CSE 6730 Project 2 Final Report
Parallel Discrete Event Simulation & Backstroke

Chayong Lee, Christopher Toth, Douglas Black
Georgia Institute of Technology
{chayong82, ctoth6, dougblac}@gatech.edu

1 Project Description

Implementing parallel discrete event simulation (PDES) [4] has never been an easy task. There are many well known problems that come from the nature of parallelism such as managing concurrent threads execution or Heisenbugs, however, the major problem of implementing PDES is synchronization. In PDES, events are not synchronized by a global clock but rather scheduled irregularly. This introduces the necessity of a rollback mechanism in PDES.

Backstroke [10] is a new open source framework for the automatic generation of reverse code for functions written in C++. Because writing an efficient rollback by hand is difficult, the primary purpose of Backstroke is to enable reverse computation for fast, efficient rollback in optimistic parallel discrete event simulation. In this project, wed like to benchmark the performance of the reverse code generation using Backstroke and evaluate how it can affect on the simulation performance in terms of time and space-wise compared to the hand-written code.

This project consists of two parts. First part is to create a parallel discrete event simulation using Time Warp as the synchronization mechanism on ROSS [3]. Second part is to benchmark the simulation performance between the hand-written code and Backstroke-generated code.

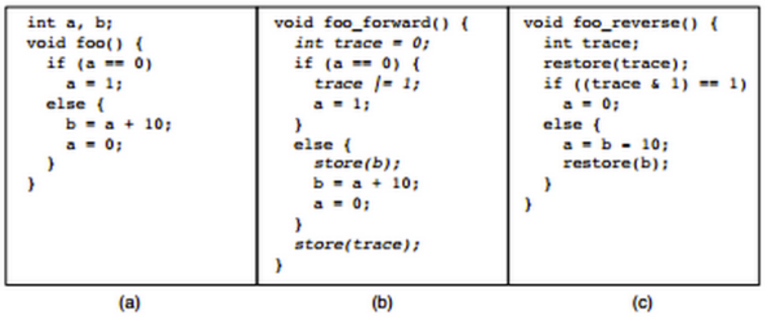
2 Background

2.1 ROSS

ROSS [3] is an open source, parallel, discrete event simulation tool for modeling large scale systems. There are multiple synchronization techniques that can be used when developing a PDES; ROSS uses the Time Warp protocol [6]. ROSS has been heavily optimized and is very efficient when modeling large scale systems. ROSS handles all of the parallelism and communication between different Logic Processes (LP's) so all we had to focus on was writing the event scheduling and handling code. The disadvantage of using ROSS is that it maintains our event list internally, and we are not allowed access to it during execution.

2.2 Backstroke

Backstroke [10] is a reverse-code generation tool for C++. This makes it especially useful for developing the rollback code for our aircraft simulation.



```
int a, b;
void foo() {
  if (a == 0)
    a = 1;
  else {
    b = a + 10;
    a = 0;
  }
}
```

(a)

```
void foo_forward() {
  int trace = 0;
  if (a == 0) {
    trace |= 1;
    a = 1;
  }
  else {
    store(b);
    b = a + 10;
    a = 0;
  }
  store(trace);
}
```

(b)

```
void foo_reverse() {
  int trace;
  restore(trace);
  if ((trace & 1) == 1)
    a = 0;
  else {
    a = b - 10;
    restore(b);
  }
}
```

(c)

Figure 1: Backstroke Code Segment

As shown in figure 1, in box (a), we have original method code. Using Backstroke, we achieve boxes (b), and (c), which constitute and forward and reverse version of this method. In the forward version, we store crucial state variables and in the reverse, we restore these same state variables. Backstroke is very useful for PDES, but it has some drawbacks. Backstroke does not currently support processing of loops or arrays, so we had to avoid these two constructs throughout our code so that it would be Backstroke compatible.

3 Related Work

Correct, optimized rollback is a natural field of interest in parallel discrete event simulations. It is not always as easy as simply decrementing a previously incremented integer. In many cases, some variables have simply changed their value and to implement a correct rollback, previous values must be stored. The difficulty of such a problem becomes immediately obvious. As such, it has been a topic of much interest among those attempting to implement PDES.

Previous attempts have been made to use method-reversal libraries to generate rollback code. In fact, attempts have even used Backstroke specifically. In order to not simply redo what others have already accomplished, we will use the conclusions from previous attempts to bolster our own methodology rather than simply substitute it in. Hou, Fujimoto, and Vuduc explored setting up a SSA graph for values of the simulation and a specialized algorithm to search the graph for appropriate rollback values. They used Backstroke in their implementation and compared its efficiency to incremental state saving, copy state saving, and a reverse C compiler [2] [5]. They concluded that Backstroke was indeed the most efficient. This is promising for our own implementation, which will be done in Backstroke.

Other approaches include examining the assignment for various variables. If possible, a logical previous/next value for the variable is decided upon but if this fails, the value is simply stored and loaded again in the event of a rollback. This approach is simpler than using reverse code- generation but it is slower because it often fails to decide logically on appropriate values and ends up storing many variables. Our implementation will be better at deciding logically on appropriate forward/backward values for variables thanks to Backstroke.

Wieland analyzes the Detailed Policy Assessment Tool (DPAT) [9] [8], which is a practical real world aviation simulation uses optimistic simulation technology. He points out several facts how DPAT avoids some of the problems associated with optimistic simulation.

As far as airport simulations, it is an incredibly popular simulation problem. There is a myriad of prior work on the subject and we will have no shortage of previous implementations to examine. PDES implementations are somewhat rarer, due to their inherently increased complexity, but such examples to exist. To cite such examples would be of little benefit, because they are quite bountiful.

4 Simulation Description

A program will be developed that will simulate air traffic in the United States. In order to do this, rules for flight must be developed. This includes airport runway operation rules, and rules for inter-region travel.

Maximum Throughput Capacity (MTC) is the fundamental measure of the capacity of the runway. It is measured in movements (arrivals and departures) per hour. The most significant factor affecting capacity is spacing between successive aircraft in both takeoff and landing streams. Of course, there are many factors that play a role in determining capacity. Runway system capacity depends on:

- Runway configuration
- Environment on which aircraft operate (wind level, cloud ceiling)
- Availability of NAVAIDS & ATC facilities and operating requirements/standards
- Characteristics of demand
- Other air traffic control rules

There are many flight rules that can be considered in the development of a simulation. Among these include minimum following distances for different aircraft sizes, minimum following distances depending on weather conditions and navigation equipment, amount of time required to take-off and land, and minimum amounts of time between successive take-offs and landings [11]. However, for the simplicity of this simulation, it will be assumed that all aircraft are the same size, have the same up-to-date navigation equipment and will all operate alike. Because of the complexity of implementing many of these flight rules, a single minimum assumed time-space gap will be considered for all departing and arriving aircraft. Also, it is assumed that there will be sufficient airspace such that collisions between aircraft will be avoided while en route.

The location of airports within a region and individual airport characteristics must be determined. Only airports designated as primary commercial airports will be considered for this simulation. This includes airports with more than 10000 yearly enplanements. Furthermore, there are four airport sizes, as designated by the percentage of all US enplanements as shown in table 1 [12].

Many airports have runways that cross each other for variable wind conditions - these runways are never used at the same time. For the simplicity of our simulation, and due to the difficulty of finding airport runway orientations, it is assumed that

Table 1: Hub type Enplanement Definitions

Hub Type	Enplanements
Large	Greater than 1%
Medium	0.25 - 1%
small	0.05 - 0.25 %
Non-Hub	Less than 0.05 %

the wind direction does not play a role in the number of useable runways, and that planes will be able to takeoff/land in the direction of the orientation of the maximum number of useable runways). A survey of different airport types reveals the distribution of the maximum number of runways that can be used concurrently in the following table:

Table 2: Airport Number of Runways

Hub Type	Runways
Large	2 - 6
Medium	1 - 3
small	1 - 2
Non-Hub	1

Knowing the number of enplanements at each airport will be particularly useful in determining the number of flights expected to take-off and land from a particular airport. Also, knowing the number of runways each airport has will be important in determining capacity of that airport. There is expected to be a correlation between the number of runways and the number of enplanements (demand and capacity).

Air traffic control is divided into 20 regions in the continental United States. Given a list of 328 primary commercial service airports from the Federal Aviation Administration, each airport was placed into its appropriate region.

Once the number or airports in each region was determined, a region size (based on the distribution of airport types within that region) could be assigned. More weight was given to regions that had more large hubs, and less emphasis was placed on non-hub airports. Figure 2 shows the distribution of airport types for all 20 regions.

For the simulation, the size of a region is equivalent to the number of aircraft that can be in that regions airspace at a given time. If airspace capacity has been

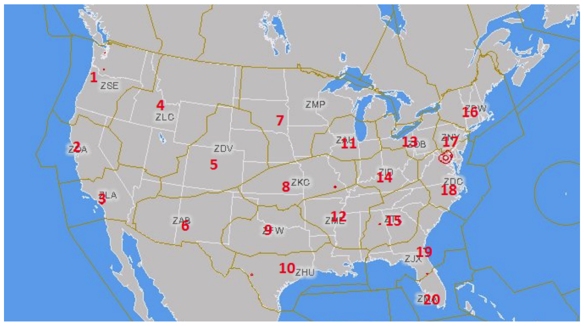


Figure 2: Location of Airport Region

Table 3: Regional Airport Distribution

Region	L	M	S	N	Total	Region Size	Region	L	M	S	N	Total	Region Size
1	1	1	1	16	19	M	11	2	1	3	7	13	M
2	1	4	1	9	15	M	12	0	2	3	7	12	S
3	3	3	3	10	19	L	13	1	3	4	9	17	M
4	1	0	2	17	20	M	14	1	2	4	4	11	S
5	1	0	1	15	17	S	15	2	0	5	8	15	M
6	0	1	2	2	5	S	16	3	3	6	13	25	L
7	1	1	4	28	34	L	17	2	0	2	4	8	S
8	1	3	3	11	18	L	18	3	2	3	8	16	L
9	1	1	3	9	14	S	19	1	1	7	10	19	M
10	1	4	3	13	21	L	20	3	2	1	4	10	M

reached, no aircraft may enter that regions airspace from another region, or take off from an airport in that region until another plane has exited the region or landed at an airport in that region. If an aircraft attempts to enter a region (a request), but cannot due to capacity constraints, this is considered a request rejection. The aircraft will have to make another request at the conclusion of the next event. Thus, queues form at airports (for planes waiting to take off) and at boundaries between regions. The effectiveness of air traffic operations can be estimated by analyzing the total number of events and the number of rejected requests (due to the fact that a runway is in use or an air traffic control region is at capacity).

Lastly, it will be important to describe inter-regional travel. To do this, spatial relationships must be made between neighboring regions. Approximate distance between central locations in neighboring regions are shown in figure 3.

The total distance an aircraft travels from one region to another can be determined by finding a path from the figure above. Because it is of interest for aircraft to take the shortest possible route, Dijkstras algorithm is applied to determine a planes route from one region to another.

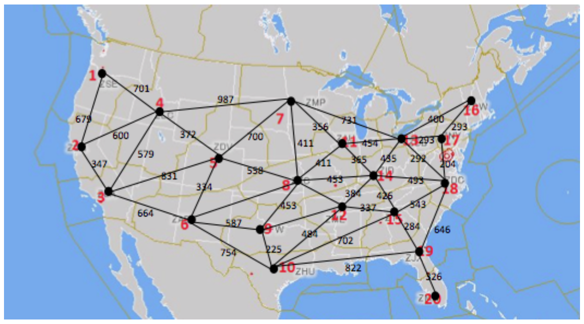


Figure 3: Approximate Distance Between Central Locations

5 PDES Design

In order to create a realistic model, we decided to use two logic processors (LP). One LP represents an airport and the other LP represents a region controller. An airport LP contains 9 state variables and a region controller LP contains 6 state variables. We used 13 different event types and run Dijkstra shortest path algorithm to send an airplane along path to the destination airport. Two limitation existed. We can not use an array. We can not use a loop. This is due to Backstroke doesn't handle arrays and loops in generating a reverse code. Figure 4 shows the simulation flow as seen from the perspective of a single flight. The black circle represents an airport and the red circle represents a region controller. The model contains a request/response structure between two different types of LP. This structure requires to include the sender's information when it schedules an events, so that the receiver LP can response back to the sender LP. We used a message structure in ROSS, which is a wrapper of an event which allows to store data for the event. The straight arrow represents a time for scheduling an event and the dotted arrow indicates a delay.

6 Software Architecture

The simulation is built on top of ROSS. As shown in figure 5, the application consists of four major modules, Aircraft, Weather/Disaster, Airport and Region Controller. Aircrafts are sent between the airport and the region controller. Airport and region controller access to the Weather/Disaster module upon a request. Airport contains a local traffic manager, which checks the number of runways in use. The local traffic manager computes a taxi in/out time and calculate a delay based on the airport size. Region controller contains an air traffic manager, which tracks the number

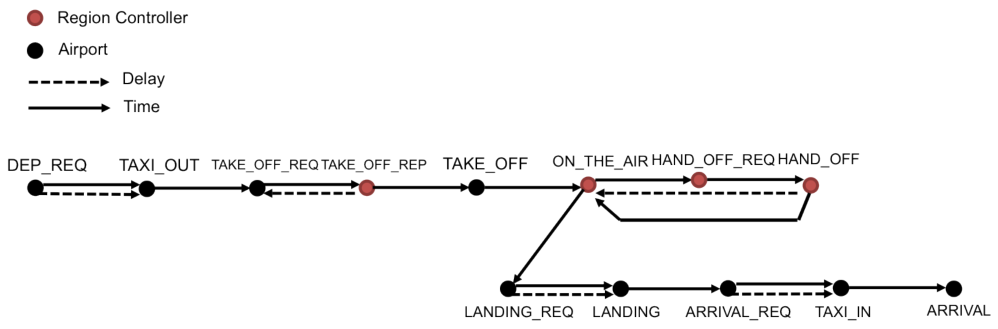


Figure 4: Air Traffic Simulation Flow

of airplanes in the region. The controller handles a hand-off request and computes a hand-off time/delay based on the weather condition and the current capacity of airplanes in the region. The controller follows the hand-off protocol to transit an airplane from one region to the next region. Due to the limitation of using arrays in Backstroke, we used Backstroke random number generator which performs a state-saving for a random seed using a stack instead of ROSS random number generator.

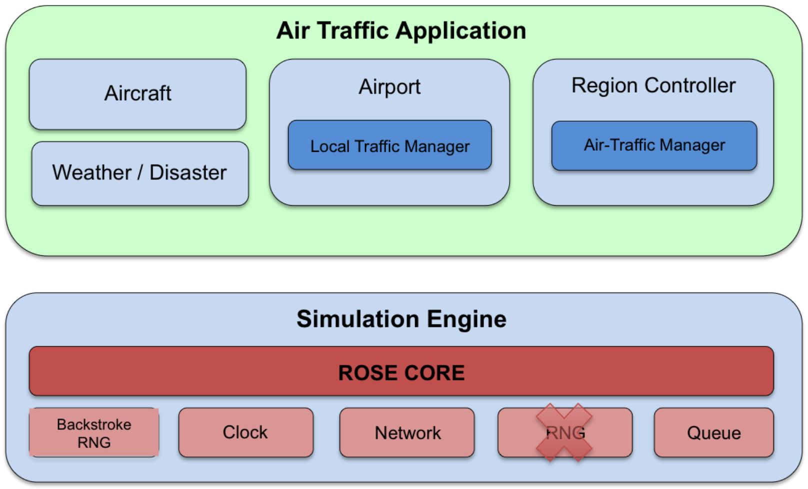


Figure 5: Software Architecture of Air Traffic Simulation

7 Simulation Results

The computer executing the simulation software has the following specifications:

Table 4: Sequential Execution Simulation

Net Events Processed	2354171
Total Transit Accepted	14012
Total Transit Rejected	14288
Total Departure Requests Accepted	9840
Total Departure Requests Rejected	1377210
Total Arrival Requests Accepted	9840
Total Arrival Requests Rejected	177018

- 2 x Intel Xeon X5550 quad-core Nehalem processors @ 2.66GHz (Hyperthreading enabled)
- 24GB Memory
- 2 x Nvidia Tesla T10 boards (half of an S1070)
- 250GB 7200RPM SATA hard drive

As described in the previous section the simulation uses 328 airports in 20 regions. A first run of the simulation uses 9840 airplanes (30 per airport). Large airports were assigned to have 6 runways, medium airports were given 3 runways, small airports 2 runways, and non- bub airports1 runway. Region capacity was also set constant at a maximum of 100 planes per region. Some statistics from the simulation are shown in table 4.

Additional experiments were executed increasing the airspace capacity of each region and the number of runways at each airports. Three additional experiments scale up the capacity and number of runways by 2, 4, and 8. Figure 6, 7, 8 show the number of rejected departure requests, execution time, and total events processed.

As expected, by increasing airspace capacity and airport capacity (increasing the number of runways) the number of rejected departure requests is considerably reduced. Also, execution time and total events processed also decrease. This can be attributed to the fact that increasing airspace and airport capacity causes fewer denials for departures, landing, and inter-region travel.

8 Parallel Benchmark

We used a traditional airport simulation [4] to benchmark the parallel performance between hand-generated code and Backstroke generated code. We had a problem

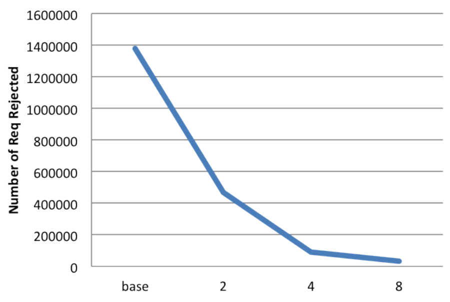


Figure 6: Number of Rejected Departure Requests

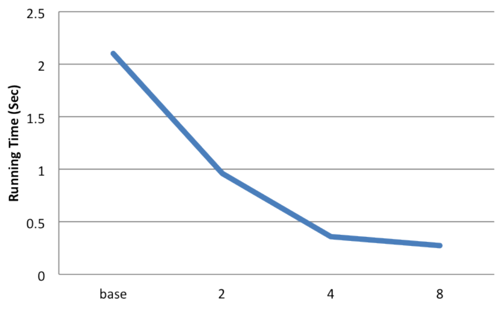


Figure 7: Sequential Execution Time

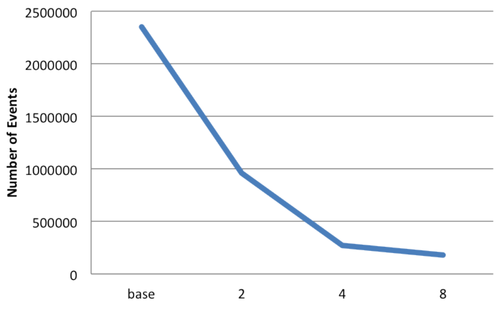


Figure 8: Number of Net Events

of using our air-traffic model on ROSS platform. In order to communicate between events, ROSS uses a message, which is a wrapper for the model data. In the air-traffic simulation, events are not independent. Each LP needs to know where the events come from so that it can send back the result to the sender LP. During this process, we observed that messages started getting corrupted when we run the simulation for long time until it executes millions events. We left fixing this issue as a future work and we used a simple traditional airport model to benchmark parallel run between hand-generated reverse code and Backstroke generated reverse code.

We run the experiments with different number of LPs so that we can evaluate how different number of LPs affects on the simulation performance. Each LP initializes 100 airplanes at the beginning of the simulation run. The airport simulation creates two-dimensional grid based on the total number of LPs. Each cell represents an airport and each airport can send airplanes to its neighbor. The airport simulation has three different events types, ARRIVAL, DEPARTURE, and LAND. The simulation starts with scheduling DEPARTURE events corresponding to the number of airplanes. At the end of DEPARTURE event, it schedules ARRIVAL events.

Table 5: Parallel Benchmark Data with 256 LPs

Type	Node	LP	End Time	Run Time	Events Processed	Rolled Back	Net Events Processed	Efficiency	Time Overhead
Hand-Coded	2	256	1.00E+06	9.76096	18578689.3	1067610.3	17511079	94.2535767	Baseline
Hand-Coded	4	256	1.00E+06	5.78666	19942270.4	2431191.4	17511079	87.80885686	Baseline
Hand-Coded	8	256	1.00E+06	3.68173	22033591.8	4522512.8	17511079	79.47459947	Baseline
Hand-Coded	16	256	1.00E+06	6.47283	32160469.5	14649390.5	17511079	54.45352656	Baseline
Backstroke	2	256	1.00E+06	11.17306	18576048.8	1064969.8	17511079	94.26697371	1.1447
Backstroke	4	256	1.00E+06	6.62435	19930653	2419574	17511079	87.86003794	1.1448
Backstroke	8	256	1.00E+06	4.27653	21985371.6	4474292.6	17511079	79.64888893	1.1616
Backstroke	16	256	1.00E+06	7.82869	31670419.2	14159340.2	17511079	78.45787558	1.2095

Table 6: Parallel Benchmark Data with 1024 LPs

Type	Node	LP	End Time	Run Time	Events Processed	Rolled Back	Net Events Processed	Efficiency	Time Overhead
Hand-Coded	2	1024	1.00E+06	43.36618	73577927.7	3628929.7	69948998	95.06790989	Baseline
Hand-Coded	4	1024	1.00E+06	24.4147	77866515.7	7917517.7	69948998	89.83193539	Baseline
Hand-Coded	8	1024	1.00E+06	14.05327	83075096.9	13126098.9	69948998	84.19971964	Baseline
Hand-Coded	16	1024	1.00E+06	12.89474	89245643.9	19296645.9	69948998	78.37808181	Baseline
Backstroke	2	1024	1.00E+06	49.52485	73579007	3630009	69948998	95.06651537	1.1420
Backstroke	4	1024	1.00E+06	27.79106	77867116.6	7918118.6	69948998	89.83124213	1.1383
Backstroke	8	1024	1.00E+06	16.3482	83061024.3	13112026.3	69948998	84.21398893	1.1633
Backstroke	16	1024	1.00E+06	14.80047	89154854.9	19205856.9	69948998	78.45787558	1.1478

ARRIVAL events end up scheduling LAND events. LAND events re-schedule DEPARTURE events so that the simulation can end on GVT.

Table 5 shows the parallel benchmark results with 256 LPs. Table 6 summarizes the parallel benchmark results with 1024 LPs. The experiment results are calculated from an average of 10 runs. The total number of events rolled back is increased as we increase the number of cores. Increasing cores causes more outer-processor communication, as a result, it increases the number of events rolled back. Consequently decreases the efficiency. However, increasing cores in general gives more speed up. Backstroke generated reverse code runs about 13% to 16% slower than the hand-written reverse code. We found out a bug in building SSA in Backstroke, which does not handle constructive assignments correctly. In theory, Backstroke should use a reverse computing mechanism to reverse those constructive assignments but Backstroke performs a state-saving mechanism. This makes the simulation slows down due to the memory access.

Compare to the efficiency of the simulation run with 256 LPs, the efficiency of the simulation run with 1024 LPs is improved. Increasing LPs produces more-inter communication and it reduces the chance for events to be scheduled to other processors, resulting in a higher efficiency. Both parallel runs with 256 LPs and 1024 LPs are valid since the total number of committed events is identical.

As we can see from the figure 9, the simulation execution time decreased as we increased the number of cores. In the case of benchmark with 256 LPs, we experienced a more overhead on using 16 cores, which results in decreasing the simulation

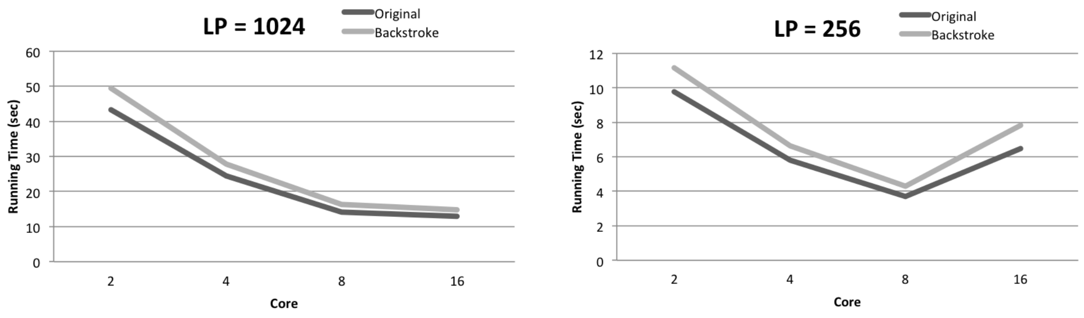


Figure 9: Simulation Performance Using Different Numbers of Processors

performance.

9 Additional Application

Airports often use simulation of arrivals and departures to evaluate operations. Depending on the runway layout of the airport, time of day, and the state of airplanes arriving and departing, different priorities may be given to landing aircraft and planes waiting to take off. The simulation could be used to predict changes in operation under adverse weather conditions and varying types of aircraft. One last application may simulate the grounding of all flights in the event of a national disaster such as 9/11. It would be of great interest to evaluate different strategies to find the quickest way to land all airborne aircraft at the same time. Of course, the accuracy of any air traffic simulation is as good as it the rules it uses. Creating an accurate simulation that provides the most meaningful results relies on the use of accurate flight rules, and providing detailed state variables that must be considered when flying.

10 Task Assignments

10.1 Application Development

- Architecture & PDES Model Design - Chayong, Chris
- Runway & traffic management strategies Doug
- Decision maker plan - Chris
- Airport Application Implementation - Chayong, Chris, Doug

- ROSS modification - Chayong

10.2 Performance Benchmark

- Simulation performance benchmark - Chayong
- Execution time benchmark - Chayong
- Code analysis = Chayong

References

[1] Bain, W.L *Air Traffic Simulation: An Object Oriented, Discrete Event Simulation on the Intel iPSC/2 Parallel System*, Distributed Memory Computing Conference, 1990., Proceedings of the Fifth , vol., no., pp.95-100, 8-12 Apr 1990

[2] Carothers, C. D., K.S. Perumalla, and R.M. Fujimoto *Efficient optimistic parallel simulations using reverse computation*, ACM Transactions on Modeling and Computer Simulation (TOMACS), vol. 9, 1999, p. 224253

[3] Carothers, C.D.; Bauer, D.; Pearce, S *ROSS: a high-performance, low memory, modular time warp system*, Parallel and Distributed Simulation, 2000. PADS 2000. Proceedings. Fourteenth Workshop on , vol., no., pp.53-60, 2000

[4] Fujimoto, R. M. *Parallel and Distributed Simulation Systems*, Wiley, 2000

[5] Fujimoto, Richard, Richard Vuduc, Cong Hou, David Jefferson, Daniel Quinlan, and George Vulov *A New Method For Program Inversion*, College of Computing, Web. 29 Mar. 2012

[6] Jefferson, D. R. *Virtual Time*, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 7, 3, pp. 404-425, July 1985

[7] Sherry, Lance *Runway Capacity*, George Mason University - Center for Air Transportation Systems Research, 2009

[8] Wieland, F *Practical parallel simulation applied to aviation modeling*, Parallel and Distributed Simulation, 2001. Proceedings. 15th Workship on , vol., no., pp.109-116, 2001

[9] Wieland, F *Parallel simulation for aviation applications*, Simulation Conference Proceedings, 1998. Winter , vol.2, no., pp.1191-1198 vol.2, 13-16 Dec 1998

[10] Vulov, G.; Cong Hou; Vuduc, R.; Fujimoto, R.; Quinlan, D.; Jefferson, D *The Backstroke framework for source level reverse computation applied to parallel discrete event simulation*, Simulation Conference (WSC), Proceedings of the 2011 Winter , vol., no., pp.2960-2974, 11-14 Dec. 2011

[11] *Aeronautical Information Manual: Official Guide to Basic Flight Information and ATC Procedures*, Federal Aviation Administration, February 2012

[12] *Enplanements at Primary Airports (Rank Order) CY10, Air Carrier Activity Information System*, Federal Aviation Administration, October 2011