuracy
Atlanta (ATL)	92.35
Los Angeles (LAX)	81.01
Chicago (ORD)	88.96
Dallas (DFW)	88.09
Denver (DEN)	89.09
New York (JFK)	85.64
San Francisco (SFO)	87.63
Charlotte (CLT)	92.96
Las Vegas (LAS)	85.92
Phoenix (PHX)	88.19

Table 4.5: Accuracy of day-to-day model - LSTM.

Airport	Accuracy
Atlanta (ATL)	93.45
Los Angeles (LAX)	85.19
Chicago (ORD)	90.49
Dallas (DFW)	89.65
Denver (DEN)	87.52
New York (JFK)	85.64
San Francisco (SFO)	88.83
Charlotte (CLT)	93.25
Las Vegas (LAS)	85.76
Phoenix (PHX)	88.09

Table 4.6: Accuracy of day-to-day model - GRU.

Airport	Accuracy
Atlanta (ATL)	93.43
Los Angeles (LAX)	83.96
Chicago (ORD)	90.47
Dallas (DFW)	89.19
Denver (DEN)	89.55
New York (JFK)	86.63
San Francisco (SFO)	88.28
Charlotte (CLT)	93.14
Las Vegas (LAS)	86.04
Phoenix (PHX)	87.97

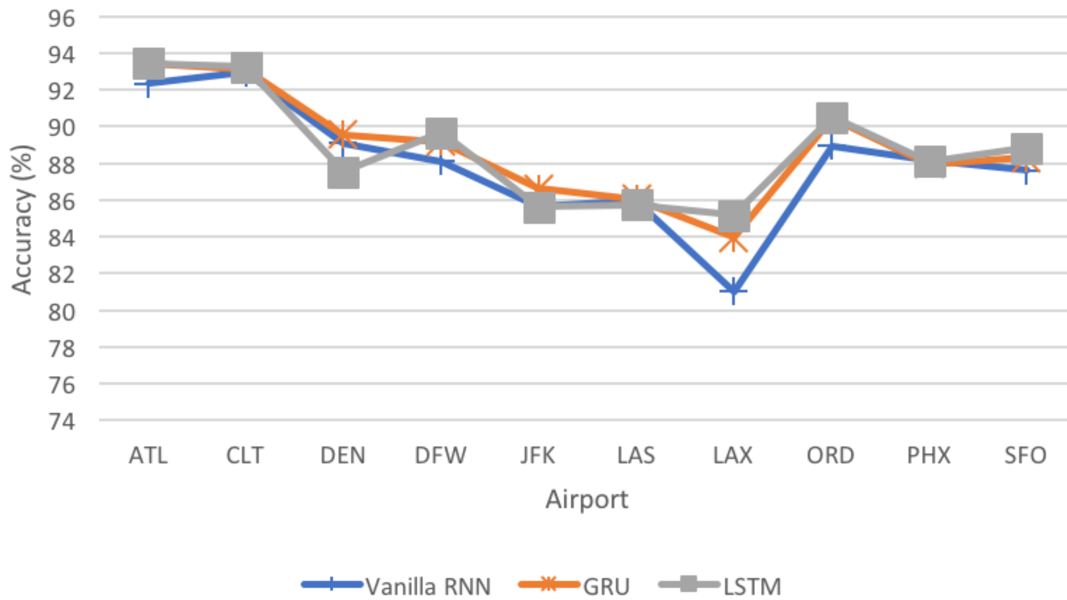

Figure 4.7: Accuracy comparison for different recurrence units.

In order to verify the effects of the different number of epochs and the different number of stacked layers, two more experimentations have been performed. For this experiment, the historical flight data from JFK airport is utilized. As a recurrent unit, LSTM is used for the experiment. Except for the variables under experiment, all the other variables are set to the same as the previous experiment which are 7 for the sequence length and 10 for the delay threshold value. Firstly, given these settings, the number of epochs is varied from 25 to 1000. From this experiment, it can be analyzed how the accuracy changes by the number of epochs. Figure 4.8, Figure 4.9 and Table 4.7 show the result of the experiment. It is observed that the accuracy increases consistently by increasing the number of epochs until 200 epochs. This means that the LSTM architecture has many parameters to optimize and there are still potential gains to acquire. The accuracy still increases when the number of epoch increases beyond 200 epochs, but it also shows a diminishing return. The maximum achievable accuracy is around 98.8 %.

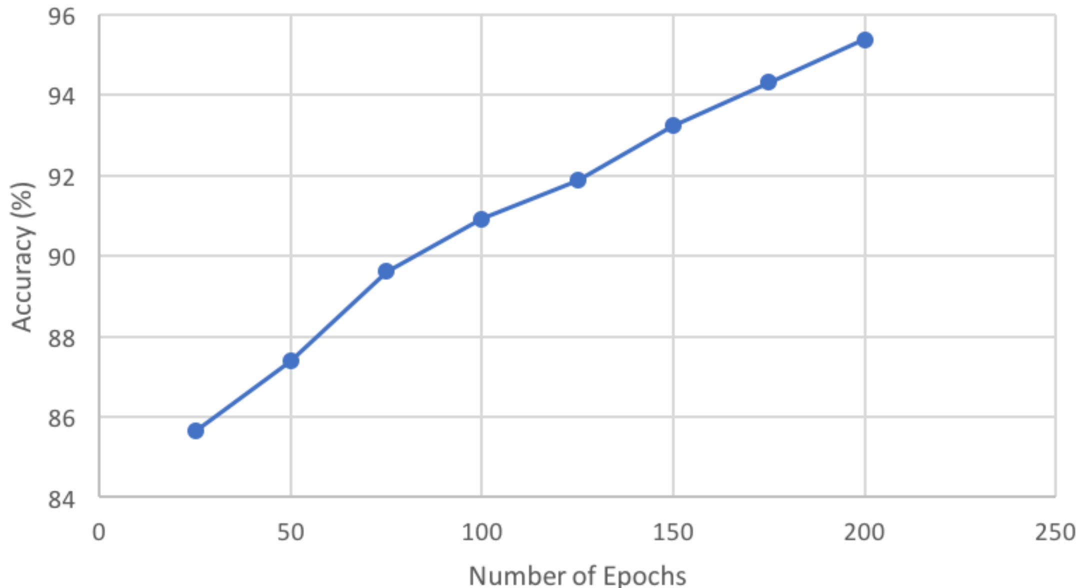

Figure 4.8: Accuracy changes with increasing number of epochs (until 200 epochs).

Table 4.7: Accuracy of day-to-day model - different epochs.

Epochs	Accuracy
25	85.64
50	87.38
75	89.62
100	90.92
125	91.88
150	93.24
175	94.32
200	95.37
300	96.63
400	97.47
500	97.98
600	98.15
700	98.37
800	98.57
900	98.76
1000	98.82

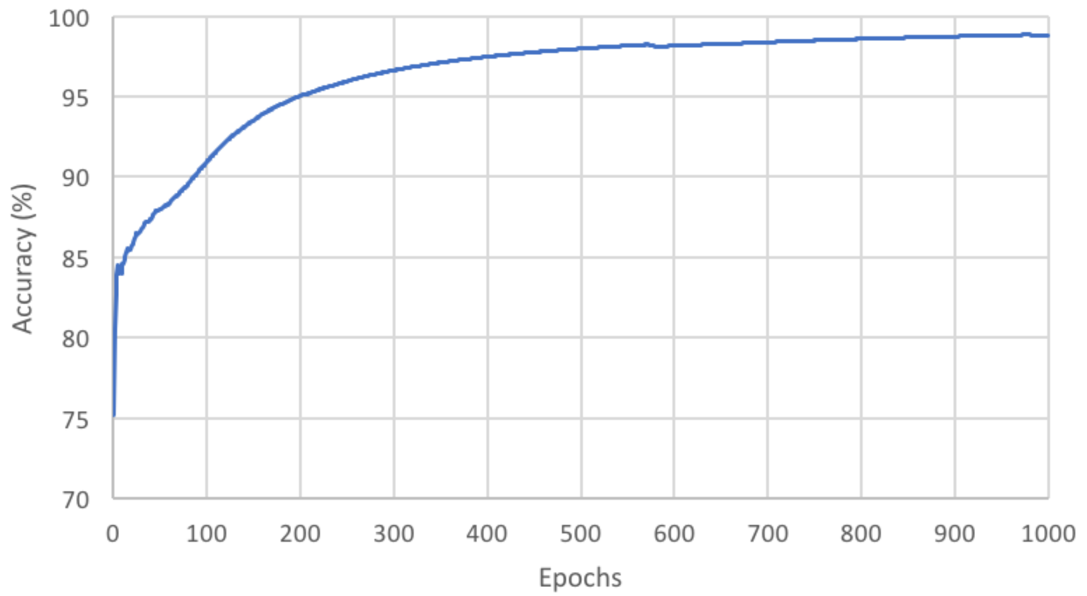

Figure 4.9: Accuracy changes with increasing number of epochs (until 1000 epochs).

As another test case, the number of stacked recurrent layers is varied while the other parameters are fixed. In this case, the number of epochs to train as 100. The number of recurrent layers is varied from 1 to 8. The results are shown in Figure 4.10 and Table 4.8. It is observed that the accuracy increases when multiple stacked layers are utilized compared to one layer. However, it gives a diminishing return when the number is greater than 6. This can be explained that more layers capture the detailed pattern of the data better. However, the accuracy is saturated at some point and the increased number of layers does not improve the accuracy beyond the point.

Table 4.8: Accuracy of day-to-day model - number of stacked layers.

Number of layers	Accuracy
1	90.92
2	93.77
3	93.21
4	93.68
5	93.82
6	92.62
7	93.47
8	93.32

Based on the trained models for Atlanta airport and JFK airport, two example prediction results were acquired. For the comparison of the accuracy, they are shown together with the actual delay data. Both are using 10 minutes as the threshold value and 100 days of prediction data has been computed. Figure 4.11 shows the result from Atlanta airport. As can be seen in the figure, the model's prediction results are almost same as the actual delay data except for several mismatches. For the comparison, the worst performing model in Table 4.5 which is JFK airport also tested for 100 days. It is shown in Figure 4.12. Even for the worst case model, it predicts the delays quite well except for a few mismatches.

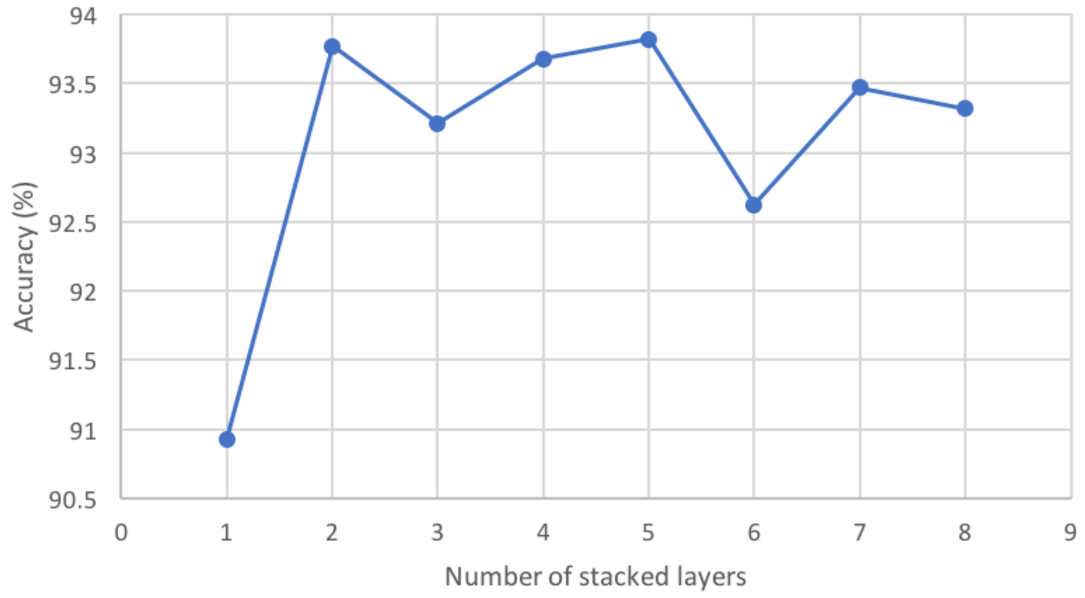

Figure 4.10: Accuracy changes with increasing number of layers.

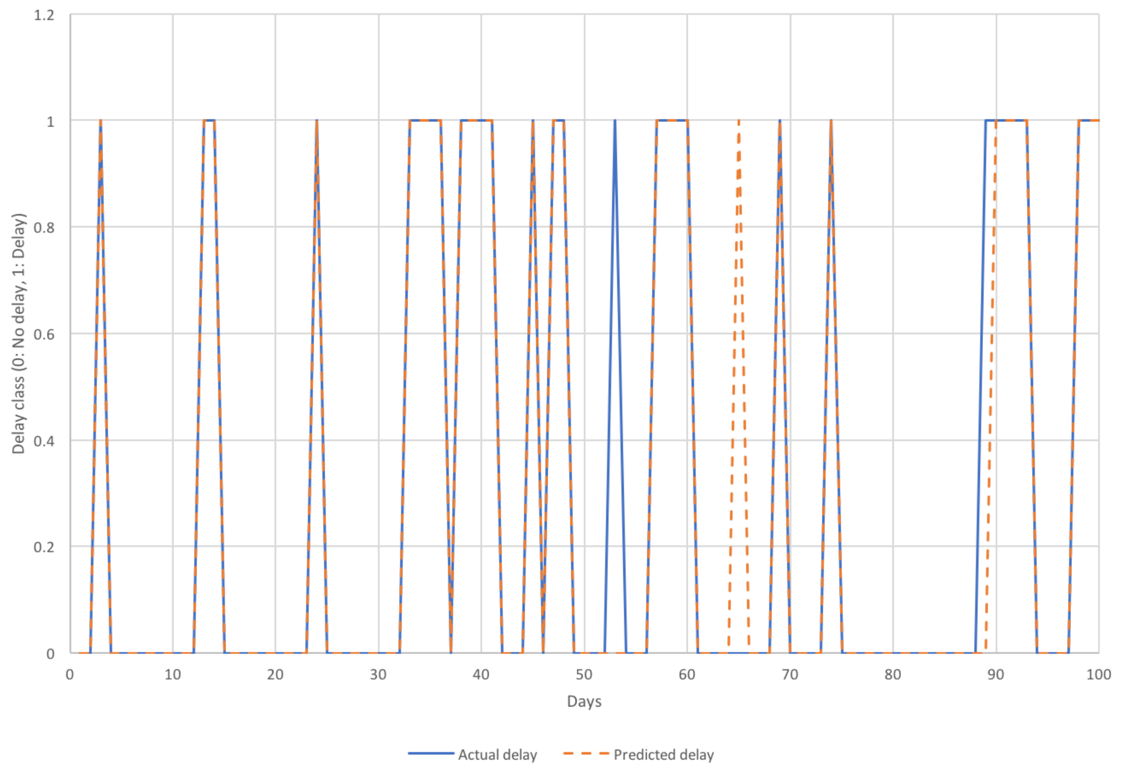

Figure 4.11: Actual delay vs Predicted delay (ATL).

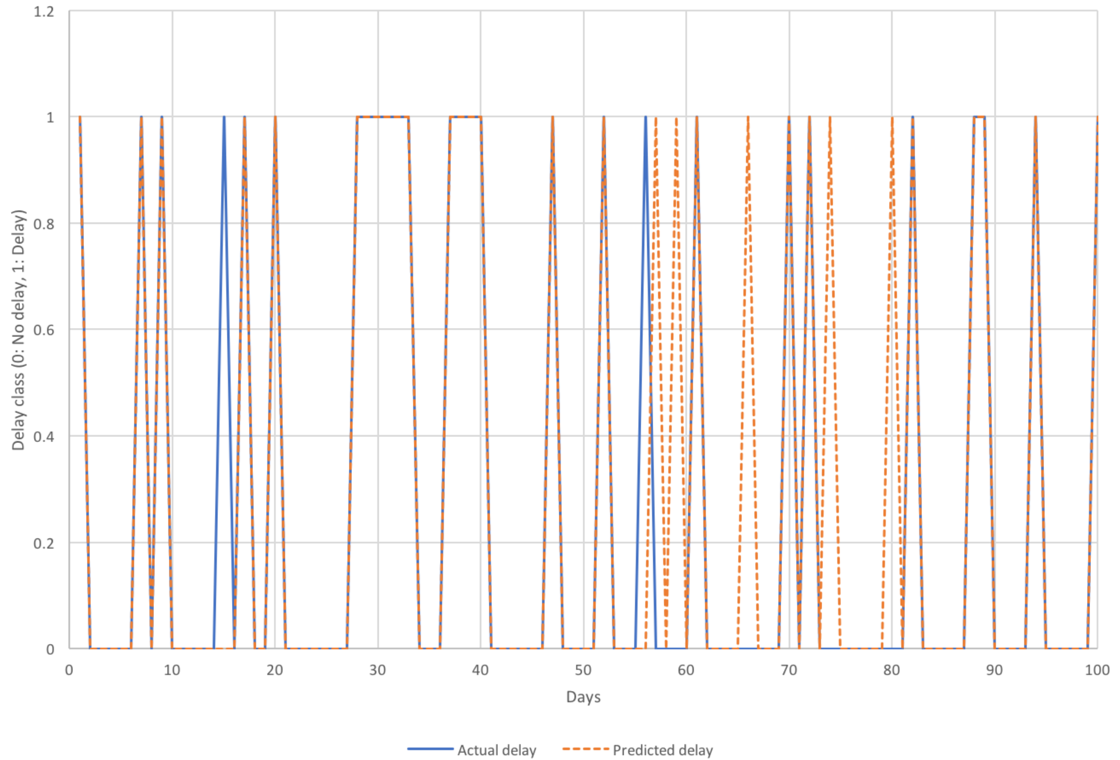

Figure 4.12: Actual delay vs Predicted delay (JFK).

4.4.3 Individual flight delay model

By combining the delay status of a single day, historical flight data and weather data, the model for individual OD pair was trained. The networks described in Section 4.3 and Figure 4.5 was utilized. In the deep layered fully connected nodes, the number of layers, the number of hidden nodes in each layer, epoch and the batch size are varied and the accuracy is tested. At every iteration, the number of samples used for the training is the batch size. Table 4.9 shows the accuracies acquired for different settings. From the results, the model achieved high accuracy ranging from 86% to 87%. It also shows that the increased number of layers is contributing to improve accuracy. Similar to the observation acquired from the previous experiment, it is observed that more epochs make the model more accurate. And, it is a common observation in most of the previous data analytics tasks [103]. This suggests that the RNN approaches and models will perform better in the future by accumulating

Table 4.9: Accuracy of individual flight delay models.

Layers	Number of hidden nodes for each layer	Epoch	Accuracy
1	133	22	85.32
2	133 $\rightarrow$ 100	22	86.57
3	133 $\rightarrow$ 200 $\rightarrow$ 15	22	86.71
4	133 $\rightarrow$ 200 $\rightarrow$ 100 $\rightarrow$ 15	22	86.93
5	133 $\rightarrow$ 300 $\rightarrow$ 200 $\rightarrow$ 100 $\rightarrow$ 15	22	86.99
5	133 $\rightarrow$ 300 $\rightarrow$ 200 $\rightarrow$ 100 $\rightarrow$ 15	228	87.40
5 (mini-batch)	133 $\rightarrow$ 300 $\rightarrow$ 200 $\rightarrow$ 100 $\rightarrow$ 15	228	87.42

more data.

4.4.4 Generalization of day-to-day model for different airports

The day-to-day model trained for one specific airport has been applied to 10 different airports. From the first part of this section, the best performing parameter settings are found using Atlanta airport's air traffic data. Therefore, the model trained with the air traffic data from Atlanta airport is applied to other 9 airports to evaluate the generalization performance of the model to the other airports. Table 4.10 shows the accuracy results for 10 airports. It shows that all the accuracy values are over 80% except for JFK airport. However, the accuracies for the other airports are usually lower than the original Atlanta airport. Thus, even though the model can be generalized well, the model needs to be trained with the specific data set to get the best accuracy. This is because each data set has its own characteristics. In order to verify that an additional training with the specific data from each individual airport helps improving the model, several numbers of additional trainings were performed for JFK airport. The result is shown in Table 4.11. When additional trainings were applied, the accuracy increases greatly. When 50 more iterations are performed, the accuracy increases almost 6%. When 150 additional iterations were applied, it shows a similar accuracy to the model trained originally from JFK airport.

Table 4.10: Accuracy of day-to-day model for different airports.

Airport	Accuracy
Atlanta (ATL)	93.45
Los Angeles (LAX)	82.63
Chicago (ORD)	88.07
Dallas (DFW)	83.33
Denver (DEN)	88.42
New York (JFK)	74.74
San Francisco (SFO)	86.14
Charlotte (CLT)	80.35
Las Vegas (LAS)	81.93
Phoenix (PHX)	81.93

Table 4.11: Additional training with a model from Atlanta airport - JFK airport.

Additional training	Accuracy
0	74.74
50	80.66
100	83.66
150	88.99

4.5 Conclusion

From this study, it is shown that the RNN architectures can improve the accuracy of the airport delay prediction models. In particular, by applying LSTM and GRU architecture to the prediction model, a highly accurate day-to-day delay prediction model can be acquired. Sequential characteristics of the data can be modeled efficiently using the recurrent models. Then, the most accurate delay states for individual flights have been acquired by feeding the delay status of a day to the individual flight delay model. It gives state-of-the-art results in predicting individual flight delays. The next steps are to apply other deep architectures to the prediction and analysis task of flight delays. It may yield important patterns in flight delay data.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this thesis, a new methodology to support air traffic management decisions using time and spatial parallel simulation algorithms and recurrent neural networks models has been presented. The specific contributions of this thesis are summarized in Section 5.1. And, directions for the future research are presented in Section 5.2.

5.1 Contributions

In order to implement an intelligent decision support system for the air traffic management, two major issues need to be addressed. For the fast exploration of a huge decision space, a simulation model for the decision support need to be fast to investigate all the possible different scenarios quickly. Also, for the accurate analysis of the air traffic flow and the decision support for the air traffic management, the decision support tool has to predict the future scenario correctly by utilizing the available data. This thesis has addressed these two problems. The specific contributions of the thesis are listed as follows.

- **Analysis of the existing decision support methodologies for the air traffic management**

  Existing models for the air traffic management have been categorized and analyzed based on their different approaches. In particular, various simulation based models are investigated with their conceptual modeling techniques which are queueing network based discrete event simulation, fluid flow model, system dynamics model and agent-based models. Queueing network based discrete event simulation is picked as a reference model and a prototype model for the air traffic network system has been formulated.

- **In-depth study and formulation of parallel discrete event simulation algorithms**

  To accelerate the constructed discrete event simulation model, various different parallel simulation algorithms have been studied. Two different paradigms of building a parallel simulation program which are spatial parallelism and time parallelism are explained and investigated. In many applications as well as the simulation of the NAS, spatial parallel simulation algorithms have been used mainly. Conservative synchronization algorithms and optimistic synchronization algorithms, which are two major synchronization algorithms for the spatial parallelism, have been studied and explained.

- **Time-parallel simulation algorithm for the air traffic networks**

  As a novel approach to the simulation of the air traffic network systems, a time-parallel simulation algorithm has been introduced and implemented. Especially, certain properties of air traffic network system, which are pre-defined schedule based operation and hub network based model, make the time-parallel algorithm promising for the application. The proposed algorithm has been realized with the proposed prototype simulation program. Using the implemented simulation program, speed ups brought by the time-parallel simulation algorithm have been investigated. In particular, the algorithm shows a sub-linear scalability with an ideal scenario. At the same time, the algorithm shows a relatively smaller scalability in the scenario of a limited airport capacity.

- **Time and spatial hybrid parallel simulation algorithm for the air traffic networks**

  To more accelerate the simulation of the air traffic network, both time-parallel and spatial parallel algorithms have been integrated to build a hybrid model which can utilize the benefits of both parallel algorithms at the same time. The details of the algorithm has been explained and implemented using the prototype simulation pro-

gram. The implemented parallel simulation algorithm results in a significant speed up with a real world air traffic scenario.

- **Formulation of the data modeling for the air traffic management**

  For acquiring more accurate simulation results and getting a better prediction model for air traffic delays, a new method to model the variables existing inside the air traffic network system has been proposed. By utilizing historical air traffic data and weather data, a new model is capable of predicting the future delays of the national airspace system. Day-to-day delay status model can be utilized for the input variable for the air traffic simulation. Also, it can be utilized for the input variable to another model which can predict delays of each individual flight.

- **Recurrent neural networks approach to the flight delay prediction**

  Recurrent neural networks is well-known for its effectiveness of modeling a sequential variables. Air traffic delay data is a kind of sequential data. Especially, the day-to-day model defined in this thesis is well fitted for the recurrent neural networks. Therefore, several different kinds of recurrent neural networks, which are vanilla RNN, LSTM and GRU, are investigated and implemented. This is also a novel approach to the prediction of air traffic delays. By setting several different parameters for the recurrent models, the accuracies of the model have been measured using the real world historical flight and weather data. Overall, the model achieved a significant improvement in predicting the future sequences and LSTM and GRU models show a promising performance.

5.2 Future Research

To build an even faster and more accurate decision support tool, the research presented in this thesis can be extended in several ways as follows.

- **Analysis of the simulation performance with enlarged future air traffic scenario**

In this thesis, real world air traffic schedule data is utilized to verify the speed up and improvement of the proposed algorithm. The benefit from the parallel simulation algorithms could become larger in an enlarged air traffic situation. Because the amount of the air traffic is increasing in a significant pace, optimizing the algorithm with the increased scenario could bring more impacts. For that research, a new method to generate a reasonable future air traffic data also needs to be developed.

- **Improvement of the spatial parallel simulation algorithms**

  From the experimentations, there exist a quite amount of rollback operations while spatial parallel simulation algorithm is running. By addressing this issue, a more efficient spatial parallel simulation algorithm can be implemented. Conservative simulation algorithms can be evaluated to see if they are showing a more efficient parallelism. Also, to reduce the amount of rollback operations, a new algorithm which is restricting too much optimism can be studied. Given that a huge amount of historical air traffic data is available, time warp simulation can limit the optimistic executions based on the statistics of historical data.

- **Dynamic workload balancing across time and spatial regions**

  Another observation from the experimentation is that it is important to distribute workload evenly. Especially, when there are multiple rounds of time-parallel simulations to fix the computations up, some LPs which do not have extra rounds computation just wait until the other LPs finish their extra rounds. This causes an inefficient utilization of such LPs. Furthermore, it is observed that the split point for the time-parallel simulation is important to avoid extra rounds of simulations. From these observations, a dynamic workload balancing algorithm might improve the parallel simulation significantly. If the more time parallelism is needed, the algorithm automatically increase the number of time parallelism while reducing the number of spatial parallelism. Also, it can distribute the time intervals again while multiple

simulation rounds are going on. Finally, it can find an optimal time division point optimally as well.

- **Extension of artificial neural networks approach to the air traffic data modeling**
  To get a more accurate model for the air traffic network, every input variable can be modeled using an artificial neural networks. For instance, the loading and unloading time of passengers at an airport gate can be varied depending on the different dates of the week and dates of the month. Similarly, waiting times for the allocation of gates can be different. These variables also can be modeled using an ANN and a more accurate input variables can be acquired as a result. On the other hand, different kinds of ANN models can be tested and applied. As one example, Convolutional Neural Networks (CNN) is good at extracting the best features from the high dimensional data. Thus, it can be utilized to find a best feature from the air traffic data.

Appendices

APPENDIX A

APPLICATION PROGRAMMING INTERFACE OF SIMULATION SOFTWARE

A.1 Time and Space Parallel Simulation

A part of software especially for the application independent parts is based on the software developed for the *NETWORK TRAFFIC SIMULATOR* from the *CSE 6730* class held in Spring 2014 at Georgia Institute of Technology. So, all the source files might share below general comments.

Common header comment

```
//////////////////////////////////////////////////////////////////////////
///                                                                    ///
///                Time and space parallel NAS simulator               ///
///                                                                    ///
///                by                                                  ///
///                     Young Jin Kim                                  ///
///                                                                    ///
///                A part of software is from                          ///
///                NETWORK TRAFFIC SIMULATOR                           ///
///                CSE / ECE - 6730 Sp. 2014                           ///
///                                                                    ///
///                by                                                  ///
///                    Christopher Hood                                ///
///                     Young Jin Kim                                  ///
///                     David Scripka                                  ///
///                                                                    ///
//////////////////////////////////////////////////////////////////////////
```

Logger.hpp

```
#ifndef LOGGER_HPP
#define LOGGER_HPP

#include <string>

#include <iostream>
#include <fstream>
#include <stdio.h>
#include <stdarg.h>

#include <map>

class Event;

class Logger
{
public:

static Logger* getLogger(const std::string& loggerName);

void writeLog(const std::string& log);

void writeLog(const char* format, ...);

void eventLog(const Event* event);

void queueEventLog(const Event* event);
void rollback( double time );
void commitLog( double time );
```

```
private:

static std::map<std::string, Logger*> loggers_;

Logger(const std::string& loggerName);

~Logger();

std::string    loggerName_;
std::ofstream  of_;

std::multimap<double, std::string> logQueue_;


// TODO file size check and make new file
};


#endif /* LOGGER_HPP */
```

Event.hpp

```
#ifndef EVENT_HPP
#define EVENT_HPP

#include "Handler.hpp"
#include "Logger.hpp"


/////////////////////////////////////////////////////////////////
/// \brief     Wrapper class for an event.
///
/////////////////////////////////////////////////////////////////
class Event
```

```
{
public:

///////////////////////////////////////////////////////////////
/// \brief      constructor
///
///////////////////////////////////////////////////////////////
Event( double time, const Handler *pHandler )
: time_(time)
{
if (pHandler) pHandler_ = pHandler->clone();
else          pHandler_ = 0x0;
}


///////////////////////////////////////////////////////////////
/// \brief      copy constructor
///
///////////////////////////////////////////////////////////////
Event( const Event &copy )
: time_(copy.time_)
{
if (copy.pHandler_) pHandler_ = copy.pHandler_->clone();
else                pHandler_ = 0x0;
}


///////////////////////////////////////////////////////////////
/// \brief      assignment
///
///////////////////////////////////////////////////////////////
Event& operator=( const Event &rhs )
{
```

```
if (this != &rhs)
{
time_ = rhs.time_;
delete pHandler_;
if (rhs.pHandler_) pHandler_ = rhs.pHandler_->clone();
else               pHandler_ = 0x0;
}
return *this;
}



/////////////////////////////////////////////////////////////////
/// \brief    dtor
///
/////////////////////////////////////////////////////////////////
~Event() { delete pHandler_; }


/////////////////////////////////////////////////////////////////
/// \brief    Handle the handler associated with this event
///
/////////////////////////////////////////////////////////////////
void   handle() { pHandler_->handle(); }


/////////////////////////////////////////////////////////////////
/// \brief    Get the time associated with this event
///
/////////////////////////////////////////////////////////////////
double time() const { return time_; }
```

};

```
////////////////////////////////////////////////////////////////
/// \brief      Sorting operator
///
////////////////////////////////////////////////////////////////
bool    operator < ( const Event &rhs ) const
{ return time_ < rhs.time_; }


////////////////////////////////////////////////////////////////
/// \brief      Peek at the handler
///
////////////////////////////////////////////////////////////////
Handler* pHandler() const {return pHandler_;}


////////////////////////////////////////////////////////////////
/// \brief      Peek at the handler
///
////////////////////////////////////////////////////////////////
void    writeLog()
{
Logger::getLogger("Event")->eventLog(this);
}

private:

double  time_;          // Sim time at which this event occurs

// Pointer to object from which Handle function called
Handler *pHandler_;

};
```

```
#endif
```

EventList.hpp

```
#ifndef EVENTLIST_HPP
#define EVENTLIST_HPP

#include "Event.hpp"

/////////////////////////////////////////////////////////////
/// \brief      Abstract base class to encompass an
///             event scheduler.
///
/////////////////////////////////////////////////////////////
class EventList
{
public:

EventList() {}
virtual ~EventList() {}

virtual void     add( Event event ) = 0;
virtual Event    pop() = 0;
virtual bool     empty() = 0;

virtual Event    head() = 0;
virtual uint32_t size() = 0;

virtual bool     annihilate( const Event &event,
                     const bool noexception = false ) = 0;

virtual EventList* clone() const = 0;

protected:
```

```
};

#endif
```

ConfigFile.hpp

```
#ifndef CONFIGFILE_HPP
#define CONFIGFILE_HPP

#include <iostream>
#include <string>
#include <sstream>
#include <map>
#include <cstdlib>


///////////////////////////////////////////////////////////////
/// \brief      General class to handle reading config files
///
///////////////////////////////////////////////////////////////
class ConfigFile
{
public:

ConfigFile() { comment_ = '#'; }
~ConfigFile();

void parse( const std::string &file );

void print( std::ostream &os );

template <class T>
void add( const std::string &arg, T &var );
```

```
template <class T>
void add( const std::string &arg, T &var, bool (*check)(T) );


private:

class VarCapsuleBase;

template <class T>
class VarCapsule;

// data members
std::map< std::string, VarCapsuleBase* > vars_;
char comment_;

};


//////////////////////////////////////////////////////////////////
/// \brief      Base class to handle encapsulation of variable
///             reference.
///
//////////////////////////////////////////////////////////////////
class
ConfigFile::VarCapsuleBase
{
public:
virtual ~VarCapsuleBase() {}
virtual void assign( std::string ) = 0;
virtual void print( std::ostream & ) = 0;
};
```

```
////////////////////////////////////////////////////////////////
/// \brief      Sub class to actually hold data.
///
////////////////////////////////////////////////////////////////
template <class T>
class
ConfigFile::VarCapsule
: public ConfigFile::VarCapsuleBase
{
public:

virtual ~VarCapsule() {}

VarCapsule()
{
pVar_ = check_ = 0x0;
read_ = false;
}

VarCapsule( T *p, bool (*c)(T) )
{
pVar_  = p;
check_ = c;
read_  = false;
}

// Assign the variable from a string
virtual void assign( std::string str )
{
std::istringstream is ( str.c_str() );
is >> *pVar_;
if ( is.fail() ) throw str;
```

```
read_ = true;
if ( check_ && !check_(*pVar_) ) throw this;
}

// print to a stream
virtual void print( std::ostream &os )
{
os << *pVar_;
if ( read_ ) os << " [read]";
else os << " [default]";
}


private:

T     *pVar_;          //  Reference to variable
bool (*check_)(T);  //  Check callback
bool read_;
};



///////////////////////////////////////////////////////////////
/// \brief     Add a variable to the config file keys
///
/// \param     arg:  Variable key
/// \param     var:  Reference of place to write the result
///                  during parsing
///
///////////////////////////////////////////////////////////////
template <class T>
void
ConfigFile::add( const std::string &arg, T &var )
```

```
{
vars_[ arg ] = new VarCapsule<T>( &var, 0x0 );
}



///////////////////////////////////////////////////////////////
/// \brief      Add a variable to the config file keys
///
/// \param      arg:     Variable key.
/// \param      var:     Reference of place to write the result
///                      during parsing.
/// \param      check:   Callback that checks the result.
///
///////////////////////////////////////////////////////////////
template <class T>
void
ConfigFile::add( const std::string &arg, T &var, bool (*check)(T) )
{
vars_[ arg ] = new VarCapsule<T>( &var, check );
}




#endif
```

CircularQueue.hpp

```
#ifndef CIRCULARQUEUE_HPP
#define CIRCULARQUEUE_HPP

#include <vector>
#include <stdint.h>
#include <cstdlib>

```

```
////////////////////////////////////////////////////////////////
/// \brief      Circular queue. Unlike most STL containers,
///             this class provides NO implicit memory
///             allocations. The queue is only moved if you
///             explicitly request more memory.
///
////////////////////////////////////////////////////////////////
template <class T>
class
CircularQueue
{
public:

class iterator;
class const_iterator;

CircularQueue();
CircularQueue( uint32_t capacity );
CircularQueue( const CircularQueue<T> &copy );
CircularQueue<T>& operator = ( const CircularQueue<T> &copy );

void            reserve( uint32_t capacity, const T &copy = T() );
int32_t         size() const;
int32_t         capacity() const;
bool            full() const;
bool            empty() const;

void            push( const T &obj );
void            pop();
void            popBack();

iterator        begin();
```

```
const_iterator begin() const;
iterator       end();
const_iterator end() const;

private:

uint32_t       vlength() const;

int32_t        size_;
iterator       begin_;
iterator       end_;
std::vector<T> vector_;

};



/////////////////////////////////////////////////////////////a////
/// \brief     Base class for a circular iterator which
///            operates in contiguous memory.
///
/////////////////////////////////////////////////////////////////
template <class T>
class
CircularIteratorBase
{
public:

CircularIteratorBase() {}

// These operators can function in between different
// children of the base class (i.e. b/w const and non-const)
bool    operator == ( const CircularIteratorBase<T> &rhs ) const;
bool    operator != ( const CircularIteratorBase<T> &rhs ) const;
```

```
int32_t operator - ( const CircularIteratorBase<T> &rhs ) const;


protected:

/////////////////////////////////////////////////////////////
/// \brief      A modulo operation that always returns the
///             same sign as the second argument, unlike the
///             built-in ''%'' operator.
///
/////////////////////////////////////////////////////////////
inline static
int32_t mod( int32_t a, int32_t b )
{
return b ? (( ( a % b ) + b ) % b) : 0;
}

/////////////////////////////////////////////////////////////
/// \brief      See if rhs has the same base memory region
///
/////////////////////////////////////////////////////////////
bool sameBase( const CircularIteratorBase<T> &rhs ) const
{
bool same ( (pMemBegin_ == rhs.pMemBegin_) &&
(pMemEnd_   == rhs.pMemEnd_) );
if (!same) throw *this;
return same;
}

/////////////////////////////////////////////////////////////
/// \brief      Set pHere_ to lie inside the memory region
///
/////////////////////////////////////////////////////////////
```

```
void set()
{
int32_t N = pMemEnd_ - pMemBegin_;
int32_t n = pHere_ - pMemBegin_;
pHere_     = pMemBegin_ + mod(n, N);
}

/////////////////////////////////////////////////////////////
/// \brief      Copy routine used by the child classes
///
/////////////////////////////////////////////////////////////
void doCopy( const CircularIteratorBase<T> &copy )
{
pMemBegin_ = copy.pMemBegin_;
pMemEnd_   = copy.pMemEnd_;
pHere_     = copy.pHere_;
}

T *pMemBegin_;
T *pMemEnd_;
T *pHere_;
};



/////////////////////////////////////////////////////////////
/// \brief      Another base class to provide the functionality
///
/// \param      T   The base template parameter
/// \param      Q   The qualified template parameter. Should be
///                 either "T" or "const T"
///
/////////////////////////////////////////////////////////////
template <class T, class Q>
```

```
class
CircularIterator
: public  CircularIteratorBase<T>
{
public:

CircularIterator();
CircularIterator( Q *pMb, Q *pMe, Q *pHe );

// Note that in this context, the literal ``const';
// qualifier means that the actual address values
// won't change, NOT that the contents of the address
// won't change. The const qualification on the
// address contents is handled by the Q parameter.

Q&   operator *  () const;
Q*   operator -> () const;

// These operators only work with members of your own child class
CircularIterator<T,Q>  operator =  ( const CircularIterator<T,Q> &rhs );

CircularIterator<T,Q>& operator ++ ();     // prefix
CircularIterator<T,Q>  operator ++ (int);  // postfix
CircularIterator<T,Q>& operator -- ();     // prefix
CircularIterator<T,Q>  operator -- (int);  // postfix

CircularIterator<T,Q>  operator +  ( int32_t n ) const;
CircularIterator<T,Q>  operator -  ( int32_t n ) const;
int32_t                operator -  ( const CircularIteratorBase<T> &rhs ) const;

CircularIterator<T,Q>& operator += ( int32_t n );
CircularIterator<T,Q>& operator -= ( int32_t n );
```

```
protected:

typedef CircularIteratorBase<T> B;

};



/////////////////////////////////////////////////////////////////
/// \brief      ctor
///
/////////////////////////////////////////////////////////////////
template <class T, class Q>
CircularIterator<T,Q>::CircularIterator()
{
B::pMemBegin_ = 0x0;
B::pMemEnd_   = 0x0;
B::pHere_     = 0x0;
}

template <class T, class Q>
CircularIterator<T,Q>::CircularIterator( Q *pMb, Q *pMe, Q *pHe )
{
// We const cast here. Now, we just have to be
// very careful to not edit the contents of these
// addresses if Q is const!
B::pMemBegin_ = const_cast<T*> ( pMb );
B::pMemEnd_   = const_cast<T*> ( pMe );
B::pHere_     = const_cast<T*> ( pHe );
B::set();
}
```

```
///////////////////////////////////////////////////////////////
/// \brief      operators
///
///////////////////////////////////////////////////////////////
template <class T>
bool
CircularIteratorBase<T>::operator == ( const CircularIteratorBase<T> &rhs ) const
{
return sameBase(rhs) && (pHere_ == rhs.pHere_);
}

template <class T>
bool
CircularIteratorBase<T>::operator != ( const CircularIteratorBase<T> &rhs ) const
{
return !(*this == rhs);
}

template <class T>
int32_t
CircularIteratorBase<T>::operator - ( const CircularIteratorBase<T> &rhs ) const
{
if (!sameBase(rhs)) return -1;

// The first ptr (this) is the head. The second is the tail
// ALWAYS returns positive!
int32_t N   = pMemEnd_ - pMemBegin_;
return mod( pHere_ - rhs.pHere_, N );
}

template <class T, class Q>
Q&
CircularIterator<T,Q>::operator * () const
```

```
{
return *B::pHere_;
}

template <class T, class Q>
Q*
CircularIterator<T,Q>::operator -> () const
{
return B::pHere_;
}

template <class T, class Q>
CircularIterator<T,Q>
CircularIterator<T,Q>::operator = ( const CircularIterator<T,Q> &rhs )
{
if (this != &rhs)
{
B::pMemBegin_ = rhs.pMemBegin_;
B::pMemEnd_   = rhs.pMemEnd_;
B::pHere_     = rhs.pHere_;
B::set();
}
return *this;
}

template <class T, class Q>
CircularIterator<T,Q>&
CircularIterator<T,Q>::operator ++ ()
{
// prefix
B::pHere_++;
B::set();
return *this;
```

```
}

template <class T, class Q>
CircularIterator<T,Q>
CircularIterator<T,Q>::operator ++ (int)
{
// postfix
CircularIterator<T,Q> ret( *this );
B::pHere_++;
B::set();
return ret;
}

template <class T, class Q>
CircularIterator<T,Q>&
CircularIterator<T,Q>::operator -- ()
{
// prefix
B::pHere_--;
B::set();
return *this;
}

template <class T, class Q>
CircularIterator<T,Q>
CircularIterator<T,Q>::operator -- (int)
{
// postfix
CircularIterator<T,Q> ret( *this );
B::pHere_--;
B::set();
return ret;
}
```

```
template <class T, class Q>
CircularIterator<T,Q>
CircularIterator<T,Q>::operator + ( int32_t n ) const
{
CircularIterator<T,Q> ret( *this );
ret.pHere_ += n;
ret.set();
return ret;
}

template <class T, class Q>
CircularIterator<T,Q>
CircularIterator<T,Q>::operator - ( int32_t n ) const
{
CircularIterator<T,Q> ret( *this );
ret.pHere_ -= n;
ret.set();
return ret;
}

template <class T, class Q>
int32_t
CircularIterator<T,Q>::operator - ( const CircularIteratorBase<T> &rhs ) const
{
return *static_cast<const CircularIteratorBase<T>*>(this) - rhs;
}


template <class T, class Q>
CircularIterator<T,Q>&
CircularIterator<T,Q>::operator += ( int32_t n )
{
```

```
B::pHere_ += n;
B::set();
}

template <class T, class Q>
CircularIterator<T,Q>&
CircularIterator<T,Q>::operator -= ( int32_t n )
{
B::pHere_ -= n;
B::set();
}










/////////////////////////////////////////////////////////////////
/// \brief      Circular queue iterator implememtation
///
/////////////////////////////////////////////////////////////////
template <class T>
class
CircularQueue<T>::iterator
: public CircularIterator<T, T>
{
public:

iterator();
iterator( T *pMb, T *pMe, T *pHe );
iterator( CircularIterator<T, T> it );
```

```
protected:

typedef CircularIteratorBase<T> B;

};

/////////////////////////////////////////////////////////////////
/// \brief      Circular queue const iterator implememtation
///
/////////////////////////////////////////////////////////////////
template <class T>
class
CircularQueue<T>::const_iterator
: public CircularIterator<T, const T>
{
public:

const_iterator();
const_iterator( const T *pMb, const T *pMe, const T *pHe );
const_iterator( CircularIterator<T, const T> it );

// Make an const iterator from an iterator
const_iterator( const CircularIteratorBase<T> &copy );
const_iterator& operator=( const CircularIteratorBase<T> &rhs );

protected:

typedef CircularIteratorBase<T> B;

};
```

```
///////////////////////////////////////////////////////////////
/// \brief    ctor
///
///////////////////////////////////////////////////////////////
template <class T>
CircularQueue<T>::iterator::iterator()
: CircularIterator<T,T>()
{}

template <class T>
CircularQueue<T>::iterator::iterator( T *pMb, T *pMe, T *pHe )
: CircularIterator<T,T>(pMb, pMe, pHe )
{}

template <class T>
CircularQueue<T>::iterator::iterator( CircularIterator<T, T> it )
: CircularIterator<T, T>( it )
{}

template <class T>
CircularQueue<T>::const_iterator::const_iterator()
: CircularIterator<T,const T>()
{}

template <class T>
CircularQueue<T>::const_iterator::const_iterator(
        const T *pMb,
        const T *pMe,
        const T *pHe )
: CircularIterator<T,const T>(pMb, pMe, pHe )
{}
```

```
template <class T>
CircularQueue<T>::const_iterator::const_iterator(
        CircularIterator<T, const T> it )
: CircularIterator<T, const T> ( it )
{}

template <class T>
CircularQueue<T>::const_iterator::const_iterator(
        const CircularIteratorBase<T> &copy )
{
doCopy( copy );
}

template <class T>
typename CircularQueue<T>::const_iterator&
CircularQueue<T>::const_iterator::operator = (
        const CircularIteratorBase<T> &rhs )
{
if (this != &rhs)
{
doCopy( rhs );
}
return *this;
}




/////////////////////////////////////////////////////////////////
/// \brief      Circular queue implememtation
///
/////////////////////////////////////////////////////////////////
```

```
///////////////////////////////////////////////////////////////
/// \brief      Default ctor
///
///////////////////////////////////////////////////////////////
template <class T>
CircularQueue<T>::CircularQueue()
{
size_ = 0;
}

///////////////////////////////////////////////////////////////
/// \brief      Reserve ctor
///
///////////////////////////////////////////////////////////////
template <class T>
CircularQueue<T>::CircularQueue( uint32_t capacity )
{
size_  = 0;
vector_.resize( capacity + 1 );
T *p   = &vector_[ 0 ];
begin_ = iterator( p, p + vlength(), p );
end_   = begin_;
}

///////////////////////////////////////////////////////////////
/// \brief      copy ctor
///
///////////////////////////////////////////////////////////////
template <class T>
CircularQueue<T>::CircularQueue( const CircularQueue<T> &copy )
{
size_   = copy.size_;
```

```
vector_ = copy.vector_;
T *p = &vector_[ 0 ];
const T *c = &copy.vector_[ 0 ];
const_iterator copyMemBegin ( c, c + copy.vlength(), c );
int32_t nb = copy.begin_ - copyMemBegin;
int32_t ne = copy.end_ - copyMemBegin;
begin_ = iterator( p, p + vlength(), p+nb );
end_   = iterator( p, p + vlength(), p+ne );
}

/////////////////////////////////////////////////////////////
/// \brief      assignment oper
///
/////////////////////////////////////////////////////////////
template <class T>
CircularQueue<T>&
CircularQueue<T>::operator = ( const CircularQueue<T> &rhs )
{
if (this != &rhs)
{
size_   = rhs.size_;
vector_ = rhs.vector_;
T *p = &vector_[ 0 ];
const T *c = &rhs.vector_[ 0 ];
const_iterator rhsMemBegin ( c, c + rhs.vlength(), c );
int32_t nb = rhs.begin_ - rhsMemBegin;
int32_t ne = rhs.end_ - rhsMemBegin;
begin_ = iterator( p, p + vlength(), p+nb );
end_   = iterator( p, p + vlength(), p+ne );
}
return *this;
}
```

```
////////////////////////////////////////////////////////////////
/// \brief      Resize the circular queue.
///
/// \param      newCapacity:  Number of elements to reserve
///
////////////////////////////////////////////////////////////////
template <class T>
void
CircularQueue<T>::reserve( uint32_t capacity, const T &copy )
{
std::vector<T> newVector ( capacity + 1, copy );

size_ = std::min( size_, int32_t(capacity) );

// Copy the data into the beginning of newVector
iterator it = begin_;
for ( int32_t i = 0; i < size_; ++i, ++it )
{
newVector[ i ] = *it;
}

// Swap the contents
vector_.swap ( newVector );

// Reset the iterators
T* p = &vector_[ 0 ];
begin_ = iterator( p, p + vlength(), p );
end_   = iterator( p, p + vlength(), p + size_ );

}

////////////////////////////////////////////////////////////////
/// \brief      Get the current number of valid elements
```

```
///              (number of pushed objects)
///
/////////////////////////////////////////////////////////////
template <class T>
int32_t
CircularQueue<T>::size() const
{
return size_;
}

/////////////////////////////////////////////////////////////
/// \brief      Get the maximum size as set with "reserve"
///
/////////////////////////////////////////////////////////////
template <class T>
int32_t
CircularQueue<T>::capacity() const
{
return vlength() - 1;
}

/////////////////////////////////////////////////////////////
/// \brief      Get the true memory size
///
/////////////////////////////////////////////////////////////
template <class T>
uint32_t
CircularQueue<T>::vlength() const
{
return vector_.size();
}

/////////////////////////////////////////////////////////////
```

```
/// \brief      Query if the queue is full
///
/////////////////////////////////////////////////////////////
template <class T>
bool
CircularQueue<T>::full() const
{
return size_ >= capacity();
}

/////////////////////////////////////////////////////////////
/// \brief      Query if the queue is empty
///
/////////////////////////////////////////////////////////////
template <class T>
bool
CircularQueue<T>::empty() const
{
return size_ == 0;
}

/////////////////////////////////////////////////////////////
/// \brief      Push a new object onto the end of the queue
///
/// \param      obj:  Element to push onto end
///
/////////////////////////////////////////////////////////////
template <class T>
void
CircularQueue<T>::push( const T &obj )
{
*end_++ = obj;
size_   = end_ - begin_;
```

```
}

////////////////////////////////////////////////////////////
/// \brief      Pop an element from the front of the queue
///             Popping on an empty queue will produce weird
///             behavior. Don't do it!
///
////////////////////////////////////////////////////////////
template <class T>
void
CircularQueue<T>::pop()
{
++begin_;
size_ = end_ - begin_;
}

////////////////////////////////////////////////////////////
/// \brief      Pop an element from the BACK of the queue
///             Popping on an empty queue will produce weird
///             behavior. Don't do it!
///
////////////////////////////////////////////////////////////
template <class T>
void
CircularQueue<T>::popBack()
{
--end_;
size_ = end_ - begin_;
}

////////////////////////////////////////////////////////////
/// \brief      Get the front iterator
///
```

```
////////////////////////////////////////////////////////////////
template <class T>
typename CircularQueue<T>::iterator
CircularQueue<T>::begin()
{
return begin_;
}

////////////////////////////////////////////////////////////////
/// \brief      Get the front iterator
///
////////////////////////////////////////////////////////////////
template <class T>
typename CircularQueue<T>::const_iterator
CircularQueue<T>::begin() const
{
return const_iterator(begin_);
}

////////////////////////////////////////////////////////////////
/// \brief      Get the rear iterator (one past the end)
///
////////////////////////////////////////////////////////////////
template <class T>
typename CircularQueue<T>::iterator
CircularQueue<T>::end()
{
return end_;
}

////////////////////////////////////////////////////////////////
/// \brief      Get the rear iterator (one past the end)
///
```

```
////////////////////////////////////////////////////////////////
template <class T>
typename CircularQueue<T>::const_iterator
CircularQueue<T>::end() const
{
return const_iterator(end_);
}


#endif
```

Handler.hpp

```
#ifndef HANDLER_HPP
#define HANDLER_HPP

#include <stdint.h>
#include <ParallelManager.hpp>

////////////////////////////////////////////////////////////////
/// \brief     Abstract base class for an event handler.
///
////////////////////////////////////////////////////////////////
class Handler
{
public:

virtual const char* typeId() const = 0;

Handler() : id_( uniqueId_++ ) {}
Handler( const Handler &copy ) : id_ (copy.id_), destId_ (copy.destId_)
{}
virtual ~Handler() {}

```

```
static void startingId( uint32_t id ) { uniqueId_ = id; }

////////////////////////////////////////////////////////////
/// \brief      Get the id
///
////////////////////////////////////////////////////////////
uint32_t      id() const { return id_; }

////////////////////////////////////////////////////////////
/// \brief      Pure virtual function to handle an event.
///
////////////////////////////////////////////////////////////
virtual void handle() = 0;

////////////////////////////////////////////////////////////
/// \brief      Pure virtual function to determine if the
///             resultant event must be scheduled remotely.
///
////////////////////////////////////////////////////////////
virtual bool isRemote() const
{ return destId_ != ParallelManager::spaceRank_; }

////////////////////////////////////////////////////////////
/// \brief      Pure virtual function to determine the target
///             process of the handler.
///
////////////////////////////////////////////////////////////
virtual uint32_t targetLp() const = 0;

virtual Handler* clone() const = 0;

int32_t destId_ = -1;
```

```
protected:

uint32_t id_;

static uint32_t uniqueId_;

};



////////////////////////////////////////////////////////////////
/// \brief      Equivalence operator
///
////////////////////////////////////////////////////////////////
inline bool operator==(const Handler &lhs, const Handler &rhs)
{
return lhs.id() == rhs.id();
}



#endif
```

MultiEventList.hpp

```
#ifndef MULTIEVENTLIST_HPP
#define MULTIEVENTLIST_HPP

#include "EventList.hpp"
#include <vector>

////////////////////////////////////////////////////////////////
/// \brief      Multiple event list to deal with multiple LPs.
///             the event list using STL vector.
///
////////////////////////////////////////////////////////////////
```

```
class MultiEventList
{
public:
MultiEventList();
MultiEventList( const MultiEventList &copy );
~MultiEventList();

MultiEventList& operator=( const MultiEventList &rhs );

void  createLists( uint32_t listCnt, EventList* templ );

void  add( uint32_t fromLP, Event event );

bool  empty( uint32_t fromLP );

// Doesn't pop the event
bool  head( Event &retval ) const;

// Pops the event
Event getFastestEvent();

// Returns total event count in all of the list
uint32_t getTotalEventCnt();

private:

void finalize();
bool getIndexOfFastestTime( uint32_t &retval ) const;

std::vector<EventList*> multiList_;

};
```

```
#endif /* MULTIEVENTLIST_HPP */
```

SetEventList.hpp

```
#ifndef SETEVENTLIST_HPP
#define SETEVENTLIST_HPP

#include "EventList.hpp"
#include <set>

////////////////////////////////////////////////////////////////
/// \brief      Simple (and inefficient) implementation of
///             the event list using STL's "set" template.
///
///
////////////////////////////////////////////////////////////////
class SetEventList
: public EventList
{
public:
SetEventList();
virtual ~SetEventList();

virtual void      add( Event event );
virtual Event     pop();
virtual bool      empty();

virtual Event     head();
virtual uint32_t  size();

virtual bool      annihilate(
        const Event &event,
```

```
        const bool noexception = false );


virtual EventList* clone() const { return new SetEventList( *this ); }

private:

std::multiset<Event> set_;

};

#endif
```

RandomGen.hpp

```
#ifndef RANDOMGEN_HPP
#define RANDOMGEN_HPP

#include <stdint.h>


///////////////////////////////////////////////////////////////
/// \brief      Random generator. The uniform generator is
///             linear congruential with the same parameters
///             as *rand48 on most *NIX systems.
///
///////////////////////////////////////////////////////////////
class RandomGen
{
public:

RandomGen();
RandomGen( uint32_t seed );

```

```
int32_t  rand();
double   uniform();
bool     bernoulli( double p );
double   normal();
double   normal( double mu, double sig );
double   normalPos( double mu, double sig );
double   exponential( double lambda );

private:

uint64_t advance();

static const uint64_t a;
static const uint64_t c;
static const uint64_t m;

uint64_t r_;

};

#endif
```

Simulator.hpp

```
#ifndef SIMUALTOR_HPP
#define SIMUALTOR_HPP

#include "Handler.hpp"
#include "EventList.hpp"
#include "TscWallClock.hpp"

/////////////////////////////////////////////////////////////////
/// \brief     Abstract base class for the simulator
///            implementation.
```

```
///
////////////////////////////////////////////////////////////////
class Simulator
{
public:

Simulator() : pEventList_ (0x0), logging_(false) {}
virtual ~Simulator() {};

// Initialize the object using an Event List type T
template <class T>
void                    initializeList();

// The child's initializatino routine
virtual void            initialize() = 0;

// Add a neighbor and the appropriate channel delay
virtual void            addNeighbor( uint32_t lp, double lookAhead ) = 0;

// Get the current simulation time
virtual double          simTime() const = 0;

// Schedule an event from a local context
virtual void            scheduleLocal(
         double time,
         const Handler *pHandler ) = 0;

// APIs for spatial parallelism
// Log a remote send (NOTE: default does nothing!)
virtual void            logSentMessage(
         double time,
         const Handler *pHandler ) {}
```

```
// Schedule an event from a remote context
virtual void        scheduleRemote(
        uint32_t lp,
        double time,
        const Handler *pHandler,
uint8_t flag0, uint8_t flag1 ) = 0;

// Run
virtual void        run() = 0;

// Run with tracing
virtual void        run( void (*myfunc) (void ) ) = 0;

// Allocate a new clone
virtual Simulator* clone() const = 0;

// Set whether to use logging or not
void                logging( bool logging ) { logging_ = logging; }

// It's up to the implementation to decide when to start and
// stop the special timer
int64_t             specialTimer() { return specialTimer_.elapsed(); }

bool                 logging_;

protected:

EventList            *pEventList_;
TscWallClock         specialTimer_;

};
```

```
////////////////////////////////////////////////////////////////
/// \brief      Initialize the simulator object by creating
///             a new event list object
///
/// \param      T:  (template parameter) The type of event
///                 list to use.
///
////////////////////////////////////////////////////////////////
template <class T>
void
Simulator::initializeList()
{
pEventList_ = new T ();
initialize();
}


#endif
```

TscWallClock.hpp

```
#ifndef TSCWALLCLOCK_H
#define TSCWALLCLOCK_H

#include <stdint.h>
#include <sys/time.h>

////////////////////////////////////////////////////////////////
/// \brief      Wall clock timer object which utilizes the
///             TSC register on x86 only. Use on other
///             platforms results in null.
///
////////////////////////////////////////////////////////////////
class TscWallClock
```

```
{
public:

TscWallClock();
~TscWallClock() {}
void     start();
void     pause();
void     reset();
int64_t elapsed() { return int64_t(real_); }

private:

uint64_t real_;

struct timespec start_, end_;

};



#endif
```

SequentialSimulator.hpp

```
#ifndef SEQUENTIALSIMULATOR_HPP
#define SEQUENTIALSIMULATOR_HPP

#include "Simulator.hpp"
#include "EventList.hpp"

//////////////////////////////////////////////////////////////////
/// \brief     Implementation of a simulator that runs
///            sequentially on one LP.
///
```

```
//////////////////////////////////////////////////////////////
class SequentialSimulator
: public Simulator
{
public:

SequentialSimulator();
SequentialSimulator( const SequentialSimulator &copy );
SequentialSimulator& operator=( const SequentialSimulator &rhs );
virtual ~SequentialSimulator();

virtual void        initialize();
virtual void        addNeighbor( uint32_t lp, double lookAhead ) {}
virtual double      simTime() const;
virtual void        scheduleLocal( double time, const Handler *pHandler );
virtual void        scheduleRemote(
        uint32_t lp,
        double time,
        const Handler *pHandler,
uint8_t flag0, uint8_t flag1 );
virtual void        run();
virtual void        run( void (*myfunc) (void ) );

virtual Simulator* clone() const
{ return new SequentialSimulator( *this ); }

protected:

double      simTime_;

};

#endif
```

ParallelManager.hpp

```
#ifndef PARALLELMANAGER_HPP
#define PARALLELMANAGER_HPP

#include <stdint.h>
#include <mpi.h>
#include <vector>
#include <queue>

//////////////////////////////////////////////////////////////////
/// \brief     Wrapper class for message passing interface
///            and handling distributed environment
///
///
//////////////////////////////////////////////////////////////////
class ParallelManager
{
public:



static void     initialize(
        int &argc,
        char ** &argv,
        const int& spaceDivision,
        const int& timeDivision );

static void     finalize();

// by default, all the communications are assumed as space Comm
// time comm will be added

template <class T>
```

```
static void      isend( const T &obj, int dst, int tag );

template <class T>
static void      recv( T &obj, int src, int tag );

template <class T>
static MPI_Request isendNb( T &obj, int dst, int tag );

static void      probeBlock();

static bool      probeRecv( int tag, int &from );

static bool      probeRecvAny( int &tag, int &from );

static uint32_t systemCount() { return systemCount_; }

static uint32_t systemId() { return systemId_; }

static uint32_t sent();                // Get sum for all LPs
static uint32_t sent( uint32_t lp );

static uint32_t received();            // Get sum for all LPs
static uint32_t received( uint32_t lp );

static MPI_Comm spaceComm_;
static MPI_Comm timeComm_;

static int spaceRank_;
static int spaceSize_;
static int timeRank_;
static int timeSize_;

private:
```

```
static std::vector<uint32_t> tx_;
static std::vector<uint32_t> rx_;

static int systemCount_;
static int systemId_;

template <class T>
struct SendObject;

template <class T>
class SendQueue;

};

/////////////////////////////////////////////////////////////////
/// TEMPLATE MEMBERS
///



/////////////////////////////////////////////////////////////////
/// \brief      Receive a single object (blocking).
///
///
/// \param      obj Object in which to put
/// \param      tag Message queue tag
///
/////////////////////////////////////////////////////////////////
template <class T>
void
ParallelManager::recv( T &obj, int src, int tag )
{
MPI_Status st;
```

```
MPI_Recv( &obj, sizeof(T), MPI_BYTE, src, tag, spaceComm_, &st );

rx_[ src ]++;

}



////////////////////////////////////////////////////////////////
/// \brief      Helper struct for memory management
///
////////////////////////////////////////////////////////////////
template <class T>
struct
ParallelManager::SendObject
{
T              *pData_;
MPI_Request rq_;
};


////////////////////////////////////////////////////////////////
/// \brief      Wrapper class for a queue.
///
////////////////////////////////////////////////////////////////
template <class T>
class
ParallelManager::SendQueue
{
public:
~SendQueue()
{
while (!queue_.empty())
```

```
{
delete queue_.front().pData_;
queue_.pop();
}
}

SendObject<T>& front() { return queue_.front(); }
void           pop() { queue_.pop(); }
void           push( const SendObject<T> &p ) { queue_.push(p); }
uint32_t       size() const { return queue_.size(); }

private:
std::queue< SendObject<T> > queue_;
};



///////////////////////////////////////////////////////////////
/// \brief      Send a single object (non-blocking). You can
///             queue multiple sends in a row with no worry.
///
/// \param      obj Single object to send (copies it)
/// \param      dst Destination LP
/// \param      tag Message queue tag
///
///////////////////////////////////////////////////////////////
template <class T>
void
ParallelManager::isend( const T &obj, int dst, int tag )
{
//      SendObject<T> sendObject;
//      sendObject.pData_ = new T ( obj );
//
//      // Perform non-blocking send
```

```
//     MPI_Send( sendObject.pData_, sizeof(T), MPI_BYTE, dst, tag,
//               spaceComm_); //, &(sendObject.rq_) );
//
//     delete sendObject.pData_;

// Each send function manages its own list of
// send buffers. This eases the memory management
// aspect.
static SendQueue<T> buffers;

// The object encompassing the data for this send operation
SendObject<T> sendObject;

// Check if we can take the top off the queue
int frontSendCompleted = 0;

if ( buffers.size() )
{
MPI_Test(
        &(buffers.front().rq_),
        &frontSendCompleted,
        MPI_STATUS_IGNORE );
}

if ( frontSendCompleted )
{
// Reuse the memory allocated for the front
sendObject.pData_ = buffers.front().pData_;
*(sendObject.pData_) = obj;

// Remove the front
buffers.pop();
}
```

```
else
{
// Make some new memory instead
sendObject.pData_ = new T ( obj );
}


// Perform non-blocking send
MPI_Isend( sendObject.pData_, sizeof(T), MPI_BYTE, dst, tag,
spaceComm_, &(sendObject.rq_) );

tx_[ dst ]++;

// Add the object to the back
buffers.push( sendObject );

}


////////////////////////////////////////////////////////////////
/// \brief     The raw version which doesn't buffer anything
///
////////////////////////////////////////////////////////////////
template <class T>
MPI_Request
ParallelManager::isendNb( T &obj, int dst, int tag )
{
MPI_Request ret;
// Perform non-blocking send
MPI_Isend( &obj, sizeof(T), MPI_BYTE, dst, tag,
spaceComm_, &ret );

tx_[ dst ]++;
```

```
return ret;
}


#endif
```

RemoteReceiver.hpp

```
#ifndef REMOTERECEIVER_HPP
#define REMOTERECEIVER_HPP

// #define COUT_MESSAGE_TRACE

#include <cstdlib>
#include <stdint.h>
#include <cstring>
#include <map>
#include "Handler.hpp"
#include "ParallelManager.hpp"

//////////////////////////////////////////////////////////////////
/// \brief     This class is used to handle the creation of
///            unique id numbers for each instantiation
///            of the RemoteReceiver template, and is also
///            used to define a way to update reach receiver
///            with one method
///
//////////////////////////////////////////////////////////////////
class RemoteReceiveAggregator
{
public:

typedef void (*ReceiveCallback) ( uint32_t );
```

```
struct Callbacks
{
ReceiveCallback rx;
};

typedef std::map<uint32_t, Callbacks> CallbackMap;



static void receiveAll();
static void sendAll( double time, double *lpBias );

class RemoteMessageBase;

protected:

static uint32_t assignReceiver( ReceiveCallback cb );

static void     scheduleRemoteWrapper( uint32_t lp, double time,
const Handler *pHandler,
uint8_t flag0, uint8_t flag1);

static double   simTimeWrapper();
static bool     getTagInRange(int &tag, int &src);

static CallbackMap& callbackList();
static int&         receiverClassCount();




struct RmbComp;

static std::multimap<RemoteMessageBase*, uint32_t, RmbComp>
sendQueue_;
```

```
};

////////////////////////////////////////////////////////////////
/// \brief    A base structure for sending event data remotely
///
////////////////////////////////////////////////////////////////
class
RemoteReceiveAggregator::RemoteMessageBase
{
public:

RemoteMessageBase() {}
RemoteMessageBase(double t, uint8_t f0, uint8_t f1)
: flag0_(f0), flag1_(f1), time_(t) {}
RemoteMessageBase( const RemoteMessageBase &copy )
: flag0_(copy.flag0_), flag1_(copy.flag1_), time_(copy.time_)
{}
virtual ~RemoteMessageBase() {}

virtual void send( uint32_t lp ) = 0;
virtual const Handler* handler() const = 0;

uint8_t flag0_;
uint8_t flag1_;
double  time_;

};



////////////////////////////////////////////////////////////////
/// \brief    A structure to compare pointers to remote messages
///
```

```
////////////////////////////////////////////////////////////////
struct
RemoteReceiveAggregator::RmbComp
{
bool operator()(const RemoteMessageBase *lhs,
const RemoteMessageBase *rhs) const
{
return lhs->time_<rhs->time_;
}
};


////////////////////////////////////////////////////////////////
/// \brief      Template class to define an interface for sending
///             events remotely.
///
////////////////////////////////////////////////////////////////
template <class T>
class RemoteReceiver
: public RemoteReceiveAggregator
{
public:

class RemoteMessage;

// Send an event to an lp (actually just queue)
static void scheduleSend( uint32_t lp, double time, const T &handler,
uint8_t flag0, uint8_t flag1 );

// Perform a send straight away (used in this module only!)
static void send( const RemoteMessage &out, uint32_t lp );

// This gets any pending event and schedules it locally using
```

```
// SimulatorExecutive::scheduleRemote()
static void receive( uint32_t src );


private:

static uint32_t receiverClassId_;

};


////////////////////////////////////////////////////////////
/// \brief      Identifier tag for a handler type T
///
////////////////////////////////////////////////////////////
template <class T>
uint32_t
RemoteReceiver<T>::receiverClassId_
        = assignReceiver(RemoteReceiver<T>::receive);



////////////////////////////////////////////////////////////
/// \brief      A structure for sending event data remotely
///
////////////////////////////////////////////////////////////
template <class T>
class
RemoteReceiver<T>::RemoteMessage
: public RemoteReceiveAggregator::RemoteMessageBase
{
public:
```

```
RemoteMessage() {}
RemoteMessage( double time, const T &handler, uint8_t f0, uint8_t f1 )
: RemoteMessageBase(time,f0,f1), handler_(handler) {}
RemoteMessage( const RemoteMessage &copy )
: RemoteMessageBase(copy), handler_(copy.handler_)
{}
virtual ~RemoteMessage() {}

virtual void send( uint32_t lp );
virtual const Handler* handler() const {return &handler_;}


T        handler_;

};




////////////////////////////////////////////////////////////////
/// \brief      Queue up data to send
///
/// \param      lp:         Which lp to send to
/// \param      time:       Event sim time
/// \param      handler:    Event handler object
///
////////////////////////////////////////////////////////////////
template <class T>
void
RemoteReceiver<T>::scheduleSend(
        uint32_t lp, double time, const T &handler,
uint8_t flag0, uint8_t flag1)
{
```

```
// Since scheduleSend is the only function called from
// outside of this module, if we NEED the template
// to actually any particular things we need to assert
// them here. This is purely syntactical and for the
// compiler. No effect on run-time.
if (receiverClassId_) {}


RemoteMessage *msg = new RemoteMessage( time, handler, flag0, flag1 );

// Create the output structure and enqueue
sendQueue_.insert( std::pair<RemoteMessage*, uint32_t>(msg, lp) );
}





///////////////////////////////////////////////////////////////
/// \brief     Straightaway send data. Not an interface function.
///
/// \param     out:       Remote message to send
/// \param     lp:        Which lp to send to
///
///////////////////////////////////////////////////////////////
template <class T>
void
RemoteReceiver<T>::send( const RemoteMessage &out, uint32_t lp )
{
ParallelManager::isend( out, lp, receiverClassId_ );
}
```

```
///////////////////////////////////////////////////////////////
/// \brief      Do the physical sending of data
///
/// \param      lp:   lp to send to
///
///////////////////////////////////////////////////////////////
template <class T>
void
RemoteReceiver<T>::RemoteMessage::send( uint32_t lp )
{
RemoteReceiver<T>::send( *this, lp );
}



///////////////////////////////////////////////////////////////
/// \brief      Receive up to one pending events and schedule
///             them.
///
///             One-shot blocking receive that ALWAYS returns
///             only after a reception.
///
///////////////////////////////////////////////////////////////
template <class T>
void
RemoteReceiver<T>::receive(  uint32_t src )
{

// Get the message
RemoteMessage inbuf;
ParallelManager::recv( inbuf, src, receiverClassId_ );

// Schedule the message (we have to use this wrapper function to
```

```
// actually make this compilable... forward declarations can't fix
// this problem)

//scheduleRemoteWrapper( src, inbuf.time_, &inbuf.handler_ );

// todo: better fix (we do this to ``fix'' the VTABLE
T handler( inbuf.handler_ );
scheduleRemoteWrapper( src, inbuf.time_, &handler,
inbuf.flag0_, inbuf.flag1_ );

}



#endif
```

StateDataHistory.hpp

```
#ifndef STATEDATAHISTORY_HPP
#define STATEDATAHISTORY_HPP

#include <stdint.h>
#include <utility>
#include "CircularQueue.hpp"



///////////////////////////////////////////////////////////////
/// \brief      Base class to define a class which holds and
///             manipulates state data via a reference to
///             the actual data and a history structure.
///
///////////////////////////////////////////////////////////////
class StateDataHistoryBase
{
public:
```

```
StateDataHistoryBase() {}
virtual ~StateDataHistoryBase() {}

// Set the max history size
virtual void     maxsize( uint32_t size ) = 0;

// Max maximum capacity
virtual uint32_t maxsize() const = 0;

// Get number of enqueued elements
virtual uint32_t size() const = 0;

bool full() const { return size() == maxsize(); }

// Create a new history entry at time by copying the current value
virtual void     push( double time ) = 0;

// Free up old entries up until time
virtual void     free( double time ) = 0;

// Delete newer entries to go back to time and set the current state
virtual void     rollback( double time ) = 0;

virtual StateDataHistoryBase* clone() const = 0;

};



////////////////////////////////////////////////////////////////
/// \brief     Class implementation for arbitrary data type
///
```

```
////////////////////////////////////////////////////////////////
template <class T>
class StateDataHistory
: public StateDataHistoryBase
{
public:

// Pair of a time value and a state object
typedef std::pair<double, T> HistoryType;

StateDataHistory( T *pState );
virtual ~StateDataHistory();

virtual void     maxsize( uint32_t size );
virtual uint32_t maxsize() const;
virtual uint32_t size() const;
virtual void     push( double time );
virtual void     free( double time );
virtual void     rollback( double time );

virtual StateDataHistoryBase* clone() const
{ return new StateDataHistory(*this); }

private:

// Queue of history values
CircularQueue<HistoryType> queue_;

// Pointer to current working state (is shallow copied)
T *currentState_;

};
```

```
////////////////////////////////////////////////////////////
/// \brief      ctor
///
////////////////////////////////////////////////////////////
template <class T>
StateDataHistory<T>::StateDataHistory( T *pState )
{
currentState_ = pState;
}

////////////////////////////////////////////////////////////
/// \brief      dtor
///
////////////////////////////////////////////////////////////
template <class T>
StateDataHistory<T>::~StateDataHistory()
{
// nada
}

////////////////////////////////////////////////////////////
/// \brief      Set the max size
///
/// \param      size: new maximum size (capacity)
///
////////////////////////////////////////////////////////////
template <class T>
void
StateDataHistory<T>::maxsize( uint32_t size )
{
queue_.reserve(size);
}
```

```
////////////////////////////////////////////////////////////////
/// \brief      Get the max size
///
/// \return     Current capacity
///
////////////////////////////////////////////////////////////////
template <class T>
uint32_t
StateDataHistory<T>::maxsize() const
{
return queue_.capacity();
}

////////////////////////////////////////////////////////////////
/// \brief      Query the number of enqueued history entries
///
/// \return     Current number of history entries
///
////////////////////////////////////////////////////////////////
template <class T>
uint32_t
StateDataHistory<T>::size() const
{
return queue_.size();
}

////////////////////////////////////////////////////////////////
/// \brief      Push the current value at "time"
///
/// \param      time: Time value associated with current state
///                   being pushed.
///
```

```
///////////////////////////////////////////////////////////////
template <class T>
void
StateDataHistory<T>::push( double time )
{
queue_.push( HistoryType(time, *currentState_) );
}

///////////////////////////////////////////////////////////////
/// \brief      Free old entries up until "time"
///
/// \param      time: Time value up to which to free older
///                   entries. Frees all entries marked
///                   as at or before time.
///
///////////////////////////////////////////////////////////////
template <class T>
void
StateDataHistory<T>::free( double time )
{
while ( !queue_.empty() && queue_.begin()->first <= time )
{
queue_.pop();
}
}

///////////////////////////////////////////////////////////////
/// \brief      Rollback newer entries down until "time", and
///             then set the current value of currentState to be
///             the newest entry.
///
/// \param      time: Time value down to which to free newer
///                   entries. Deletes all entries greater than
```

```
///                    time, then takes the next entry as the
///                    current state before removing it from
///                    the list.
///
////////////////////////////////////////////////////////////////
template <class T>
void
StateDataHistory<T>::rollback( double time )
{
if ( queue_.empty() ) return;

typename CircularQueue<HistoryType>::iterator back = queue_.end() - 1;
while ( queue_.size() > 1 && back->first > time )
{
queue_.popBack();
back--;
}

// Set the current state
// Note that rollback() ALWAYS rollback at least once!
*currentState_ = back->second;
queue_.popBack();
}



#endif
```

AntiMessageHandler.hpp

```
#ifndef ANTIMESSAGEHANDLER_HPP
#define ANTIMESSAGEHANDLER_HPP

#include "Handler.hpp"
#include "ParallelManager.hpp"
```

```
////////////////////////////////////////////////////////////
/// \brief      Encompasses an anti-message. Note that when
///             properly constructed, an anti-message and its
///             complementary normal message will yield
///             equivalence even though the they share no
///             "real" parameters. This is because they share
///             the same unique id.
///
////////////////////////////////////////////////////////////
class AntiMessageHandler
: public Handler
{
public:

virtual const char* typeId() const
{ return "AntiMessageHandler"; }

AntiMessageHandler() {}
AntiMessageHandler( const Handler &copy )
: Handler(copy), targetLp_( copy.targetLp() ) {}

virtual ~AntiMessageHandler() {}

virtual void     handle()   {}

virtual bool     isRemote() const
{ return ParallelManager::spaceRank_ != targetLp_; }

virtual uint32_t targetLp() const { return targetLp_; }

virtual Handler* clone() const
{ return new AntiMessageHandler(*this); }
```

```
protected:

uint32_t targetLp_;

};

#endif
```

TimeWarpSimulator.hpp

```
#ifndef TIMEWARPSIMULATOR_HPP
#define TIMEWARPSIMULATOR_HPP

#include <list>
#include <deque>
#include "Simulator.hpp"
#include "StateDataHistory.hpp"
#include "CircularQueue.hpp"
#include "NullMessageHandler.hpp"

/////////////////////////////////////////////////////////////////
/// \brief     TW simulator implementation
///
/////////////////////////////////////////////////////////////////
class TimeWarpSimulator
: public Simulator
{
public:

static uint16_t rollbackcounter;

TimeWarpSimulator();
TimeWarpSimulator( const TimeWarpSimulator &copy );
```

```
TimeWarpSimulator& operator=( const TimeWarpSimulator &rhs );
virtual ~TimeWarpSimulator();

virtual void        initialize();
virtual void        addNeighbor( uint32_t lp, double lookAhead );
virtual double      simTime() const;
virtual void        scheduleLocal( double time, const Handler *pHandler );
virtual void        logSentMessage( double time, const Handler *pHandler );
virtual void        scheduleRemote(
        uint32_t lp,
        double time,
        const Handler *pHandler,
uint8_t flag0, uint8_t flag1 );
virtual void        run();
virtual void        run( void (*myfunc) (void) );

virtual Simulator* clone() const
{ return new TimeWarpSimulator(*this); }


// User interface function special for Time Warp
template <class T>
static void         attach( T &state, uint32_t historySize );


protected:

// Enum describing types of global control to perform
enum GlobalControl
{
GC_NONE = 0,
GC_SYNC = 1,
ENUM_MX = 2
```

```
};

// Enum describing termination states
enum State
{
NOT_DONE     = 0,
LOCAL_DONE   = 1,
GLOBAL_DONE = 2,
NUM_STATES
};

// Enum describing flag0 of the message structure
enum MessageFlag
{
NORM_MSG = 0,
ANTI_MSG = 1,
NULL_MSG = 2,
NUM_MSGS
};

// Class to hold input queue messages
struct EventState
{
EventState() : event_(0.0,0x0) {}
EventState(const Event &e) : event_(e) {}
Event              event_;
std::deque<Event> outQueue_;
};


// Private methods
void          rollback( double time, const Handler *pHandler = 0x0 );
void          globalControl( GlobalControl gc );
```

```
GlobalControl  checkGlobalControl();
void           commitGvt( double newGvt );
void           sendAntiMessage( const Event &event );
void           annihilate( double time, const Handler *pHandler );
void           initiateGlobalControl();
State          localState();
void           sendNullMessages();
bool           checkLpsState( State state );

// Member objects
double                   simTime_;
double                   gvt_;
bool                     terminating_;
uint32_t                 historySize_;
StateDataHistoryBase *pStateHistory_;   // State queue
// lookahead values for doing remote sends
std::vector<double>  lookAhead_;
// List of states for my neighbors
std::vector<State>    stateList_;
// Buffer for initiating global control
GlobalControl            txBuf_;

// Processed events queue (events I handled)
CircularQueue<EventState> processedQueue_;

// Hold anti-message temporarily, if it is not found
std::list<Event>              antiMsgHolder_;




// Constant objects
// Okay to use the tag 0 since the handler tags start at 1
```

```
static const int       gcTag_ = 0;

// Static members (only used for initial attach)
static StateDataHistoryBase *pInitialHistoryToAttach_;
static uint32_t              initialHistorySize_;

};



///////////////////////////////////////////////////////////////
/// \brief      Static template user interface to attach
///             state data to the simulator.
///             NOTE: This attach must be performed BEFORE the
///                   call to initialize!
///
/// \param      state: Reference to state data
/// \param      historySize: size of the 3 queues
///
///////////////////////////////////////////////////////////////
template <class T>
void
TimeWarpSimulator::attach( T &state, uint32_t historySize )
{
// This value will be cloned into the actual object.
// Note that this object will not be freed during main(). That is okay!
pInitialHistoryToAttach_ = new StateDataHistory<T>( &state );

initialHistorySize_ = historySize;
}



#endif
```

SimulatorExecutive.hpp

```
#ifndef SIMULATOREXECUTIVE_HPP
#define SIMULATOREXECUTIVE_HPP

#include <cstdlib>
#include <iostream>
#include "Handler.hpp"
#include "Simulator.hpp"
#include "RemoteReceiver.hpp"
#include "SimApplication.hpp"

////////////////////////////////////////////////////////////////
/// \brief      Static class encompassing the simulation
///             executive. Note: since this is static,
///             there is only ONE per LP.
///
////////////////////////////////////////////////////////////////
class SimulatorExecutive
{
public:

static uint16_t startTime;
static uint16_t endTime;

static std::string scheduleFile_;
static std::uint16_t spaceDivision_;
static std::uint16_t timeDivision_;

static uint32_t randSeed_;
static double dupRatio_;

static double simTime();
```

```
static void   run();
static void   run( void (*myfunc)(void) );

template <class T, class T2>
static void   initialize(
        std::string schedulFile,
        const uint16_t spaceDivision,
        const uint16_t timeDivision = 1);

static void   finalize();

template <class T>
static void   schedule( double time, const T &handler );

// Note that in contrast to schedule(), scheduleRemote() takes
// a pointer to the handler. The origin for this discrepancy lies
// in forward declaration problems with the headers,  nothing more.
static void   scheduleRemote(
        uint32_t lp,
        double time,
        const Handler *pHandler,
uint8_t flag0, uint8_t flag1 );

static void   addNeighbor( uint32_t lp, double lookAhead );

static void   logging( bool logging );

// Get the value of the "special" timer after running.
// The special timer metric depends on the sim impl
static int64_t specialTimer();

private:
```

```
static Simulator *pSimulator_;

static size_t timeRound_;

// post fix-up computation for time parallelism
static bool   checkCompleted(bool &isThisLpSkip);
static void   postProcess();
static void   doFixup( const size_t numLPs, short* times, short* updates );
static void   postProcess_naive();
static void   postProcess_gather();
static void   postProcess_scan();

};



/////////////////////////////////////////////////////////////
/// \brief     Initialize the simulator to a known state.
///
/// \param     T:    (template parameter) Which simulator type
///                  to use.
/// \param     T2:   (template parameter) Which event list type
///                  to use;
///
/////////////////////////////////////////////////////////////
template <class T, class T2>
void
SimulatorExecutive::initialize(
        std::string schedulFile,
        const uint16_t spaceDivision,
        const uint16_t timeDivision)
{
delete pSimulator_;
```

```
pSimulator_ = new T ();
pSimulator_->initializeList<T2> ();

scheduleFile_ = schedulFile;
spaceDivision_ = spaceDivision;
timeDivision_ = timeDivision;

// We need to set the handler starting id so they don't overlap
// between LPs

Handler::startingId( (0xFFFFFFFF / ParallelManager::systemCount())
* ParallelManager::systemId() );



// All are connected
for (size_t i = 0; i < ParallelManager::spaceSize_; ++i)
{
if (i != ParallelManager::spaceRank_)
{
std::cout << "set lookahead: "
        << nassimulator::SimApplication::sendHorizon
        << std::endl;
addNeighbor( i, nassimulator::SimApplication::sendHorizon );
// addNeighbor( i, 1 );
}
// else
}

}



///////////////////////////////////////////////////////////////
/// \brief     Schedule an event using an object.
```

```
///
/// \param      time      Time (s) at which event occurs
/// \param      handler   Event handler object
///
/////////////////////////////////////////////////////////////
template <class T>
void
SimulatorExecutive::schedule( double time, const T &handler )
{

if ( pSimulator_ )
{

if ( handler.isRemote() )
{

// Send with null flags and a message Id
RemoteReceiver<T>::scheduleSend(
        handler.targetLp(),
        time,
        handler, 0, 0 );
pSimulator_->logSentMessage( time, &handler );
}
else
{
// test log
// std::cout << "schedule time: " << time << std::endl;
// std::cout << testcount++ << std::endl;
// It's okay to pass this reference because the new Event
// object clones the handler object
pSimulator_->scheduleLocal(time, &handler);
}
}
```

```
else
{
std::cerr << "ERROR: Simulator not initialized. Exiting." << std::endl;
exit( EXIT_FAILURE );
}
}

#endif
```

Flight.hpp

```
#ifndef FLIGHT_HPP
#define FLIGHT_HPP

#include <string.h>
#include <string>
#include <sstream>
#include <iostream>

#include "Logger.hpp"



namespace nassimulator {

// @TODO month and date to get daylight savings.
/////////////////////////////////////////////////////////////
/// \brief      LocalTime data class
///
/////////////////////////////////////////////////////////////
class LocalTime
{
public:

static LocalTime GetNewTime( const LocalTime &begin,
```

```
const size_t elapedMin,
const double utcOffset )
{
// std::cout << "begin.hour_: " << begin.hour_ << std::endl;
// std::cout << "begin.minute_: " << begin.minute_ << std::endl;
// std::cout << "elapsedMin: " << elapedMin << std::endl;
// std::cout << "utcOffset: " << utcOffset << std::endl;

size_t newHour = begin.hour_ + elapedMin / 60;
size_t newMins = begin.minute_ + elapedMin % 60;
if (newMins > 59) {
newHour++;
newMins -= 60;
}

newHour += ( utcOffset - begin.utcOffset_ );

LocalTime newTime( newHour, newMins, utcOffset );

return newTime;
}

static LocalTime GetNewTime( const size_t simTime,
const double utcOffset )
{
size_t newHour = simTime / 60;
size_t newMins = simTime % 60;

newHour += ( utcOffset + 4 );

LocalTime newTime( newHour, newMins, utcOffset );

return newTime;
```

```
}

static int GetDiff( LocalTime time1, LocalTime time2 )
{
return time1.getMostEastTimeMinute() - time2.getMostEastTimeMinute();
}

/////////////////////////////////////////////////////////////
/// \brief      default constructor
///
/////////////////////////////////////////////////////////////
LocalTime() {}

/////////////////////////////////////////////////////////////
/// \brief      constructor
///
/////////////////////////////////////////////////////////////
LocalTime(
        std::string timestring,
        double utcOffset,
        bool plusoneday = false )
: utcOffset_(utcOffset)
{
size_t len = timestring.length();
if ( len < 3 )
{
hour_ = 0;
minute_ = atoi(timestring.c_str());
}
else
{
hour_ = atoi( timestring.substr(0, len - 2).c_str() );
minute_ = atoi( timestring.substr( len - 2, 2 ).c_str() );
```

```
}

if (plusoneday)
hour_ += 24;
}

///////////////////////////////////////////////////////////////////
/// \brief constructor
///
///////////////////////////////////////////////////////////////////
LocalTime( size_t hour, size_t minute, double utcOffset )
: utcOffset_(utcOffset), hour_(hour), minute_(minute) { }

///////////////////////////////////////////////////////////////////
/// \brief assignment
///
///////////////////////////////////////////////////////////////////
LocalTime& operator=( const LocalTime &rhs )
{
if (this != &rhs)
{
this->hour_ = rhs.hour_;
this->minute_ = rhs.minute_;
this->utcOffset_ = rhs.utcOffset_;
}
return *this;
}

// This is puerto rico time (UTC-4), Estern time is also UTC -4 during DST
size_t getMostEastTimeMinute() const
{
return 60 * (hour_ - (int)( 4.0 + utcOffset_ )) + minute_;
}
```

```
size_t hour_;

size_t minute_;

double utcOffset_;

};

/////////////////////////////////////////////////////////////////
/// \brief      Airport data class
///
/////////////////////////////////////////////////////////////////
class AirportInfo
{
public:

/////////////////////////////////////////////////////////////////
/// \brief      default constructor
///
/////////////////////////////////////////////////////////////////
AirportInfo() {}

AirportInfo(
std::string airportId,
std::string airportFullName,
std::string cityName,
double longitude,
double latitude,
double utcOffset,
std::string daylightSaving
)
: airportId_(airportId),
```

```
airportFullName_(airportFullName),
cityName_(cityName),
longitude_(longitude),
latitude_(latitude),
utcOffset_(utcOffset),
daylightSaving_(daylightSaving) {}

std::string airportId_;
std::string airportFullName_;
std::string cityName_;

double longitude_;
double latitude_;

double utcOffset_;

std::string daylightSaving_;

};

/////////////////////////////////////////////////////////////////
/// \brief    Flight status data class
///
/////////////////////////////////////////////////////////////////
class FlightStat
{
public:

/////////////////////////////////////////////////////////////////
/// \brief    default constructor
///
/////////////////////////////////////////////////////////////////
FlightStat() {}
```

```
};

////////////////////////////////////////////////////////////
/// \brief      Flight data class
///
////////////////////////////////////////////////////////////
class Flight
{
public:

////////////////////////////////////////////////////////////
/// \brief      default constructor
///             @TODO date information?
////////////////////////////////////////////////////////////
Flight() {}

////////////////////////////////////////////////////////////
/// \brief      copy ctor
///
////////////////////////////////////////////////////////////
Flight( const Flight &copy )
{
strncpy(this->carrierId_, copy.carrierId_, 8);
strncpy(this->tailNum_, copy.tailNum_, 8);
this->flightNum_ = copy.flightNum_;
this->originAirportId_ = copy.originAirportId_;
this->destAirportId_ = copy.destAirportId_;
this->crsDepTime_ = copy.crsDepTime_;
this->crsArrTime_ = copy.crsArrTime_;
this->actDepTimeRec_ = copy.actDepTimeRec_;
```

```
this->actArrTimeRec_ = copy.actArrTimeRec_;
this->crsFlightTime_ = copy.crsFlightTime_;
this->depDelRec_ = copy.depDelRec_;
this->arrDelRec_ = copy.arrDelRec_;

// result data
this->actDepTime_ = copy.actDepTime_;
this->actArrTime_ = copy.actArrTime_;

// for time parallel simulator
this->isInTimeBin_ = copy.isInTimeBin_;
}

/////////////////////////////////////////////////////////////
/// \brief       constructor
///              @TODO date information?
/////////////////////////////////////////////////////////////
Flight(
std::string carrierId,
std::string tailNum,
size_t flightNum,
size_t originAirportId,
size_t destAirportId,
LocalTime crsDepTime,
LocalTime crsArrTime,
LocalTime actDepTimeRec,
LocalTime actArrTimeRec,
size_t crsFlightTime,
int depDelRec,
int arrDelRec )
: /* carrierId_(carrierId),
tailNum_(tailNum), */
flightNum_(flightNum),
```

```
originAirportId_(originAirportId),
destAirportId_(destAirportId),
crsDepTime_(crsDepTime),
crsArrTime_(crsArrTime),
crsFlightTime_(crsFlightTime),
actDepTimeRec_(actDepTimeRec),
actArrTimeRec_(actArrTimeRec),
depDelRec_(depDelRec),
arrDelRec_(arrDelRec) {

strncpy(carrierId_, carrierId.c_str(), 8);
strncpy(tailNum_, tailNum.c_str(), 8);

}



// size_t GlobalTime() const { return 0; }

//////////////////////////////////////////////////////////////
/// \brief      dtor
///
//////////////////////////////////////////////////////////////
~Flight() { }



//////////////////////////////////////////////////////////////
/// \brief      assignment
///
//////////////////////////////////////////////////////////////
Flight& operator=( const Flight &rhs )
{
if (this != &rhs)
{
```

```
strncpy(this->carrierId_, rhs.carrierId_, 8);
strncpy(this->tailNum_, rhs.tailNum_, 8);
this->flightNum_ = rhs.flightNum_;
this->originAirportId_ = rhs.originAirportId_;
this->destAirportId_ = rhs.destAirportId_;
this->crsDepTime_ = rhs.crsDepTime_;
this->crsArrTime_ = rhs.crsArrTime_;
this->actDepTimeRec_ = rhs.actDepTimeRec_;
this->actArrTimeRec_ = rhs.actArrTimeRec_;
this->crsFlightTime_ = rhs.crsFlightTime_;
this->depDelRec_ = rhs.depDelRec_;
this->arrDelRec_ = rhs.arrDelRec_;

// result data
this->actDepTime_ = rhs.actDepTime_;
this->actArrTime_ = rhs.actArrTime_;

// for time parallel simulator
this->isInTimeBin_ = rhs.isInTimeBin_;
}
return *this;
}

///////////////////////////////////////////////////////////////
/// \brief       Peek at the flight information
///
///////////////////////////////////////////////////////////////
void     writeLog()
{
std::ostringstream buf;
buf << "Flight: " << carrierId_ << flightNum_;
buf << ", Tail Num: " << tailNum_;
buf << ", origin: " << originAirportId_;
```

```
buf << ", crsDepTime: " << crsDepTime_.hour_
         << ":" << crsDepTime_.minute_;
buf << ", crsDepTime(Estern): " << crsDepTime_.getMostEastTimeMinute();
buf << ", dest: " << destAirportId_;
buf << ", crsArrTime: " << crsArrTime_.hour_
         << ":" << crsArrTime_.minute_;
buf << ", crsArrTime(Estern): " << crsArrTime_.getMostEastTimeMinute();
buf << ", isInTimeBin_: " << isInTimeBin_;
// buf << ", isInSpaceBin_: " << isInSpaceBin_;
Logger::getLogger("Flight")->writeLog(buf.str());
}

// input data - fixed from the first time
char carrierId_[8] = {0, };
char tailNum_[8] = {0, };
// std::string carrierId_;
// std::string tailNum_;
size_t flightNum_;
size_t originAirportId_;
size_t destAirportId_;
LocalTime crsDepTime_;
LocalTime crsArrTime_;
LocalTime actDepTimeRec_;
LocalTime actArrTimeRec_;
size_t crsFlightTime_;
int depDelRec_;
int arrDelRec_;

// result data
LocalTime actDepTime_;
LocalTime actArrTime_;

// for time parallel simulator
```

```
bool isInTimeBin_ = false;
// for space parallel simulator
// bool isInSpaceBin_ = false;

private:

};

class CompareFlight
{
public:
bool operator() (Flight const &a, Flight const &b)
{
return a.actDepTimeRec_.getMostEastTimeMinute()
        > b.actDepTimeRec_.getMostEastTimeMinute();

}

};

struct
{
bool operator() (Flight const &a, Flight const &b)
{
return a.actDepTimeRec_.getMostEastTimeMinute()
        < b.actDepTimeRec_.getMostEastTimeMinute();
}

} sCompareFlight;



} // nassimulator
```

```
#endif
```

AirportManager.hpp

```
#ifndef AIRPORTMANAGER_HPP
#define AIRPORTMANAGER_HPP

#include <list>
#include <vector>
#include "Flight.hpp"

namespace nassimulator {



struct delayPropInfo {

delayPropInfo() : delayProp(0), numFlights(0) {}
delayPropInfo(
        uint32_t delayProp,
        uint32_t numFlights) : delayProp(delayProp),
                        numFlights(numFlights) {}

uint32_t delayProp;
uint32_t numFlights;
};

struct airportState {

airportState(uint16_t timestamp) : timestamp_(timestamp),
                waittime_(0) { waitflights_.clear(); }

uint16_t timestamp_;
uint16_t waittime_;
```

```
std::vector<std::string> waitflights_;
};

////////////////////////////////////////////////////////////////
/// \brief      Airport Manager class
///
////////////////////////////////////////////////////////////////
class AirportManager
{
public:

static const size_t LANDING_TIME = 0;
static const size_t TAXI_TIME = 0;

std::vector<airportState> airportStates;
//   uint16_t startTime;

////////////////////////////////////////////////////////////////
/// \brief      constructor
///
////////////////////////////////////////////////////////////////
explicit AirportManager(
        size_t airportId,
        std::string airportname,
        uint16_t orderInTime )
: airportId_(airportId),
        airportname_(airportname),
        orderInTime_(orderInTime)
{
// Add information to the list for distributing airports across LPs



}
```

```
///////////////////////////////////////////////////////////////
/// \brief      dtor
///
///////////////////////////////////////////////////////////////
~AirportManager() { }

void addInFlight( Flight flight ) { inFlights.push_back(flight); }
void addOutFlight( Flight flight ) { outFlights.push_back(flight); }

bool isInRunwayEmpty() { return inRunwayEmpty_; }
void setInRunwayEmpty( bool isEmpty ) { inRunwayEmpty_ = isEmpty; }
bool isOutRunwayEmpty() { return outRunwayEmpty_; }
void setOutRunwayEmpty( bool isEmpty ) { outRunwayEmpty_ = isEmpty; }

bool hasNextInFlight() { return (inFlights.size() > 0); }
Flight popNextInFlight()
{
Flight nextFlight;

if (inFlights.size() > 0)
{
nextFlight = inFlights.front();
inFlights.pop_front();
}

return nextFlight;
}

size_t getLandingTime() { return LANDING_TIME; }
void fixupRemoveFlight(std::string flightname) {}
void fixupAddFlight(std::string flightname, uint16_t delay) {}
```

```
size_t getOrderInTime() { return orderInTime_; }
std::string airportname_;

private:

static void addNewAirport();
static float altMin_;
static float altMax_;
static float latMin_;
static float latMax_;

double  airportId_;
uint16_t  orderInTime_;
bool    inRunwayEmpty_ = true;
bool    outRunwayEmpty_ = true;

std::list< Flight > inFlights;
std::list< Flight > outFlights;

};

} // nassimulator


#endif // AIRPORTMANAGER_HPP
```

AirportTrafficImporter.hpp

```
#ifndef AIRTRAFFICIMPORTER_HPP
#define AIRTRAFFICIMPORTER_HPP

#include <string>
#include <map>
#include <vector>
```

```
#include <queue>
#include <memory>

#include "EventList.hpp"
#include "Flight.hpp"
#include "AirportManager.hpp"

namespace nassimulator {

enum ScdItem {

Year = 0,
Month = 1,
DayofMonth,
DayOfWeek,
FlightDate,
UniqueCarrier,
AirlineID,
Carrier,
TailNum,
FlightNum,
OriginAirportID,
OriginAirportSeqID,
OriginCityMarketID,
Origin,
OriginCityName,
OriginState,
OriginStateFips,
OriginStateName,
OriginWac,
DestAirportID,
DestAirportSeqID,
DestCityMarketID,
```

```
Dest,
DestCityName,
DestState,
DestStateFips,
DestStateName,
DestWac,
CRSDepTime,
DepTime,
DepDelay,
DepDelayMinutes,
DepDel15,
DepartureDelayGroups,
DepTimeBlk,
TaxiOut,
WheelsOff,
WheelsOn,
TaxiIn,
CRSArrTime,
ArrTime,
ArrDelay,
ArrDelayMinutes,
ArrDel15,
ArrivalDelayGroups,
ArrTimeBlk,
Cancelled,
CancellationCode,
Diverted,
CRSElapsedTime,
ActualElapsedTime,
AirTime,
Flights,
Distance,
DistanceGroup,
```

```
CarrierDelay ,
WeatherDelay ,
NASDelay ,
SecurityDelay ,
LateAircraftDelay ,
FirstDepTime ,
TotalAddGTime ,
LongestAddGTime ,
DivAirportLandings ,
DivReachedDest ,
DivActualElapsedTime ,
DivArrDelay ,
DivDistance ,
Div1Airport ,
Div1AirportID ,
Div1AirportSeqID ,
Div1WheelsOn ,
Div1TotalGTime ,
Div1LongestGTime ,
Div1WheelsOff ,
Div1TailNum ,
Div2Airport ,
Div2AirportID ,
Div2AirportSeqID ,
Div2WheelsOn ,
Div2TotalGTime ,
Div2LongestGTime ,
Div2WheelsOff ,
Div2TailNum ,
Div3Airport ,
Div3AirportID ,
Div3AirportSeqID ,
Div3WheelsOn ,
```

```
Div3TotalGTime,
Div3LongestGTime,
Div3WheelsOff,
Div3TailNum,
Div4Airport,
Div4AirportID,
Div4AirportSeqID,
Div4WheelsOn,
Div4TotalGTime,
Div4LongestGTime,
Div4WheelsOff,
Div4TailNum,
Div5Airport,
Div5AirportID,
Div5AirportSeqID,
Div5WheelsOn,
Div5TotalGTime,
Div5LongestGTime,
Div5WheelsOff,
Div5TailNum

};



struct stateOrder
{
stateOrder()=default;

stateOrder(
        uint16_t regionid,
        uint16_t orderinregion,
        uint16_t lpCount,
        uint16_t absorder,
```

```
        uint16_t spacedivision )
{
this->regionid_ = regionid;
this->orderinregion_ = orderinregion;
this->absorder_ = absorder;
calcLpOrder(spacedivision, lpCount);
};

stateOrder( const stateOrder& copy )
{
this->regionid_ = copy.regionid_;
this->orderinregion_ = copy.orderinregion_;
this->absorder_ = copy.absorder_;
this->lporderintime_ = copy.lporderintime_;
}

stateOrder& operator=( const stateOrder &rhs )
{
if (this != &rhs)
{
this->regionid_ = rhs.regionid_;
this->orderinregion_ = rhs.orderinregion_;
this->absorder_ = rhs.absorder_;
this->lporderintime_ = rhs.lporderintime_;
}
return *this;
}

void calcLpOrder(
uint16_t spacedivision,
uint16_t lpCount
)
{
```

```
if (spacedivision == 5)
{
this->lporderintime_ = this->regionid_;
}
else if (spacedivision == 53)
{
this->lporderintime_ = this->absorder_;
}
else if (spacedivision < 54)
{
// @TODO check
uint16_t div = 53 / spacedivision;
uint16_t remainder = 53 % spacedivision;

if (absorder_ < (div + 1)*remainder)
{
this->lporderintime_ = this->absorder_ / (div + 1);
}
else
{
this->lporderintime_
        = (absorder_ - (div + 1)*remainder) / div + remainder;
}
}
else
{
std::cerr
        << "Cannot divide more than 53 spatial regions for now."
        << std::endl;
}
// lpid_
}
```

```
uint16_t regionid_;
uint16_t orderinregion_;
uint16_t absorder_;
uint16_t lporderintime_;

};

// ///////////////////////////////////////////////////////////
/// \brief      Static class to import air traffic data
///
// ///////////////////////////////////////////////////////////
class AirTrafficImporter
{
public:

// @TODO Import from database?

static void ReadAirportData(
const std::string& fileName,
std::map< std::string, AirportInfo > &airports,
uint32_t lpCount = 1,
uint32_t lpId = 0,
uint32_t timedivision = 1
);

static void Import_v2(
const uint32_t rndSeed,
double duplicateRatio,
const std::string& fileName,
uint16_t& startTime,
uint16_t& endTime,
uint16_t& timeorder,
uint16_t& spaceorder,
```

```
std::map< std::string, Flight > &flightsMap,
std::map< std::string,
        std::priority_queue<Flight, std::vector<Flight>,
                            CompareFlight> > &schedules,
std::map< size_t, std::shared_ptr<AirportManager> > &airportManagers,
std::map< std::string, delayPropInfo > &delayPropMap,
uint32_t lpCount = 1,
uint32_t lpId = 0,
uint32_t timedivision = 1,
uint32_t spacedivision = 1
);

private:

static std::vector<Flight> entireFlights_;
static bool dataInitialLoaded_;
static std::map< std::string,
std::priority_queue<Flight, std::vector<Flight>,
CompareFlight> >
schedulemapKeep;

static uint32_t virtualFlightNo;
static uint32_t virtualTailNo;

};



} // nassimulator

#endif
```

SimApplication.hpp

```
#ifndef SIMAPPLICATION_HPP
```

```
#define SIMAPPLICATION_HPP

#include <queue>
#include <memory>
#include "Event.hpp"
#include "Handler.hpp"
#include "Flight.hpp"
#include "AirportManager.hpp"
#include "SimApplicationStateData.hpp"

namespace nassimulator {



/////////////////////////////////////////////////////////////////
/// \brief      Static application interface. Serves as a
///             wrapper interface for an instance in sim
///             time of the application state data.
///
/////////////////////////////////////////////////////////////////
class SimApplication
{
public:

static uint16_t timeorder;
static uint16_t spaceorder;
static uint16_t spaceregion;
static uint16_t sendHorizon;

// simulator options
static size_t DUMMY_COUNT;
static size_t FIXUP_METHOD;
static size_t INITIAL_DIST;
```

;

```
// debug counter
static size_t arrCounter;

static std::map< size_t, std::shared_ptr<AirportManager> >
        airportManagers;
static std::map< std::string, AirportInfo > airportData;
static std::map< std::string, Flight > flightsMap;
static std::map< std::string,
std::priority_queue<Flight, std::vector<Flight>,
CompareFlight> >
schedulemap;
static std::map< std::string,
std::vector<Flight> >
schedulemap_complete;

static std::map< std::string, uint32_t > fixupMsg;

static std::map< std::string, delayPropInfo > delayPropMap;
static std::map< std::string, uint32_t > delayUpdatedFromPrev;



size_t status_;

// Wrapper/helper functions

// Helper
static SimApplicationStateData& state() { return state_; }

private:

static SimApplicationStateData state_;

};
```

```
class NasHandler : public Handler
{
public:
Flight flight_;

NasHandler( ) : Handler() {}
NasHandler( Flight flight ) : flight_(flight) {}
NasHandler( const NasHandler& copy )
: Handler(copy), flight_(copy.flight_)
{}

};



class DepartedHandler : public NasHandler
{
public:
virtual const char* typeId() const { return "DepartedHandler"; }

// ctor
DepartedHandler() : NasHandler() {}
DepartedHandler( Flight flight ) : NasHandler(flight) {}
DepartedHandler( const DepartedHandler& copy )
: NasHandler(copy)
{}

// dtor
virtual ~DepartedHandler() {}

// Handle method
virtual void handle();
```

```
// check if remote event
virtual bool isRemote() const;

// get target lp
virtual uint32_t targetLp() const;

virtual DepartedHandler* clone() const
{ return new DepartedHandler( *this ); }

};

class ArrivedHandler : public NasHandler
{
public:
virtual const char* typeId() const { return "ArrivedHandler"; }

// ctor
ArrivedHandler() : NasHandler() {}
ArrivedHandler( Flight flight ) : NasHandler(flight) {}
ArrivedHandler( const ArrivedHandler& copy )
: NasHandler(copy)
{}

// dtor
virtual ~ArrivedHandler() {}

virtual void handle();

virtual bool isRemote() const;

virtual uint32_t targetLp() const;

virtual ArrivedHandler* clone() const
```

```
{ return new ArrivedHandler( *this ); }

};

class LandedHandler : public NasHandler
{
public:
virtual const char* typeId() const { return "LandedHandler"; }

// ctor
LandedHandler() : NasHandler() {}
LandedHandler( Flight flight ) : NasHandler(flight) {}
LandedHandler( const LandedHandler& copy )
: NasHandler(copy)
{}

// dtor
virtual ~LandedHandler() {}

virtual void handle();

virtual bool isRemote() const;

virtual uint32_t targetLp() const;

virtual LandedHandler* clone() const
{ return new LandedHandler( *this ); }

};

class DivertedHandler : public NasHandler
{
public:
```

```
virtual const char* typeId() const { return "DivertedHandler"; }

// ctor
DivertedHandler() : NasHandler() {}
DivertedHandler( Flight flight ) : NasHandler(flight) {}
DivertedHandler( const DivertedHandler& copy )
: NasHandler(copy)
{}

// dtor
virtual ~DivertedHandler() {}

virtual void handle();

virtual bool isRemote() const;

virtual uint32_t targetLp() const;

virtual DivertedHandler* clone() const
{ return new DivertedHandler( *this ); }

};

class TaxiinHandler : public NasHandler
{
public:
virtual const char* typeId() const { return "TaxiinHandler"; }

// ctor
TaxiinHandler() : NasHandler() {}
TaxiinHandler( Flight flight ) : NasHandler(flight) {}
TaxiinHandler( const TaxiinHandler& copy )
: NasHandler(copy)
```

```
{}

// dtor
virtual ~TaxiinHandler() {}

virtual void handle();

virtual bool isRemote() const;

virtual uint32_t targetLp() const;

virtual TaxiinHandler* clone() const
{ return new TaxiinHandler( *this ); }

};

} // namespace nassimulator

#endif
```

REFERENCES

[1] F. A. Forecast. (2016). Forecast 2016–2036, (visited on 01/18/2017).

[2] M. Ball, C. Barnhart, M. Dresner, M. Hansen, K. Neels, A. Odoni, E. Peterson, L. Sherry, A. A. Trani, and B. Zou, “Total delay impact study: A comprehensive assessment of the costs and impacts of flight delay in the united states”, 2010.

[3] M. D. Moore and K. H. Goodrich, “High speed mobility through on-demand aviation”, in *2013 aviation technology, integration, and operations conference*, 2013, p. 4373.

[4] M. Jones, P. Perfect, M. Jump, and M. White, “Investigation of novel concepts for control of a personal air vehicle”, in *American helicopter society 70th annual forum, montréal, québec, canada*, 2014.

[5] I. Chakraborty, B. G. Lozano, T. Nam, and D. N. Mavris, “A preliminary study of high lift system design and actuation for a personal air vehicle concept”, in *14th aiaa aviation technology, integration, and operations conference*, 2014, p. 2855.

[6] J. Page, J. Olsen, and A. Isikveren, “Design of a light, four-seat, zero-emissions aircraft”, in *Aiac16: 16th australian international aerospace congress*, Engineers Australia, 2015, p. 396.

[7] S. Athènes, P. Averty, S. Puechmorel, D. Delahaye, and C. Collet, “Atc complexity and controller workload: Trying to bridge the gap”, in *Proceedings of the international conference on hci in aeronautics*, AAAI Palo Alto, CA, 2002, pp. 56–60.

[8] T. W. Vossen, R. Hoffman, and A. Mukherjee, “Air traffic flow management”, in *Quantitative problem solving methods in the airline industry*, Springer, 2012, pp. 385–453.

[9] M. Ball, C. Barnhart, G. Nemhauser, and A. Odoni, “Air transportation: Irregular operations and control”, *Handbooks in operations research and management science*, vol. 14, pp. 1–67, 2007.

[10] S. J. Rassenti, V. L. Smith, and R. L. Bulfin, “A combinatorial auction mechanism for airport time slot allocation”, *The bell journal of economics*, pp. 402–417, 1982.

[11] M Ball and K Hoffman, “Nextor congestion management project: Interim report”, Technical report, NEXTOR, The National Center of Excellence for Aviation Operations Research, Tech. Rep., 2005.

[12] M. O. Ball, L. M. Ausubel, F. Berardino, P. Cramton, G. Donohue, M. Hansen, and K. Hoffman, “Market-based alternatives for managing congestion at new yorks laguardia airport”, AirNeth Annual Conference, The Hague, 2007.

[13] J. I. Daniel, “Congestion pricing and capacity of large hub airports: A bottleneck model with stochastic queues”, *Econometrica: Journal of the econometric society*, pp. 327–370, 1995.

[14] E. Pels and E. T. Verhoef, “The economics of airport congestion pricing”, *Journal of urban economics*, vol. 55, no. 2, pp. 257–277, 2004.

[15] J. L. Schank, “Solving airside airport congestion: Why peak runway pricing is not working”, *Journal of air transport management*, vol. 11, no. 6, pp. 417–425, 2005.

[16] R. Horonjeff, F. X. McKelvey, *et al.*, *Planning and design of airports*. McGraw-Hill New York, 1962.

[17] R. Hoffman and M. O. Ball, “A comparison of formulations for the single-airport ground-holding problem with banking constraints”, *Operations research*, vol. 48, no. 4, pp. 578–590, 2000.

[18] O. Richetta and A. R. Odoni, “Dynamic solution to the ground-holding problem in air traffic control”, *Transportation research part a: Policy and practice*, vol. 28, no. 3, pp. 167–185, 1994.

[19] A. Mukherjee, “Dynamic stochastic optimization models for air traffic flow management”, *Institute of transportation studies*, 2004.

[20] FAA. (1993). Simmod, (visited on 07/01/2017).

[21] Jeppesen. (1993). Taam, (visited on 07/01/2017).

[22] D. Long, D. Lee, J. Johnson, E. Gaier, and P. Kostiuk, “Modeling air traffic management technologies with a queuing network model of the national airspace system”, 1999.

[23] S ZAIDMAN, “National airspace system performance analysis capability naspac”, in *Agifors proceedings*, 1988.

[24] L. A. Wojcik, “Airspace and airport system simulation with dpat”, 2000.

[25] C. Roof, T. Thrasher, C. Hall, E. Dinges, R. Bea, A. Hansen, S. Balasubramaniam, A. Nguyen, B. Kim, P. Hollingsworth, *et al.*, “Aviation environmental design tool (aedt): System architecture”, John A. Volpe National Transportation System Center, Tech. Rep., 2007.

[26] NASA. (2003). Aces, (visited on 07/01/2017).

[27] L. Meyn, T. Romer, K. Roth, L. Bjarke, and S. Hinton, "Preliminary assessment of future operational concepts using the airspace concept evaluation system", in *4th aviation technology, integration and operations forum, chicago, il*, 2004.

[28] NASA. (2001). Facet, (visited on 07/01/2017).

[29] K Bilimoria, B. Sridhar, G. B. Chatterji, K Sheth, and S. Grabbe, "Facet: Future atm concepts evaluation tool", *Air traffic control quarterly*, vol. 9, no. 1, pp. 1–20, 2001.

[30] K. Palopo, G. B. Chatterji, M. D. Guminsky, and P. C. Glaab, "Shadow mode assessment using realistic technologies for the national airspace system (smart nas) test bed development", in *Aiaa aviation forum*, 2015.

[31] A. Kazda and R. E. Caves, *Airport design and operation*. Emerald Group Publishing Limited, 2010.

[32] A. Blumstein, "The landing capacity of a runway", *Operations research*, vol. 7, no. 6, pp. 752–763, 1959.

[33] W. J. Swedish, "Upgraded faa airfield capacity model. volume ii. technical description of revisions", Tech. Rep., 1981.

[34] B. O. Koopman, "Air-terminal queues under time-dependent conditions", *Operations research*, vol. 20, no. 6, pp. 1089–1114, 1972.

[35] P. A. Kivestu, "Alternative methods of investigating the time dependent m/g/k queue", PhD thesis, Massachusetts Institute of Technology, 1976.

[36] K. M. Malone, "Dynamic queueing systems: Behavior and approximations for individual queues and for networks", PhD thesis, Massachusetts Institute of Technology, 1995.

[37] T. Le, "Total airportsim: A new generation airport simulation model", *Transportation research e-circular*, 2002.

[38] R. M. Fujimoto, *Parallel and distributed simulation systems*. Wiley New York, 2000, vol. 300.

[39] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs", *Software engineering, ieee transactions on*, no. 5, pp. 440–452, 1979.

[40] B. Thomas, S. S. Rizvi, and K. M. Elleithy, “Reducing null messages using grouping and status retrieval for a conservative discrete-event simulation system”, in *Proceedings of the 2009 spring simulation multiconference*, Society for Computer Simulation International, 2009, p. 120.

[41] B. Wang, Y. Zhai, Z. Wang, H. Zhang, and D. Qing, “Enhanced null message algorithm for hybrid parallel simulation systems with large disparity in time step”, in *Distributed simulation and real time applications (ds-rt), 2016 ieee/acm 20th international symposium on*, IEEE, 2016, pp. 61–68.

[42] D. M. Nicol, C. C. Michael, and P. Inouye, “Efficient aggregation of multiple lps in distributed memory parallel simulations”, in *Proceedings of the 21st conference on winter simulation*, ACM, 1989, pp. 680–685.

[43] D. M. Nicol, “The cost of conservative synchronization in parallel discrete event simulations”, *Journal of the acm (jacm)*, vol. 40, no. 2, pp. 304–333, 1993.

[44] E. W. Lynch and G. F. Riley, “Hardware supported time synchronization in multicore architectures”, in *Principles of advanced and distributed simulation, 2009. pads’09. acm/ieee/scs 23rd workshop on*, IEEE, 2009, pp. 88–94.

[45] J. Liu and R. Rong, “Hierarchical composite synchronization”, in *Proceedings of the 2012 acm/ieee/scs 26th workshop on principles of advanced and distributed simulation*, IEEE Computer Society, 2012, pp. 3–12.

[46] D. R. Jefferson, “Virtual time”, *Acm transactions on programming languages and systems (toplas)*, vol. 7, no. 3, pp. 404–425, 1985.

[47] Y.-B. Lin, B. R. Preiss, W. M. Loucks, and E. D. Lazowska, “Selecting the checkpoint interval in time warp simulation”, in *Acm sigsim simulation digest*, ACM, vol. 23, 1993, pp. 3–10.

[48] R. Rönngren and R. Ayani, “Adaptive checkpointing in time warp”, in *Acm sigsim simulation digest*, ACM, vol. 24, 1994, pp. 110–117.

[49] Y.-B. Lin and B. R. Preiss, “Optimal memory management for time warp parallel simulation”, *Acm transactions on modeling and computer simulation (tomacs)*, vol. 1, no. 4, pp. 283–307, 1991.

[50] R. M. Fujimoto and K. S. Panesar, “Buffer management in shared-memory time warp systems”, *Acm sigsim simulation digest*, vol. 25, no. 1, pp. 149–156, 1995.

[51] J. Wang and C. Tropper, “Optimizing time warp simulation with reinforcement learning techniques”, in *Simulation conference, 2007 winter*, IEEE, 2007, pp. 577–584.

[52] R. M. Fujimoto, I. Nikolaidis, and C. A. Cooper, "Parallel simulation of statistical multiplexers", *Discrete event dynamic systems*, vol. 5, no. 2-3, pp. 115–140, 1995.

[53] A. G. Greenberg, B. D. Lubachevsky, and I. Mitrani, "Algorithms for unboundedly parallel simulations", *Acm transactions on computer systems (tocs)*, vol. 9, no. 3, pp. 201–221, 1991.

[54] J. J. Wang and M. Abrams, "Approximate time-parallel simulation of queueing systems with losses", in *Proceedings of the 24th conference on winter simulation*, ACM, 1992, pp. 700–708.

[55] T. Kiesling and S. Pohl, "Time-parallel simulation with approximative state matching", in *Proceedings of the eighteenth workshop on parallel and distributed simulation*, ACM, 2004, pp. 195–202.

[56] T. H. D. Thi, J.-M. Fourneau, and F. Quessette, "Time-parallel simulation for stochastic automata networks and stochastic process algebra", in *International conference on analytical and stochastic modeling techniques and applications*, Springer, 2014, pp. 140–154.

[57] L. Grasedyck, C. Löbbert, G. Wittum, A. Nägel, V. Schulz, M. Siebenborn, R. Krause, P. Benedusi, U. Küster, and B. Dick, "Space and time parallel multigrid for optimization and uncertainty quantification in pde simulations", in *Software for exascale computing-sppexa 2013-2015*, Springer, 2016, pp. 507–523.

[58] Y. Qu and X. Zhou, "Large-scale dynamic transportation network simulation: A space-time-event parallel computing approach", *Transportation research part c: Emerging technologies*, vol. 75, pp. 1–16, 2017.

[59] B. A. Pearlmutter, "Gradient calculations for dynamic recurrent neural networks: A survey", *Ieee transactions on neural networks*, vol. 6, no. 5, pp. 1212–1228, 1995.

[60] K.-i. Funahashi and Y. Nakamura, "Approximation of dynamical systems by continuous time recurrent neural networks", *Neural networks*, vol. 6, no. 6, pp. 801–806, 1993.

[61] Y. J. Kim, D. N. Mavris, and R. M. Fujimoto, "Time-parallel simulation of air traffic networks", in *Winter simulation conference (wsc), 2017*, IEEE, 2017.

[62] Y. J. Kim, S. Choi, S. Briceno, and D. Mavris, "A deep learning approach to flight delay prediction", in *Digital avionics systems conference (dasc), 2016 ieee/aiaa 35th*, IEEE, 2016, pp. 1–6.

[63] M. Molina, S. Carrasco, and J. Martin, "Agent-based modeling and simulation for the design of the future european air traffic management system: The experience

of cassiopeia", in *International conference on practical applications of agents and multi-agent systems*, Springer, 2014, pp. 22–33.

[64] O. J. Pinon, *A methodology for the valuation and selection of adaptable technology portfolios and its application to small and medium airports*. 2012.

[65] Y. J. Kim, O. J. Pinon-Fischer, and D. N. Mavris, "Parallel simulation of agent-based model for air traffic network", in *Aiaa modeling and simulation technologies conference*, 2015, p. 2799.

[66] F. Wieland, "Parallel simulation for aviation applications", in *Proceedings of the 30th conference on winter simulation*, IEEE Computer Society Press, 1998, pp. 1191–1198.

[67] S. Lee, A. Pritchett, and D. Goldsman, "Hybrid agent-based simulation for analyzing the national airspace system", in *Proceedings of the 33nd conference on winter simulation*, IEEE Computer Society, 2001, pp. 1029–1036.

[68] M. Hybinette and R. M. Fujimoto, "Cloning parallel simulations", *Acm transactions on modeling and computer simulation (tomacs)*, vol. 11, no. 4, pp. 378–407, 2001.

[69] P. Heidelberger and H. S. Stone, "Parallel trace-driven cache simulation by time partitioning", in *Proceedings of the 22nd conference on winter simulation*, IEEE Press, 1990, pp. 734–737.

[70] J. Post, J. Gulding, K. Noonan, D. Murphy, J. Bonn, and M. Graham, "The modernized national airspace system performance analysis capability (naspac)", *Weather*, vol. 4, p. 5, 2008.

[71] B. Bagdatli and D. Mavris, "Use of high-level architecture discrete event simulation in a system of systems design", in *Aerospace conference, 2012 ieee*, IEEE, 2012, pp. 1–13.

[72] C. Iwata and D. Mavris, "Object-oriented discrete event simulation modeling environment for aerospace vehicle maintenance and logistics process", *Procedia computer science*, vol. 16, pp. 187–196, 2013.

[73] P. K. Menon, G. D. Sweriduk, and K. D. Bilimoria, "New approach for modeling, analysis, and control of air traffic flow", *Journal of guidance, control, and dynamics*, vol. 27, no. 5, pp. 737–744, 2004.

[74] A. Borshchev and A. Filippov, "From system dynamics and discrete event to practical agent based modeling: Reasons, techniques, tools", in *Proceedings of the 22nd international conference of the system dynamics society*, 2004.

[75] A. A. Tako and S. Robinson, “The application of discrete event simulation and system dynamics in the logistics and supply chain context”, *Decision support systems*, vol. 52, no. 4, pp. 802–815, 2012.

[76] S. R. Conway, “An agent-based model for analyzing control policies and the dynamic service-time performance of a capacity-constrained air traffic management facility”, in *Icas 2006-25th congress of the international council of the aeronautical sciences hamburg, germany*, 2006, pp. 3–8.

[77] G. J. Couluris, C. Hunter, M. Blake, K. Roth, D. Sweet, P. Stassart, J. Phillips, and A. Huang, “National airspace system simulation capturing the interactions of air traffic management and flight trajectories”, in *Aiaa guidance, navigation, and control (gnc) conference*, 2003.

[78] M. O. Ball and G. Lulli, “Ground delay programs: Optimizing over the included flight set based on distance”, *Air traffic control quarterly*, vol. 12, no. 1, pp. 1–25, 2004.

[79] J.-T. Wong and S.-C. Tsai, “A survival model for flight delay propagation”, *Journal of air transport management*, vol. 23, pp. 5–11, 2012.

[80] B. of transports statistics. (2017). Research and innovative technology administration (rita)/transtats, (visited on 01/18/2017).

[81] N. Cetin, A. Burri, and K. Nagel, “A large-scale agent-based traffic microsimulation based on queue model”, in *In proceedings of swiss transport research conference (strc), monte verita, ch*, Citeseer, 2003.

[82] FAA. (Oct. 2009). Traffic flow management in the national airspace system.

[83] *New york times, http://www.nytimes.com/.*

[84] A. Sodani, “Knights landing (knl): 2nd generation intel® xeon phi processor”, in *Hot chips 27 symposium (hcs), 2015 ieee*, IEEE, 2015, pp. 1–24.

[85] B. Manley and L. Sherry, “Analysis of performance and equity in ground delay programs”, *Transportation research part c: Emerging technologies*, vol. 18, no. 6, pp. 910–920, 2010.

[86] J. Ferguson, A. Q. Kara, K. Hoffman, and L. Sherry, “Estimating domestic us airline cost of delay based on european model”, *Transportation research part c: Emerging technologies*, vol. 33, pp. 311–323, 2013.

[87] C. N. Glover and M. O. Ball, "Stochastic optimization models for ground delay program planning with equity–efficiency tradeoffs", *Transportation research part c: Emerging technologies*, vol. 33, pp. 196–202, 2013.

[88] Y. Tu, M. O. Ball, and W. S. Jank, "Estimating flight departure delay distributionsa statistical approach with long-term trend and short-term pattern", *Journal of the american statistical association*, vol. 103, no. 481, pp. 112–125, 2008.

[89] N. Xu, G. Donohue, K. B. Laskey, and C.-H. Chen, "Estimation of delay propagation in the national aviation system using bayesian networks", in *6th usa/europe air traffic management research and development seminar*, Citeseer, 2005.

[90] J. J. Rebollo and H. Balakrishnan, "Characterization and prediction of air traffic delays", *Transportation research part c: Emerging technologies*, vol. 44, pp. 231–241, 2014.

[91] S. Choi, Y. J. Kim, S. Briceno, and D. N. Mavris, "Prediction of weather-induced airline delays based on machine learning algorithms", in *Digital avionics systems conference (dasc), 2016 ieee/aiaa 35th*, IEEE, 2016.

[92] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald, and E. Muharemagic, "Deep learning applications and challenges in big data analytics", *Journal of big data*, vol. 2, no. 1, pp. 1–21, 2015.

[93] H. Kashyap, H. A. Ahmed, N. Hoque, S. Roy, and D. K. Bhattacharyya, "Big data analytics in bioinformatics: A machine learning perspective", *Arxiv preprint arxiv:1506.05101*, 2015.

[94] Y. Lv, Y. Duan, W. Kang, Z. Li, and F.-Y. Wang, "Traffic flow prediction with big data: A deep learning approach", *Intelligent transportation systems, ieee transactions on*, vol. 16, no. 2, pp. 865–873, 2015.

[95] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks", in *International conference on machine learning*, 2013, pp. 1310–1318.

[96] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks", in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, IEEE, 2013, pp. 6645–6649.

[97] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation", *Arxiv preprint arxiv:1406.1078*, 2014.

[98] R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio, "How to construct deep recurrent neural networks", *Arxiv preprint arxiv:1312.6026*, 2013.

[99] *National oceanic and atmospheric administration, http://www.noaa.gov/.*

[100] G. E. Dahl, T. N. Sainath, and G. E. Hinton, "Improving deep neural networks for lvcsr using rectified linear units and dropout", in *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, IEEE, 2013, pp. 8609–8613.

[101] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting", *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[102] L. Bottou, "Large-scale machine learning with stochastic gradient descent", in *Proceedings of compstat'2010*, Springer, 2010, pp. 177–186.

[103] X.-W. Chen and X. Lin, "Big data deep learning: Challenges and perspectives", *Access, ieee*, vol. 2, pp. 514–525, 2014.