

AD-A258 851



AFIT/GCS/ENG/GCS92D-19



DEVELOPMENT OF A PROTOCOL USAGE GUIDELINE
FOR CONSERVATIVE PARALLEL SIMULATIONS

THESIS

Prescott John Van Horn
Captain, USAF

AFIT/GCS/ENG/GCS92D-19



Approved for public release; distribution unlimited

93 1 04 030

AFIT/GCS/ENG/GCS92D-19

DEVELOPMENT OF A PROTOCOL USAGE GUIDELINE
FOR CONSERVATIVE PARALLEL SIMULATIONS

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science (Computer Systems)

Prescott John Van Horn, B.S.Ch.E.
Captain, USAF

December, 1992

Approved for public release; distribution unlimited

Acknowledgements

I would like to thank my thesis advisor, Dr. Thomas Hartrum, and committee members, Maj Eric Christensen, USA and Maj Tom Wailes, USAF, for their assistance and guidance through the course of this most memorable (sic) endeavor. Additionally, I want to thank Mr. Rick Norris for keeping the "cubes" up and running, and for puttin' up with us PSWGers. A special thanks goes out to MSgt Dan "Boner" Archibald for introducing and getting me out to *Tanks* every so often just to get loose and relaxed for an obligatory "Rock is an ...", and to Vicky Archibald for preparing all those meals. In addition, acknowledgements go out to Col Stephen F. Austin, Martin Birch, Friz Freleng and Chuck Jones, fellow PSWGers Andy "That-A-Boy Emmitt!" Breeden, and Dave "Think Minimal Effort" Daniel, Whitehall Laboratories, the Miller Brewing Co., and most of all to the United States Air Force for their incomparable and world renowned personnel system.

Prescott John Van Horn

DTIC CLASSIFICATION 1

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Table of Contents

	Page
Acknowledgements	ii
Table of Contents	iii
List of Figures	viii
List of Tables	x
Abstract	xii
 I. Introduction	 1
1.1 Background	1
1.1.1 Conservative Methods.	3
1.1.2 Optimistic Methods.	3
1.1.3 SPECTRUM Simulation Testbed.	4
1.2 Problem Statement	5
1.3 Research Objective	5
1.4 Research Questions	5
1.5 Scope	5
1.6 Thesis Outline	5
 II. Current Issues	 7
2.1 Introduction	7
2.2 Conservative Simulation Approaches	7
2.2.1 The Basic Chandy-Misra Algorithm.	7
2.2.2 The Chandy-Misra Algorithm with NULL Messages. . .	10
2.2.3 The Carrier NULL Approach.	12

	Page
2.2.4 A Conditional Chandy-Misra Approach.	13 °
2.2.5 Chandy-Misra with Channel Clocks.	14
2.2.6 The Chandy-Misra Algorithm Variants.	14
2.3 Optimistic Simulation Approaches	16
2.3.1 The Basic Time Warp Algorithm.	16
2.3.2 Time Warp Variations.	17
2.4 Hybrid Simulation Approaches	18
2.4.1 Shared Resource Algorithm for Distributed Simulation (SRADS).	18
2.4.2 The Speculative Algorithm.	18
2.5 Alternative Simulation Approaches	19
2.6 Simulation Performance Issues	20
2.6.1 Partitioning.	20
2.6.2 Speedup.	20
2.6.3 Timing Simulations.	21 °
2.7 Simulation Testbeds	22
2.7.1 SPECTRUM.	22
2.7.2 TCHSIM.	23
2.8 Summary	23
III. Simulation Environment	24
3.1 The Intel iPSC/2	24
3.2 SPECTRUM	25
3.3 Simulations	25
3.3.1 8 LP Carwash With Feedback.	25
3.3.2 8 LP Carwash Without Feedback.	29
3.3.3 7 LP Carwash With Feedback.	30
3.4 Simulation Filters	30

	Page
3.4.1 Chandy-Misra NULL Message Filter.	30
3.4.2 Modified Chandy-Misra NULL Message Filter.	33
3.4.3 Chandy-Misra NULL Message Filter with Safetimes. . .	33
3.4.4 Shared Resource Algorithm for Distributed Simulation (SRADS).	34
IV. Queuing Experiment Design, Preparation, and Analysis	36
4.1 Introduction	36
4.2 Purpose	36
4.3 Experiment Design	36
4.3.1 Dynamic Simulation Parameters.	37
4.3.2 Static Simulation Parameters.	39
4.4 Experiment Preparation	40
4.4.1 Timing Instrumentation.	40
4.4.2 Message Counting Instrumentation.	41
4.5 Hammell's Results	42
4.6 Experimental Results	43
4.6.1 Introduction.	43
4.6.2 Effects of LP Partitioning.	44
4.6.3 Effects of Simulation Filter Types.	48
4.6.4 Effects of Model Configuration and Run Time.	51
4.6.5 Effects of SPECTRUM.	55
4.7 Summary	56
V. Summary and Conclusions	58
5.1 Summary	58
5.2 Filter Assignment Guidelines	59
5.3 LP Assignment Guidelines	60
5.3.1 4 Node Assignments.	60

	Page
5.3.2 2 Node Assignments.	62
5.4 Hypercube Allocation	63
5.5 Conclusions	63
5.6 Recommendations for Further Research	66
Appendix A. Process Manager Module	68
A.1 LP Initialization Function	68
A.2 Message Manager Function	68
A.3 Advance Time Function	69
A.4 Message Information Function	69
Appendix B. Node Manager Module	70
B.1 Initialization and Termination Function	70
B.2 Communications Function	70
B.3 Memory Management Function	71
Appendix C. 8 LP Carwash, With Feedback, Experimental Data	72
C.1 1 Node Configuration	72
C.2 2 Node Configurations	75
C.3 4 Node Configurations	79
C.4 8 Node Configuration	81
Appendix D. 8 LP Carwash, Without Feedback, Experimental Data	83
D.1 Without Feedback Case 1 Configurations	83
D.2 Without Feedback Case 2 Configurations	87
Appendix E. 7 LP Carwash, With Feedback, Experimental Data	91
E.1 1 Node Configuration	91
E.2 2 Node Configurations	95
E.3 4 Node Configurations	99
E.4 7 Node Configuration	100

	Page
Bibliography	102
Vita	104

List of Figures

Figure	Page
1. Chandy-Misra Algorithm	9
2. Deadlocked Cyclic Network	10
3. Deadlocked Acyclic Network	11
4. Waiting Dependency Loop	13
5. Conditional Chandy-Misra Algorithm	14
6. SPECTRUM Testbed Structure	26
7. 8 LP Carwash Simulation With Feedback Schematic	27
8. 8 LP Carwash Simulation Without Feedback Schematic	30
9. 7 LP Carwash Simulation With Feedback Schematic	31
10. Tandem Simulation Schematic	46
11. 8 LP Carwash Simulation With Feedback Schematic	60
12. 7 LP Carwash Simulation With Feedback Schematic	61
13. 8 Node Hypercube LP Placement	64
14. 4 Node Hypercube LP Placement	64
15. 1 Node Wall Times - 8 LPs (secs)	73
16. Nullwash Messages Sent - 8 LPs, $t = 1000$	73
17. Nullwash Messages Posted and Received - 8 LPs, $t = 1000$	74
18. Delwash Messages Sent - 8 LPs, $t = 1000$	74
19. Delwash Messages Posted and Received - 8 LPs, $t = 1000$	75
20. Safewash Messages Sent - 8 LPs, $t = 1000$	76
21. Safewash Messages Posted and Received - 8 LPs, $t = 1000$	76
22. SRADwash Messages Sent - 8 LPs, $t = 1000$	77
23. SRADwash Messages Posted and Received - 8 LPs, $t = 1000$	77
24. 8 Node Wall Times - 8 LPs (secs)	82
25. 1 Node Wall Times, Arcs Feedback - 8 LPs (secs)	84

Figure	Page
26. Safewash Messages Sent, Arcs Feedback - 8 LPs, $t = 1000$	85
27. Safewash Messages Posted and Received, Arcs Feedback - 8 LPs, $t = 1000$	85
28. 8 Node Wall Times, Arcs Feedback - 8 LPs (secs)	86
29. 1 Node Wall Times, w/o Feedback - 8 LPs (secs)	87
30. Safewash Messages Sent, w/o Feedback - 8 LPs, $t = 1000$	88
31. Safewash Messages Posted and Received, w/o Feedback - 8 LPs, $t = 1000$	88
32. 8 Node Wall Times, w/o Feedback - 8 LPs (secs)	89
33. 1 Node Wall Times - 7 LPs (secs)	91
34. Nullwash Messages Sent - 7 LPs, $t = 1000$	92
35. Nullwash Messages Posted and Received - 7 LPs, $t = 1000$	93
36. Delwash Messages Sent - 7 LPs, $t = 1000$	93
37. Delwash Messages Posted and Received - 7 LPs, $t = 1000$	94
38. Safewash Messages Sent - 7 LPs, $t = 1000$	94
39. Safewash Messages Posted and Received - 7 LPs, $t = 1000$	95
40. SRADwash Messages Sent - 7 LPs, $t = 1000$	96
41. SRADwash Messages Posted and Received - 7 LPs, $t = 1000$	96
42. 7 Node Wall Times - 7 LPs (secs)	101

List of Tables

Table	Page
1. iPSC/2 Specifications	24
2. Source LP Destinations and Departure Factors (8 LPs)	27
3. Source LP Arrival and Delay Times (8 LPs)	28
4. Wash LP Delay Times and Departure Factors (8 LPs)	29
5. Rewash Rates and Delay Times (7 & 8 LPs)	29
6. Source LP Destinations and Departure Factors (7 LPs)	31
7. Source LP Arrival and Delay Times (7 LPs)	31
8. Wash LP Delay Times and Departure Factors (7 LPs)	32
9. 2 Node Tandem Model Results, $t = 10000$	46
10. 4 Node Tandem Model Results, $t = 10000$	46
11. Overall Node Wall Times - 8 LPs (secs), $t = 1000$	49
12. Overall Node Wall Times, Arcs Feedback - 8 LPs (secs), $t = 1000$	50
13. Overall Node Wall Times, w/o Feedback - 8 LPs (secs), $t = 1000$	51
14. Overall Node Wall Times - 7 LPs (secs), $t = 1000$	51
15. 8 LP With Feedback Speedups - 1 Node vs. 8 Node	52
16. 8 LP With w/o Feedback Speedups - 1 Node vs. 8 Node	52
17. Hypercube LP Placement Wall Times (secs), $t = 10000$	65
18. 1 Node Wall Times - 8 LPs (secs)	72
19. 2 Node Wall Time Ranges - 8 LPs (secs), $t = 1000$	78
20. 4 Node Wall Time Ranges - 8 LPs (secs), $t = 2000$	80
21. 8 Node Wall Times - 8 LPs (secs)	81
22. 1 Node Wall Times, Arcs Feedback - 8 LPs (secs)	83
23. 8 Node Wall Times, Arcs Feedback - 8 LPs (secs)	86
24. 1 Node Wall Times, w/o Feedback - 8 LPs (secs)	87
25. 8 Node Wall Times, w/o Feedback - 8 LPs (secs)	89

Table	Page
26. 1 Node Wall Times - 7 LPs (secs)	91
27. 2 Node Wall Time Ranges - 7 LPs (secs), $t = 1000$	97
28. 4 Node Wall Time Ranges - 7 LPs (secs), $t = 2000$	99
29. 7 Node Wall Times - 7 LPs (secs)	100

Abstract

The objective of distributed simulation is to speedup simulation execution by partitioning the simulation processing load over multiple processors. This thesis reviews current synchronization protocol methods for distributed simulations, and proposes guidelines for obtaining optimal conservative simulation partitionings using empirical evidence.

An analysis is performed using three protocol variations of the Chandy-Misra NULL message algorithm, two using a pending message blocking strategy, and the other using a safetime blocking strategy. A fourth protocol evaluated is based on the SRADS algorithm proposed by Reynolds. The analysis involves a study of all possible 2 and 4 node configurations, for three queuing simulations, using all possible protocol and model pairings. A fourth queuing model is then used to independently validate results.

In the end, the safetime version of the Chandy-Misra protocol is demonstrated to provide better overall performance than the other protocols evaluated. Partitioning guidelines developed established a relationship between process configurations and load balancing. It is seen that separating highly communicative processes onto different nodes, or locating highly communicative processes on the same node with fewer processes, provided the optimal 4 node configurations, while reducing the number of intra-node process connections provided for the optimal 2 node configurations.

DEVELOPMENT OF A PROTOCOL USAGE GUIDELINE FOR CONSERVATIVE PARALLEL SIMULATIONS

I. Introduction

1.1 Background

Webster's dictionary defines a *simulation* as a "training device that duplicates artificially the conditions likely to be encountered in some operation". Although most people never think about computer simulations as something commonplace, simulations are in fact a very real and vital part of our everyday world. From teaching a pilot to fly and predicting tomorrow's weather, to training Olympic athletes and planning for global war, computer simulations are a critical component in each of these activities.

Traditionally, system simulations like those above have all been done in a sequential manner on a single central processing unit (CPU), meaning that the entire simulation is executed by the same CPU. In addition, for discrete-event simulations, the system being simulated is only executed at specific discrete points in time, after which the simulation clock is then advanced to the next operation or computation. This sequential mode also implied that only a single simulation computation could be done at any one point in time. To store the operations, or events, discrete-event simulations usually make use of an event list structure, which is normally ordered in increasing time. As each event was removed from the front of the event list and processed, the simulation time was advanced to the time of the event. This idea of processing events in increasing time is important. In real-world systems, time is always advancing forwards; going backwards in time cannot happen. Therefore, simulations cannot allow "old" or past events to be carried forward.

Although simulations, such as those above, have solved many critical real-world problems, they suffered from two main problems: they had to be executed in a sequential manner, on a single CPU with no concurrency, and second, as a result, they were relatively slow as compared to the real-time system they were supposed to be simulating. For example,

tomorrow's weather forecast would be ready in three days, or by the time an aircraft simulation responded to the pilot's input, the plane would be nose deep in the ground. In order to solve these types of problems, the idea of using a parallel computer was evolved. In particular, by allowing the simulation to be partitioned into several independently executing parts or processes, and distributed across several CPU's, the simulation's processes could be run concurrently at an overall speed faster than on a single CPU. Thus, any real-world system that included individual components or modules which interact independently, at discrete times, could be simulated.

In general, the partitionings of the real-world system form a series of interacting modules, or physical processes (PP), that communicate with each other by passing messages. And as such, each specific PP behavior, at time x , is influenced only by the messages received from the other PPs after time x . In addition, there is also a defined simulation model logical process (LP) corresponding to every PP. An LP can safely simulate the actions of a PP up to time x if the LP knows the initial state, and all the messages that the corresponding PP received up to time x .

Of course, it would be nice if all randomly selected partitionings, for a specific simulation, produced optimal results. But the method chosen to partition a simulation is extremely critical. How a simulation is partitioned will not only effect the control of each LP within the simulation, but also effect the level of speedup attainable by the simulation. Thus, one of the basic goals of discrete-event distributed simulations is to improve and optimize simulation performance, with respect to wall time and efficiency of operation over the sequential version, through system partitioning.

To control the simulation's message passing, that is, to ultimately insure that all the messages are received and processed in an increasing time ordered manner, two basic message synchronization protocol methods have been developed: conservative and optimistic. In the conservative method approach, the simulation *avoids* the possibility of an out-of-place message by determining when it is safe to process the current message, or by determining when all the messages that could affect the current message have been processed. Optimistic methods use a *detection and recovery* approach; errors are first de-

tected (messages from the past), then the simulation is taken back to a time before the error occurred, reset using the old message as a guide, and then allowed to proceed.

1.1.1 Conservative Methods. Some of the first conservative protocol methods were developed during the late 1970's by Chandy-Misra (7) and Bryant (4). In these approaches, increasing time-stamped messages are sent to an LP if the originating LP has determined, from examining all pending input messages, that it is safe to send the message. The major problem encountered with this method is that if the LP determines that a message cannot be sent, the LP must block and wait until a message is received on all input lines, and then determine which message, if any, can be processed; this can lead to deadlock situations if appropriate actions are not taken. Deadlock situations arise when one LP is waiting on another LP to proceed, which in turn is waiting on the first LP to proceed, bringing the simulation to a halt. One way to avoid deadlocks is to allow the LPs to send NULL messages (messages containing no simulation information except time) at specific points during the simulation. These messages specify that the originating LP will not send any messages in the time interval between the last message received, and the time of the NULL message.

The basic NULL message method, as with most protocol methods, has many variants. Some of these are:

- 1) Sending NULL messages when a receiving LP demands a NULL message, rather than at specified points during the simulation.
- 2) Sending NULL messages only after a specified number of real messages.
- 3) Sending NULL messages only after it has been determined that a real message will not be sent out in some specific future time interval.
- 4) Sending NULL messages to every other connected LP anytime a NULL message is sent.

1.1.2 Optimistic Methods. Optimistic methods, most notably Time Warp, were first originated by Jefferson in 1985 (16). In this method, if an LP receives an message from the past, the LP will "rollback" the simulation time to the time the past message should

have occurred, and restarted. In particular, under the Time Warp method, each message sent has two time values, a send time and a receive time. The receive time represents the time of arrival at the receiver, i.e., the same, and only time used in the conservative methods. The send time equals the local clock, defined as the minimum receive time of all unprocessed messages, of the originating LP when the message was sent. LPs then process messages and advance the local simulated time for as long as they have *any* messages waiting, and unlike the conservative methods, the LP does so without waiting to receive messages on *all* its input lines. Because of this, the LP's local clock may get ahead of its predecessors' local clocks, and may receive a message from the past. If this happens, the LP "rolls back" its simulation time and restarts the simulation to take into account the new message. In order to cancel the effects of the previous (now erroneous) messages, the LP also sends "antimessages" to its successors. Also, unlike the conservative methods, the states of each LP since the last correct time must be stored; this enables the simulation to be correctly rolled back.

1.1.3 SPECTRUM Simulation Testbed. With these various types of message protocols present, there is still not enough current comparative data to determine which protocol is best for a particular application. The SPECTRUM (Simulation Protocol Evaluation on a Concurrent Testbed with ReUsable Modules) testbed has been designed to support the empirical comparative study of parallel simulation protocols and applications under a controlled environment (26). SPECTRUM is composed of four programs (12): an application program, a process manager, a node manager, and the protocol or filter program itself. The process manager provides the interface to the user-supplied application program, and includes modules for clock advancement, message management, and initialization. The node manager provides the interface between the process manager and the parallel computer. The filter provides the modules to implement the message protocol, and is interfaced with the process and node managers. With this, SPECTRUM can be used to easily switch-out the application and filter programs, with other application and filter programs with common SPECTRUM interfaces, providing for the quick turnaround and baselined environment needed for empirical comparison studies.

1.2 Problem Statement

Using empirical evidence, can a given simulation model be analyzed prior to implementation, and based on the analysis, the best protocol and LP allocation strategy be predicted to obtain the optimal simulation wall time?

1.3 Research Objective

The objective of this research is to develop a guideline to determine, based on model characteristics, the best protocol and LP allocation strategy for use with a simulation model.

1.4 Research Questions

To reach the stated objective, this thesis research specifically looks at the following questions:

- 1) How do protocol variations affect model performance and wall time?
- 2) How can the models be partitioned and allocated across several nodes, based on the application and message protocol, to optimize wall time and performance?
- 3) Given an untried simulation model, can the most efficient partitioning and protocol be predicted for use with that model?

1.5 Scope

This research effort consists of empirical studies performed on discrete-event simulations and simulation protocols. Specifically, the dependencies between message protocols and simulation models are explored using the SPECTRUM testbed. All empirical studies are performed on hardware implementing a distributed memory architecture.

1.6 Thesis Outline

Chapter I serves to introduce the concepts and methods of parallel simulations and of simulation testbeds. Chapter II contains an overview of current literature. Chapter

III discusses in detail the various environments, both hardware and software, and protocol algorithms, used in this research. Chapter IV discusses the approach, methodology, and design of the experiments, and contains the analysis of the experimental results, and Chapter V includes experimental conclusions and future research recommendations.

II. Current Issues

2.1 Introduction

As alluded to in Chapter I, there are many variants of the conservative and optimistic message synchronization protocols. This review will begin by discussing several of the more well-documented message protocols, as well as briefly mentioning some that are fairly new and less well known. This will be followed by a discussion of the more important simulation performance issues and experimental testbeds. In general, one of the main concerns in comparing simulations is not only how to measure individual performances against other simulations, but also what to measure. To compound this problem, there has been no easy method to compare simulation performances on a common environment or testbed. This has been alleviated, in part, by the creation of the SPECTRUM and TCHSIM testbeds. These testbeds allow simulations, and their protocols, to be easily switched with other simulations and protocols, leaving all support systems intact, thus giving the common environment needed for comparative studies.

2.2 Conservative Simulation Approaches

2.2.1 The Basic Chandy-Misra Algorithm. In the basic Chandy-Misra algorithm (7), each LP can only execute its own code and two commands: send and receive. In sending, an LP places a message on an output line, blocks until the destination LP acknowledges receipt of the message, then proceeds with its own code execution. In receiving, an LP opens any one of many possible input lines from which to receive a message, but may have to wait until the message arrives along the chosen opened line. These messages are delivered, and received at the destination LP, in the order in which they were sent.

Every message between PP's is simulated between the corresponding LP's using the following message sequence format:

$$(time_1, msg_1), (time_2, msg_2), \dots, (time_N, msg_N)$$

and according to the following rules (6):

1) $0 \leq time_1 \leq time_2 \dots \leq time_N$.

2) A PP must have sent msg_N to another PP at $time_N$.

3) A PP must have sent no other messages to another PP besides $msg_1, msg_2, \dots, msg_N$.

This means that the sequence of messages sent by an LP, up to $time_N$, must correspond exactly to the actual message sequence sent by the corresponding PP. Also, given that the LP time is defined as the minimum time of all the incoming lines, all subsequent messages sent or received by an LP, at *any* point during the simulation, must have a time greater than the LP time. The LP computes the time value of a message going out by using, as a lower bound, the time of the next transmitted message along that line, and computes the time value of the incoming line using the time of the last message the LP received along that line. Thus, to summarize (6):

1) An LP must guarantee that all subsequent input messages, along each input line, will have a time greater than or equal to the LP time.

2) An LP cannot send any message on any output line where the message time is less than the LP time.

3) An LP need not send a message out on every line, every time, if the LP can guarantee that the times on all subsequent outgoing messages will be greater than the LP time.

Normally, during a simulation execution, an LP alternates between computing and waiting to communicate. When waiting, an LP will follow these rules (6):

1) An LP must wait, before any computations are performed, until the LP has received a message on *all* input lines whose time is greater than or equal to the LP time.

2) An LP must wait on *all* output lines, on which there is a message to be sent, until each message time is greater than or equal to the LP time.

In other words, an LP cannot compute any output based on only one incoming message and with pending messages on less than the total number of input lines. Similarly, an LP must wait to send a message on an output line which has a queued message, until all other output lines are also ready. Whenever an LP does receive a message, it compares the

times of all the incoming messages and chooses the minimum value to work with, and to update its local clock. If the LP time changes, the LP advances the simulation to the new time, and then computes, for each outgoing line, any messages that it will need to send with times up through the new local time. In some cases the outgoing lines will have no messages destined for them, so no messages are sent. Thus, the basic message algorithm (20) for LP_i is shown in Figure 1.

```

Initialize  $LP_i$  time  $T_i = 0$ ;
While the simulation is not complete, loop.
{
    /*Simulate  $PP_i$  up to  $T_i$ .*/
    For each outgoing line, compute the following sequence of messages:

         $(time_1, msg_1), (time_2, msg_2), \dots, (time_n, msg_n)$ 

    /*Where  $time_1 < time_2 \dots < time_n$ , and  $PP_i$  sends  $msg_i$  at  $time_i$  along line.*/

    /*1) All messages sent by  $PP_i$  up to  $T_i$  can be deduced by  $LP_i$  and sent.
    2) Also some messages to be sent beyond  $T_i$  may be predicted by  $LP_i$  and sent.
    3) Only new messages that have not been sent before are sent, and some or all of
    these message sequences may be empty.*/

    Send each message, in the sequence, along the appropriate line;

    /*Receive any messages and update  $T_i$  until  $T_i$  changes value.*/

    /*Initialize  $T'_i$ .*/
     $T'_i = T_i$ ;

    While  $T'_i = T_i$  loop.
    {
        Wait to receive messages along all incoming lines;
        Upon receipt of a message, update all  $LP_i$  internal states and
        recompute  $T_i$ , the minimum overall incoming line clock value;
    }
}

```

Figure 1. Chandy-Misra Algorithm

A major problem with this basic algorithm is that deadlocks can occur when all the LPs wait on each other to send a message, as in a cyclic network (Figure 2), or in an acyclic network (Figure 3), when LP1 only sends messages to LP3. When this happens, LP3 cannot proceed until it receives a message from LP2, which, in this case, will never occur. In both cases, though, no computation is done and the simulation is never advanced. In a more formal definition (20), deadlock occurs when *all* of the following conditions hold:

1) Every LP is either waiting to receive a message or has been terminated.

2) At least one LP is waiting to receive a message.

3) For any LP that is waiting to receive a message, there is no message in transit to that LP.

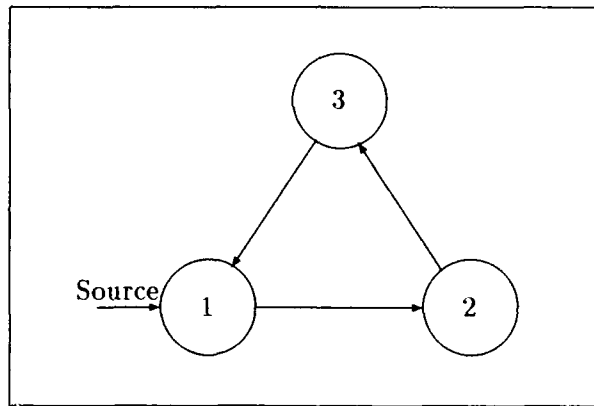


Figure 2. Deadlocked Cyclic Network

2.2.2 The Chandy-Misra Algorithm with NULL Messages. A popular method to avoid the deadlock problem is to allow LPs to send NULL messages at specific points during the simulation (20). These messages indicate that an LP will not send any messages during the time interval between the last message processed and the encoded time of the NULL message. As a result, any future message received by an LP will have a time exceeding that of the NULL message time. Reception of a NULL message is treated in the same manner as with any other message, causing the destination LP to update its state and local clock. For example, if it is required that the simulation be correct up to time x , then every LP

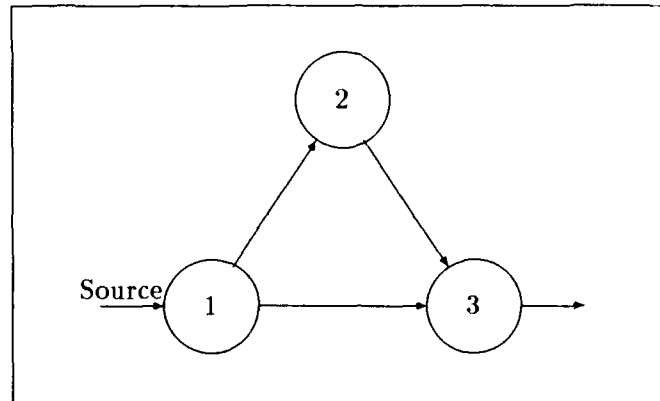


Figure 3. Deadlocked Acyclic Network

must send out messages until the LP time equals time x . If no non-NULL messages are available, NULL messages must be sent until time x is reached.

Suppose now that the LP can predict, that after a message sequence, it will not send out any message until some future time x . At this point the LP will send a NULL message, after the message sequence, with the predicted future time x encoded. Since the LP knows its own state until time x , it can predict all messages, real and NULL, at least until time x . This means that for each outgoing line, the last message time will be greater than or equal to the original LP time, and that NULL messages will be the only last messages sent.

Although by using NULL messages the simulation will never deadlock, significant numbers of NULL messages could be sent which are not necessarily needed or used. For example, a non-processed NULL message has no effect if it is followed by a real, or another NULL message, with a greater time stamp. One proposal to reduce the number of NULL messages sent is to delay the transmission of NULL messages for some arbitrary time, hoping that any future non-NULL messages received by an LP will make it unnecessary to transmit the NULL messages at all (20). This time delay ranges from no delay, which results in NULL messages transmitted all the time (the basic algorithm), through infinity, in which no NULL messages are transmitted, resulting in deadlock possibilities. Another proposal would be to send out NULL messages only after some number of non-NULL messages are sent, or after a specific amount of simulation time has elapsed (9).

The basic problem with both of the above proposals, and those like them, is that there is no optimal delay time or number of non-NULL messages. It all depends on the simulation, the computer architecture, the amount of memory, etc. A more palatable approach is to use NULL message cancellation methods (22). In this approach, for example, when an LP receives a NULL message and then later receives another message, NULL or non-NULL, with a greater timestamp, the previous NULL can be discarded if it has not yet been processed. An alternative method, for cases in which LPs loopback on themselves, is to also not send NULL messages back to themselves in *any* case.

2.2.3 The Carrier NULL Approach. Recently, an improvement on the basic NULL message Chandy-Misra algorithm has been brought to light (5). This method, known as the “carrier NULL message” method, not only advances the simulation time but also stores global knowledge about the system being simulated. This method relies on the fact that a good simulation lookahead scheme can reduce message traffic by being “smarter” about the messages sent, thus achieving higher performances and speed-ups. The simplistic lookahead ability of the basic Chandy-Misra algorithm is thus improved through analyzing the global knowledge carried within the specialized carrier NULL messages.

The algorithm is based on a concept called *global waiting dependencies*, which eliminates the requirement that an LP must wait until all input lines have message times greater than or equal to the LP time before processing a message. Referring to Figure 4, normally LP1 cannot send a message to LP2 unless LP2 first sends a message to LP3, etc. This type of loop is what is called a waiting dependency loop. But under this algorithm, LP2 does not have to wait for LP1 in order to send a message to LP3; it can process a message from the source input, and send the message to LP3, without concern about whether there may be a message coming from LP1 with a older time.

To provide this certainty, carrier NULL messages are used to provide the earliest time of any possible real messages which can break the waiting dependency loop. A carrier NULL message is of the form (*time*, *route info*, *lookahead info*, carrier NULL), where *time* is the timestamp of the message, *route info* is the LP's identifier, and *lookahead info* is the earliest time of all the possible messages which can break the waiting dependency loop.

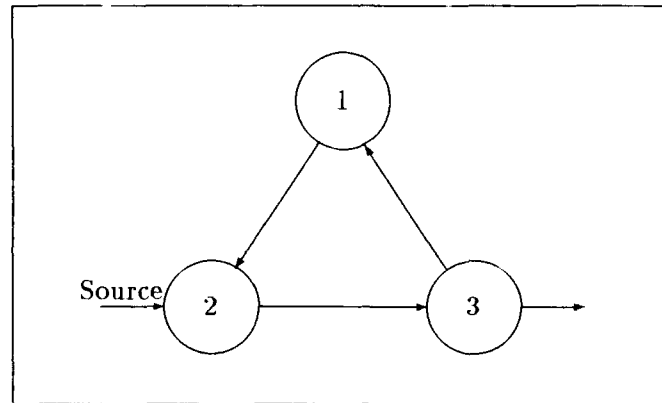


Figure 4. Waiting Dependency Loop

Carrier NULL messages are treated the same as any other messages; the timestamp on the carrier NULL message causes the LP to update its clock and process any messages. In addition, the LP may now safely predict and process any future simulation messages up to the lookahead time given by the carrier NULL message. The new prediction is then sent to other downstream LPs via a NULL message, with a time equal to the original *lookahead info* time.

2.2.4 A Conditional Chandy-Misra Approach. In an effort to eliminate altogether the need for any type of NULL message to avoid deadlocks, a conditional algorithm has been proposed (8). The major differences from the basic NULL message algorithms are:

- 1) The use of messages for other than interprocess communications.
- 2) The use of messages which need not duplicate those in the PP.
- 3) The use of conditional messages, which may or may not be transmitted next depending on the time stamp and the state of the other messages.

Basically, the event list in any sequential simulation can be thought of as a list of *conditional messages*, in which some particular message is the next message to be executed on the condition there are no earlier messages. In this algorithm, the event list is used to determine *definite messages* (those which *will* occur in the system being simulated, and thus are safe to process) from conditional messages (those which *may or may not* occur.

and thus are not guaranteed to be safe to process). A definite message is defined as the next message in the event list that has the minimum time of all the LP's pending messages. The basic algorithm is shown in Figure 5.

```

While the simulation is not complete, loop for each LP.
{
    1) Compute and execute as many definite messages as far into the
       future as possible from stored conditional messages.

    2) Update and send a message containing the conditional message
       time after computing all possible definite messages.

    3) Wait to receive and store a message from all LPs containing
       the conditional message time.
}

```

Figure 5. Conditional Chandy-Misra Algorithm

2.2.5 Chandy-Misra with Channel Clocks. Proicou (23) modified the basic Chandy-Misra with NULL messages algorithm to include what he calls "channel clocks". Channel clocks are used to store the time of the last message received or sent along an input or output line. Before an LP can proceed, it must be sure, by checking the input channel clocks, that its next message time is less than the minimum times of the last messages received from *all* its input LPs. In doing this, the LP ensures that no message can arrive with a time earlier than the message being processed. The output channel clocks are used to reduce the number of NULLs sent by eliminating those NULLs with times less than or equal to the current output channel clock time for that line.

2.2.6 The Chandy-Misra Algorithm Variants. In addition to the approaches already discussed, there are also several other approaches to the basic Chandy-Misra algorithm which are being studied and evaluated (28). These schemes follow the given rules:

- 1) When an LP sends a message, the destination LP will acknowledge receipt.

2) If an LP receives a message, the originating LP either sends or queues-up another message.

3) If no messages are being received at an LP, a NULL message pointer is returned, and the destination LP selects a queued output message to send. If no messages are queued, the destination LP may send a non-NULL message to *its* predecessors requesting that any queued messages be sent.

2.2.6.1 Eager Events, Lazy NULL Messages. In this method, all NULL messages are queued-up, and any non-NULL messages produced are combined with the queued NULL messages, and sent immediately. If the destination LP happens to return a NULL pointer, the NULL message with the earliest time is then sent.

2.2.6.2 Indefinite-Lazy, Single Event. For this algorithm, *all* messages are queued-up and sent *only* when the destination LP returns a NULL pointer. At this point, the queue with the earliest message time is selected to generate messages up to the *first* message, if any, or NULL message.

2.2.6.3 Indefinite-Lazy, Multiple Event. This is identical to the indefinite-lazy, single event variant, except the queue with the earliest message time is selected to generate messages up to the *last* message, if any, or NULL message.

2.2.6.4 Demand-Driven. In demand driven, all output messages from the originating LP are queued-up, and no NULL messages are sent unless the destination LP demands one. Therefore, an LP which may have to create numerous NULL messages can save extra work if none of its successors make requests for NULL messages. When a destination LP returns a NULL pointer, the originating LP that lags the furthest behind selects the destination for the demand message. Upon receipt of this message, if the queue is not empty, the LP sends all messages in the queue. If the queue is empty, the LP generates another demand message to its predecessors.

2.2.6.5 Demand-Driven, Adaptive. In this approach, a message threshold is established and associated with each output line. Messages generated by an LP are queued

up to the threshold. When the threshold is reached, the entire contents of the queue are sent. The demand message method works as above, but it additionally causes the threshold level to decrease.

2.3 Optimistic Simulation Approaches

2.3.1 The Basic Time Warp Algorithm. Under the Time Warp (TW) algorithm (16), a message received by an LP does not necessarily have to be in increasing timestamp order, as mandated by the Chandy-Misra algorithms, and as such, messages can be received in any order. This implies that the TW algorithm takes an optimistic approach in advancing the simulation time, and assumes that each message received is the next correct message, and proceeds to process each message as received. Whenever a message is received with a timestamp less than some other message already processed, the algorithm was too optimistic and must correct its mistake. It does this in the following manner:

- 1) backup or "rollback" the simulation to a time just before the incorrect message time;
- 2) execute the former incorrect message at the correct simulation time;
- 3) start re-executing any messages with timestamps greater than the former incorrect message time in timestamp order, then
- 4) cancel the effects of any output messages that were sent with times greater than the former incorrect message time prior to simulation rollback.

The rollback and canceling output message processes in the TW algorithm are special cases and must be handled in a very specific manner. To support the rollback process, TW regularly saves the condition of the state of each LP. These states are stored in a queue associated with each LP and are accessed whenever it is necessary to perform a rollback. The cancellation of previous, now erroneous, messages is accomplished by means of "antimessages". In TW, every message is considered to have either a + sign or a - sign. Two messages that are identical, but opposite in sign are said to be opposite to each other, thus the term "antimessage". Whenever TW sends a message to another LP, an antimessage is also created and stored in the LPs output queue. As long as the LP does not

rollback, the antimessages are never used and are eventually destroyed. If, however, the LP must rollback, all the antimessages with times greater than the rolled back simulation time will be sent out (otherwise known as aggressive cancellation) to cancel any messages at the other LPs.

Another important concept in the TW algorithm is that of Global Virtual Time (GVT). GVT is defined as the smallest time among all the unprocessed messages. No messages with times less than the GVT can be rolled back. In other words, the GVT is the time in which the simulation has safely reached and is sure that no messages less than the GVT will be received.

2.3.2 Time Warp Variations. As with the Chandy-Misra algorithms, there are several TW variations and modifications available (10).

2.3.2.1 Lazy Cancellation. In the aggressive cancellation TW algorithm, as mentioned above, whenever an LP must rollback, antimessages are sent out immediately to cancel any real messages sent before the rollback time. In lazy cancellation, LPs do not immediately send out the antimessages. Instead, the LPs wait to see if the new execution will produce some of the same messages. If an identical message is created, there is no need to send the antimessage. If, on the other hand, a message is not created, after its time the antimessage is sent.

2.3.2.2 Lazy Reevaluation. This method is similar to lazy cancellation, except it deals with the LPs' state and not its messages. This variation looks at the state of the LP before a rollback, and again after a rollback. If the re-execution of rolled back messages leads to an identical state as before, the method "jumps forward" over them and continues simulation execution.

2.3.2.3 Direct Cancellation. Under this TW variation, whenever a message schedules another message, a pointer is connected to both messages. If the second message is determined to be canceled, it can be found with ease by following the pointer from

the first message. Without this, as under the other TW algorithms, the algorithms must perform a search to locate the messages to cancel.

2.4 Hybrid Simulation Approaches

2.4.1 Shared Resource Algorithm for Distributed Simulation (SRADS). SRADS was proposed by Reynolds (24) to improve upon simulation performance degradation caused by other methods used to prevent deadlock, and by the blocking of processes awaiting message time requirements in order to proceed. As will be seen, SRADS is both conservative, by blocking LPs for synchronization, and optimistic, by assuming the next message time.

In the SRADS algorithm, it is assumed that the communication line between LPs is a shared resource, and that the shared resource is a facility for message storage, or a *shared facility*. Every LP connected to a shared facility may either read from or write into the facility, given the following constraint: all the other LPs connected to the same facility have times which are greater than or equal to the requesting LP. When an LP needs to access a shared facility, it sends out POLL messages encoded with the current time of the LP at pre-determined frequencies to the facility. The LP will then block while waiting for an ACK message from the facility indicating that its time has advanced to at least that of the POLL message. Thus, the POLL message is used to synchronize connected LPs by ensuring that the sending LP does not get behind the receiving LP. In addition, the POLL message is used to aggressively advance the simulation time in the absence of any other information. If an LP's next message is a POLL message, it will go ahead and advance its clock to that of the POLL message, block, and then wait for an ACK message. With this type of aggressive strategy, an LP may receive a message from the past called a *time slip*. This aggressiveness is based on two assumptions made by an LP (27):

- 1) Messages will only arrive at regularly scheduled times. Thus, between POLL message times the LPs can advance at their own rates.
- 2) The LP can advance its clock to the next POLL message time.

2.4.2 *The Speculative Algorithm.* Mehl (19) introduces a new distributed simulation method in which an optimistic approach is combined with a conservative method, producing what is called a “speculative simulation” method. In this method, a conservative algorithm is modified so that during its non-computing times it processes future messages in advance, before their correct execution time. These messages are processed on a copy of the current LP state, so that the current local state or event queues are not modified, and stored in a local buffer. If it turns out that the early execution was correct, the local state and event queues can be updated quickly, saving time during the computing phases. These early executions are called “speculative executions” because the algorithm “speculates” that the advanced processing will be correct.

2.5 *Alternative Simulation Approaches*

As seen above, most simulation algorithm research has concentrated on modifying either of the two traditional classes of simulation approaches: conservative or optimistic. Reynolds (25) has proposed that there are many other alternative approaches to simulations that have yet to be addressed. The proof of this statement lies in a set of nine design variables which defines a simulation algorithm. By studying different combinations of these variables, it was shown that current simulation approaches can be correctly derived from these variables, as well as an infinite variety of other unexplored approaches. These design variables are described as:

- 1) Partitioning: How are LPs distributed across the nodes.
- 2) Adaptability: Allowing an LP to dynamically change the protocol algorithm it is using based on some chosen criteria.
- 3) Aggressiveness: Allowing the *initial* processing of messages not to be in a time-ordered sequence.
- 4) Accuracy: Requiring that the processing of messages *ultimately* be in a time-ordered sequence.
- 5) Risk: Allowing inaccurate or out-of-sequence messages to be passed on.
- 6) Knowledge Embedding: The amount of knowledge embedded within messages.

- 7) Knowledge Dissemination: Allowing LPs to transmit knowledge to other LPs.
- 8) Knowledge Acquisition: Allowing LPs to request knowledge from other LPs.
- 9) Synchrony: The degree that an LP works independently from other LPs.

2.6 *Simulation Performance Issues*

As stated before, the goal of distributed parallel simulation is to reduce the time needed to produce results as compared to the same simulation executed on a sequential computer. Two factors, more than any other, affect how well a simulation is transformed from sequential processing to parallel processing: partitioning and speedup measurement.

2.6.1 Partitioning. Partitioning is the act of splitting up a simulation into separate components or LPs. Each LP will, in turn, perform some portion of the total simulation. How this partitioning is accomplished not only affects the management of each process within the simulation, but also affects the synchronization between each of the processes. Partitioning, in general, can be broken down into four basic issues: synchronization, granularity, data partitioning, and functional partitioning. (21)

1) Synchronization: Synchronization is the exchanging of messages between LPs.

2) Granularity: Granularity is the ratio of the amount of processing an LP can perform without the need to synchronize with another LP. In other words, a very large-grained process does not need to "talk" with other LPs, whereas a fine-grain process is constantly exchanging messages with others.

3) Data Partitioning: Data partitioning replicates the simulation across every LP but splits up the data onto the LPs which need the data.

4) Functional Partitioning: Functional partitioning, as its name implies, splits up the simulation functionally across each LP, and data is passed along from one LP to another, when needed, to complete the simulation.

2.6.2 Speedup. Parallel simulation performance is usually measured by the decrease in time over the same simulation executed sequentially. This time difference, known as

speedup, is formally measured by the ratio of the time to execute a process sequentially to the time to execute the same simulation in parallel. Therefore, theoretically, if a simulation takes t seconds to run sequentially, then when running in parallel with n LPs, it should take only t/n seconds (linear speedup). But because of computer scheduling, context switching, and synchronization delays, linear speedup is rarely achieved.

2.6.3 Timing Simulations. One of the major problems with measuring and reporting simulation speedup is in how the timings are gathered. The methods for timing simulations can cause wide variations in the computed speedup, and as such, the comparison of speedup values between simulations, especially on different architectures, can be a comparison between apples and oranges. Generally, either wall or CPU clock timings are used to gather information, but a new log-based method has been proposed which, according to the author, will eliminate the bias of the CPU and simulation. (17)

2.6.3.1 Wall Clock Timings. Wall clock timings, especially when more than one simulation is competing for a CPU, are generally considered to be too coarse for extremely accurate simulation timings. The main reason for this is that the wall clock cannot separate the time that an LP is working and the time that an LP is swapped out. As such, this time increases as the CPU load increases.

2.6.3.2 CPU Clock Timings. As an alternative, CPU clock timings can provide a more accurate method of obtaining simulation times. A problem with CPU clock timings is that depending on the type of communication scheme used (e.g., busy-wait). System loading can play a small role in increasing the overall time.

2.6.3.3 Log Timings. In this new scheme, each process creates a log containing all of its communications. During this logging phase, the LP denotes the CPU time before and after each send or receive. After the simulation terminates, an algorithm processes each of LP's logs, and compares a set global time to the entries in the log and calculates elapsed times.

2.7 Simulation Testbeds

In the past, researchers have performed experiments with various simulation algorithms and compared results from those experiments without much of a common baseline. As such, comparisons were, to say the least, not very good without an extensive understanding of the architectures and conditions of the experiments. In an effort to eliminate this discrepancy and create some kind common environment to compare algorithms, the idea of a testbed was formulated. SPECTRUM (26) (12), and TCHSIM (14) are two such testbeds.

2.7.1 SPECTRUM. The SPECTRUM testbed preliminary results have generally been positive, and have uncovered several unknown design issues (26). Reynolds' initial theory was that the effectiveness of a simulation algorithm is dependent on the simulation. His findings seem to corroborate this theory (in fact the dependence is much more than expected) and the simulation designer must be aware of the algorithms' limitations. Reynolds additionally devised a method to package the protocols into "filter" units which could be changed during and/or after a simulation run. This significantly increased the flexibility of the testbed by reducing the overhead of changing simulation protocols to merely relinking the simulation code over the static testbed environment. By means of SPECTRUM, Reynolds additionally identified several simulation variables which affect the choice and design of simulation protocols:

- 1) Determinism: The degree to which a simulation is deterministic.
- 2) Queuing: Amount and type of queuing system employed by the simulation.
- 3) Processing Delays: How many processing delays occur during the simulation?
- 4) Causality: Does every message that arrives at an LP directly cause another message to be sent?
- 5) State Change Characteristics: How often do variables change state during the simulation?
- 6) Balance: How uniform is the processing requirement across each LP?
- 7) Activity Level: How many LPs are busy at any point in time?

8) Connectivity: To what extent can messages on one LP affect messages on other LPs?

2.7.2 TCHSIM. TCHSIM (Thomas C. Hartrum SIMulation) is an AFIT object-oriented simulation testbed, interfaced with SPECTRUM, designed to provide a general, discrete-event simulation environment to allow experimentation with several application models without having to reconfigure the simulation or modify the model everytime. TCHSIM is composed of nine modules: a startup and initialization module (driver), an application module, and an interface module to several object oriented functions (clock, event, next event queue, and general support) and the SPECTRUM process and node managers. (14)

2.7.2.1 QUESIM. QUESIM (QUEuing SIMulation) is a general queuing simulation program, built around TCHSIM, that provides an easy method to create customized network queuing applications. QUESIM is built around seven node types: a source node that generates arrivals, a server node with a local event list and delay time, a routing node that sends messages out with no delay, a merge node that receives several messages into a single output, a sink node that destroys the message, and transmit/receive nodes which transmit and receive messages for parallel simulation execution. (13)

2.8 Summary

As seen, most simulation methods have been around since the late 1970s and are very well studied and understood. Even though conservative and optimistic approaches to simulations are by far the most prevalent, current thinking is beginning to show that there may be an infinite number of other types of methods that have yet to be explored. Even by just looking at the surface of these new approaches, a better understanding of simulation protocols and applications is beginning to come to light. This is even more pronounced when using a testbed in which a simulation can be held constant and the protocol varied, and seeing the changing of performance under a controlled environment.

III. Simulation Environment

3.1 The Intel iPSC/2

The iPSC/2 is a distributed-memory, parallel supercomputer developed by Intel Scientific Computers. The AFIT version of the iPSC/2 is based on a three-dimensional (8 node) hypercube configuration which has the specifications listed in Table 1 (1). Architecturally, the iPSC/2 computer consists of two types of CPU's (nodes), compute and I/O, along with a front-end processor called the System Resource Manager (SRM) or *host*. The nodes are setup as a processor/memory pair, with their physical memory distinct from that of the host and the other nodes. In addition, all the nodes are connected via a Direct Connect Module (DCM) which coordinates node communication activities. In general, the DCM provides the nodes with direct, high-speed bidirectional message passing capabilities through calls to the Intel system routines *csend* and *crecv*. In other words, when a node sends a message to another node, the message goes directly to the receiving node without disturbing any of the other nodes. As a result of the DCM, communication times between any two nodes are theoretically uniform. In addition, each node can access both the host file system and the iPSC/2 Concurrent File System, and since a concurrent file is distributed over several disk drives, different nodes can access the same file simultaneously.

(15)

Table 1. iPSC/2 Specifications

SRM CPU	Intel 80386, 32-bit
Numeric Coprocessor	Intel 80387
Clock	16 MHz
Operating System	Host: AT&T UNIX, Version V Node: NX/2
Hard Disk	140 MB
Memory	SRM: 8 MB Node: 4 MB

3.2 SPECTRUM

The SPECTRUM (25) testbed is a system for designing and evaluating parallel simulation protocols on a common environment. SPECTRUM is composed of four modules (Figure 6) (12): an application program, a process manager, a node manager, and the protocol module (filter). The process manager (Appendix A) provides the interface to the user-supplied application program, and includes support for clock advancement, message management, and initialization. The node manager (Appendix B) provides the interface between the process manager and the parallel computer, and provides functions for memory management and communications. The filter provides the modules to implement the simulation communication synchronization protocol, and is called directly by the process manager, and indirectly by the node manager when communication is necessary between LPs. Although using the filter module is entirely optional (the process and node managers can handle message communication without it), there would be a dramatic risk increase for simulation deadlock. Finally, in order for SPECTRUM to understand how the LPs are networked together, SPECTRUM references an application specific "arcs" file, containing connection information and line delay times.

3.3 Simulations

3.3.1 8 LP Carwash With Feedback. The 8 LP carwash simulation with feedback (Figure 7), is a simple deterministic queuing model composed of eight processes; three sources: LP0, LP1, LP2; four routers (washes): LP3, LP4, LP5, LP6; and a sink (exit): LP7. In general, cars are "created" by the source processes, sent to the wash processes, and then routed to the exit process, where the cars are either destroyed, or "rewashed" by routing them back (feedback) to selected source processes. The deterministic nature of the carwash simulation is necessary in order to verify the "correctness" of the simulation after modifications to any part of the carwash, or underlying code. Also, as can be seen from Figure 7, LP1 has only one input line (from itself), and is therefore considered a "free-runner", meaning it can create messages as fast as it can without waiting on any other LP.

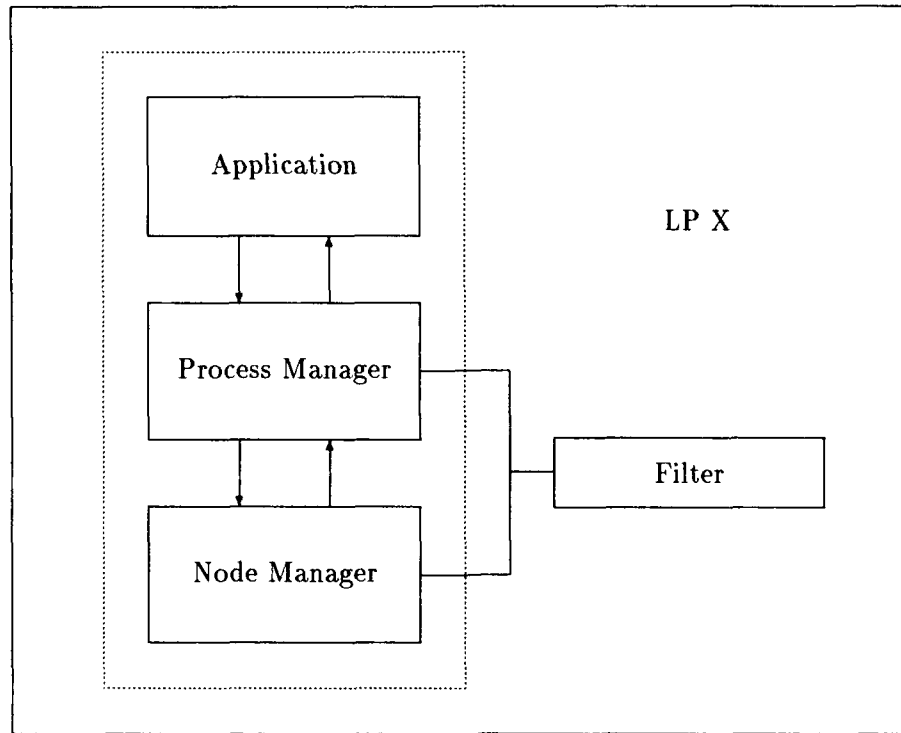


Figure 6. SPECTRUM Testbed Structure

The 8 LP carwash simulation used by AFIT is a modified version of the original software received from the University of Virginia (UVA). The main differences are:

- 1) The departure times of the cars are recomputed deterministically to guarantee no simultaneous arrivals at any downstream LP.
- 2) The car numbers are uniquely computed by each source LP for ease in tracking: LP0 cars start at 0, LP1 cars start at 1000, and LP2 cars start at 2000.
- 3) The simulation ends on a time, and not on a message, to smooth out the simulation termination.

3.3.1.1 Carwash Source Processes. The source processes create the cars, receive cars for rewash, and determine to which “wash” LP (according to the car number in Table 2), the cars will be sent. If this is a new car, as opposed to a rewash, the source processes set the new car arrival time with a time equal to the current simulation time

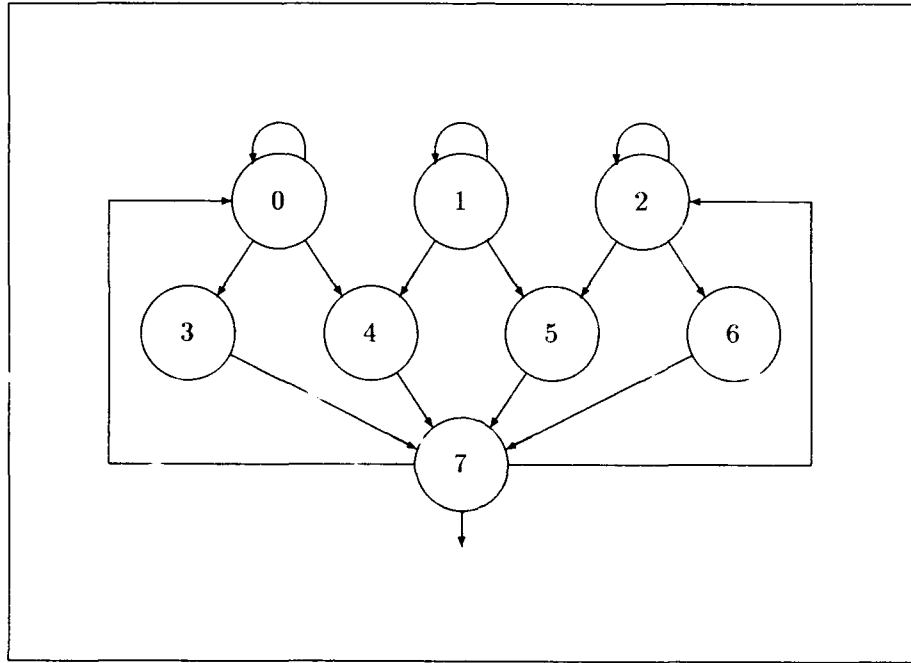


Figure 7. 8 LP Carwash Simulation With Feedback Schematic

Table 2. Source LP Destinations and Departure Factors (8 LPs)

<i>Source LP</i>	<i>Destination LP</i>	<i>Car Number</i>	<i>Departure Factor</i>
Source 0	Wash 1	Odd Numbers	1 Units
	Wash 2	Even Numbers	2 Units
Source 1	Wash 2	Odd Numbers	1 Units
	Wash 3	Even Numbers	2 Units
Source 2	Wash 3	Odd Numbers	1 Units
	Wash 4	Even Numbers	2 Units

plus the arrival interval time (Table 3). The source processes then determine the new car departure time by comparing the simulation time to the LP time. If the simulation time is *greater than or equal to* the LP time, the delay time (Table 3) is added to the *simulation* time. If the simulation time is *less than* the LP time, the delay time is added to the *LP* time. In either case, the departure time and the LP time are processed for determinism by modulo 2 arithmetic, and with the addition of a constant departure factor (Table 2), then reset. If the car was a rewash, the source processes determine the destination and departure time as above, but add the rewash delay time (Table 3), instead of the delay time, to the LP or simulation time.

Table 3. Source LP Arrival and Delay Times (8 LPs)

<i>Source LP</i>	<i>Destination LP</i>	<i>Arrival Interval</i>	<i>Delay Time</i>	<i>Rewash Delay Time</i>
Source 0	Wash 1	4 Units	5 Units	10 Units
	Wash 2	4 Units	8 Units	10 Units
Source 1	Wash 2	5 Units	4 Units	N/A
	Wash 3	5 Units	8 Units	N/A
Source 2	Wash 3	8 Units	9 Units	14 Units
	Wash 4	8 Units	2 Units	14 Units

3.3.1.2 Carwash Wash Processes. The wash processes add the wash delay time for each car according to Table 4, then routes the car to the exit process. The wash processes compute the car's departure time by comparing the simulation time to the LP time. If the simulation time is *greater than or equal to* the LP time, the wash delay time is added to the *simulation* time. If the simulation time is *less than* the LP time, the wash delay time is added to the *LP* time. In either case, the departure time and the LP time are reset for determinism by modulo 4 arithmetic, and with the addition of a constant wash departure factor (Table 4).

3.3.1.3 Carwash Exit Process. The exit process either "destroys" the cars, or "rewashes" the cars by deterministically routing selected cars back to one of two source processes according to Table 5. If the car is going to be rewash, its departure time is computed by comparing the simulation time to the LP time. If the simulation time

Table 4. Wash LP Delay Times and Departure Factors (8 LPs)

<i>Source LP</i>	<i>Destination LP</i>	<i>Wash Delay Time</i>	<i>Wash Departure Factor</i>
Wash 1	Exit	2 Units	4 Units
Wash 2	Exit	4 Units	5 Units
Wash 3	Exit	5 Units	6 Units
Wash 4	Exit	3 Units	7 Units

is *greater than or equal to* the LP time, the exit delay time (Table 5) is added to the *simulation* time. If the simulation time is *less than* the LP time, the exit delay time is added to the *LP* time. In either case, the departure time and the LP time are processed, and reset, for determinism by modulo 2 arithmetic so that all rewashes have even times. If the car is not going to be rewashed, the car is destroyed.

Table 5. Rewash Rates and Delay Times (7 & 8 LPs)

<i>Source LP</i>	<i>Destination LP</i>	<i>Rewash Car Rate</i>	<i>Exit Delay Time</i>
Exit	Source 0	3rd of Every 4th Car	6 Units
	Source 2	1st, 2nd, 4th of Every 4th Car	8 Units
	Destroyed	All Others	N/A

3.3.2 8 LP Carwash Without Feedback. The 8 LP carwash simulation without feedback (Figure 8), is also a simple deterministic queuing model composed of eight processes; three sources: LP0, LP1, LP2; four routers (washes): LP3, LP4, LP5, LP6; and a sink (exit): LP7. In general, cars are created by the source processes, sent to the wash processes, and then routed to the exit process, where the cars are *always* destroyed. Also, as can be seen from Figure 8, LP0, LP1, and LP2 have only one input line (from themselves), and are all therefore considered “free-runners”, meaning they all can create messages as fast as they can, without waiting on any other LPs. All of the other characteristics of the simulation are identical with the 8 LP feedback version, except that there are no rewashes.

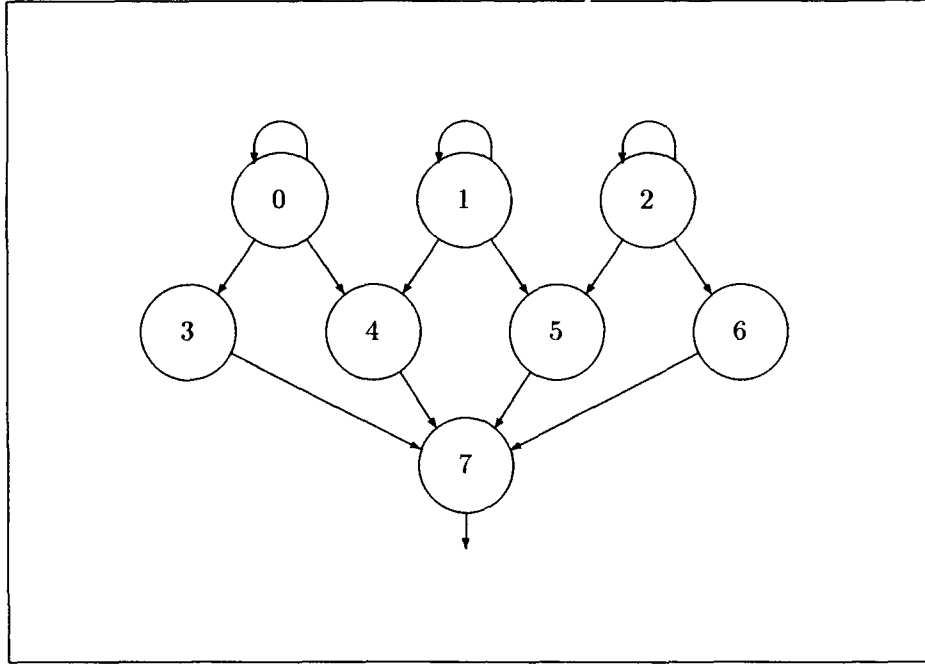


Figure 8. 8 LP Carwash Simulation Without Feedback Schematic

3.3.3 7 LP Carwash With Feedback. The 7 LP carwash simulation with feedback (Figure 9), is a simple deterministic queuing model composed of seven processes; three sources: LP0, LP1, LP2; three routers (washes): LP3, LP4, LP5; and a sink (exit): LP6. The 7 LP carwash simulation was based on the modified UVA 8 LP carwash model, and besides the reduction in the number of LPs, the AFIT modifications and the general LP logic is the same as in the 8 LP version. But, because of the different configuration, modifications in the various source and wash delay and factor times were required, and are listed in Tables 6, 7, and 8.

3.4 Simulation Filters

3.4.1 Chandy-Misra NULL Message Filter. AFIT uses an implementation of the NULL-message protocol proposed by Chandy and Misra in (7), and modified by UVA. In this protocol, an LP must block until there is at least one pending message on *all* of its incoming lines (a pending message blocking strategy). Also, if two LPs share more than one

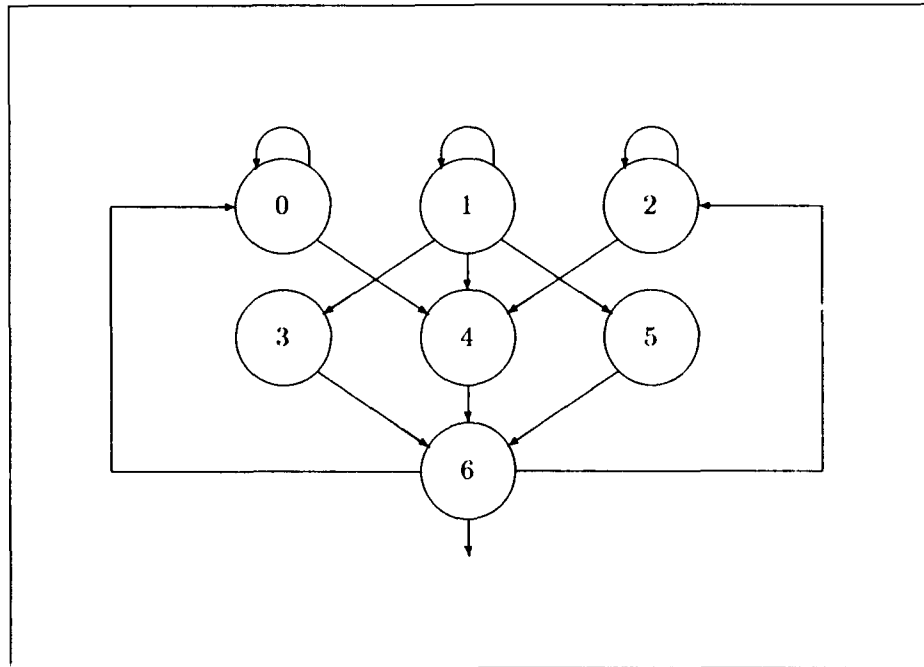


Figure 9. 7 LP Carwash Simulation With Feedback Schematic

Table 6. Source LP Destinations and Departure Factors (7 LPs)

<i>Source LP</i>	<i>Destination LP</i>	<i>Car Number</i>	<i>Departure Factor</i>
Source 0	Wash 2	All Numbers	2 Units
Source 1	Wash 1	Other Numbers	1 Units
	Wash 2	Mod 2 Numbers	2 Units
	Wash 3	Mod 3 Numbers	3 Units
Source 2	Wash 2	All Numbers	1 Units

Table 7. Source LP Arrival and Delay Times (7 LPs)

<i>Source LP</i>	<i>Destination LP</i>	<i>Arrival Interval</i>	<i>Delay Time</i>	<i>Rewash Delay Time</i>
Source 0	Wash 2	4 Units	5 Units	10 Units
Source 1	Wash 1	5 Units	4 Units	N/A
	Wash 2	5 Units	6 Units	N/A
	Wash 3	5 Units	8 Units	N/A
Source 2	Wash 2	8 Units	9 Units	14 Units

Table 8. Wash LP Delay Times and Departure Factors (7 LPs)

<i>Source LP</i>	<i>Destination LP</i>	<i>Wash Delay Time</i>	<i>Wash Departure Factor</i>
Wash 1	Exit	2 Units	4 Units
Wash 2	Exit	4 Units	5 Units
Wash 3	Exit	3 Units	6 Units

line, the filter will send only one NULL message to the receiving LP. Similarly, a receiving LP needs to wait for a message from each LP, not from each incoming line. Additionally, the filter (*null_mess_ftr*) incorporates a NULL message cancellation algorithm. If the message received was a non-NULL message, the filter will step through the event list deleting all NULL messages with times less than or equal to the non-NULL message. In this version, there are four times that NULL messages are sent:

1) During startup (*null_init_ftr*). For each LP, NULL messages are sent on each output line with the minimum delay time of each line encoded.

2) When an LP receives a NULL message (*null_get_ftr*) with *new* information (message time greater than the LP simulation time). The LP sends out NULL messages on all output lines encoded with the simulation time plus the line delay time.

3) When an LP receives a NULL message (*null_get_ftr*) with *old* information (message time less than or equal to the LP simulation time). The LP sends out NULL messages on all output lines, encoded with the time of the NULL message plus the line delay time, only if this updated time gives *new* information.

4) When a non-NULL or NULL message is sent to another LP (*null_post_ftr*). All other outgoing lines will receive a NULL message with the same time as the original outgoing message.

To aid in simulation termination, a termination filter (*null_term_ftr*) is defined. This filter accepts a real message, whose time is greater than or equal to a pre-defined simulation termination time, and then sends a real message to all outgoing lines from the terminating LP indicating that the LP is terminating. A sixth filter component, *null_time_ftr*, is also

defined, but it does not perform any function except to print an error message if a message arrives from the past.

3.4.2 Modified Chandy-Misra NULL Message Filter. A slight modification to the basic UVA Chandy-Misra filter, described above, is also used by AFIT. In this modification, an additional NULL message cancellation strategy is employed, so that prior to each time a NULL message is sent, a check is made to see if the NULL message is addressed back to the originating LP. If it is, there is no need to send the NULL message and the NULL message is discarded. All other filter components perform the same function as above.

3.4.3 Chandy-Misra NULL Message Filter with Safetimes. AFIT also uses a variation of the basic UVA Chandy-Misra filter implemented by Proicou (23). In this variation, before an LP can proceed, the LP must ensure that the time of the message to be processed is less than the minimum time of the last message received from *all* its input lines (safetime blocking strategy). The line "safetime" records this minimum time of the last message along an input or output line. The input safetimes are used to monitor the simulation progress, while the output safetimes are used to eliminate redundant NULL messages (messages where the message time is less than or equal to the destination output line safetime). This latter use of safetimes was added for the case when a NULL message is sent to more than one downstream LP. Unlike the general UVA Chandy-Misra protocol, an LP is not required to block until there is at least one pending message on all its incoming lines, but only by the constraint that the earliest next message to be processed, in the event list, must be less than or equal to the minimum safetime. But, like the modified UVA Chandy-Misra protocol, if a message received (`null_mess_ftr`) was a non-NULL message, the filter will step through the event list deleting all NULL messages with times less than or equal to the non-NULL message. In this variation, there are also four times that NULL messages are sent:

- 1) During startup (`null_init_ftr`). For each LP, NULL messages are sent on each output line with the minimum delay time of each line encoded.

2) When an LP receives a non-NULL or NULL message (`null_get_ftr`). If the output line safetime is less than the simulation time plus the line delay time, the filter updates the output line safetime by sending out NULL messages on all the LP's output lines, encoded with the simulation time plus the line delay time.

3) When an LP receives a NULL message (`null_get_ftr`) with *new* information. The LP sends out NULL messages, on all other output lines, encoded with the NULL message time plus the line delay time.

4) When a non-NULL or NULL message is sent to another LP (`null_post_ftr`). All other outgoing lines will receive a NULL message with the same time as the original outgoing message.

To aid in simulation termination, a termination filter (`null_term_ftr`) is defined. This filter accepts a real message, whose time is greater than or equal to a pre-defined simulation termination time, and then sends a real message to all outgoing lines from the current LP indicating that the LP is terminating. A sixth filter component, `null_time_ftr`, is also defined, but it does not perform any function except to print an error message if a message arrives from the past.

3.4.4 Shared Resource Algorithm for Distributed Simulation (SRADS). This protocol filter is also a UVA variant of the original SRADS proposed by Reynolds (24). In this version, called a buffered SRADS, an LP may send a message at any time, being restricted only by the size of the input buffer. SRADS uses two types of control messages: POLLS and ACKs. POLLS are used to query a downstream LP to see if it has advanced to a given time, while ACKs are sent from the downstream LP to the polling LP when the LP time is greater than or equal to the received POLL time. POLL messages are sent to all connected predecessor LPs at scheduled times (`srads_time_ftr`), determined by a preset polling frequency, when the message time is greater than or equal to the minimum next poll time of all the predecessor LPs, or when the last known message times of all the predecessor LPs is less than the message time. This second condition keeps the LPs "in-synch", meaning that the simulation time is up to the current message time. A caveat in the use of the SRADS filter is that although UVA preset the polling frequencies and offsets in the simulation arcs

file, and preset the delay time in the `srads_get_ftr`, no fine-tuning was performed in order to match AFIT's version of the carwash simulation.

ACK messages are sent at three times during the simulation:

- 1) When an LP receives a POLL message (`srads_mess_ftr`) and its time is less than or equal to the simulation time, an ACK is sent with the simulation time encoded.

- 2) When an LP receives an ACK (`srads_mess_ftr`) and there are ACKs which can be sent out to other LPs, ACKs are sent with the next time to acknowledge a POLL encoded.

- 3) When the simulation needs to advance its time (`srads_time_ftr`), ACKs are sent with the new time encoded .

The initialization filter component, `srads_init_ftr`, defines and initializes arrays used to track next poll times, last message times, and time-to-signal-poll times. The get event filter, `srads_get_ftr`, blocks (time blocking strategy) until a message is found, then returns the message to the application. In addition, to aid in terminating the simulation, a termination filter (`srads_term_ftr`) is defined. This filter accepts a real message, whose time is greater than or equal to a pre-defined simulation termination time, and then sends a real message to all outgoing lines from the current LP indicating that the LP is terminating. The post event filter is not used in the SRADS protocol.

IV. *Queuing Experiment Design, Preparation, and Analysis*

4.1 *Introduction*

This chapter describes in detail the experiment design and preparation needed to accomplish an empirical analysis of queuing models with various SPECTRUM filters. It relates experimental results and performance analysis around several simulation parameters which are used to define optimization guidelines for utilizing queuing models employing defined SPECTRUM filters. It presents the validation of the proposed guidelines using a 7 LP carwash model version. And finally, Hammell's results (11) are presented as a basis for a SPECTRUM comparison.

4.2 *Purpose*

The purpose of this experiment was to analyze and compare the four filters described in Chapter III, and to determine and define:

- 1) How run time, SPECTRUM filters, and simulation model configurations affect simulation wall time?
- 2) How simulation models can be partitioned and allocated across several nodes to optimize wall time and performance?

Additionally, in conjunction with this experiment and as a secondary goal, a comparison against Hammell's results, using a modified SPECTRUM was performed.

4.3 *Experiment Design*

The experiment design was centered around two types of simulation parameters: dynamic and static. Dynamic parameters are those parameters which were *varied* during a particular experiment, and include the LP partitioning and the simulation run time. Static parameters are those parameters which were held *constant* during a particular experiment, and include the simulation model configuration, the simulation filter type, and SPECTRUM.

In addition, to ensure that any modifications to the simulation system did not cause any adverse effects to the standard simulation execution, a test output file was generated and verified against a baselined output file. All filters, except the SRADS filter, produced the same output prior to an experiment start, and were all verified against the baselined output file, and against Hammell's experimental output when applicable.

4.3.1 *Dynamic Simulation Parameters.*

4.3.1.1 *LP Partitioning.* The LP partitioning method for the 2 and 4 node hypercube configurations was based upon Hammell's calculations (11). Hammell calculated that the number of possible 2 and 4 node combinations, for an 8 LP application, was based on:

$$C(n, r) = \frac{n!}{r!(n-r)!}$$

which defines the number of r combinations of n distinct objects, thus *8 choose 4* for 70 combinations, and *8 choose 2* for 2520 combinations. In addition, to eliminate duplicate combinations, Hammell made the assumption that the same combination of LPs on different nodes would give identical results. Although Intel states that message passing is performed with "uniform latency" (15), there has, in the past, been some discussions and indications to disprove this statement. These indications seem to favor placement of LPs on the nodes in a nearest-neighbor fashion, rather than in a logical or random placement. Since factoring in hypercube placement would dramatically increase the number of experimental LP configurations required while not adding to the overall thesis goal, it was decided to accept the claim as true, but with the caveat of reducing any impacts by always placing the LPs on the same nodes. As a result, in the 8 and 7 node cases, LPs 0 through 6 or 7 were placed on nodes 0 through 6 or 7 respectively; in the 4 node configurations, nodes 0 through 3 were always used; and for the 2 node configurations, nodes 0 and 1 were used. Using this caveat, there are now 105 possible partitionings on a 4 node hypercube, and 35 possible partitionings on a 2 node hypercube. For the 7 LP application, the possible partitionings are the same as in the 8 LP case, except without an eighth LP. To change

the LP partitioning during the experiment, it was simply a matter of allocating the LPs to the nodes as required during the *host* program execution.

The 8 LP carwash queuing model was partitioned into 1, 2, 4, and 8 node configurations, with multiple LPs resident on each node for the 1, 2, and 4 node configurations. Since the AFIT hypercube has 8 nodes, and this carwash model has 8 LPs, the partitioning for the 8 node configuration was a one-for-one match, with one LP resident per node. For the 1 node configuration, all 8 LPs were placed on a single node and was used as the sequential baseline for any speedup measurements. In the 2 and 4 node configurations, all possible combinations were tested, as described above, using 4 LPs resident per node during the 2 node tests, and 2 LPs resident per node during the 4 node tests.

The 7 LP carwash queuing model was also partitioned in the same manner, this time into 1, 2, 4, and 7 node configurations, with multiple LPs resident on each node for the 1, 2, and 4 node configurations. Since the AFIT hypercube has more than 7 nodes, the partitioning for the 7 node configuration was also a one-for-one match, with one LP resident on a node. The 1, 2, and 4 node configurations were partitioned as in the 8 LP case, minus the eighth LP, so during 2 node cases, one node contained 3 LPs, and the other 4 LPs. During the 4 node cases, three nodes contained 2 LPs, and one node contained 1 LP.

4.3.1.2 Simulation Run Time. Simulation run times are expressed in pseudo-seconds, and are used to simulate a length of time for executing a simulation. For the course of this experiment, run times of 500, 1000, 2000, 5000, and 10000 seconds were selected. Since the 1, 7, and 8 node carwash configurations were only partitioned one way, all five run times were used for a comparison study. But during the 2 and 4 node experiments, run times of 1000 and/or 2000 were selected to reduce the number of experimental runs to a manageable level, and to keep the wall times low enough to allow enough time for an exhaustive partitioning study.

In order to modify the run times during the experiments, it was necessary to edit the *globals.h* file, within SPECTRUM, and change the defined *MAXTIME* variable to the desired run time length, then compile and re-link the simulation code.

4.3.2 Static Simulation Parameters.

4.3.2.1 *Simulation Model Configuration.* Overall, there were four variations of the carwash model, all of which are described in Chapter III, that were used in this experiment:

- 1) The “standard” AFIT 8 LP version, with feedback (Figure 7 in Chapter III).
- 2) A modified 8 LP “no feedback” version (case 1), designed to destroy all cars from LP7, but with the arcs file *not* modified, so it still allowed non-real messages to be transferred to/from LPs 0 and 2, to/from LP7 (Figure 8 in Chapter III).
- 3) A modified 8 LP “no feedback” version (case 2), designed to destroy all cars from LP7, but with the arcs file modified, so it does *not* allow *any* messages to be sent to LPs 0 and 2, from LP7 (Figure 8 in Chapter III).
- 4) A 7 LP version, with feedback, used to validate the 8 LP carwash results (Figure 9 in Chapter III).

4.3.2.2 *Simulation Filter Type.* For each carwash configuration, all four of the filters described in Chapter III were used to collect timing and message count measurements (described below) for the various simulation run times. The following convention is used throughout the rest of this thesis to reference these four filters and associated carwash application systems:

- 1) Nullwash: Basic Chandy-Misra protocol, from UVA, with NULL messages.
- 2) Delwash: Modified Chandy-Misra protocol with NULL messages addressed back to the originating LP being deleted, and not sent.
- 3) Safewash: Chandy-Misra variant using safetimes on input and output arcs.
- 4) SRADwash: SRADS protocol described by Reynolds. Again, a caveat in the use of the SRADS filter is that although UVA preset the polling frequencies and offsets in the simulation arcs file, and preset the delay time in the `srads_get_fltr` filter module, no fine-tuning was performed to attempt an improved match to AFIT’s version of the carwash simulation, or to improve performance.

4.3.2.3 SPECTRUM. The version of SPECTRUM used during this experiment was a modified version of that used by AFIT in the past. In this version, SPECTRUM terminates an LP based upon a pre-defined maximum time, instead of a defined end-event. In doing so, an LP, after termination, does not need to keep processing messages until it reaches the end-event; instead the LP can discontinue processing messages as soon as it reaches the maximum time. This version of SPECTRUM was compared against the same experiments Hammell performed in order to determine any improvements in performance.

4.4 Experiment Preparation

To measure the effects of the different filters on simulation performance, both the carwash models and filters were modified to include performance measurement instrumentation to compute the wall times of each LP, and to count real and non-real message traffic into and out of each LP. The results of each type of measurement were downloaded to each respective LPs log file when the LP terminated, and printed. This instrumentation was in addition to the limited instrumentation already located within the node manager and host simulation loading programs.

4.4.1 Timing Instrumentation. Although the host program provides timing measurement instrumentation for the wall time of each LP, this time includes the time for the LP to initialize and terminate. To obtain a more accurate LP wall time measurement based solely on the LP's activity during the simulation run itself, and not including initialization and termination time, instrumentation was added to the carwash application programs using the Intel *mclock* routine (1), which returns a node clock time in milliseconds. This routine was called just after each LP's initialization was complete (*lp_init* module) to obtain a starting wall time of the simulation LP, and then again just prior to the LP's call to the *lp_terminate* module, to obtain the LP's stopping wall time. The two wall times were differenced and then divided by 1000 to obtain a total wall time in seconds.

To obtain further accuracy, each LP was loaded onto a node using the Intel *load* command, but with the addition of the *-H* option (1). This option will load an LP onto a node, but will halt the node and not execute the LP yet. Once *all* the LPs were loaded

onto the specified nodes, the Intel command *startcube* (1) was given to start all the nodes simultaneously. Using this method, after several test executions, it became apparent that it was not necessary to rerun each simulation experiment several times to obtain an average wall time, as the wall times collected were within acceptable tolerances. All times collected and reported were the maximum time gathered for the simulation execution over all of the LPs' wall times.

4.4.2 Message Counting Instrumentation. Although the node manager program can provide simple message count measurements for each LP, the usefulness of these counts is limited by the fact that they can only provide a *total* count of *all* messages, either sent to nodes via *csend* or received from nodes via *crecv*. Potential concerns in relying only on this information include:

- 1) No breakout of message type, either sent or received.
- 2) No count of messages not transmitted to other LPs (posted).
- 3) Messages may or may not get received (left waiting on the Intel message buffer).
- 4) Even if received, messages may never get processed by either the application or the filters, and just left on the LP queue when the simulation terminates.
- 5) Messages may get received and then just get deleted prior to processing (NULL messages) when they don't meet a certain event time criteria.

To improve the accuracy, as well as improving the capability of the overall system, more sophisticated message counting instrumentation, including message counts to determine either directly, or indirectly, the situations above, was implemented in all the carwash application programs and all four filter programs.

The additions to the application programs included:

- 1) Tracking *real* messages being sent.
- 2) Tracking *real* messages being posted to the originating LP (applicable only to the source LP's).
- 3) Tracking *real* messages removed from the simulation (applicable only to the exit LP's).

4) Logging all count results (application and filter) to the LP's message log when the LP terminates.

The additions to the filter programs included:

- 1) Tracking *NULL* messages that are deleted.
- 2) Tracking *NULL*, *POLL*, and *ACK* messages being sent.
- 3) Tracking *NULL*, *POLL*, and *ACK* messages being posted back to the originating LP (applicable only to the source LP's).
- 4) Tracking *NULL*, *POLL*, *ACK*, and *real* messages received by the originating LP and processed (applicable only to the source LP's).
- 5) Tracking *NULL*, *POLL*, *ACK*, and *real* messages received from another LP and processed.

4.5 Hammell's Results

As mentioned before, Hammell's experimental results were based upon an older version of SPECTRUM that terminated an LP based on an end-event, and additionally, only on the 8 LP carwash with feedback model. Because of this type of termination, and the fact that LP1 is considered a "free-runner", LP1 produced an inordinate amount of messages waiting for an end-event message until it terminated. Hammell reported simulation wall times, for runs of $t = 1000$, from 6.35 to 11.65 seconds for the 4 node configurations, and from 8.18 to 36.02 seconds for the 2 node configurations. Message counts for *just* LP1, for the same executions, were reported as 3922 messages received and 11769 messages sent (these were message counts only from the node manager program as described above).

In the 4 node configurations, Hammell reported that nodes with both LP1 and LP6 resident performed the best, followed by LP1 and LP3, LP1 and LP2, LP1 and LP0, LP1 and LP7, LP1 and LP4, and LP1 and LP5. These configurations all gradually increased in wall time, but still were relatively the same. The worst partitionings, with over a 2 second jump in wall time, were when both LP4 and LP5 were resident on the same node. This is because LP1, he conjectured, acting as a free-runner feeds all its messages to both LP4 and LP5. As a result, he concluded that a direct connection between performance

and number of messages received is established, and that the simulation performance is affected by how the *total* number of messages received are distributed across all the nodes. The more messages received on a node(s), as compared with the other nodes (what he calls “balance”), the worse the performance.

For the 2 node configurations, Hammell reported that the best performance was obtained when LP1 was resident with either LP4 *or* LP5, but not both. This was followed by configurations of LP1, LP4, and LP5 resident together, and then to the worst performance when LP4 and LP5 were resident together, while LP1 was on the other node. This final configuration caused an increase in wall time of over 8 seconds. From these results, Hammell concluded that it is not the distribution of messages across the nodes, as in the 4 node experiment, that causes poor performance, but just the total number of messages received over all the nodes. He also stated that this simpler conclusion was due to the fact that 2-node communication is less complex than 4-node communication.

4.6 Experimental Results

4.6.1 Introduction. Assuming, as a start, that Hammell’s conclusions are correct, then for a 1 node case, the simulation wall time should be based solely on the total number of messages received over all the nodes; and for the 7 or 8 node cases, the simulation wall time should be based on the distribution or balance of the messages received across all the nodes, and less on the total number of messages received.

As seen, Hammell used as a simulation communication/processing load indicator, the number of messages received at the nodes. As mentioned above, that doesn’t relate how many messages were actually processed or sent by the LPs, although the three are related. In this research, three other indicators were examined to aid in more accurately measuring and determining the processing and communication load of the simulation:

- 1) Number of messages that were actually “seen” or processed by each LP.
- 2) Number of messages that were sent by each LP, between nodes.
- 3) Number of messages that were posted by each LP.

A fourth factor, filter computational load, which includes algorithmic computation and

message setup, can have an impact on the wall time, but it is difficult to quantify and was inferred if all other factors were equal.

Based on this, for the 1 node configurations, the wall time should rely on all factors: the number of messages processed, the number of messages sent and posted, the number of messages received, and the computational load caused by the filter. On the opposite side, for the 8 and 7 node configurations, since the simulation is distributed with 1 LP per node, the overhead of processing messages and filter computational load should have a smaller effect towards the wall time, as compared to the communications overhead of sending, posting, and receiving messages across all the nodes. For the 2 and 4 node configurations, there should be varied impacts from all factors, but relying more on communication factors, and less on processing factors.

Detailed data, data discussions, and figures of each type of carwash configuration experiment are presented in the appendices. Appendix C discusses the 8 LP carwash with feedback, Appendix D discusses both the 8 LP carwash without feedback cases, and Appendix E looks at the 7 LP carwash validation results.

4.6.2 Effects of LP Partitioning. It was soon discovered that predicting the optimum 4 node configurations strictly by looking at an application schematic diagram was very difficult, although problem areas could be tagged for further investigation. Optimum 2 node configurations seemed to be much easier better to predict from only schematic diagrams since processing loads were the motivation factor, and there were fewer configurations to investigate. The reason for the 4 node difficulty is that 4 node configurations are more dependent on the communication loads than the 2 node cases, along with the added difference in LP message delay and sending time intervals. If all LPs sent messages at the same time intervals, the number of messages sent and received could more easily be predicted, and effects on downstream LPs could be more easily determined. But, as it currently stands in the carwash model, the time intervals vary considerably from LP to LP, making performance predictions much more difficult.

The most successful way of determining the optimum 4 node configurations, or at least configurations which were toward the top of the scale, was to use a message count

(sent, posted, and received) from each filter, in conjunction with the simulation schematic, to locate LPs which had multiple incoming paths and thus were involved in complex communication paths. These LPs were designated as "critical LPs", and once located, the concept of keeping these LPs separated on different nodes, while balancing the node communication loads, proved to overall produce the best results. Additionally, it was found that in cases where the number of LPs per node is not balanced, placing the critical LPs together on the node with the fewest LPs, along with balancing the node communication loads, also proved to produce good results. These results differed from Hammell's final 4 node conclusions, in that although the higher the total number of messages received overall indicates a higher communication overhead, it is actually the balance of messages sent *and* received from each critical LP, while keeping all the nodes in balance.

Determining the best 2 node cases seemed much simpler, in that it was not the communication loads or critical LP factors which influenced the LP configurations, but reducing the intra-node communications between *all* the LPs. In this manner, optimum configurations could be determined by strictly analyzing the simulation schematics. Again, this finding altered Hammell's conjecture that it was the total number of messages received on the nodes.

To verify the effects of communication and processing loads on both the 2 and 4 node configurations, an ideal simulation model was generated. In this model, each LP only communicates with one downstream LP in a tandem configuration (Figure 10). The experiment was run using the *nullwash* filter at $t = 10000$. The messages sent from each LP were measured as follows: LPs 0 and 1: 5001, LPs 2 and 3: 2500, LPs 4, 5, and 6: 1250. The results of each experiment are presented in Tables 9 and 10.

As seen in the 2 node case, when interacting LPs were placed together on the same node (second configuration), resulting in low internode communication loads, the wall time was the worst, whereas in the first configuration, when the internode communication load was the highest, the wall time was the best. This implies that for 2 nodes, processing loads take precedence over the communication load. In the 4 node case, the same implication can be made by looking at the first two configurations, but when the communication load is balanced, as in the third configuration, the wall time decreases even more. Thus, for

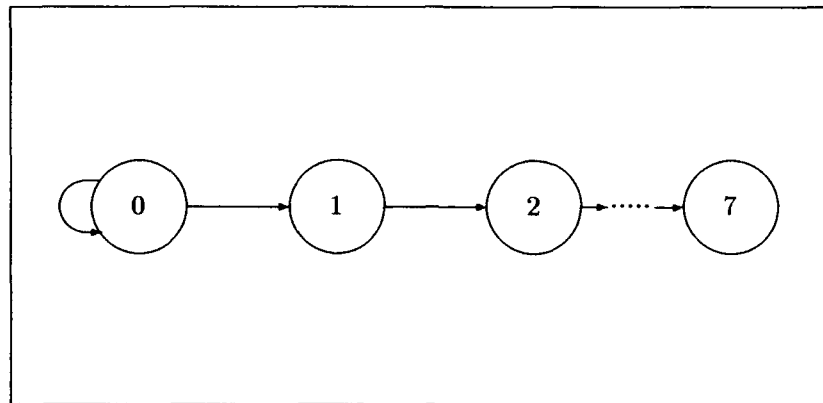


Figure 10. Tandem Simulation Schematic

the 4 node configurations, this implies that the communication load is ultimately more important than the processing load.

Table 9. 2 Node Tandem Model Results, $t = 10000$

<i>Node 0 LPs</i>	<i>Node 1 LPs</i>	<i>Time (secs)</i>
0,2,4,6	1,3,5,7	82.9
0,1,2,3	4,5,6,7	113.5

Table 10. 4 Node Tandem Model Results, $t = 10000$

<i>Node 0 LPs</i>	<i>Node 1 LPs</i>	<i>Node 2 LPs</i>	<i>Node 3 LPs</i>	<i>Time (secs)</i>
0,1	2,3	4,5	6,7	308.1
0,2	4,6	1,3	5,7	79.4
0,7	1,6	2,5	3,4	77.3

4.6.2.1 8 LP Carwash With Feedback. From the 8 LP carwash experiments, LPs 4, 5, and 7, were shown to have the greatest impact on the simulation wall time. The reason behind this is the way that the 8 LP carwash model is set up. Referring back to Figure 7 in Chapter III, LPs 4, 5, and 7 are each merge LPs with multiple incoming communication paths. This tends to form much more complex communication paths than the LP 0, 3, and 7 path, which is basically a single input, single output path. LP1 only

communicates, and thus sends all its messages to LPs 4 and 5, which are also getting messages from LPs 0 and 2. LPs 4 and 5, in turn, only communicate with LP7, which is also receiving messages from LPs 3 and 6. Thus, LPs 4, 5, and 7, can be considered the simulation's critical LPs, since each of the LPs have multiple incoming arcs.

For the 2 node configurations, results indicated that processing overhead within the nodes plays the major role in the wall time determination. The more intra-node connections on both nodes, the worst the simulation. The 4 node configurations imply the opposite of the 2 node results: reduce and balance the impact of the communication load. The more unbalanced the nodes sending messages, the worse the simulation. Also, it is strictly the critical LPs which determined simulation performance, unlike the 2 node configurations in which all LPs have some influence.

4.6.2.2 8 LP Carwash Without Feedback, Case 1. For the 2 and 4 node configurations, identical optimal and non-optimal configurations were obtained, as in the feedback version, and are explained from the fact that the only difference from the feedback version is the lack of real messages sent, those from LP7 to LPs 0 and 2. Additionally, since the ratio of real messages to NULL messages is small, removing the real messages sent from LP7 did not effect the overall simulation message counts and processing loads.

4.6.2.3 8 LP Carwash Without Feedback, Case 2. For both the 2 and 4 node configurations, wall time results also followed the same best and worst partitions that showed up in the feedback case. The identical partitions are explained from the fact that the only difference from the feedback version is the lack of two arcs, those from LP7 to LPs 0 and 2. To reiterate, referring back to Figure 8 in Chapter III, LPs 4, 5, and 7 are each merge LPs with multiple incoming communication paths. LPs 4, 5, and 7, again can be considered the simulation's critical LPs. Since the deletion of the feedback arcs did not effect the processing loads of the critical LPs, or the overall message production of the critical LPs, the same partitions repeated themselves again.

4.6.2.4 7 LP Carwash With Feedback. In this type of simulation, where there is an odd number of LPs, a different situation presented itself. Here, not all the nodes

will have the same number of LPs resident. For the 2 node configurations, one node will have 3 LPs resident and the other will have 4 LPs resident. For the 4 node configurations, three nodes will have 2 LPs resident and the other will have 1 LP resident. This type of situation causes the simulation to react differently as the various combinations of LPs are tried. To begin, in an ideal case when 1 LP is present per node, the simulations will be faster, when compared to the same simulation when multiple LPs are present per node. This is because each node can now devote itself entirely to only processing one LP, rather than processing multiple LPs. The same logic can apply when one node contains one less LP than the other node(s). Since the node has fewer LPs to process, placing the highly active or critical LP(s) on those nodes provided a better configuration, ultimately speeding up the simulation.

From the 2 and 4 node experiments, LPs 4 and 6 have the greatest impact on the simulation wall time. The reason behind this is the way that the 7 LP carwash model is set up. Referring back to Figure 9 in Chapter III, LPs 4 and 6 are both merge LPs with multiple incoming communication paths. This tends to form much more complex communication paths than the LP 1, 3, and 6 path, which is basically a single input, single output path. LP1 communicates, and thus sends all its messages to LPs 3, 4 and 5, while LPs 0 and 2 only communicate with LP4. LP4 in turn, only communicates with LP6, which is also receiving messages from LPs 3 and 5. LPs 4 and 6 can be considered the simulation's critical LPs.

Again, in the 2 and 4 node configurations, the use of the critical LPs, keeping the simulation balanced, and isolating intra-node communication paths, again proved to be the factor in determining optimal LP partitioning. Additionally, in cases of odd numbers of LPs, placing the highly active or critical LP(s) on those nodes with fewer LPs aided in providing a better configuration.

4.6.3 Effects of Simulation Filter Types.

4.6.3.1 8 LP Carwash With Feedback. Table 11 offers an overview of the simulation wall times, for each filter, in each node configuration. For the 2 and 4 node con-

figurations, wall times are for the best and worst configuration times obtained through the experiment. In general, each of the Chandy-Misra type filters performed the same during a simulation run time of 1000. Not until the simulation runs are increased, do the differences between the filters become more evident and comparisons are easily distinguished.

Table 11. Overall Node Wall Times - 8 LPs (secs), $t = 1000$

<i>Configuration</i>	<i>Nullwash</i>	<i>Delwash</i>	<i>Safewash</i>	<i>SRADwash</i>
1 Node	4.01	3.41	4.18	No Data
2 Nodes	1.70-2.75	1.44-2.36	2.06-4.01	7.25-17.58
4 Nodes	1.18-1.59	1.18-1.49	1.16-1.80	3.69-6.54
8 Nodes	1.13	1.09	0.89	2.26

4.6.3.2 8 LP Carwash, Without Feedback, Case 1. Generally, for the 8 LP carwash, with feedback case 1, the wall times for the filters (Table 12) for higher run times were faster than with feedback. But why were the wall times generally slower for lower run times? A possible explanation is that since the filters are not sending real messages from LP7, and the ratio of real to NULL messages is so small, it takes longer run times in order to make a measurable difference in message counts and times. Additionally, there is an indication that real messages, in the feedback mode, cause the greatest impact in simulation performance for the *nullwash* and *delwash* filters. An explanation is that both these filters, because of their blocking strategy, now wait less when NULL messages are arriving at a more or less constant interval. Before, when real messages were sent back to LPs 0 and 2, the filters would receive messages at a less constant interval and therefore the LPs will wait longer for pending messages.

In the *safewash* filter, the LPs only need to check the input arc time, so the impact is far less when a few real messages are not sent to LPs 0 and 2.

4.6.3.3 8 LP Carwash, Without Feedback, Case 2. As seen, the 8 LP carwash simulation could perform better without feedback at all (Table 13) than with feedback, depending on the filter and run time used. This implies that the presence of feedback loops could reduce the speedup attainable, at least in queuing type models. Why then do

Table 12. Overall Node Wall Times, Arcs Feedback - 8 LPs (secs), $t = 1000$

<i>Configuration</i>	<i>Nullwash</i>	<i>Delwash</i>	<i>Safewash</i>	<i>SRADwash</i>
1 Node	3.74	2.91	3.08	No Data
2 Nodes	2.01-3.45	1.57-2.53	1.84-2.96	11.34-20.00
4 Nodes	1.51-1.82	1.35-1.69	1.27-1.91	5.04-6.31
8 Nodes	1.25	1.21	1.06	2.27

the *nullwash* and *delwash* filters perform worse without feedback than with feedback? The answer, unfortunately, cannot be found by looking at the message counts, since they imply at least equal run times. Therefore it must be in the filter and/or application code. As a side note, Lee (18) also reported that the feedback version was better than the no feedback version, but attributed the slow down to an increased number of messages produced by the 3 source LPs all acting as free-runners, but as shown above this proved not always to be the case. Knowing that LPs 0, 1, and 2 send messages at different specific intervals (for determinism, see Chapter III), it becomes a matter of LP synchronization. In the no feedback version, LPs 0 and 2 are now also free-runners and are creating messages at will, and LPs 3 - 6 receive messages from LPs 0 - 2 at different times. In the *nullwash* and *delwash* filters, before an LP can proceed it must have a message pending on each input arc. With the source LPs free-running, the wash LPs must now wait more than in the feedback version, when LPs 0 and 2 are also waiting, allowing the wash LPs to synchronize and proceed more often. The *safewash* and *SRADwash* filters allow an LP to proceed based on the input arc times and not on pending messages, permitting them to proceed without waiting for messages. Therefore, in certain cases, feedback loops slow down the source LPs enough allowing the wash LPs to stay in-synch and to proceed without waiting as often. Another interesting item is that the *SRADwash* filter performed better, longer run times attained and closer wall times to the other filters, than in the feedback version.

4.6.3.4 7 LP Carwash With Feedback. Table 14 offers an overview of the simulation wall times, for each filter, in each node configuration, and in general validated the 8 LP results.

Table 13. Overall Node Wall Times, w/o Feedback - 8 LPs (secs), $t = 1000$

<i>Configuration</i>	<i>Nullwash</i>	<i>Delwash</i>	<i>Safewash</i>	<i>SRADwash</i>
1 Node	4.18	3.38	3.16	15.50
2 Nodes	1.46-2.77	1.16-2.42	1.43-2.66	6.88-8.01
4 Nodes	0.94-1.90	0.74-1.36	0.64-1.10	2.87-3.34
8 Nodes	0.82	0.69	0.36	1.07

Table 14. Overall Node Wall Times - 7 LPs (secs), $t = 1000$

<i>Configuration</i>	<i>Nullwash</i>	<i>Delwash</i>	<i>Safewash</i>	<i>SRADwash</i>
1 Node	3.71	2.91	3.82	No Data
2 Nodes	1.61-2.62	1.43-2.06	1.57-4.78	5.71-14.54
4 Nodes	1.23-1.94	1.15-1.65	1.05-2.22	2.38-6.29
7 Nodes	1.19	1.09	0.67	1.62

4.6.3.5 Simulation Speedup. As far as speedup was concerned, the *delwash* filter did have its benefits. By reducing the numbers of NULL messages posted to the source LPs, the simulation did show signs of speedup over the *nullwash* filter of around 1.2 in the best cases. The *safewash* filter, with its safetime blocking strategy, proved to be an excellent filter, beating out the other two Chandy-Misra filters in both the 8 and 7 node cases, and achieving almost linear speedup in both cases over the 1 node configuration. While the *SRADwash* filter continually indicated, with caveats, that it is not a good filter for queuing type simulations with feedback, it did vastly improve in cases in which there was no feedback. Also, in looking at the output, the *SRADwash* filter allows events from the past, within a time range, to still get processed, resulting in the output not always being in time ordered sequence. Tables 15 and 16 show these speedup relationships of the various filters for the 1 node versus 8 node case, with and without feedback. For example, in Table 15, the 8 node version of *nullwash* was 4.48 times faster than the 1 node version of *safewash*. The 7 node relationships showed very close comparisons to the 8 node versions, and as such are not included.

4.6.4 Effects of Model Configuration and Run Time. To begin with, there were several cases in which the simulation failed to complete a specified simulation run time. In

Table 15. 8 LP With Feedback Speedups - 1 Node vs. 8 Node

	1 Node			
8 Nodes	Nullwash	Delwash	Safewash	SRADwash
Nullwash	2.09	2.14	4.48	11.00
Delwash	2.11	2.16	4.51	11.21
Safewash	3.61	3.69	7.71	16.19
SRADwash	3.16	2.34	2.71	9.55

Table 16. 8 LP With w/o Feedback Speedups - 1 Node vs. 8 Node

	1 Node			
8 Nodes	Nullwash	Delwash	Safewash	SRADwash
Nullwash	4.29	3.89	1.96	18.90
Delwash	4.18	3.79	1.92	22.46
Safewash	12.95	11.75	5.95	43.05
SRADwash	2.67	2.39	2.01	14.48

all these cases, the simulations generally produced too many messages, too fast, and tended to fill up the Intel message buffer very quickly, causing the hypercube to lock-up. This problem has been brought to the attention of Intel, but a solution has yet to be delivered.

4.6.4.1 8 LP Carwash With Feedback. Simulation wall times for 1 node tests, at least through $t = 10000$, were fairly consistent. During the $t = 500$ run time, *delwash* was the fastest, followed by *safewash*, *nullwash* and *SRADwash*, respectively. Starting around $t = 1000$, *nullwash* begins to outperform *safewash*, and finally, starting around $t = 5000$, *nullwash* begins to edge out *delwash* for the top spot. Both *safewash* and *SRADwash* failed to complete the 1 node timing tests for all run times. In this case, the 1 node configurations seem to rely simply on both the total number of messages received and on the total number of messages sent. But also, more importantly and expected, the complexity of the internal code algorithms appears to adversely effect the simulation wall time in the case of *safewash* and *SRADwash*.

For the 8 node experiments, the simulation wall times, through $t = 10000$, were very consistent, with *safewash* executing the fastest, always followed by *delwash*, *nullwash*

and *SRADwash* respectively. Only *SRADwash* failed to complete the 8 node timing tests for all run times. For 8 node configurations, the computational load seems to totally counteract any communication overhead, unless the communication load is dramatically higher (*SRADwash*). This also conforms to intuition that any complex processing algorithm will work better on 8 nodes than on 1 node, despite the number of messages passed, since the computational load is now spread over more nodes.

4.6.4.2 8 LP Carwash Without Feedback, Case 1. Wall times for the 1 node case, at the start, showed the *safewash* filter performing the best, then as the run time is increased, the *delwash* filter becomes the best. In all cases the *nullwash* and *SRADwash* filter were always the third and fourth fastest respectively, while the *delwash* and *SRADwash* failed to complete all the run time tests. These results are different than in the same configuration with feedback, in that the *nullwash* and *delwash* filters were then always competing as the best filter.

For the 8 node case, the wall times for the *nullwash*, *delwash*, and *SRADwash* filter, were faster when the run time was greater than 1000, as compared to the feedback version, while the *safewash* filter showed virtually no change. But unlike the 1 node case, the 8 node case filter wall times compared the same against the feedback case, with *safewash* generally executing the fastest, followed by *delwash*, *nullwash*, and *SRADwash*. Again, *delwash* and *SRADwash* failed to complete the tests.

Generally, the wall times, for higher run times were faster than with feedback, and for all intents and purposes, unless run times are extremely long, the Chandy-Misra filters performed the same. The speedup encountered is attributable to three things:

- 1) No real messages, and corresponding NULL messages, are sent to LPs 0 and 2.
- 2) Reduction in the number of real messages "rewashed" and sent from LPs 0 and 2.
- 3) Reduction in, and now less complex, application code to process real messages, since no real messages are sent from LP7.

But why were the wall times generally slower for lower run times? A possible explanation is that since the filters are not sending real messages from LP7, and the ratio of

real to NULL messages is so small, it takes longer run times in order to make a measurable difference in message counts and ratios.

4.6.4.3 8 LP Carwash Without Feedback, Case 2. During the 1 node tests, up through $t = 10000$, only one general difference resulted over the feedback tests. Here the *safewash* filter, now with four less arcs to track (2 output arcs from LP7 and 1 input arc each into LP0 and 2) and process, showed to be the fastest, followed by *delwash*, *nullwash*, and *SRADwash*, which was the only filter not to complete the test. Compared to the feedback version, both the *safewash* and *SRADwash* filters always outperformed the feedback version, while the *nullwash* and *delwash* filters always performed worse when the run time was greater than 1000.

During the 8 node tests, the *safewash* filter again performed the best, followed by the *delwash*, *nullwash*, and *SRADwash* filters, respectively. This time, compared to the feedback version, both the *safewash* and *SRADwash* filters were better. The *nullwash* and *delwash* filters performed better without feedback up to $t = 5000$, at which point they slowed down and became worse. The *SRADwash* filter failed to complete all the tests, but attained higher run times than with the feedback version.

In this case, when there was no feedback at all, and the simulation was using a pending message blocking strategy (*nullwash* and *delwash*), the simulation continually got out-of-synch and slowed down below that of the feedback version. When the simulation used a safetime blocking strategy (*safewash* and *SRADwash*), the performance improved over the feedback version.

4.6.4.4 7 LP Carwash With Feedback. The same conclusions reached in the 8 LP configuration were also reflected here. The 1 and 7 node configurations relied on the total number of messages posted and received, the total number of messages sent, and filter algorithm effects. But, in some cases, where the number of messages sent were so overwhelming *SRADwash*, the numbers of messages posted and received, or the effects of filter algorithms were inconsequential. Also, although the complexity of the internal filter algorithms appears to adversely effect the simulation wall time, in one case (*delwash*) with

the reduction in the number of LPs and higher run times, it appears that the algorithm's effect was cut slightly, allowing any benefits not to be canceled.

Simulation wall times with the four filters, with 1 node, at least through $t = 10000$, were very consistent. The *delwash* filter was the fastest, followed by the *nullwash*, *safewash* and *SRADwash* filters, respectively. Both *safewash* and *SRADwash* failed to complete the 1 node timing tests for all run times.

Simulation wall times, for 7 nodes, with the four filters, were consistent with *safewash* always executing the fastest, and *SRADwash* always executing the slowest. The *delwash* and *nullwash* filters reversed themselves when the run time was greater than 2000, with *delwash* performing the best at the lower run times, and *nullwash* performing the best during the longer run times. Only *SRADwash* failed to complete the 7 node timing tests for all run times.

4.6.5 Effects of SPECTRUM. The new SPECTRUM did indeed improve the performance of the carwash simulation. And although LP1 is still considered a free-runner, it does not produce the abundance of messages it once did under the old SPECTRUM, and therefore does not effect the overall simulation wall time or LP partitionings as much as it did in the past. Also, by using the new SPECTRUM, there was a not quite so clear delineation, especially in the 2 node case, between the various LP combinations effects on the wall time, as there was in Hammell's work, but certain LP combinations did emerge as being better as others.

In comparing this experiment and Hammell's results above, the item that stands out the most is that the number of messages produced and sent, especially LP1, is considerably reduced. An LP can now terminate as soon as it has reached the end time instead of producing and sending messages until it receives the end-event message. As a comparison to Hammell's LP1, in this thesis at the same $t = 1000$ run time and using the *nullwash* filter, there were only 609 messages sent and 201 messages received (a decrease of over 1900%). In fact the *total* number of messages sent for all the LPs was only 2879, and the total received was only 2435. Using the same example above, in the 2 node configurations, wall times in this experiment were produced ranging from 1.70 to 2.75 seconds (a 13 time

decrease in the worst case), and for the 4 node configurations, wall times were produced ranging from 1.18 to 1.59 seconds (a 7 time decrease over the worst case).

4.7 Summary

The results of the experiments, in most cases, were not so unexpected. The new SPECTRUM did improve the performance of the carwash simulation dramatically and Hammell's results were validated overall. Although the optimal configurations were different, due to the improved SPECTRUM and to LP1's altered performance, balancing the simulation's communication or processing load was still the key factor.

It was seen that the four filters did not perform identically. The *nullwash* and *delwash* filters performed similarly since the only difference was the check for, and deletion, of NULL messages addressed back to the originator. And, generally, the *delwash* filter performed slightly better than its counterpart because of the reduction in NULL message transmissions. The *safewash* filter was overall the better of all the filters, in the 7 or 8 node case, based in part by its different blocking strategy. But it usually performed worse when multiple LPs were present on each node, apparently due to the more complex filter algorithm. The *SRADwash* filter indicated that it either was not a very good protocol for queuing type simulations, or that it needs quite a bit of work to tune it, and make it competitive with the other filters.

In determining a simulation's optimal partitioning, it appears that simulations, as part of their internal makeup (intentional or unintentional), have LPs which will always have a high communication load, both in messages received and messages sent. These LPs are critical and necessary in the determination of the optimal partitionings, especially in the 4 node cases where communication load is of prime importance. In the 2 node cases, where processing load outweighed communication load, reducing intra-node LP connections and isolating LPs from each other showed to be the most important.

In addition, the use of message counts involving the messages "seen" or processed by an LP didn't seem to have any benefits, nor did using the number of NULL messages deleted by the filters. A problem area that was not measured, and might be difficult to do,

V. Summary and Conclusions

5.1 Summary

Several algorithms for distributed conservative discrete-event simulations were described and tested during the course of this thesis. The *nullwash* filter uses the basic Chandy-Misra logic in that NULL messages are sent out to all other output arcs when a message is sent, and an LP will block processing until all input arcs have a message pending. A modified version (*delwash*) of the above filter was also tested. The difference is that, in this version, NULL messages addressed back to the originator will not be sent. The third filter, *safewash*, although very similar to those above, uses the input arc times (safetimes) to determine whether to block processing or not, and will not send a NULL message if the time of the output arc to the message destination is greater than or equal to the NULL message time. And finally, a completely different type of filter was also tried in the case of the *SRADwash* filter. This type of filter uses POLL and ACK control messages, instead of NULLs, to synchronize the simulation, and instead of sending control messages forward, looks backwards to see if it is alright to proceed. The *SRADwash* filter also uses arc times in order to verify if the simulation can proceed, or block processing.

In addition to testing three new filters with the "standard" 8 LP carwash simulation, two other new variations of the carwash simulation were also tested, with all the filters, to gather additional simulation performance data. These new versions were:

- 1) An 8 LP carwash without feedback loops from LP7 to LPs 0 and 2.
- 2) An 8 LP carwash with feedback loops from LP7 to LPs 0 and 2, but not allowing real messages to be entered onto the loops.

And finally, an improved version of SPECTRUM, designed to terminate an LP based on a specified end-time rather than an end-event, was also tested in conjunction with all the filters and simulations. Results obtained using the new SPECTRUM were quite encouraging, and successfully improved the performance of the carwash simulation by a large margin. Along with this, Hammell's results were validated to a point overall, although the optimum 2 and 4 node configurations were different due to the improved SPECTRUM.

is the filter algorithm processing effects. In several cases all other factors were the same, and this effect had to be conjectured as the reason for poor performance, especially during the 1 node cases where this factor is the most of concern.

In general, balancing the simulation's communication load on 4 node configurations, and reducing the number of intra-node connections in 2 node configurations, produced the optimal LP combinations.

It was good to see that all four filters did not perform identically, so that comparisons could be made. The *nullwash* and *delwash* filters performed and acted the same since the only difference was the check for, and deletion, of NULL messages addressed back to the originator. Generally, the *delwash* filter performed slightly better than its counterpart because of the reduction in NULL message transmissions. The *safewash* filter was generally the better of all the filters when executed on 8 nodes. When multiple LPs on the nodes were used, the *safewash* filter performed equal to or below that of the *nullwash* and *delwash* filters. This is conjectured to be caused by the increased processing that the *safewash* filter must perform, which doesn't get reduced to a minimum level until executed on 8 nodes. The *SRADwash* filter was a genuine disappointment. Although the polling frequencies used came from UVA and were supposedly correct, no additional effort was spent in fine tuning the filter for AFIT. Generally, it always performed far worse than the other filters, and produced out-of-order data. This should be alleviated, by what degree is unknown, by modifying the polling frequencies and the "get filter" delay time to more match the simulation model's message interval times.

In an effort to validate experimental results and guidelines, a 7 LP carwash model was developed. This simulation was designed to provide a slightly different model configuration, but still retain some of the 8 LP model characteristics (feedback, merge nodes, free-running LPs, etc.). In all cases, the 7 LP simulation verified the guidelines discovered by the 8 LP simulation. In fact, the 7 LP simulation brought to light the idea of placing the critical LPs onto a node with the fewest LPs to aid in improving performance.

5.2 Filter Assignment Guidelines

Although all the filters, in general, gave similar partitioning results, it was seen that in simulations with no, or limited feedback, the *safewash* and *SRADwash* filters performed better than the *nullwash* and *delwash* filters. This is attributed to the use of a safetime blocking strategy, rather than a pending message blocking strategy, allowing the LPs to

proceed in a more smooth manner, and not as often get out-of-synch and block as the pending message strategy appears to accomplish.

5.3 LP Assignment Guidelines

Empirical experiments have provided some guidelines to use when assigning LPs to nodes using simple queuing models. The experiments have shown that in the 2 node cases, the allocation problem is based on the processing load of the two nodes, while in the 4 node cases, the partitioning is based on the communication loads. Even though the guidelines apply to the 2 and 4 node cases, the overall primary guideline, at least for 8 nodes, is to always use the maximum number of nodes available. This way the LPs can be allocated closer to the optimum one-to-one ratio to the nodes. In the discussions that follow, Figures 11 and/or 12 will be used to illustrate the guidelines for clarity.

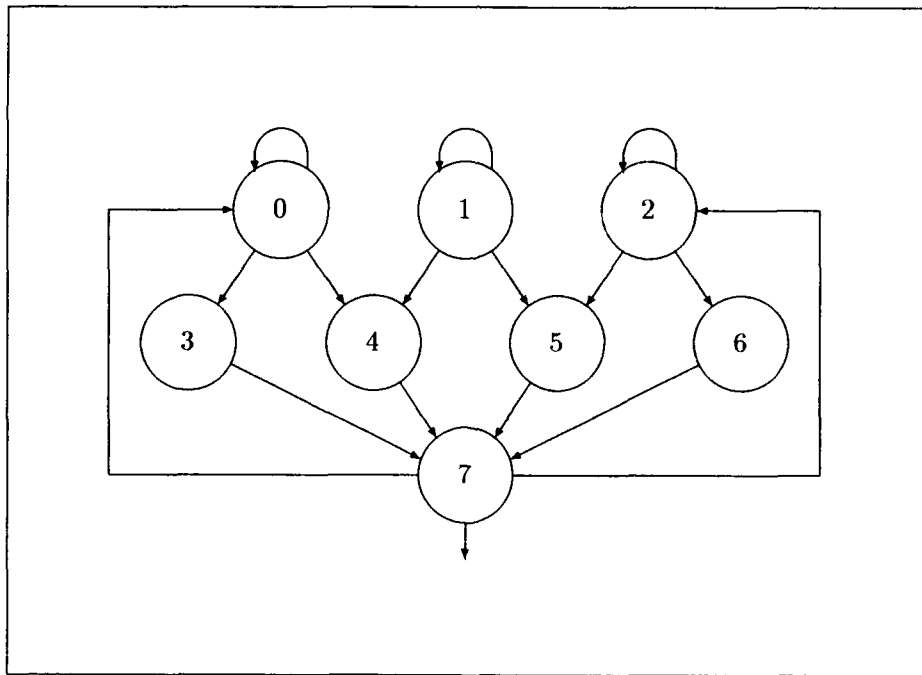


Figure 11. 8 LP Carwash Simulation With Feedback Schematic

5.3.1 4 Node Assignments. As mentioned in Chapter IV, predicting the optimum 4 node configurations by only looking at an application schematic diagram is very difficult,

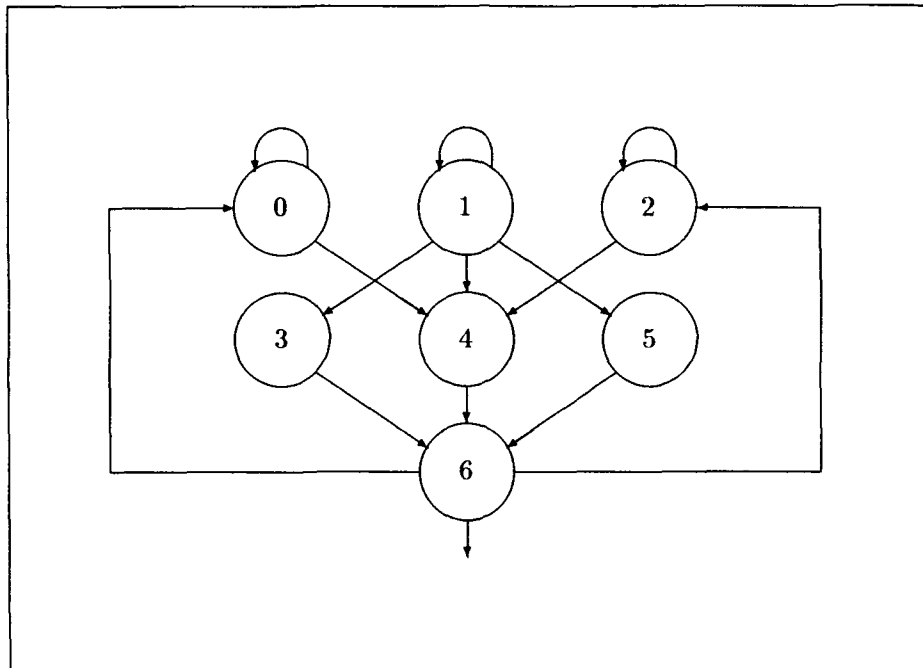


Figure 12. 7 LP Carwash Simulation With Feedback Schematic

but it is a start. Begin by looking for LPs (critical LPs) which have multiple incoming arcs (highly merged). These LPs will tend to have a high communication load, and with the blocking algorithms imposed, will tend to slow the simulation much more than a single-input/single-output LP. Next, using message count instrumentation, execute runs to obtain message counts at all the LPs in the simulation. By looking for peaks of messages sent and received at the LPs, but not posted, verify or modify the initial prediction of the critical LPs. The prevailing guideline in assigning LPs to 4 nodes is balancing LP communication loads across *all* nodes: the more balanced the message loads of the LPs on each node, the better the performance. Following this guideline, along with applying the following rules will give one of the best configurations.

1) Separate the critical LPs as much as possible onto different nodes. In the 8 LP case with LPs 4, 5, and 7 acting as the critical LPs, a good configuration could be 0,6/1,5/2,4/3,7, while a bad configuration could be 0,6/1,2/4,5/3,7. As seen, just switching

two LPs (4 and 5), so that they are together on the same node, can result in a bad configuration. Therefore care must be taken in LP assignments.

2) Place critical LPs on nodes with the fewest number of LPs, starting with the critical LP with the highest communication load. In the 7 LP case, LPs 4 and 6 are the critical nodes, with LP4 having the highest number of messages sent and received. Here, the best configuration could be 0,2/1,5/3,6/4, whereas the worst combination could be 0,4/1/2,3/5,6. In this case, because of the nodes message imbalance, and one node with only one LP, LPs 4 and 6 are located on different nodes but produce the worst combination.

5.3.2 2 Node Assignments. Unlike the 4 node guidelines, the 2 node guidelines are concerned with processing load rather than communication load. Because of this, the use of the message counts are very limited, and will not be used as often. To reiterate, when determining partitions, always consider both nodes, a switch in one LP on the wrong node can turn a good combination into a bad combination. The following guidelines are listed by priority, but also must be looked at in total.

1) Reduce LP feed-forward connections on the same node through partitioning, or by reducing the number of LPs on the node. In the 8 LP case, through partitioning, a good combination would be 0,1,6,7/2,3,4,5, whereas a bad combination is 0,2,6,7/1,3,4,5. For the 7 LP case, through LP reduction, a good combination would be 0,1,2,6/3,4,5, while a bad configuration is 0,1,3,4/2,5,6.

2) Totally isolate LPs involved in feedback on the same node. In the 8 node case, this would mean combinations with LPs 0, 2, and 7 on the same node, and for the 7 node case, LPs 0, 2, and 6 on the same node. In this case, feedback loops are formally defined as arcs which connect the exit, or sink, LP, with one or more of its' predecessor LPs, and also sends real and/or NULL messages towards the predecessor LPs. Thus, feedback LPs are those LPs at either end of the feedback arcs.

3) Reduce connections into critical LPs on the same node. For example, avoid placing LPs 1, 4, and 5, or LPs 4, 5, and 7 on the same node.

4) Increase the connections out of critical LPs on the same node. A good example would be to place LPs 0, 2, and 6, in the 7 LP carwash, on the same node.

5) If it is unavoidable to put LP connections on the same node, select the connections that have the lowest message traffic.

5.4 Hypercube Allocation

In an effort to prove or disprove, and document the results, whether the same LP combinations on different hypercube nodes cause an adverse effect to wall time, a small set of experiments were designed and performed. As stated in Chapter IV, Intel claims that *any* node-to-node message transfer is of uniform latency. As a start, the 8 LP carwash, without feedback, was allocated and executed on the hypercube in a nearest-neighbor fashion (Figure 13). In this manner, the placement is the closest match to the actual carwash simulation layout (LP1 connected to LPs 4 and 5, LP2 connected to LP6, etc.). The only mismatch, because of the hypercube structure, is that LP6 is not connected to LP7, but since this LP is a minor player in the simulation, the affects are minor. In addition, the 8 LP carwash, without feedback, was allocated and executed on the hypercube in a farthest-neighbor fashion. Here, the LPs were placed on nodes in a manner to gain a worst placement (LP1 connected to LPs 2 and 7, LP5 connected to LPs 0 and 4, etc.). As an additional experiment, a 4 node test was also performed following the same procedures as in the 8 node case. As seen in Figure 14, the nearest-neighbor case, again setup like the carwash schematic, has LP1 connected to LPs 4 and 5, all the wash LPs either connected or resident with LP7, etc. Also, a farthest-neighbor configuration was also setup as shown. Results from the experiments are shown in Table 17, and based from them, it appears that Intel's claim does not hold up, as the wall times between the two respective configurations were measurably different. Thus, a good starting point for an additional research project would be to investigate further just how much an effect different cube allocations influence the wall time.

5.5 Conclusions

In general, partitioning guidelines developed established a relationship between LP configuration and load balancing. The experiments showed that reducing and balancing the communication load by separating highly communicative LPs onto different nodes,

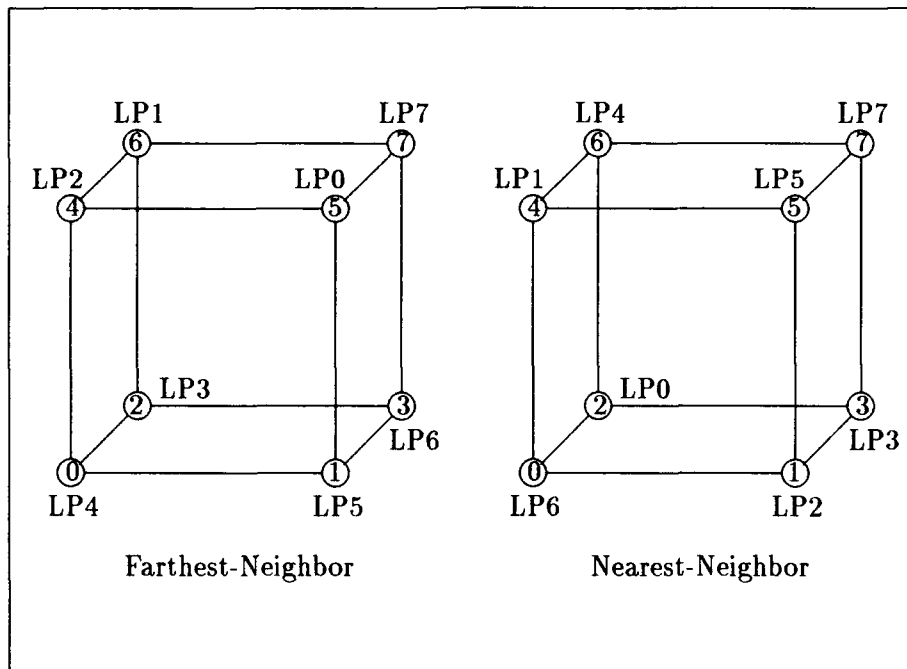


Figure 13. 8 Node Hypercube LP Placement

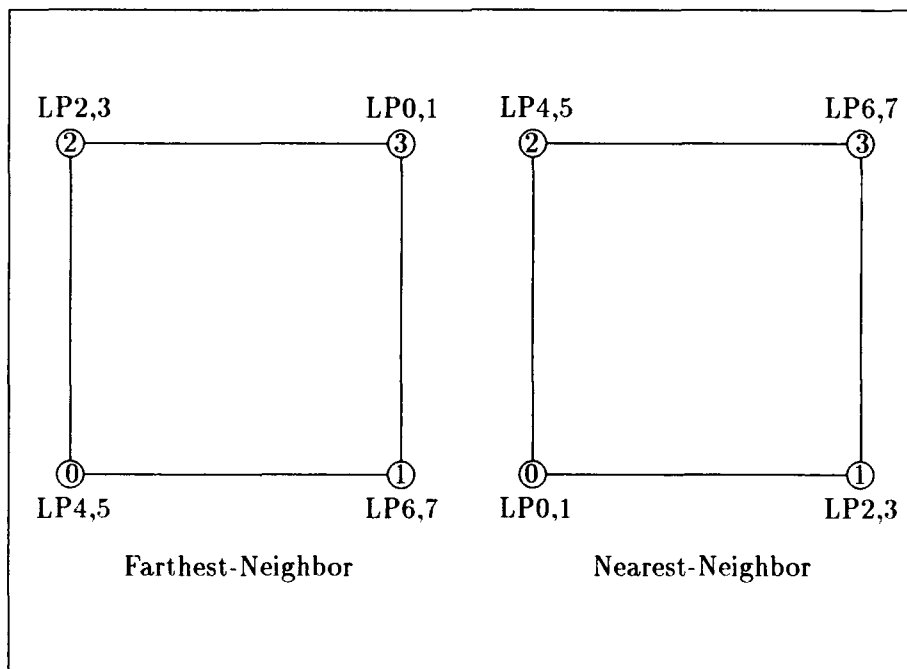


Figure 14. 4 Node Hypercube LP Placement

Table 17. Hypercube LP Placement Wall Times (secs), $t = 10000$

<i>Placement</i>	<i>Nullwash</i>	<i>Safewash</i>
8 Node Nearest	50.34	12.40
8 Node Farthest	65.72	14.00
4 Node Nearest	60.82	29.08
4 Node Farthest	63.22	30.16

or by locating the highly communicative processes on the same node with fewer LPs, provided the optimal 4 node configurations. In the 2 node configurations, reducing the number of intra-node process connections provided for the optimal configurations through the reduction of the processing load.

For the 4 node cases, locating the optimal configuration is begun by looking for LPs (critical LPs) which have multiple incoming arcs. Then by using message counts, isolating and verifying the initial critical LP prediction. The prevailing guideline in assigning LPs to 4 nodes is balancing LP communication loads across *all* the nodes: the more balanced the message loads of the LPs on each node, the better the performance. In an expansion of the above guideline, the easiest method to balance the communication load is to separate the critical LPs as much as possible onto different nodes. Or, in cases where the number of LPs per node is unbalanced, place the critical LPs on the node with the fewest number of LPs.

As mentioned above, guidelines for the 2 node configurations, are concerned with only the processing load. Because of this, the use of the message counts are very limited, and are not used as often. In these cases, the primary guideline is to reduce the LP feed-forward intra-node connections through partitioning, or by reducing the number of LPs on the same node.

This thesis also showed that in simulations with no, or limited feedback, the use of a safetime blocking strategy (i.e., *safewash*) was preferable for better performance, than a pending message blocking strategy (i.e., *nullwash* or *delwash*). The reason is that safetime filters allow the LPs to proceed in a smoother manner by not allowing the simulation to get out-of-synch and block as often.

Finally, to determine whether *any* node-to-node message transfers have a uniform latency, a small set of experiments was performed. And as suspected, Intel's claim did not hold up, since the wall times between a nearest-neighbor and farthest-neighbor configuration was markedly different.

5.6 *Recommendations for Further Research*

There are many additional areas to continue further research in empirical analysis of AFIT simulations. Probably the most beneficial would be the measurement and study of the filters and application code processing loads. This would help further explain, and possibly prove or disprove, some earlier conclusions based upon processing loads of individual LPs, and will show whether the processing loads are reduced enough, going from 1 to 8 nodes, to make a significant impact.

In addition, there are still several areas in which the simulation and/or filter code can be subtly changed and tested. It would be interesting to perform some of these changes and see if the same conclusions from this thesis still hold. These include:

- 1) Studying the effects of spin-loops in the application code (in conjunction with processing load measurements).

- 2) Increase the number of messages, in SPECTRUM, taken from Intel buffer at one time. And if a fix is ever received from Intel for the buffer overflow problem, seeing if this change causes any adverse effects to the simulations.

- 3) Modify the SRADS polling frequency, offset, and delay time, to see if a more accurate time ordered output could be obtained, and if the run time could be decreased.

- 4) Currently, the carwash simulation sends messages to downstream LPs in an evenly distributed manner, although at different delay times, meaning connected downstream LPs will each ultimately receive the same number of messages. Change the method of determining which downstream LPs messages are addressed into a more uneven manner. This could also be performed in conjunction with changing the carwash LP message delay intervals, to either be the same time, or at least in a more ordered manner.

5) Look into non-symmetric partitioning of simulations, i.e., unequal number of LPs per node.

6) Evaluate combining or merging LPs and measuring the resulting effects in performance.

7) Investigate whether using different synchronization protocols on different LPs, in the same simulation, is possible, and would it improve performance.

8) Use larger simulations and increase the number of LPs and nodes, beyond eight, to see if the same guidelines apply.

Finally, a more ambitious project would be to perform the same type of analysis, as in this thesis, on both the VHDL (3) and battlesim (2) simulations. The *safewash* filter has already been modified for use with the VHDL simulation, but the others would have to be changed in a similar way.

Appendix A. *Process Manager Module*

The process manager (*lp_man*) module is composed of four basic functions: LP initialization, message management, advancing the simulation time, and displaying message information.

A.1 *LP Initialization Function*

The initialization function is composed of three modules:

1) *lp_level_init*: called at simulation startup to initialize SPECTRUM. SPECTRUM is initialized through calls to *read_lp_info* and the node manager initialization routine, *node_level_init*, which startup the LPs on the nodes.

2) *lp_init*: called by each LP to build the filter table. The filter table determines which filter routines are called and used. If no filters are to be used, NULLs are used to indicate that filters are not present. In addition, if there is an "simulation initialization" filter, it is called at this time.

3) *read_lp_info*: builds the LP information arrays, used in the SPECTRUM initialization, by reading the application arcs file. This file describes the simulation network structure, and contains each LP's delay times and ids, number of input and output LPs, number of input and output lines, ids and delay times, and input LP polling frequencies and offsets.

A.2 *Message Manager Function*

The message manager function is composed of four modules:

1) *lp_get_event*: obtains the next message from the event list. If no "get event" filter is defined, the event list is checked for a current message using the node manager routine *node_receive_pending_messages*. If no message is present, the LP will block until a message is received using the node manager routine *node_block_til_message*. If there is a current message, it is removed and the event list pointer is advanced to the next message.

2) `lp_post_event`: sends a message to an LP. If a "post event" filter is defined, it is called at this point. In any case, if the message is for the current LP, then the message is "created" via the node manager routine `node_create_event` and posted to the LP by calling `lp_post_message`. If the message is to another LP, it is sent via the node manager routine `node_send_message` to the specified LP.

3) `lp_post_message`: posts a message to an LP. If a "post message" filter is defined, it is called at this point. In any case, the message is entered into the LP's event list through a call to `lp_nq_event`.

4) `lp_nq_event`: places messages in the LP event list in time ordered sequence.

5) `lp_terminate`: terminates the LP. A "terminate LP" filter must always be defined, and is called at this point. When control is returned after coming back from the filter, the `node_terminate` routine is called to gracefully shutdown the LP.

A.3 Advance Time Function

The advance time function is composed of one module, `lp_advance_time`, which advances the LP time to the new message time. If an "advance time" filter is defined, it is called at this point. In any case, the LP time is advanced to the message time.

A.4 Message Information Function

The message information function is composed of four modules:

1) `display_lp_info`: displays the LPs array information built by `lp_level_init`.

2) `errlog`: opens the error log file "errlog" for simulation error messages and diagnostic display of simulation arcs file.

3) `log`: opens a message log file for each LP (`logx`), for simulation *REPORT* and *DEBUG* messages.

4) `display_event_list`: displays all the messages in an LP's current event list.

Appendix B. *Node Manager Module*

The node manager (cube2) module is composed of three basic functions: node initialization and termination, communications management, and memory management.

B.1 Initialization and Termination Function

The initialization and termination functions are composed of three modules:

1) `node_levelinit`: called at the startup of each node. This module assigns the simulation processes to the nodes, and starts up the simulation.

2) `node_terminate`: determines if the cause of the termination is from a final message or if the final time has been reached, and sends out a `LAST_EVENT` or `LAST_TIME` message, respectively. In either case, the LP blocks until an `END_MSG` message is received, then calls the routine `shut_down` and returns to the calling function for termination by the application program.

3) `shut_down`: terminates the LP.

B.2 Communications Function

The communication function is composed of three modules:

1) `node_receive_pending_messages`: processes pending messages. This module checks incoming messages, via `crecv` calls, for an `END_MSG` or `EVENT_MSG` message stamp. If an `END_MSG` is received, it terminates the LP through a call to the routine `shut_down` and exits, else if an `EVENT_MSG` is received, it inserts the message into the event list via a call to `lp_post_message`.

2) `node_block_till_message`: block the LP until a message is received. This is accomplished by continuously calling `node_receive_pending_messages` to check for any messages.

3) `node_send_message`: sends messages to the specified LP via `csend` calls and then releases memory through `node_trash_event`. If the message is an `END_MSG`, it then calls `shut_down` to terminate the LP.

B.3 Memory Management Function

The memory management function is composed of two modules:

- 1) `node_create_event`: allocates memory space for a new message.
- 2) `node_trash_event`: deallocates event list memory space for future use.

Appendix C. 8 LP Carwash, With Feedback, Experimental Data

C.1 1 Node Configuration

As seen in Table 18 and Figure 15, simulation wall times with the four filters, at least through $t = 10000$, are fairly consistent. During the lower run time *delwash* is the fastest, followed by *safewash*, *nullwash* and *SRADwash*, respectively. Then starting around $t = 1000$, *nullwash* begins to outperform *safewash*, and finally, starting around $t = 5000$, *nullwash* begins to edge out *delwash* for the top spot. Both *safewash* and *SRADwash* failed to complete the 1 node timing tests for all run times.

Table 18. 1 Node Wall Times - 8 LPs (secs)

Run Time	Nullwash	Delwash	Safewash	SRADwash
500	1.93	1.43	1.65	5.83
1000	4.01	3.41	4.18	No Data
2000	8.35	7.91	11.65	No Data
5000	29.45	30.17	62.95	No Data
10000	91.75	95.10	No Data	No Data

Looking at Figures 16 and 18, for the *nullwash* and *delwash* filters, it appears from the graphs that the *delwash* filter should always perform better than the *nullwash* filter. The *nullwash* filter does send out slightly more messages than the *delwash* filter, but the biggest difference is in the number of messages posted and received. In Figures 17 and 19, it can be seen that all three source LPs posted and received significantly fewer messages in the *delwash* filter. Of course this is due to the *delwash* filter deleting NULL messages addressed back to the originator. This accounts for the slower *nullwash* times during the lower run times. But since the *delwash* filter has the added code to check for posted NULL messages, it appears that the code is also adding enough processing overhead to cancel-out the benefits gained by the deletion of unneeded NULL messages during higher run times. Also, as seen above, this processing overhead appears to slowly increase the difference in wall time, relative to the *nullwash* filter, as the run time increases.

The same argument can be applied to why the *safewash* filter performed worse than both the *nullwash* and *delwash* filters. Although, the *safewash* filter sent out more messages

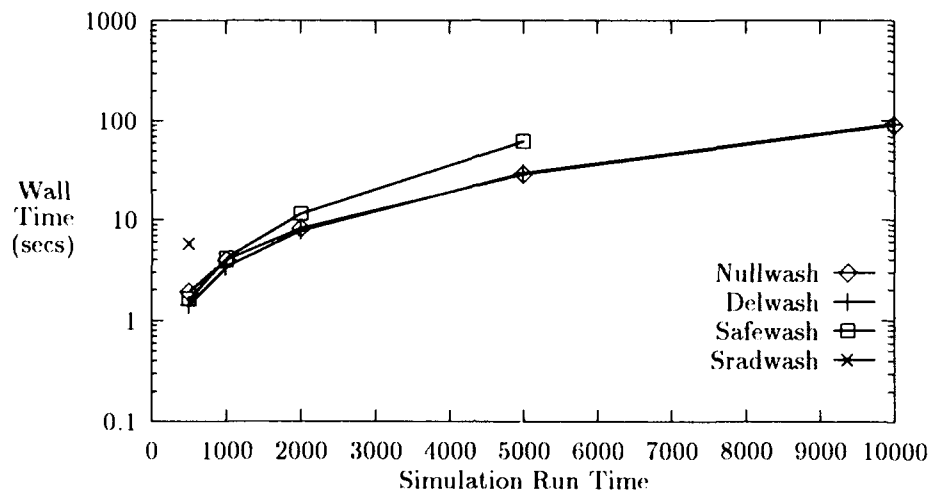


Figure 15. 1 Node Wall Times - 8 LPs (secs)

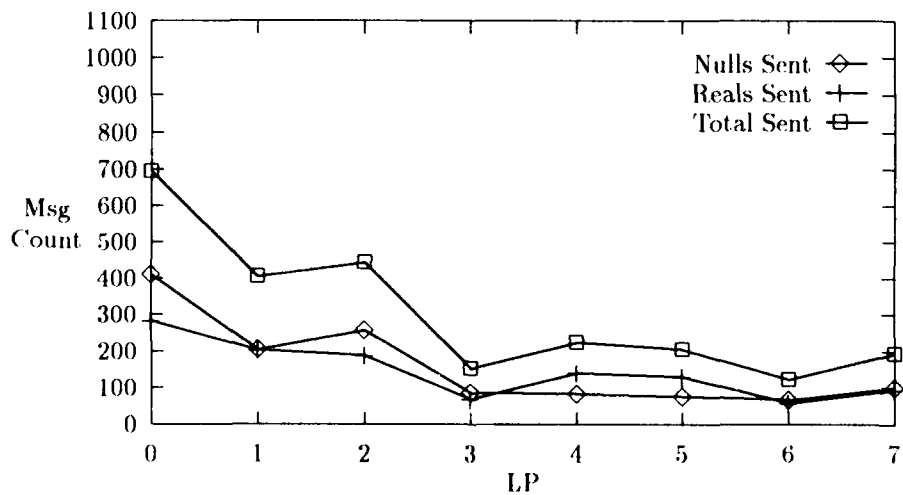


Figure 16. Nullwash Messages Sent - 8 LPs, $t = 1000$

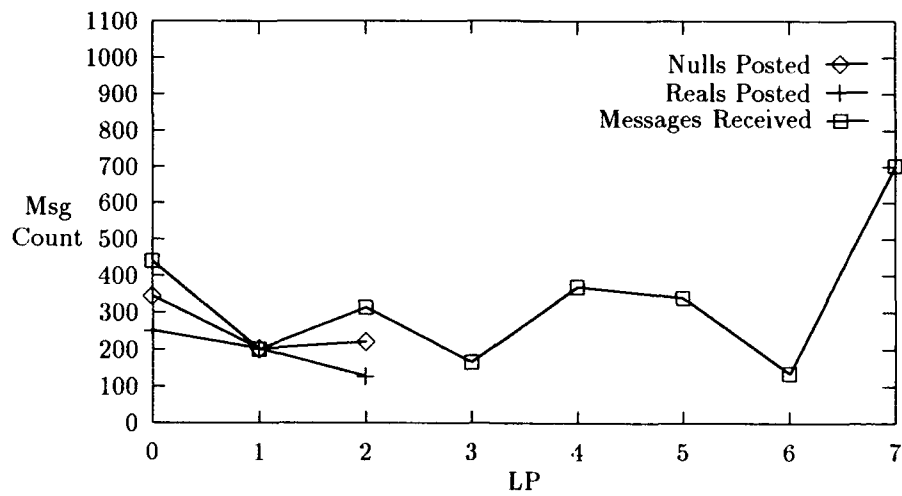


Figure 17. Nullwash Messages Posted and Received - 8 LPs, $t = 1000$

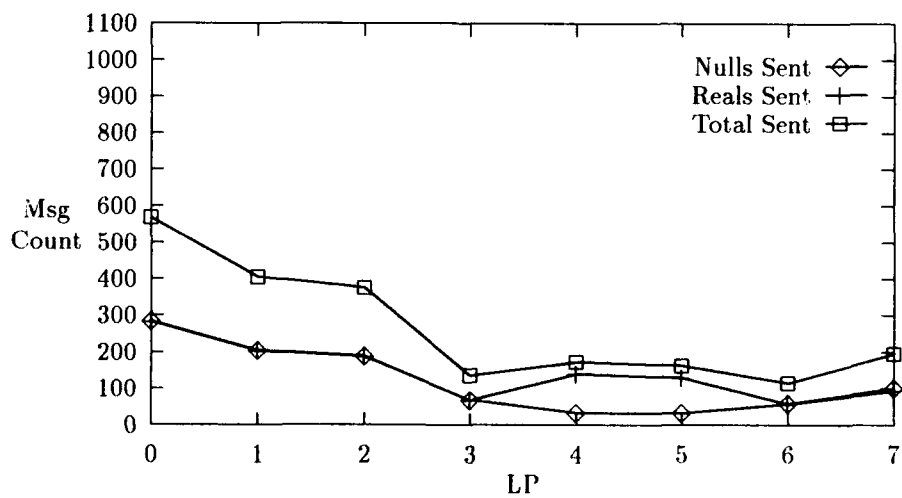


Figure 18. Delwash Messages Sent - 8 LPs, $t = 1000$

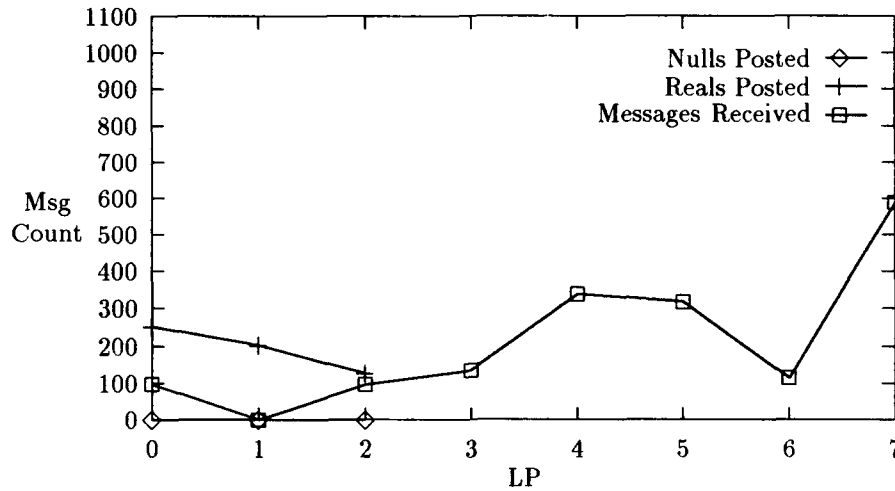


Figure 19. Delwash Messages Posted and Received - 8 LPs, $t = 1000$

(LP7 in Figure 20) than both the *nullwash* and *delwash* filters, it posted and received more messages (LPs 0 and 2 in Figure 21) than *delwash*, and fewer messages (LPs 1 and 7) than *nullwash*. Combine this with the fact that the *safewash* filter also has some added code to track and update channel times, add up to the wall time significantly increasing when executed on 1 node.

Looking at the number of sent messages (Figure 22), the *SRADwash* filter sent hundreds of more messages (LPs 4, 5, and 7) than all the other filters, and posted and received hundreds of fewer messages (Figure 23). This also indicates that the *SRADwash* filter, by its very nature of constantly polling, meaning more control messages are sent, will tend to be slower than the other filters tested.

C.2 2 Node Configurations

Table 19 shows those LP combinations that indicated a cause-effect relationship to the wall time. The x 's in the table signify that any of the other unmentioned LPs can be placed at that node, being sure not to violate previous or successive combinations.

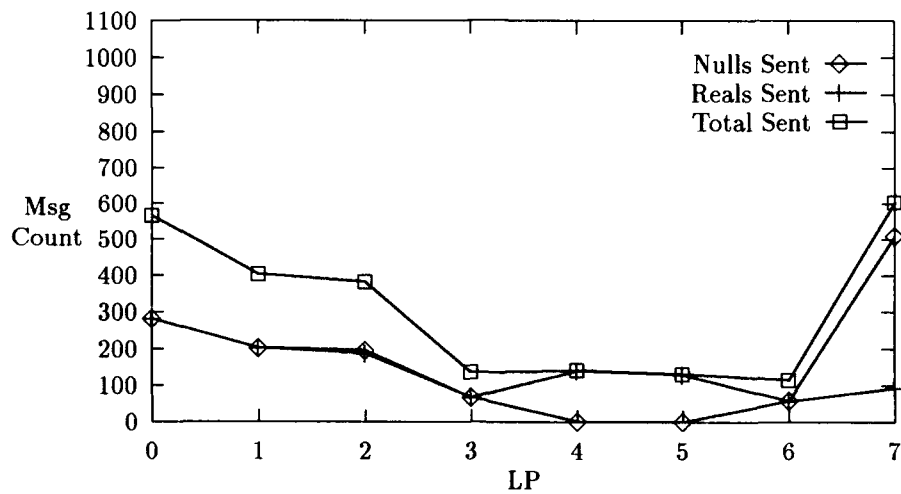


Figure 20. Safewash Messages Sent - 8 LPs, $t = 1000$

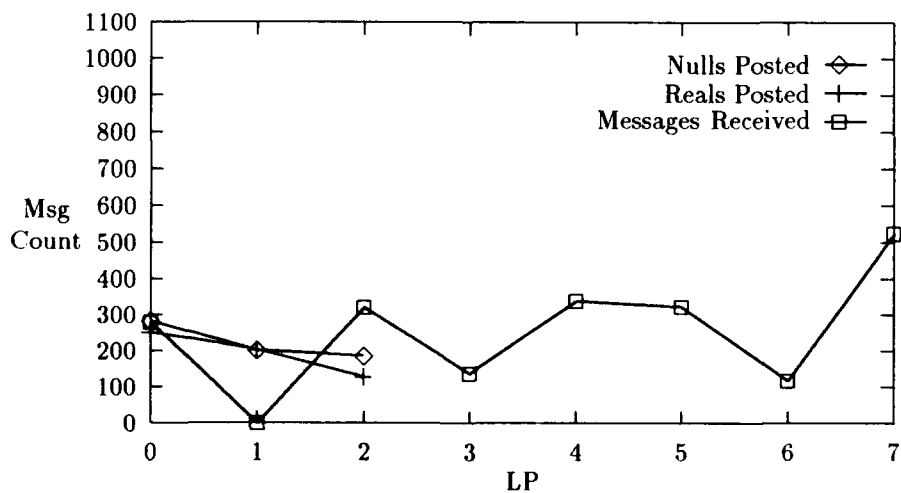


Figure 21. Safewash Messages Posted and Received - 8 LPs, $t = 1000$

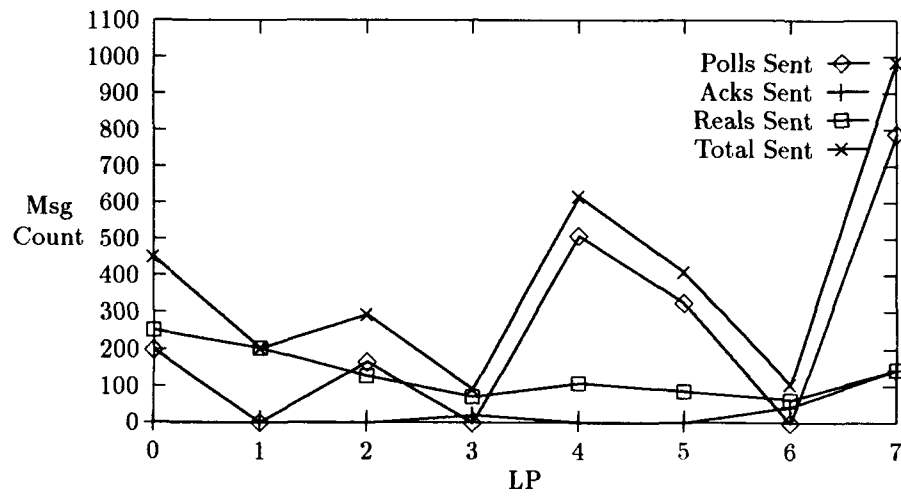


Figure 22. SRADwash Messages Sent - 8 LPs, $t = 1000$

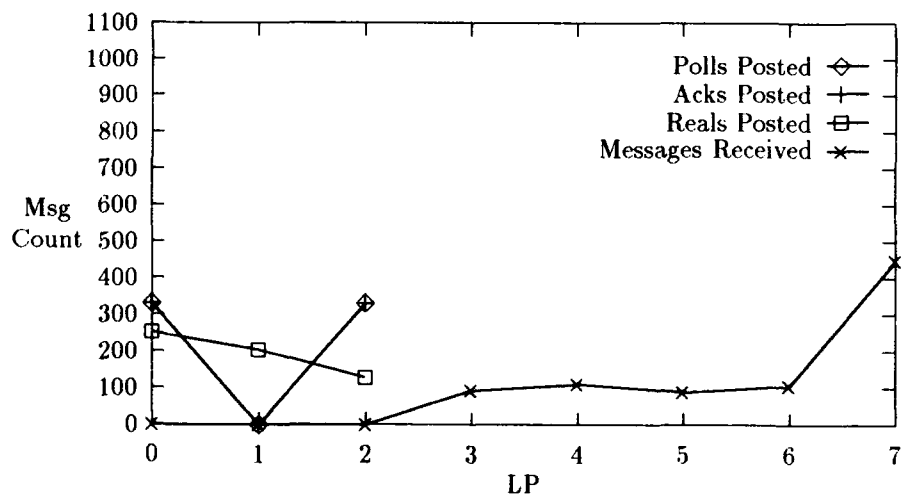


Figure 23. SRADwash Messages Posted and Received - 8 LPs, $t = 1000$

Table 19. 2 Node Wall Time Ranges - 8 LPs (secs), $t = 1000$

<i>Filter</i>	<i>Node 0 LPs</i>	<i>Node 1 LPs</i>	<i>Time (secs)</i>
Nullwash	x,4,5,x	1,x,x,7	1.70-1.79
	x,4,x,x	x,x,5,7	1.80-1.83
	x,x,5,x	x,4,x,7	1.84-1.90
	x,x,x,x	x,4,5,7	1.92-2.30
	1,4,5,x	x,x,x,7	2.41-2.75
Delwash	x,4,5,x	1,x,x,7	1.44-1.52
	x,4,x,x	x,x,5,7	1.55-1.60
	x,x,5,x	x,4,x,7	1.62-1.76
	x,x,x,x	x,4,5,7	1.79-2.01
	1,4,5,x	x,x,x,7	2.12-2.36
Safewash	x,4,5,x	1,x,x,7	2.06-2.66
	x,4,x,x	x,x,5,7	2.82-2.95
	x,x,5,x	x,4,x,7	3.04-3.13
	x,x,x,x	x,4,5,7	3.18-3.45
	1,4,5,x	x,x,x,7	3.51-4.01
SRADwash	x,4,5,x	x,x,x,7	7.25-10.43
	x,4,x,x	x,x,5,7	10.78-12.69
	x,x,5,x	x,4,x,7	12.76-14.11
	x,x,x,x	x,4,5,7	14.27-17.58

As seen from the 2 node experiments, LPs 4, 5, and 7 have the greatest impact on the simulation wall time. The reason behind this is the way that the 8 LP carwash model is set up. Referring back to Figure 7 in Chapter III, LPs 4, 5, and 7 are each merge LPs with multiple incoming communication paths. This tends to form much more complex communication paths than the LP 0, 3, and 7 path, which is basically a single input, single output path. LP1 only communicates, and thus sends all its messages to LPs 4 and 5, which are also getting messages from LPs 0 and 2. LPs 4 and 5, in turn, only communicate with LP7, which is also receiving messages from LPs 3 and 6. LPs 4, 5, and 7, can be considered the simulation's critical LPs since a majority of the communication paths are to one of these three LPs.

As seen in Table 19, for the *nullwash*, *delwash*, and *safewash* filters, the optimum LP combination occurs when LPs 4 and 5 are resident on the same node, and LPs 1 and 7 are resident on the other node. The wall time then gradually increases as various

combinations of LPs 4, 5, and 7 are located on each of the two nodes, concluding with the worst combination when LPs 4 and 5 are resident on the same node with LP1.

Looking at Figures 16, 18, and 20, when the worst combinations occur, a majority of the messages are sent intra-node. In the best cases, when LPs 4 and 5 are one node, and LPs 1 and 7 are on the other node, the messages sent are more internode. This brings up the point again of why the other LPs are minor players in the simulation performance. As mentioned before, the other LPs are not critical LPs, and to add to the discussion, while processing messages the other LPs need only wait on one input arc to proceed. Whereas, LPs 4, 5, and 7 all must wait on at least two input arcs. By waiting on only one input arc, the LPs can act as a pseudo-free-runner, meaning that they can proceed as soon as a message arrives, and can act somewhat independently without effecting the other LPs. LPs 4, 5, and 7 all must wait for more messages to arrive, which slows down the LP processing and ultimately the simulation. By placing these LPs, in various configurations on the same node, in a manner which allows them to feed each other (LP4 feeding LP7, or LP1 feeding LPs 4 and 5, etc) just increases the node processing load and simulation wall time.

The *SRADwash* filter is a different case. As noticed in Table 19 the best combinations are those with LPs 4 and 5 resident on one node and LP7 on the other node, while the worst combinations are those with LPs 4, 5, and 7 resident on the same node. Looking at Figure 22, and remembering that the *SRADwash* filter sends a vast majority of messages (POLLS) "backwards", when LPs 4, 5, and 7 are combined, the critical LPs are all on the same node, which sends a tremendous amount of messages intra-node. When LPs 4 and 5 are combined, and LP7 is on the other node, the two nodes now share more equally the total simulation messages sent, resulting in more internode communications and faster wall time. In addition, since the nodes processing load was primarily in sending messages, processing messages posted and received (Figure 23) tended to be overshadowed.

C.3 4 Node Configurations

As in the 2 node configurations, LPs 4, 5, and 7 showed the greatest influence on the simulation wall time. But, unlike the 2 node configurations, all three of the Chandy-Misra filters did not produce analogous results (Table 20). For the 4 node runs, the run time

was increased to $t = 2000$ in order to produced a better resolution between the filters. Although, in the case of the *SRADwash* filter which tended to deadlock most of the time, the run time was kept at $t = 1000$. The *safewash* filter, when LPs 4 and 5 are located on the same node, produced the best results, while in the other two Chandy-Misra filters, that same combination produced the worst results. The *safewash* filter produced the worst results when LPs 4 and 7 were located on the same node.

Table 20. 4 Node Wall Time Ranges - 8 LPs (secs), $t = 2000$

<i>Filter</i>	<i>Node 0 LPs</i>	<i>Node 1 LPs</i>	<i>Node 2 LPs</i>	<i>Node 3 LPs</i>	<i>Time (secs)</i>
Nullwash $t = 2000$	x,4	x,5	x,7	x,x	1.89-2.63
	x,x	4,x	5,7	x,x	2.63-2.79
	x,x	x,5	4,7	x,x	2.79-3.01
	x,x	4,5	x,7	x,x	3.16-3.95
Delwash $t = 2000$	x,4	x,5	x,7	x,x	2.86-3.26
	x,x	4,x	5,7	x,x	3.28-3.47
	x,x	x,5	4,7	x,x	3.48-3.64
	x,x	4,5	x,7	x,x	3.72-4.11
Safewash $t = 2000$	x,x	4,5	x,7	x,x	2.84-3.28
	x,4	x,5	x,7	x,x	3.29-4.81
	x,x	4,x	5,7	x,x	4.88-5.16
	x,x	x,5	4,7	x,x	5.17-5.95
SRADwash $t = 1000$	x,4	x,5	x,7	x,x	3.69-5.07
	x,x	4,5	x,7	x,x	5.12-5.23
	x,x	4,x	5,7	x,x	5.24-5.55
	x,x	x,5	4,7	x,x	5.56-6.54

In the *nullwash* and *delwash* filters, looking at the Figures 16 and 18, during the worst combination, when LPs 4 and 5 are located on the same node, the total messages sent are much greater than the LP7 node, and thus are out-of-balance. In the best combinations, the total messages sent are more evenly distributed.

In Figure 20, the biggest difference, as compared to the *nullwash* or *delwash* filters, is the *safewash* LP7 sends out about three times the number of messages, and in this filter, LP7 sends out more messages than any other LP. In the best case, when LPs 4 and 5 are combined, the total messages sent on that node are more equal to the nodes with LP7 resident. During the worst cases, when LP7 is combined with either LPs 4 or 5, the total

messages sent on that node becomes out-of-balance compared against the other nodes with LPs 4 or 5.

From Table 20, the worst cases for *SRADwash* are when LP4 or LP5 is resident on the same node as LP7, while the best combinations are when LPs 4, 5, and 7 are all on different nodes. Looking at Figure 22, it is seen that when LP4 or LP5 is resident with LP7, the total messages sent on that node is the greatest and makes the simulation unbalanced. When LPs 4, 5, and 7 are on different nodes, fewer and more equal numbers of messages are sent on each node, balancing the simulation.

C.4 8 Node Configuration

As seen in Table 21 and Figure 24, simulation wall times with the four filters, at least through $t = 10000$, are also consistent with *safewash* executing the fastest, always followed by *delwash*, *nullwash* and *SRADwash* respectively. Only *SRADwash* failed to complete the 8 node timing tests for all run times.

Table 21. 8 Node Wall Times - 8 LPs (secs)

<i>Run Time</i>	<i>Nullwash</i>	<i>Delwash</i>	<i>Safewash</i>	<i>SRADwash</i>
500	0.53	0.52	0.36	0.61
1000	1.13	1.09	0.89	2.26
2000	2.93	2.92	2.08	7.55
5000	14.05	13.95	8.17	No Data
10000	52.41	50.84	26.15	No Data

As in the 1 node case, the *nullwash* filter should perform slightly below the *delwash* filter, since the *nullwash* filter sends and posts more messages than the *delwash* filter, and in this case it always does, although the differences in wall times between the two filters were closer than in the 1 node tests. This difference is due to the *delwash* computational load now spread over more nodes, thus reducing the impact of the filter algorithm processing.

In the *safewash* filter, it is seen that it sends more messages than the first two filters, yet is constantly faster for 8 nodes. In this case, like the one node case, although the *safewash* algorithm sends more messages, the computational load is spread over more nodes

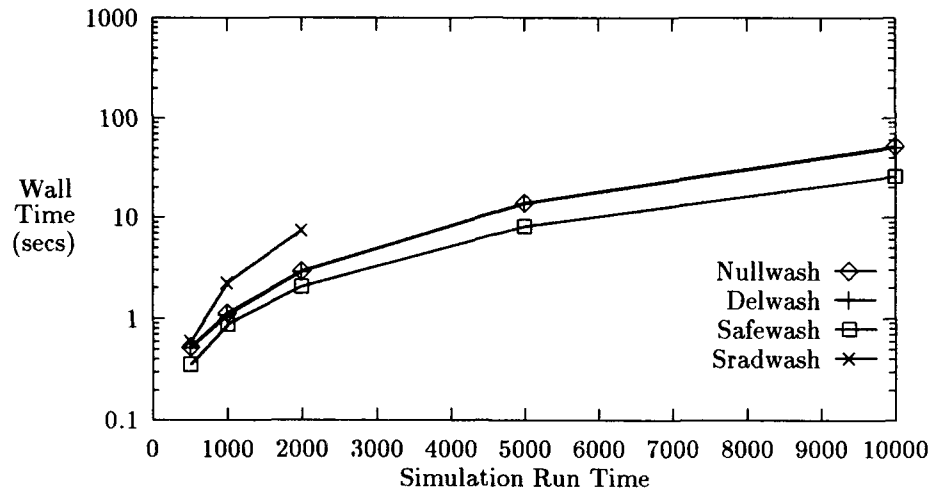


Figure 24. 8 Node Wall Times - 8 LPs (secs)

causing the simulation to proceed in a more efficient manner, and ultimately speeding up the simulation.

Looking at only the *SRADwash* messages sent, it is again seen that this filter sends hundreds of more messages relative to the other filters. And again, this causes the filter to be slower than the others tested.

Appendix D. 8 LP Carwash, Without Feedback, Experimental Data

In this experiment, two cases were considered:

1) Modifying the carwash application code not to "rewash" any cars, thus destroying all cars, but *not* modifying the arcs file so it still allows non-real messages to be sent to LPs 0 and 2 from LP7.

2) Modifying the carwash application code not to "rewash" any cars, and modifying the arcs file so it does *not* allow messages to be sent to LPs 0 and 2 from LP7.

D.1 Without Feedback Case 1 Configurations

Wall times for the 1 node case are presented Table 22 and Figure 25. At the start, the *safewash* filter performs the best, then as the run time is increased, the *delwash* filter becomes the best. In all cases, the *nullwash* and *SRADwash* filter were always the third and fourth fastest respectively, while the *delwash* and *SRADwash* failed to complete all the run time tests. These results are different than in the configuration with feedback, in that the *nullwash* and *delwash* filters were then always competing as the best filter.

Table 22. 1 Node Wall Times, Arcs Feedback - 8 LPs (secs)

Run Time	Nullwash	Delwash	Safewash	SRADwash
500	2.00	1.74	1.65	6.41
1000	3.74	2.91	3.08	No Data
2000	7.79	6.38	7.09	No Data
5000	24.95	No Data	21.39	No Data
10000	76.59	No Data	68.61	No Data

The case 1 wall times for 1 node were faster than with feedback, with the difference in wall times between the Chandy-Misra filters being smaller as compared to those with feedback. For all intents and purposes, unless run times are extremely long, the Chandy-Misra filters performed the same. The minor speedup encountered is attributable to three things:

- 1) No real messages, and corresponding NULL messages, sent to LPs 0 and 2.

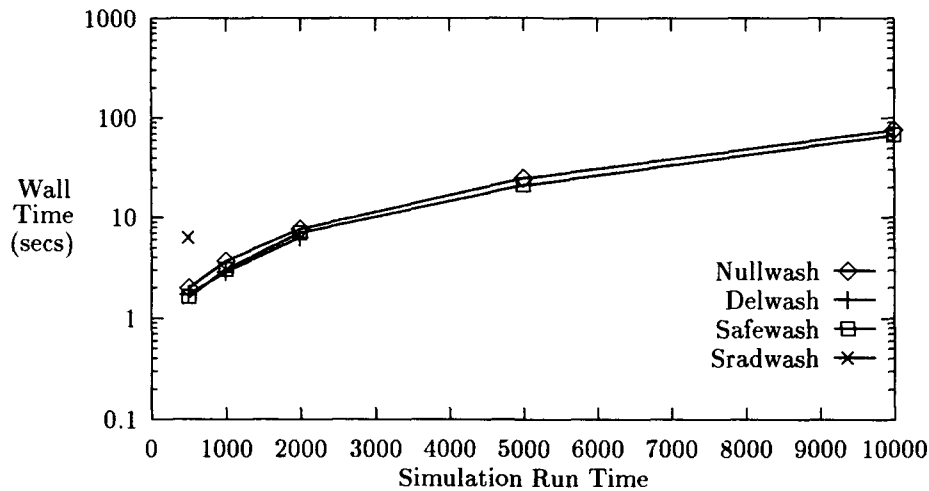


Figure 25. 1 Node Wall Times, Arcs Feedback - 8 LPs (secs)

- 2) Reduction in the number of real messages "rewashed" and sent from LPs 0 and 2.
- 3) Reduction in, and now less complex application code to process real messages, since no real messages are sent from LP7.

For the 8 node case, wall times are presented in Table 23 and Figure 28. The wall times for the *nullwash*, *delwash*, and *SRADwash* filter, were faster when the run time was greater than 1000, as compared to the feedback version, while the *safewash* filter showed virtually no change. But unlike the 1 node case, the 8 node case filter wall times compared the same against the feedback case, with *safewash* generally executing the fastest, followed by *delwash*, *nullwash*, and *SRADwash*. Again, *delwash* and *SRADwash* failed to complete the tests.

For the 2 and 4 node configurations, identical optimal and non-optimal configurations were obtained, and are explained from the fact that the only difference from the feedback version is the lack of real messages sent, those from LP7 to LPs 0 and 2. Since the ratio of real messages to NULL messages is small, removing the real messages sent from LP7 did not effect the overall simulation message counts, ratios, and load balance since LP7 still sent the same general numbers of messages.

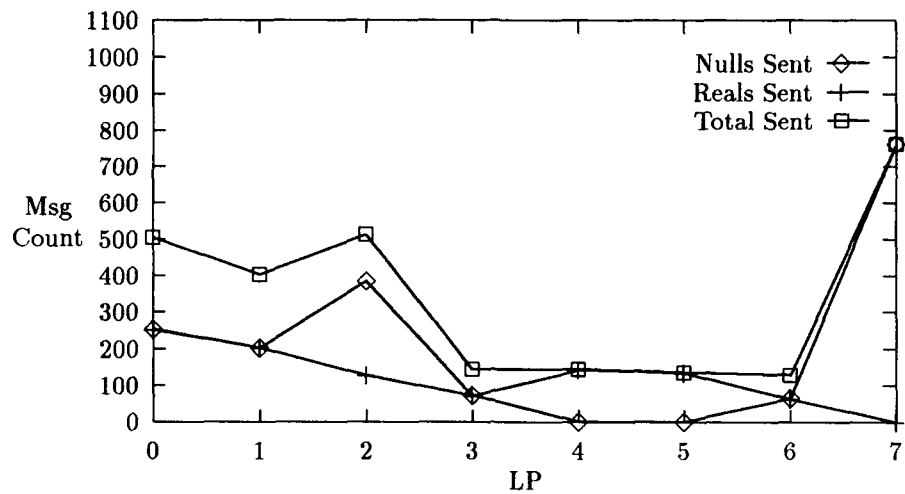


Figure 26. Safewash Messages Sent, Arcs Feedback - 8 LPs, $t = 1000$

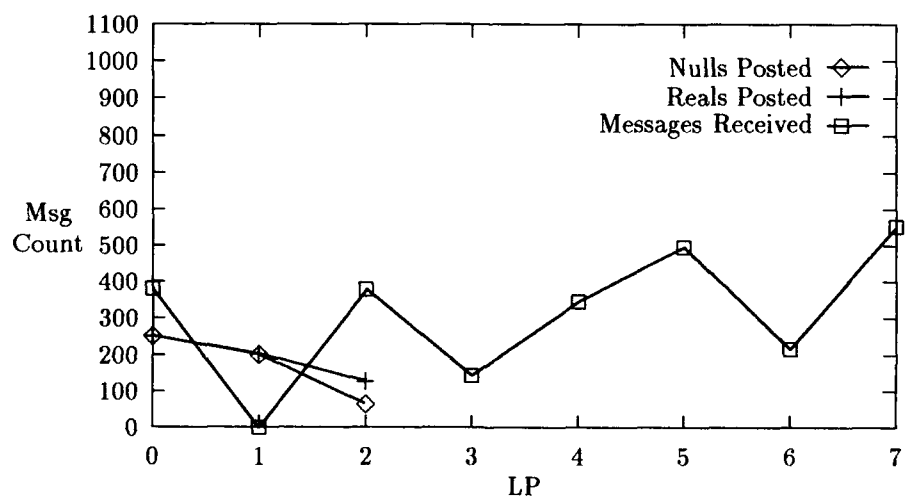


Figure 27. Safewash Messages Posted and Received, Arcs Feedback - 8 LPs, $t = 1000$

Table 23. 8 Node Wall Times, Arcs Feedback - 8 LPs (secs)

<i>Run Time</i>	<i>Nullwash</i>	<i>Delwash</i>	<i>Safewash</i>	<i>SRADwash</i>
500	0.85	0.84	0.78	0.63
1000	1.25	1.21	1.06	2.27
2000	2.51	2.54	2.34	7.17
5000	9.87	No Data	8.58	No Data
10000	33.09	No Data	28.97	No Data

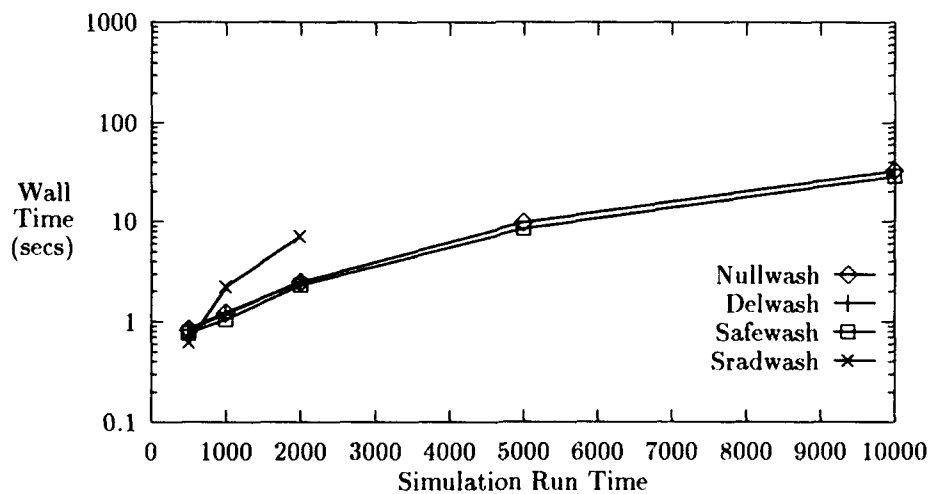


Figure 28. 8 Node Wall Times, Arcs Feedback - 8 LPs (secs)

D.2 Without Feedback Case 2 Configurations

During the 1 node tests (Tables 24 and Figures 29) up through $t = 10000$, only one general difference resulted over the feedback tests. Here the *safewash* filter, now with four less arcs to track (2 output arcs from LP7 and 1 input arc each into LP0 and 2) and process, showed to be the fastest, followed by *delwash*, *nullwash*, and *SRADwash*, which was the only filter not to complete the test. Compared to the feedback version, both the *safewash* and *SRADwash* filters always outperformed the feedback version, while the *nullwash* and *delwash* filters always performed worse when the run time was greater than 1000.

Table 24. 1 Node Wall Times, w/o Feedback - 8 LPs (secs)

Run Time	Nullwash	Delwash	Safewash	SRADwash
500	1.67	1.26	1.25	3.78
1000	4.18	3.38	3.16	15.50
2000	9.44	8.44	7.09	No Data
5000	57.41	52.05	26.35	No Data
10000	308.48	194.69	86.03	No Data

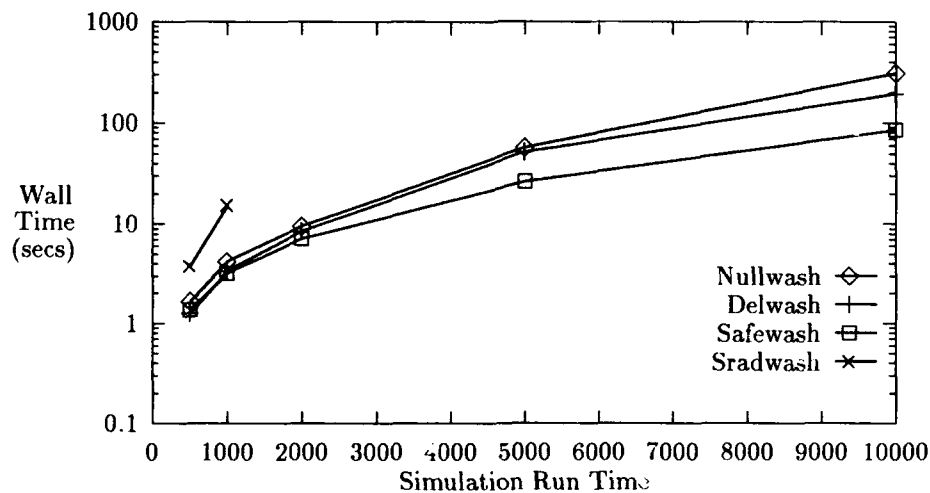


Figure 29. 1 Node Wall Times, w/o Feedback - 8 LPs (secs)

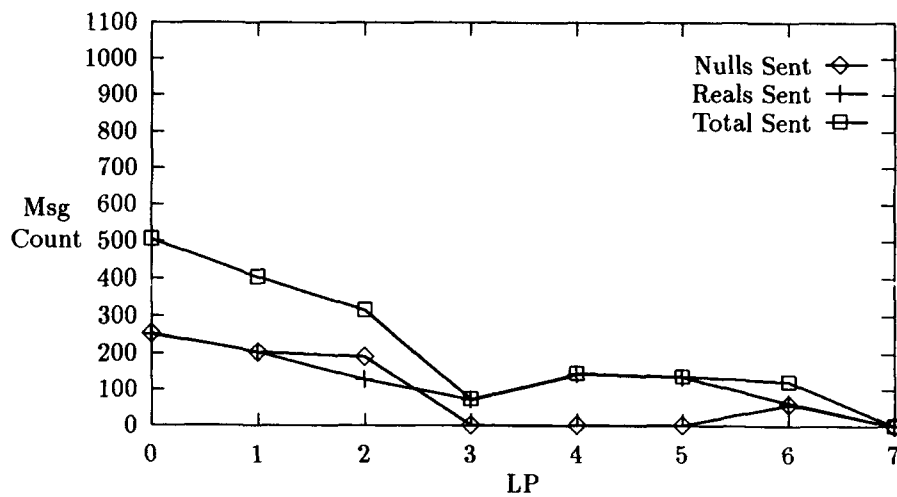


Figure 30. Safewash Messages Sent, w/o Feedback - 8 LPs, $t = 1000$

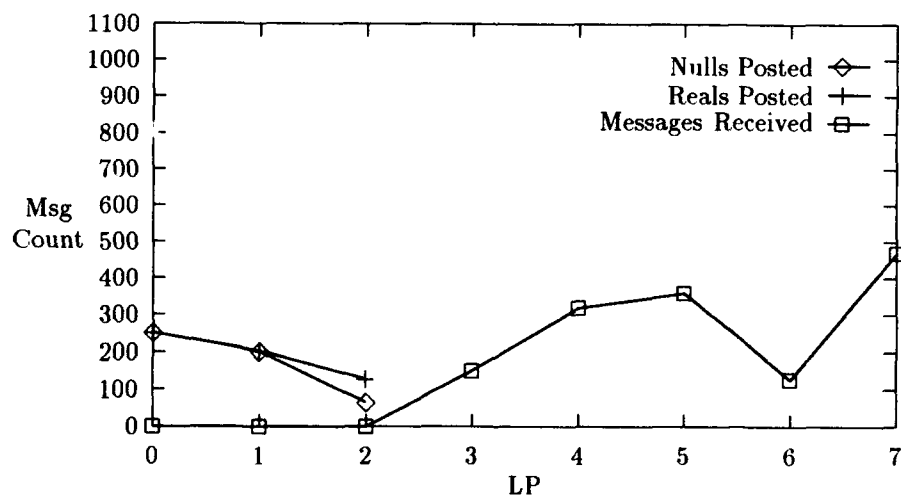


Figure 31. Safewash Messages Posted and Received, w/o Feedback - 8 LPs, $t = 1000$

During the 8 node tests (Tables 25 and Figures 32) up through $t = 10000$, the *safewash* filter again performed the best, followed by the *delwash*, *nullwash*, and *SRADwash* filters, respectively. This time, compared to the feedback version, both the *safewash* and *SRADwash* filters were better. The *nullwash* and *delwash* filters performed better without feedback up to $t = 5000$, at which point they slowed down and became worse. The *SRADwash* filter failed to complete all the tests, but attained higher comparable run times than with the feedback version.

Table 25. 8 Node Wall Times, w/o Feedback - 8 LPs (secs)

Run Time	Nullwash	Delwash	Safewash	SRADwash
500	0.27	0.23	0.17	0.43
1000	0.82	0.69	0.36	1.07
2000	2.70	2.36	0.99	3.53
5000	13.38	13.72	4.43	No Data
10000	65.33	53.11	10.21	No Data

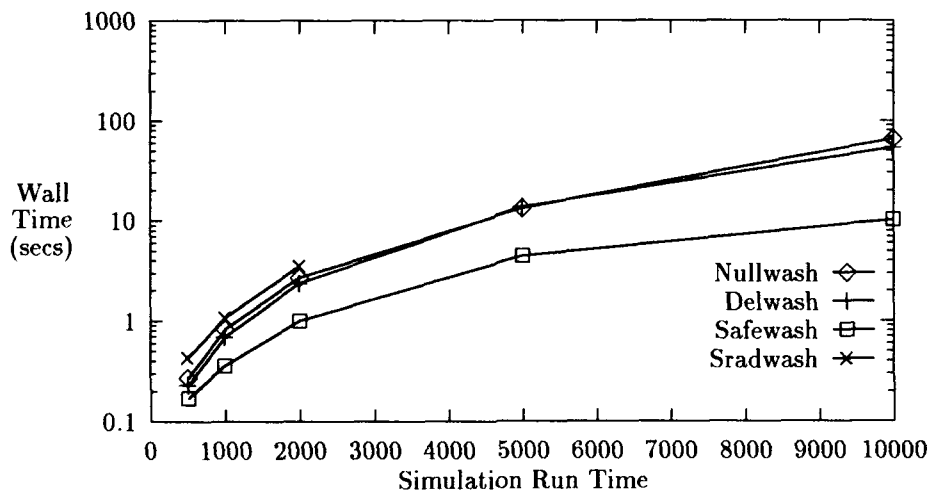


Figure 32. 8 Node Wall Times, w/o Feedback - 8 LPs (secs)

For both the 2 and 4 node configurations, wall time results followed the 8 node case, and the same best and worst partitions showed up as in the feedback case. The identical

partitions are explained from the fact that the only difference from the feedback version is the lack of two arcs, those from LP7 to LPs 0 and 2. To reiterate, referring back to Figure 8 in Chapter III, LPs 4, 5, and 7 are each merge LPs with multiple incoming communication paths. LP1 only communicates, and thus sends all its messages to LPs 4 and 5, which are also getting messages from LPs 0 and 2. LPs 4 and 5, in turn, only communicate with LP7, which is also receiving messages from LPs 3 and 6. The LPs 4, 5, and 7, can be considered the simulation's critical LPs since a majority of the communication paths are to one of these three LPs.

Since the deletion of the feedback arcs did not effect the critical LPs, or the overall message production and loading of each LP, the same partitions repeated themselves again. For example, looking at Figures 30 and 31, even though LP7 did not send any messages, it still received the same number of messages as the feedback version, thus it still had to process a large number of messages even though LP7 deleted all of them, and giving rise to the same processing load, communication load, and partitionings as the 2 and 4 node feedback results.

Appendix E. 7 LP Carwash, With Feedback, Experimental Data

E.1 1 Node Configuration

As seen in Table 26 and Figure 33, simulation wall times with the four filters, at least through $t = 10000$, are fairly consistent. The *delwash* filter is the fastest, followed by the *nullwash*, *safewash* and *SRADwash* filters, respectively. Both *safewash* and *SRADwash* failed to complete the 1 node timing tests for all run times.

Table 26. 1 Node Wall Times - 7 LPs (secs)

Run Time	Nullwash	Delwash	Safewash	SRADwash
500	1.76	1.24	1.80	4.98
1000	3.71	2.91	3.82	No Data
2000	9.01	7.27	12.77	No Data
5000	46.97	31.86	No Data	No Data
10000	157.26	111.90	No Data	No Data

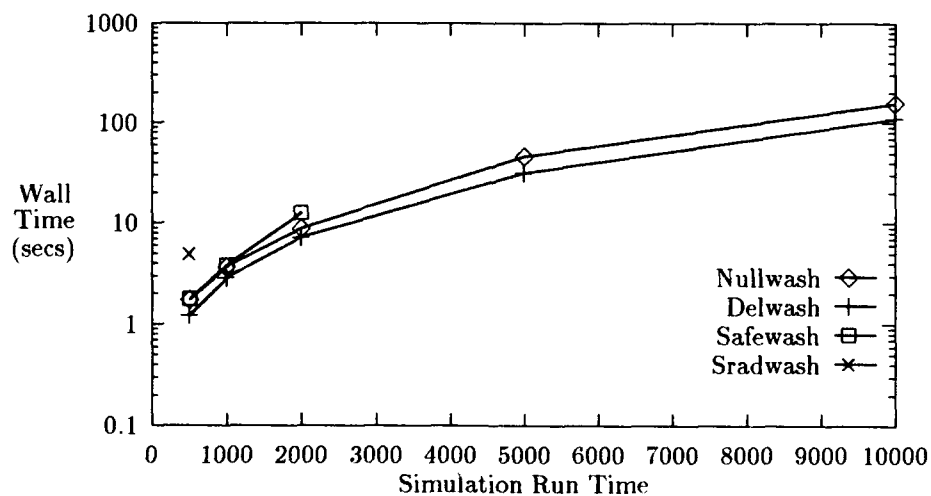


Figure 33. 1 Node Wall Times - 7 LPs (secs)

Looking at Figures 34 and 36, for the *nullwash* and *delwash* filters, it appears from the graphs that the *delwash* filter should always perform equal to the *nullwash* filter. But

again, the biggest difference is in the number of messages posted and received (Figures 35 and 37). It can be seen that in the *delwash* filter, all three source LPs posted and received significantly fewer messages than in the *nullwash* filter. This is due, of course, to the *delwash* filter deleting NULL messages addressed back to the originator. Also, unlike the 8 LP carwash, even though the *delwash* filter has the added code to check for posted NULL messages, it appears that because of the fewer number of LPs, the added processing overhead is not canceling the benefits gained by the deletion of the unneeded NULL messages during the higher run times.

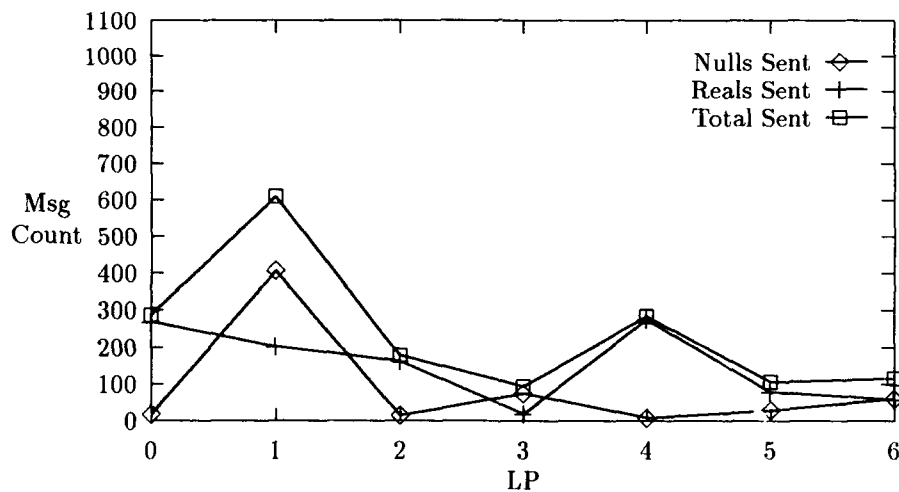


Figure 34. Nullwash Messages Sent - 7 LPs, $t = 1000$

Overall, the *safewash* filter performed worse than both the *nullwash* and *delwash* filters. And although it posted and received fewer messages (Figure 39) than both the *delwash* and *nullwash* filters, it did send out more messages (LP7 in Figure 38) than both the filters. Combine this with the fact that the *safewash* filter also has some added code to track and update channel times, all detrimentally add to the wall time when executed on 1 node.

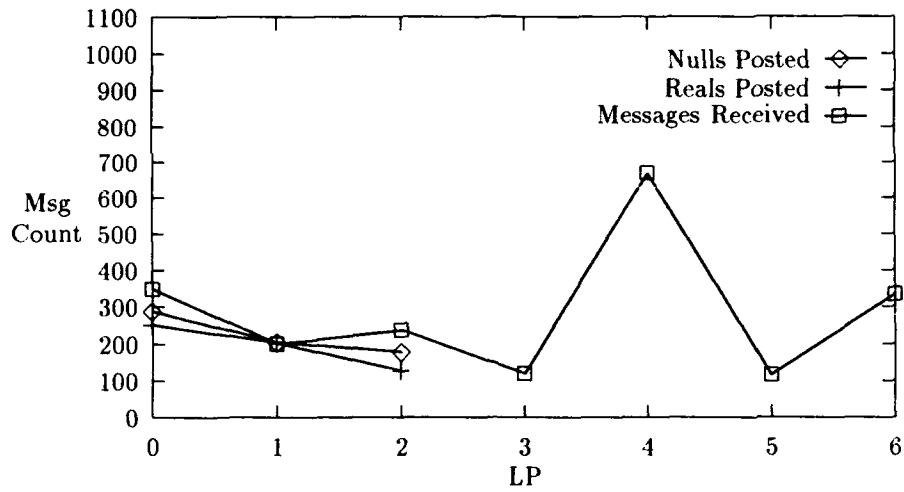


Figure 35. Nullwash Messages Posted and Received - 7 LPs, $t = 1000$

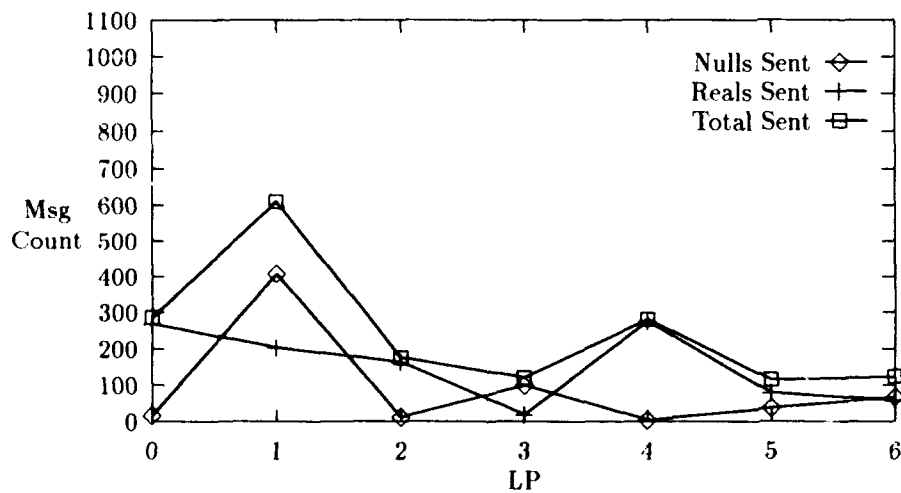


Figure 36. Delwash Messages Sent - 7 LPs, $t = 1000$

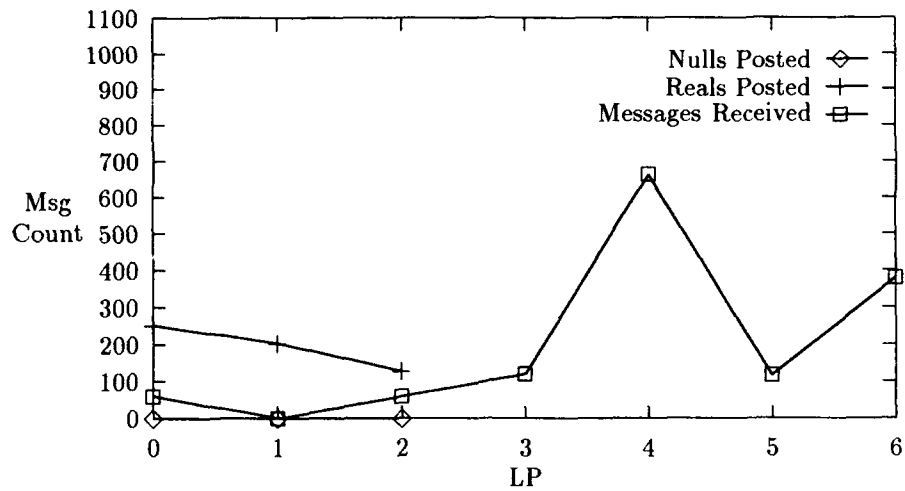


Figure 37. Delwash Messages Posted and Received - 7 LPs, $t = 1000$

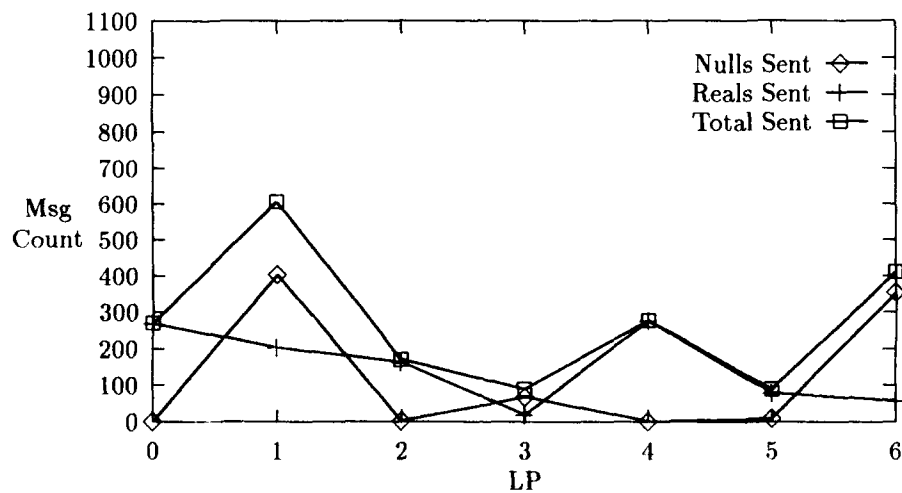


Figure 38. Safewash Messages Sent - 7 LPs, $t = 1000$

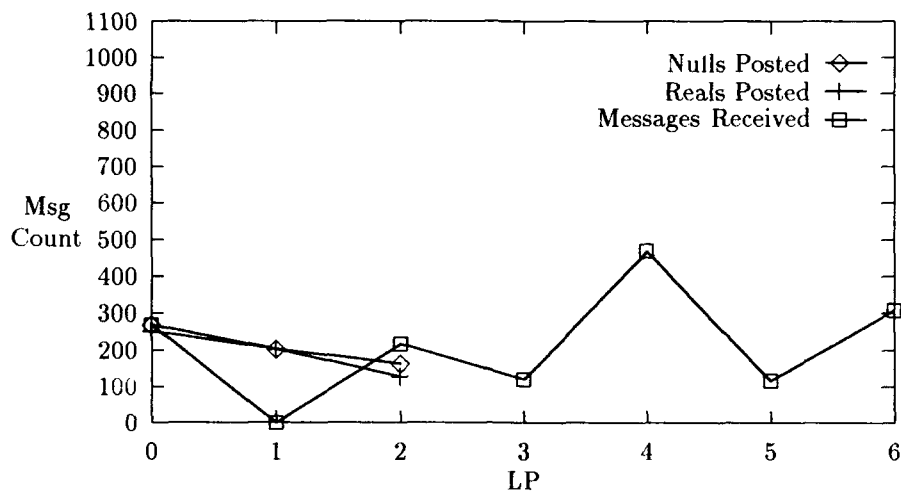


Figure 39. Safewash Messages Posted and Received - 7 LPs, $t = 1000$

Looking at the number of messages sent (Figure 40), the *SRADwash* filter sent hundreds of more messages (LPs 4 and 6) than all the other filters. Thus, it is no surprise to find this filter the worst of the four tested. But looking at the numbers of messages posted and received (Figure 41), it shows again that the *SRADwash* filter processed hundreds of fewer messages. But since the number of messages sent were so overwhelming, the numbers of messages posted and received were inconsequential.

E.2 2 Node Configurations

Table 19 again shows those LP combinations that indicated a cause-effect relationship to the wall time. The x 's in the table signify that any of the other unmentioned LPs can be placed at that node, being sure not to violate previous or successive combinations.

As seen from the 2 node experiments, LPs 4 and 6 have the greatest impact on the simulation wall time. The reason behind this is the way that the 7 LP carwash model is set up. Referring back to Figure 9 in Chapter III, LPs 4 and 6 are both merge LPs with multiple incoming communication paths. This tends to form much more complex communication paths than the LP 1, 3, and 6 path, which is basically a single input, single

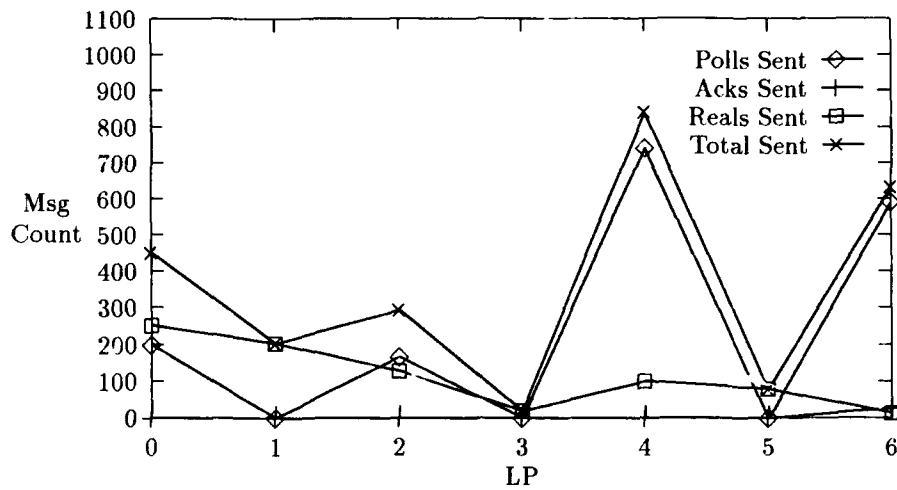


Figure 40. SRADwash Messages Sent - 7 LPs, $t = 1000$

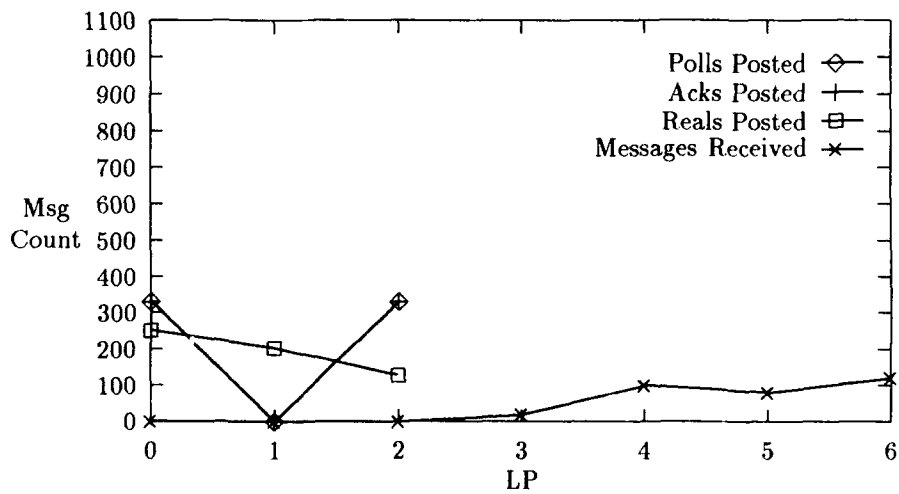


Figure 41. SRADwash Messages Posted and Received - 7 LPs, $t = 1000$

Table 27. 2 Node Wall Time Ranges - 7 LPs (secs), $t = 1000$

<i>Filter</i>	<i>Node 0 LPs</i>	<i>Node 1 LPs</i>	<i>Time (secs)</i>
Nullwash	x,4,6	x,x,x,x	1.61-1.91
	x,4,x	x,x,x,6	1.95-2.07
	x,x,x	x,x,4,6	2.08-2.22
	x,x,6	x,x,4,x	2.24-2.62
Delwash	x,4,6	x,x,x,x	1.43-1.60
	x,4,x	x,x,x,6	1.61-1.70
	x,x,x	x,x,4,6	1.72-1.80
	x,x,6	x,x,4,x	1.83-2.05
Safewash	x,4,x	x,x,x,6	1.57-2.36
	x,x,x	x,x,4,6	2.51-3.34
	x,4,6	x,x,x,x	2.51-3.34
	x,x,6	x,x,4,x	3.37-4.78
SRADwash	x,4,x	x,x,x,6	5.71-8.29
	x,x,6	x,x,4,x	8.33-9.93
	x,4,6	x,x,x,x	8.33-9.93
	x,x,x	x,x,4,6	10.15-14.54

output path. LP1 communicates, and thus sends all its messages to LPs 3, 4 and 5, while LPs 0 and 2 only communicate with LP4. LP4 in turn, only communicates with LP6, which is also receiving messages from LPs 3 and 5. LPs 4 and 6 can be considered the simulation's critical LPs, since a majority of the communication paths are to one of those two LPs.

As seen in Table 27, for the *nullwash* and *delwash* filters, the optimum LP combinations occur anytime LPs 4 and 6 are resident on the 3 LP node. The wall time then increases as LP4 is resident on the 3 LP node and LP6 is resident on the 4 LP node, and LPs 4 and 6 resident together on the 4 LP node, concluding with the worst combinations when LP4 is resident on the 4 LP node, and LP6 on the 3 LP node.

By looking at Figures 34 - 37, when the worst combinations occur, LP4, sending more than three times the number of messages, and receiving more than twice the number of messages than LP6, is also combined with three other LPs. Comparing this with LP6 resident on the 3 LP node, which uses significantly less processing power, creates a load

imbalance and slows down the simulation. In the best configurations, both of the critical LPs are resident on the 3 LP node, allowing for more intra-node processing.

For the *safewash* filter (Table 27), the best combinations occur when LP4 is resident on the 3 LP node, while LP6 is on the 4 LP node. This progresses through with LPs 4 and 6 resident together on the 4 LP node, then on the 3 LP node, finally to the worst case when LP6 is resident on the 3 LP node, and LP4 on the 4 LP node. In Figures 38 and 39, it can be seen that the same argument above can be used here for the worst case since they are the same, but why did the *nullwash* and *delwash* best case configuration become one of *safewash*'s worst cases? That is the case when LPs 4 and 6 are combined on the 3 LP node. Looking at the figures, the biggest difference is that LP6 sends out about four times the number of messages in the *safewash* filter. At this point, even though the two critical LPs are on the 3 LP node, and intra-node communication is reduced, it appears that the node is starting to become saturated with messages to the point of getting imbalanced against the 4 LP node. Thus, placing the critical LPs on the node with fewer LPs may not always give the best results.

The *SRADwash* filter is again a different case, but mirrors slightly the *safewash* filter since both LPs 4 and 6 are highly active. As noticed in Table 27, the best combinations are those with LP4 resident on the 3 LP node, and LP6 on the 4 LP node, while the worst combinations are those with LPs 4 and 6 on the 4 LP node. Looking at Figure 40, when LPs 4 and 6 are combined, the both highly active critical LPs are on the same node, which sends a tremendous amount of messages intra-node. Here is an extreme case, when both very active critical LPs are resident on the 4 LP node, making the node's workload extremely unbalanced as compared to the now lightly loaded 3 LP node. When LP4 and LP6 are separated, as in the best cases, the two nodes now share more equally, with the more active LP4 on the less worked 3 LP node, and the less active LP6 on the 4 LP node, the total messages sent intra-node, resulting in a faster wall time. In addition, it appears since the nodes processing load was primarily in sending messages, processing messages posted and received (Figure 41) tended to be overshadowed again.

E.3 4 Node Configurations

Like the 2 node configurations, LPs 4 and 6 again showed the greatest influence on the simulation wall time (Table 28). All four filters produced the best results when LP6 was resident on a 2 LP node, while LP4 was resident on the 1 LP node. The worst combinations for the *safewash* and *SRADwash* filters, when LPs 4 and 6 were resident on a 2 LP node, again produced one of the best configurations for the *nullwash* and *delwash* filters. Similar arguments, from the 2 node configurations for the best and worst combinations, can apply here as well for the 4 node configurations.

Table 28. 4 Node Wall Time Ranges - 7 LPs (secs), $t = 2000$

Filter	Node 0 LPs	Node 1 LPs	Node 2 LPs	Node 3 LPs	Time (secs)
Nullwash $t = 2000$	x,x	x,6	4	x,x	2.66-3.75
	x,x	4,6	x	x,x	3.79-4.04
	x,4	x,6	x	x,x	4.06-5.33
Delwash $t = 2000$	x,x	x,6	4	x,x	3.79-3.89
	x,x	4,6	x	x,x	3.91-3.98
	x,4	x,6	x	x,x	4.00-4.76
Safewash $t = 2000$	x,x	x,6	4	x,x	2.14-3.10
	x,4	x,6	x	x,x	3.16-5.73
	x,4	x,x	6	x,x	5.97-6.19
	x,x	4,6	x	x,x	6.25-7.63
SRADwash $t = 1000$	x,x	x,6	4	x,x	2.58-3.57
	x,4	x,x	6	x,x	3.62-3.99
	x,4	x,6	x	x,x	3.99-5.12
	x,x	4,6	x	x,x	5.17-6.34

Looking at the Figures 34 - 37, during the worst combinations when LPs 4 and 6 are located on separate 2 LP nodes, the messages LP4 sends and receives are much greater than those from LP6, increasing the workload on the node and creating a simulation imbalance. In the best combinations when LP4 is on the 1 LP node, the node processing load decreases, and distributes the processing load more evenly.

From Table 28, the worst cases for *SRADwash* and *safewash* are when LPs 4 and 6 are resident on the same 2 LP node. While the best combinations are when LPs 4 or 6 are on the 1 LP node. Looking at Figures 38, 39, and 40, it is seen that when LPs 4 and 6

are resident together, the total messages sent on that node is the greatest over all the LPs, and makes the simulation unbalanced. In these cases, LP6 is also very active, compared to the other Chandy-Misra filters, and combinations with LP6 resident on the 1 LP node stands out as one of the middle configurations. Whereas, when LPs 4 or 6 are on the 1 LP node, the node only has one LP to be concerned with, thus again balancing the overall simulation.

E.4 7 Node Configuration

As seen in Table 29 and Figure 42, simulation wall times with the four filters, at least through $t = 10000$, are consistent with *safewash* always executing the fastest, and *SRADwash* always executing the slowest. The *delwash* and *nullwash* filters reverse themselves when the run time is greater than 2000, with *delwash* performing the best at the lower run times, and *nullwash* performing the best during the longer run times. Only *SRADwash* failed to complete the 7 node timing tests for all run times.

Table 29. 7 Node Wall Times - 7 LPs (secs)

<i>Run Time</i>	<i>Nullwash</i>	<i>Delwash</i>	<i>Safewash</i>	<i>SRADwash</i>
500	0.43	0.36	0.30	0.55
1000	1.19	1.09	0.67	1.62
2000	3.52	3.43	1.83	5.49
5000	14.59	17.91	8.92	No Data
10000	44.88	67.99	22.54	No Data

As in the 1 node case, the *nullwash* filter should perform slightly below the *delwash* filter, since the *nullwash* filter sends and posts more messages than the *delwash* filter, and in this case it does in the lower run times, although the differences in wall times between the two filters were closer than in the 1 node tests. The real difference is during the higher run times when the *delwash* filter starts performing worse than the *nullwash* filter. This difference is apparently due to the *delwash* computational load, even though now spread over more nodes, still increasing the impact of the filter algorithm processing.

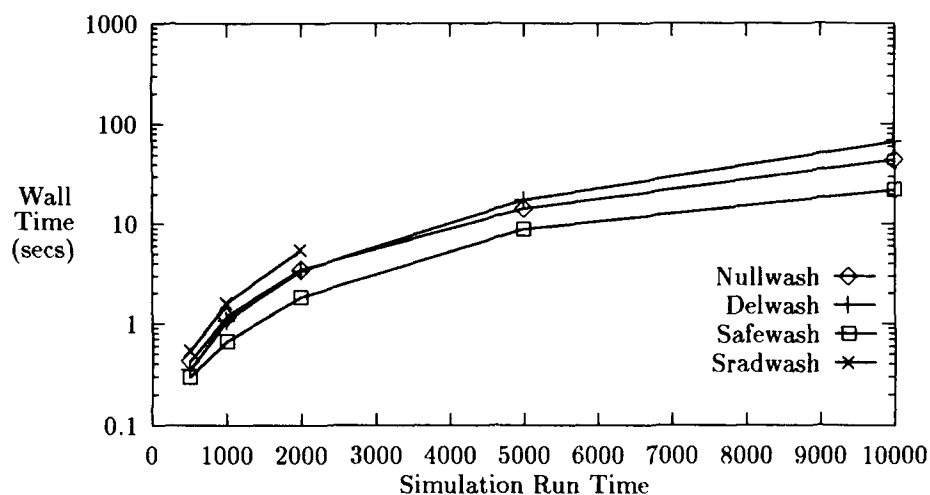


Figure 42. 7 Node Wall Times - 7 LPs (secs)

In the *safewash* filter, it is seen that it sends more messages than the first two filters, yet is constantly faster for 7 nodes. In this case, like the one node case, although the *safewash* algorithm sends more messages, the computational load is now spread over more nodes, causes the simulation to proceed at a more efficient manner, and ultimately speeding up the simulation.

Looking at only the *SRADwash* messages sent, it is again seen that this filter sends hundreds of more messages relative to the other filters (LPs 4 and 6). Again, this causes the filter to be slower than the others tested.

Bibliography

1. Beard, R. Andrew and Gary B. Lamont. *AFIT/ENG Intel Hypercube Quick Reference Manual* (Version 1.1 Edition). Air Force Institute of Technology, Feb 1990.
2. Bergman, Kenneth C. *Spatial Partitioning of a Battlefield Parallel Discrete Event Simulation*. MS thesis AFIT/GCS/ENG/92D-03, Air Force Institute of Technology, 1992.
3. Breeden, Thomas A. *Parallel Implementations of Structural VHDL Circuits on Intel Hypercubes*. MS thesis AFIT/GCE/ENG/92D-01, Air Force Institute of Technology, 1992.
4. Bryant, Randal E. "Simulation on a Distributed System." *Proceedings of the 1st Int'l Conference on Distributed Computing Systems*. 544-552. Oct 1979.
5. Cai, Wentong and Stephen J. Turner. "An Algorithm for Distributed Discrete-Event Simulation - The Carrier Null Message Approach." *Proceedings of the SCS Multi-conference on Distributed Simulation* 22. 3-8. Mar 1990.
6. Chandy, K. M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, 24(11):198-206 (May 1981).
7. Chandy, K. M. and Jayadev Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, SE-5(5):440-452 (Sep 1979).
8. Chandy, K. M. and R. Sherman. "The conditional event approach to distributed simulation." *Proceedings of the SCS Multi-conference on Distributed Simulation* 21. 93-99. Mar 1989.
9. Davis, Nathaniel J. and others. "Distributed Discrete-Event Simulation Using Null Message Algorithms on Hypercube Architectures," *Journal of Parallel and Distributed Computing*, 8:349-357 (1990).
10. Fujimoto, Richard M. "Parallel Discrete-Event Simulation," *Communications of the ACM*, 33(10):31-53 (Oct 1990).
11. Hammell, Robert J. *Empirical Analysis of a Parallel Simulation*. Project Report for CEG 721, Computer Architecture II, Wright State University, Dec 1991.
12. Hartrum, Thomas C. *AFIT Guide to SPECTRUM*. Unpublished Report, Air Force Institute of Technology, Feb 1992.
13. Hartrum, Thomas C. *A Queuing Simulation Application for Parallel Simulation with TCHSIM*. Unpublished Report, Air Force Institute of Technology, Jan 1992.
14. Hartrum, Thomas C. *TCHSIM: A Simulation Environment for Parallel Discrete Event Simulation*. Unpublished Report, Air Force Institute of Technology, Jan 1992.
15. Intel Corporation. *iPSC/2 User's Guide*, Mar 1989. Order Number: 311532-003.
16. Jefferson, David R. "Virtual Time," *ACM Transactions on Programming Languages and Systems*, 3(7):404-425 (Jul 1985).

17. Karonis, Nicholas T. "Timing Parallel Programs That Use Message Passing," *Journal of Parallel and Distributed Computing*, 14:29-36 (1992).
18. Lee, Ann K. *An Empirical Study of Combining Communicating Processes in a Parallel Discrete Event Simulation*. MS thesis AFIT/GCS/ENG/90D-08, Air Force Institute of Technology, 1990.
19. Mehl, Horst. "Speed-Up of Conservative Distributed Discrete Event Simulation Methods by Speculative Computing (Short Method)." *Proceedings of the SCS Multi-conference on Distributed Simulation* 23. 163-166. Jan 1991.
20. Misra, J. "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, 18:39-65 (Mar 1986).
21. Nutt, Gary J. "Distributed Simulation Design Alternatives." *Proceedings of the SCS Multi-conference on Distributed Simulation* 22. 51-55. Mar 1990.
22. Preiss, Bruno R., et al. "Null Message Cancellation in Conservative Distributed Simulation." *Proceedings of the SCS Multi-conference on Distributed Simulation* 23. 33-38. Jan 1991.
23. Proicou, Michael C. *A Distributed Kernel for Simulation of the VHSIC Hardware Description Language*. MS thesis AFIT/GCS/ENG/89D-14, Air Force Institute of Technology, 1989.
24. Reynolds, Paul F. "A Shared Resource Algorithm for Distributed Simulation." *Conference Proceedings of the Ninth Annual Symposium on Computer Architecture*. 259-266. Apr 1982.
25. Reynolds, Paul F. "A Spectrum of Options for Parallel Simulation." *Proceedings of the 1988 Winter Simulation Conference*. 325-332. Dec 1988.
26. Reynolds, Paul F. "Comparative Analysis of Parallel Simulation Protocols." *Proceedings of the 1989 Winter Simulation Conference*. 671-679. Dec 1989.
27. Reynolds, Paul F. "SRADS with Local Rollback." *Proceedings of the SCS Multi-conference on Distributed Simulation* 22. 161-164. Jan 1990.
28. Su, Wen-King and Charles L. Seitz. "Variants of the Chandy-Misra-Bryant distributed discrete-event simulation algorithm." *Proceedings of the SCS Multi-conference on Distributed Simulation* 21. 38-43. Mar 1989.

Vita

Captain Prescott J. Van Horn was born in Beaumont, Texas, on 28 January 1960. He obtained a Bachelor of Science Degree in Chemical Engineering from Tri-State University, Angola, Indiana, in December 1982. After getting his commission in the United States Air Force through the Officer Training School in August 1984, he attended St. Louis University in the Air Force Institute of Technology's Basic Meteorology Degree Program. In August 1985, he was assigned to the Air Force Global Weather Central, Offutt AFB, as Chief of Software Maintenance for Meteorological Displays, and later as lead software engineer and Air Weather Service acquisition liaison officer for the Automated Weather Distribution System (AWDS) program. In April 1990, he went to Osan AB, Korea, as the Wing Weather Officer to the 51st Tactical Fighter Wing, and Mission Planning Weather Officer to the 36th Tactical Fighter Squadron "The Fabulous Flying Fiends". He again entered the Air Force Institute of Technology in May 1991.

Permanent address: 3017 Chimneywood Dr
Floyds Knobs, IN 47119

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE December 1992	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE Development of a Protocol Usage Guideline for Conservative Parallel Simulations			5. FUNDING NUMBERS	
6. AUTHOR(S) Prescott John Van Horn, Capt, USAF				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583			8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/92D-19	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Capt Steve Suddarth AFOSR/NM Bldg 410 Bolling AFB Washington, D.C. 20332-6448			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The objective of distributed simulation is to speedup simulation execution by partitioning the simulation processing load over multiple processors. This thesis reviews current synchronization protocol methods for distributed simulations, and proposes guidelines for obtaining optimal conservative simulation partitionings using empirical evidence. An analysis is performed using three protocol variations of the Chandy-Misra NULL message algorithm, two using a pending message blocking strategy, and the other using a safetime blocking strategy. A fourth protocol evaluated is based on the SRADS algorithm proposed by Reynolds. The analysis involves a study of all possible 2 and 4 node configurations, for three queuing simulations, using all possible protocol and model pairings. A fourth queuing model is then used to independently validate results. In the end, the safetime version of the Chandy-Misra protocol is demonstrated to provide better overall performance than the other protocols evaluated. Partitioning guidelines developed established a relationship between process configurations and load balancing. It is seen that separating highly communicative processes onto different nodes, or locating highly communicative processes on the same node with fewer processes, provided the optimal 4 node configurations, while reducing the number of intra-node process connections provided for the optimal 2 node configurations.				
14. SUBJECT TERMS Parallel Simulation, Discrete Event Simulation, Distributed Simulation			15. NUMBER OF PAGES 117	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	