CHAPTER 5: THE CARNIVAL SIMULATOR SUITE

5-1. Introduction

Our experiments consisted of generating multi-process traces whose characteristics were based on the grid and simulating each trace in every possible configuration. In this chapter, we describe the software used to generate and run the experiments. In the course of the descriptions we will also discuss a number of assumptions and policy choices we made, which are embedded or implemented in the software. In Section 5-2, we describe Carnival, a simulator of multi-process, communicating, distributed computations. In Section 5-3, we describe GENACTIONS, which generates traces based on certain input parameters and random number generators. The input parameters to GENACTIONS include the probabilities of generating each type of trace action; these probabilities are computed from grid coordinates by GGAP (Generate GenActions Parameters), described in Section 5-4. The trace summarizer SUM, described in Section 5-5, reads a trace and prints a summary of it. The summary can be used to check that the actual characteristics of the trace are similar to the desired characteristics specified by the parameters. Summary data is also used in the analyses of the experiment outcomes. In Section 5-6, we investigate whether strings of consecutive execute actions in traces are a problem. In Section 5-7, we discuss the lengths of the traces. Section 5-8 is a summary of the chapter.

5-2. The Carnival Simulator

5-2.1. Introduction

Carnival is a highly parameterized simulation system which simulates the execution of multiple processes on multiple, possibly heterogeneous processing nodes. Processes use CPUs, perform I/Os, and send and receive messages. Carnival also simulates interference by other processes, not part of the distributed computation being simulated, as they compete for CPUs, I/O facilities, and communication bandwidth.

The inputs to Carnival include

- a file of parameters describing the computing environment
- descriptions of the heterogeneous processing nodes
- the assignment of processes to processing nodes
- a trace of actions taken by the processes to be simulated
- a trace of the outside loads on each CPU, I/O facility, and communication medium.

Carnival produces extensive output reports, listing service times and elapsed times for each processor, I/O facility, and communication medium, broken down by process and by facility.

Carnival is written in the C programming language, using the CSIM library of simulation subroutines [CSIM88]. The Carnival simulator is not itself a distributed program.

5-2.2. General Overview

Carnival simulates the operations of multiple processes. Each process is assigned to one processing node (also called a processor or CPU); there is no provision for process migration. CPUs may be heterogeneous, that is, they may have different speeds. Each processing node has one I/O facility; processes may use only the I/O facility of the processing node to which they are assigned. There is one communication facility which connects all processing nodes. The speed of communication is the same among all pairs of nodes.

Parameters describing the computing environment are read from files (or from the command line). There are many of these parameters, allowing a wide variety of distributed computer systems to be easily simulated.

An input file contains the sequence of actions to be simulated for each process. Processes may request CPU service, they may read or write to or from their I/O facility, they may send messages to other processes and receive messages from other processes, and they may terminate. These *actions* are the fundamental units of simulation activity in Carnival.

More than one process may be assigned to the same processing node, in which case the processes may have to wait to use the CPU or the I/O facility.

There is one shared communication facility, so communication traffic may have to wait to use it. The communications facility approximately simulates a packet network such as an Ethernet [MeB76], handling one packet from each message being transmitted, servicing messages in round robin order. Packetization is simulated by using round robin scheduling for the communication facility, with the timeslice computed to be one packet time based on the communication facility's bandwidth and packet size, both of which are input parameters. The communication facility is modelled by a single-server queue, so only one packet can be on the network at a time.

A second input file, this one read during simulation, describes time-varying external loads on the CPUs, I/O facilities, and the communication medium. External loads are loads on the facilities not due to the processes of the distributed computation being simulated. External load allows simulation of different, and varying, environment conditions.

The extensive environment description read during start-up, and the time-varying external load feature, as well as the trace-driven main simulation, together make Carnival very flexible.

The following basic assumptions are built into Carnival, and cannot be altered by the input files:

- there is one I/O facility per processing node
- there is one communication facility, and it connects all processing nodes
- all messages are from one sender process to one receiver process (no broadcast or multicast)
- communication speed is the same between all pairs of processing nodes
- CPUs use round robin scheduling
- all CPUs have the same timeslice, although the timeslice is a parameter and can be changed
- I/O facilities (disks) use first-come-first-served scheduling
- the communication facility handles one packet from each message, servicing messages in round robin order

Carnival goes through three phases of execution: an initialization phase, a simulation phase, and a reporting phase. During initialization, Carnival reads information about the distributed computer system to be simulated from input files. During the simulation phase, Carnival simulates the actions of the processes and the external loads; the simulation phase ends when all of the processes being simulated have terminated. In the report phase, Carnival generates extensive reports about what happened during the simulation.

5-2.3. Initialization

During its initialization phase, Carnival reads information which provides a description of the distributed computing environment Carnival is to simulate and controls Carnival's operations.

5-2.3.1. The Parameter File

The first file read during initialization contains a number of parameters, some of which control Carnival's operations, others of which describe the computing environment; it is called the *parameter file*. Figure 5-1 shows the parameter file used in our experiments.

```
# parameter file for Carnival.

# control switches (1 = on, 0 = off)
#
cpuLoadSwitch        0
ioLoadSwitch         0
commLoadSwitch       0
commCpuSwitch        1
commDelaySwitch      1

# input files
#
assgnmntFileName     arg 2       # carnival paramfile assgnmntfile actionfile
cpuSpeedFileName     cpuspeed
actionFileName       arg 3       # carnival paramfile assgnmntfile actionfile
loadFileName         load

cpu_timeslice        0.01

io_cpu_time          2.0
io_blocksize         8192
io_latency           30.0
io_transfer          8.0

send_rmt_cpu_time    2.0
send_lcl_cpu_time    1.5
rcv_cpu_time         1.0

comm_latency         1.5
comm_bandwidth       10.0
packet_size          1500
```

Figure 5-1:
Carnival Parameter File

The parameters *cpuLoadSwitch*, *ioLoadSwitch*, and *commLoadSwitch* control the external loads placed on the CPUs, I/O facilities, and communication facility, respectively. In our experiments there was no external load and these switches were always off. The other parameters are described throughout the rest of this section.

5-2.3.2. CPU Speed Declarations

The second file read during initialization, the *cpuspeed file*, gives the number of processing nodes available and the relative speed of each processor.

Carnival does not simulate individual CPU instructions, but only the amount of processor time required for tasks, so heterogeneity becomes a matter of how much time processors of different speeds require to perform tasks. Thus, specifying relative processor speed is sufficient information to allow the simulation of heterogeneous CPUs.

CPU service requests are given in terms of the amount of time a nominal standard CPU would require to accomplish the task. CPU speeds are stated relative to the nominal standard, so a speed of one is normal or standard. A CPU which is twice as fast as the standard would have a speed of two, three times as fast, three, and so on. A CPU which is half as fast as the standard would have a speed of 0.5.

Our experiments involved only identical processing nodes, so all CPUs were given a speed of one.

5-2.3.3. Process Assignments

The third file read during initialization is the *assignment file.* Carnival simulates the execution of each process on one processing node; there is no process migration. This file tells Carnival the processing node on which each process is to be executed. Some processing nodes declared in the cpuspeed file may have no processes assigned to them.

In our experiments, there was a different assignment file for each configuration.

5-2.4. Simulation

After completing the initialization phase, Carnival moves into the simulation phase during which the actions of each process are read from the action file and simulated. In addition, Carnival allows simulated external load to be placed on the CPUs, disks, and network; since this feature was not used in our experiments it is not described.†

In the following sections we describe how processes are simulated, the actions processes can take, and the action trace file. We then describe in more detail how the CPUs and the communication facility are simulated.

5-2.4.1. The Life of a Process

The main program of Carnival starts an internal simulator ‘‘process’’ (a CSIM process) for each process to be simulated, then waits for each CSIM process to signal that the process it is simulating has terminated. For each process being simulated, Carnival reads actions from the action file and simulates them, one action at a time, until a ‘‘terminate’’ action for the process is encountered. The simulation time at which each process terminates is recorded; the overall response time for the program being simulated in the configuration simulated is the termination time of the last, longest-running process.

5-2.4.2. The Action Trace File

Carnival reads two files during the simulation phase: the *action file,* to obtain actions for the processes being simulated, and the *load file,* to obtain load records for the external load managers. In this section we describe the action file.

Actions for all the processes being simulated are interleaved in the trace file. Each action includes the process number of the process which will take the action, the action type, and whatever action parameters are required.

As an example, Figure 5-2 shows a simple but complete action file for three processes.

† The external load feature will be described in a forthcoming report on Carnival.

```
# action trace file

#    proc    action       action parameters

     0       io           read          1286
     1       receive
     1       exec         9.9495695
     0       exec         186.306839
     1       send         2             8169
     0       send         1             3120
     2       exec         56.994995
     2       receive
     2       receive
     0       send         2             7838
     2       exec         12.844194
     1       terminate
     2       io           write         3198
     2       terminate
     0       terminate

# total number of actions: 15
```

Figure 5-2:
Example Action File

The actions are described in the following sections.

5-2.4.3. Actions

There are five types of actions: ‘‘execute,’’ ‘‘io,’’ ‘‘send,’’ ‘‘receive,’’ and ‘‘terminate.’’ In the following sections, we describe each type of action.

5-2.4.3.1. Execute Actions

An execute action is a request to execute on the CPU for a certain amount of service time. The single parameter is the CPU service time requested. The request is for the amount of service required on a nominal standard CPU; the requested service time is adjusted to account for the speed of the CPU actually being used.

5-2.4.3.2. Input/Output Actions

An I/O operation is a read or a write of a certain number of bytes of data; the first action parameter specifies whether the operation is a read or a write, and the second specifies the number of bytes of data to be read or written.

Setting up and beginning an I/O operation, and cleaning up afterward, each require some CPU service. The amount of CPU time required is specified by the parameter *io_cpu_time* from the parameter file. The CPU is used for half of that time before the I/O to set-up the I/O, and half afterward to clean-up after the I/O. In our experiments, *io_cpu_time* was 2.0 ms.

The I/O facilities, disks, are scheduled first-come-first-served (FCFS). Real disks often have complex scheduling algorithms, but since real disk queues are usually empty, we believe that simple FCFS is an adequate model.

The delay in performing a read consists of a wait to use the I/O facility (FCFS), plus one latency time (which includes seek delay and rotational delay), plus one disk-to-memory transfer time per block of data to be read. The number of blocks read is computed from the number of bytes to be read and the block size parameter, counting any fractional block as a full block. Latency time, transfer time, and block size are parameters read from the parameter file during initialization. The parameter names, and the values used in our experiments, are

io_cpu_time	2.0 ms
io_latency	30.0 ms
io_transfer	8.0 ms
io_blocksize	8192 bytes

Carnival does not model read-ahead.

In most systems, data to be written out to disk is not actually written to disk immediately, rather, the data to be written is moved or copied to an in-memory disk block cache. At some later, unpredictable time, the block is actually written to the disk. Thus, a process performing a write operation does not have to wait for a disk access. Carnival models zero I/O delay for writes, making the assumption that the time to move the data to the buffer cache is included in *io_cpu_time.*

5-2.4.3.3. Send Actions

Sends and receives are different action types (as opposed to reads and writes, which are both I/O actions).

A send operation sends one message of any length from the process which executes the send operation to the destination process' incoming message queue, or mailbox. The destination process and the amount of data to be sent, in bytes, are parameters of the send action. When the message arrives at the destination process, the process may read the message immediately, it may read it later, or it may never read it. Communication is asynchronous; sending a message does not imply a rendezvous between the sending and receiving processes.

The destination process may have been assigned to the same processing node as the sender, in which case the send is a *local send*. If the destination process is on a different processing node than the sender, then the send is a *remote send*.

All that is required for a local send is a certain amount of CPU service to copy or move the message from the sending process to the destination process. The amount of CPU time required is given by the parameter *send_lcl_cpu_time.*

A remote send requires a certain amount of CPU service on the sending CPU to hand the message off to the communication facility (for example, to an Ethernet controller). The amount of CPU time required is given by the parameter *send_rmt_cpu_time.* The simulation of the communication facility is described below.

If the CPU service required for local and remote sends appears in the trace as separate execute actions, the parameter *commCpuSwitch* can be turned off, in which case Carnival will not simulate use of the CPU for send or receive actions. In our experiments, *commCpuSwitch* was on, and the parameters had the values

send_rmt_cpu_time	2.0 ms
send_lcl_cpu_time	1.5 ms

The time at which each message is sent is recorded, so that the elapsed end-to-end time until the message to be received can be computed.

Sends are non-blocking. A local send operation is complete, and the sending process may continue to the next action, as soon as the message has been placed in the receiver's mailbox; this requires only a certain amount of CPU time. A remote send operation is complete when, after some CPU service time, the message has been handed off to the communication facility. Thus, completion of a remote send action may occur earlier than the time the message starts to be placed on the communication medium, and it may occur much earlier than the time the message reaches the receiver's mailbox.

5-2.4.3.4. Receive Actions

A process performing a receive operation attempts to retrieve a message from its incoming message queue, or mailbox. If there are messages in the mailbox, the receive operation removes the first message from the mailbox and returns it, without blocking. If there are no messages in the mailbox, the process will block until a message arrives, at which time the process is unblocked, and the message is removed from the mailbox and returned.

There are no parameters for the receive operation. Receive always returns the first available message. There is no way to select among the messages in the mailbox, nor is there any way to check whether there are messages in the mailbox.

Receive operations require a certain amount of CPU service at the receiving processing node to copy or move the message from the mailbox into the process. The amount of CPU time required is given by the parameter *rcv_cpu_time,* which had the value 1.0 ms in our experiments.

If the CPU service required for receives appears in the trace as separate execute actions, the parameter *commCpuSwitch* can be turned off, in which case Carnival will not simulate use of the CPU for send or receive actions. In our experiments, *commCpuSwitch* was on.

The time spent waiting to receive a message is recorded for each receive action. When there is a message in the incoming mailbox the waiting time is zero. The elapsed end-to-end time of a message includes sending and receiving CPU service, communication transit time, and time spent in the receiver's incoming mailbox.

5-2.4.3.5. Terminate Actions

Simulation of a process ends when a terminate action is encountered; no more actions will be simulated for that process. The simulation phase ends when all processes have terminated.

5-2.4.4. The Facilities

5-2.4.4.1. Simulating the CPUs

Processes may use the CPU because CPU service is explicitly requested by an execute action, or to satisfy an implicit requirement for CPU service due to an I/O, a send, or a receive action. Each process uses only the CPU of the processing node to which it was assigned. The amount of CPU service required may come from the execution time action parameter or from an environment parameter such as *send_rmt_cpu_time.*

Service times stated in the action file and the parameter file are the amount of time a task would require on a nominal CPU with a relative speed of one. The raw CPU service time is adjusted to account for the relative speed of the CPU which will actually be used, according to the formula

$$\text{adjServiceTime} = \text{rawServiceTime/speedOf(CPU)}$$

For example, the adjusted service times on a CPU which is twice as fast as the standard CPU (speed two) will be half the raw times. The adjusted times on a CPU which is half as fast as the standard (speed 0.5) will be twice the raw times.

The CPUs are scheduled according to a round-robin discipline with timeslice; see [CSIM88] for a fuller description of the discipline. The timeslice is given by the parameter *cpu_timeslice,* which is read from the parameter file during initialization. In our experiments, *cpu_timeslice* was 0.01 ms.

Adjusted service times and elapsed times are recorded, both total and by requesting action. This information is reported at the end of the simulation.

5-2.4.4.2. Simulating Communication

Carnival assumes that there is one communication medium connecting all processes, as is the case with Ethernets and rings. The communication medium is scheduled according to a round robin discipline with timeslice (see [CSIM88]); this models the increased latency wait due to contention for the medium when traffic is heavy. The timeslice is computed to be one packet-time at the real bandwidth, using the packet size given as a run-time parameter, according to the formula

comm_timeslice = packet_size/bandwidth

where the timeslice is in milliseconds, the packet size is in bytes, and the communication bandwidth is in bytes/ms. The bandwidth in bytes/ms is computed from the communication bandwidth given in the parameter file, which is stated in megabits per second. The parameter names, and the values used in our experiments, are

comm_latency	1.5 ms
comm_bandwidth	10.0 Megabits/sec
packet_size	1500 bytes

There turned out to be a bug in the conversion code, so that the bandwidth in bytes/ms is ten times larger than it should be, so 10 Mb/sec is used as though it were 100 Mb/sec. We had originally intended to model existing Ethernets at 10 Mb/sec; instead we found ourselves modelling FDDI rings at 100 Mb/sec.

For local sends, the local CPU is used for *send_lcl_cpu_time* if *CommCpuSwitch* is on, and the message is then placed in the destination process' mailbox. For remote sends, the sending process' CPU is used for *send_rmt_cpu_time* if *commCpuSwitch* is on; what happens to the message next depends on the *commDelaySwitch* parameter which is read at initialization time; in our experiments, *commDelaySwitch* was always on. If the switch is off, communication medium delays are not simulated and the message is placed in the destination process' incoming mailbox immediately. If the switch is on, then the service time for which the message will need use of the communication medium is computed according to the formula

comm_latency + (message_size/bandwidth)

The parameter *comm_latency,* stated in milliseconds, is read from the parameter file at initialization. In our experiments *comm_latency* had the value 1.5 ms. *Message_size* is an action parameter given in bytes, and bandwidth, described above, is in bytes/ms. This amount of service is then requested from the communication medium to transport the message. Thus all remote messages compete for the communication medium on a packet-by-packet basis. When service is complete, the message is added to the destination process' mailbox.

This is a fairly general model of communication, and can be used to simulate a variety of communication media. At the same time, it does not model certain details, for example, it does not model lost bandwidth due to collisions that occur in Ethernets, nor the back-off rescheduling used in Ethernet controllers; it does not model the time taken to handle the token in a token ring. Headers and trailers are not added to messages, so the message size action parameter should specify the total number of bytes sent over the wire, including headers, trailers, checksums, and so on. Time to compute checksums and so on is modelled by the CPU use required to send a message.

The sending process is delayed only by the sending CPU time (either local or remote); it is not delayed by the amount of communication time. Rather, the message is handed off to the communication system, and the sending process is free to proceed independently. The communication system does not consume additional CPU service at either the sending or the receiving CPU.

The communication system is not checked in deciding whether to terminate the simulation, so there could be messages in transit at termination.

5-2.5. Output Reports

After the simulation phase, Carnival generates several reports. There are reports on the use of each facility (CPUs, disks, and the communication facility), and there are a number of reports on communication activity. If the external load switches are on, reports on the external load are generated.

In the following sections we describe the output reports. These reports represent the outcomes of the experiments, and are the data used to derive the results described in Chapters 6, 7, and 8. Since Carnival's external load feature was not used in our experiments, the external load reports are not described here.†

5-2.5.1. Reports on Use of Facilities

Carnival prints separate reports for each of the three types of facilities: CPUs, I/O facilities, and the communication facility. The reports are similar in form; each has two sections: facility use by each process, and total usage of the facility. In the following sections we define the data reported, describe how the data for the reports is gathered and how the statistics are computed, and more fully describe the reports.

5-2.5.1.1. The Data Reported

The facility reports include the following information: number of service requests, total service time and total elapsed time, and mean service time and mean elapsed time. Numbers of service requests are called ‘‘counts.’’

For a particular process, *service time* is the amount of time the process spent actually receiving service from the facility, as opposed to time spent waiting in a queue. Service time for a facility is the total amount of service time received by all processes using the facility.

Elapsed time for a particular process is the amount of (simulated) time that passed while the process was trying to obtain service; elapsed time includes time spent waiting in queues and service time. Elapsed time for a facility is the total amount of simulation time that passed while processes were trying to obtain service, including waiting times and service times.

5-2.5.1.2. Data Collection and Computation of Statistics

During simulation, each use of a facility by a process is noted; the number of times the facility is used is counted, and both service time and elapsed time are recorded.

Use of a CPU is recorded not only by CPU and process, but also by the type of action for which the CPU is used (exec, io, send, or receive). The total CPU use for communication is also recorded; this is the sum of CPU uses for send and receive actions. In addition, total CPU use for all action types is also recorded.

† See the forthcoming report on Carnival, which will include information on the external load feature and reports.

The service times and elapsed times are recorded in CSIM *table* data structures, using the CSIM *record* operation. The results in the reports are generated by the CSIM *table* functions *table_cnt, table_sum,* and *table_mean.* (See [CSIM88].)

5-2.5.1.3. Report on CPU Use

Figure 5-3 shows an example of a report on CPU use. The first section contains the per-process information, the second reports the totals over all processes for each facility.

Report on CPU use:

Proc	CPU	type	what	execute	I/O	send	receive	comm tot	total
0	0	svc	sum	5.0000	0.0000	0.0000	1.0000	1.0000	6.0000
1	1	svc	sum	0.0000	0.0000	2.0000	0.0000	2.0000	2.0000
0	0	svc	cnt	1	0	0	1	1	2
1	1	svc	cnt	0	0	1	0	1	1
0	0	svc	mean	5.0000	0.0000	0.0000	1.0000	1.0000	3.0000
1	1	svc	mean	0.0000	0.0000	2.0000	0.0000	2.0000	2.0000
0	0	elpsd	sum	9.9900	0.0000	0.0000	2.0000	2.0000	11.9900
1	1	elpsd	sum	0.0000	0.0000	2.0000	0.0000	2.0000	2.0000
0	0	elpsd	cnt	1	0	0	1	1	2
1	1	elpsd	cnt	0	0	1	0	1	1
0	0	elpsd	mean	9.9900	0.0000	0.0000	2.0000	2.0000	5.9950
1	1	elpsd	mean	0.0000	0.0000	2.0000	0.0000	2.0000	2.0000

CPU	type	what	execute	I/O	send	receive	comm	total
0	svc	sum	5.0000	0.0000	0.0000	1.0000	1.0000	6.0000
1	svc	sum	0.0000	0.0000	2.0000	0.0000	2.0000	2.0000
2	svc	sum	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0	elpsd	sum	9.9900	0.0000	0.0000	2.0000	2.0000	11.9900
1	elpsd	sum	0.0000	0.0000	2.0000	0.0000	2.0000	2.0000
2	elpsd	sum	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Figure 5-3:
Example Report on CPU Use

5-2.5.1.4. Report on I/O Facility Use

The report on I/O facility use is very similar to the report on CPU use, except that the use is not broken down by action type; I/O facilities are used only for I/O operations.

5-2.5.1.5. Report on Communication Facility Use

The report on communication facility use is also very similar to the reports on CPU and I/O use, but it is simpler, as there is only one communication facility. Use is not broken down by action type, as all communication facility use is for communication operations. Note that this report only covers use of the communication facility; there are separate reports on communication activity.

5-2.5.2. Reports on Communication Activity

There are six reports which describe the communication activity that took place during simulation:

Messages Sent Counts
Messages Received Counts
Receive Waiting Times — Sums
Receive Waiting Times — Means
End-To-End Communication Times — Sums
End-To-End Communication Times — Means

In the following sections we describe each of these reports. Following that, we describe how the data for the reports is gathered and how the statistics in the reports are computed.

5-2.5.2.1. Message Counts

The *messages sent counts* are counts of the number of messages sent by each sending process to each destination process. Similarly, the *messages received counts* are counts of the number of messages received by each receiving process, broken down by sending process. Simulation can end with messages in transit and with unreceived messages in processes' mailboxes, so the number of messages sent by process A to process B may not be the same as the number of messages received by process B from process A. The reports also show subtotals for each sending process and each receiving process, and a grand total number of messages.

5-2.5.2.2. Receive Waiting Times

Receive waiting time is the amount of time a process spends waiting for a message to arrive, that is, the (simulated) time between starting a receive action and the arrival of a message; it does not include the receiving process' CPU use. If there is already a message in the receiving process' mailbox when it starts the receive operation, then the receive waiting time is zero.

The sum of the receive waiting times for each pair of sending and receiving processes are reported, with subtotals for each sending process and each receiving process, and a grand total time all receivers spent waiting for messages from all senders. Similarly, the mean receive waiting times are reported for each pair of sending and receiving processes, with averages over each sending process and over all receiving process, and a grand overall mean time spent waiting for each message over all senders and receivers.

5-2.5.2.3. End-to-End Communication Times

End-to-end communication time is the time it takes a message to get from the sending process to the receiving process. More precisely, it is the (simulated) time elapsed from when the sending process starts the send operation until the receiving process completes the receive operation, including delays due to sender CPU use, receiver CPU use, and delay through the communication medium. It also includes any time the message spends in the receiver's mailbox ready to be received, waiting for the receiving process to do a receive operation.

As with communication waiting times, both sums of end-to-end times and mean end-to-end times are reported, for each pair of sending and receiving processes, totals or means over each sending process and each receiving process, and grand totals or overall means over all processes.

5-2.6. Summary

Carnival is a flexible simulation tool for simulating the execution of multiple, communicating processes in a distributed, heterogeneous computer system with outside load. Carnival reads a description of the computer system to be simulated, the speeds of the processing nodes' CPUs, and the assignment of processes to processing nodes. It simulates process' actions, including executing on the CPU, doing I/O, and sending and receiving messages. When simulation of all processes is complete, reports are generated, showing service time and elapsed time for each process for each facility, and showing the number of messages sent and received, the total and mean receive waiting times, and the total and mean elapsed end-to-end times for every pair of processes, with subtotals or means for every sending process and every receiving process and grand totals or means over all processes.

We have used Carnival to run our experiments, the results of which are discussed in Chapters 6, 7, and 8.

Carnival reads traces, which describe the actions of sets of processes. The program GENACTIONS, described in the following section, generates action traces for simulation by Carnival.

5-3. GENACTIONS: Generating Trace Files

5-3.1. Introduction

The GENACTIONS program generates an action file, that is, a multiprocess trace file, which can be used as input to the Carnival simulator. The characteristics of the traces generated are determined by information read from a parameter file at start-up time and by random number generators. The random number generators are seeded using a timestamp, so each execution of GENACTIONS should produce a different trace.

In the following several sections, we describe how the GENACTIONS program operates to generate action files. Then, in Section 5-3.8, we describe how multiple streams of random numbers are generated and used in GENACTIONS.

5-3.2. The Action File

The Carnival simulator reads an *action file*, a trace of the actions of a group of processes; GENACTIONS generates these action files. The actions processes can take include ''execute,'' meaning execute using the CPU for a certain amount of service time; ''io,'' meaning perform an I/O operation, either a read or a write; ''send,'' meaning send a message to another process; ''receive,'' meaning receive a message if one has arrived, or wait for a message to arrive and then receive it; and ''terminate,'' meaning that the process is finished. The simulation ends when all processes have terminated.

Each line of the action file describes a single action to be taken by one of the processes. Actions for different processes are interleaved in the action file. Each action specifies the number of the process which is to take the action, the type of the action, and any other information required for the action; these additional items of information are called *action parameters.*

5-3.3. General Operation

When it starts, GENACTIONS reads a parameter file describing the desired characteristics of the trace to be generated. Because GENACTIONS uses random numbers in generating the trace, the actual characteristics only approximate the ones specified in the parameter file. The parameter file also contains directives that control GENACTIONS' operation. The parameters and their effects are described throughout this and the following sections.

The body of the GENACTIONS program is a loop, each iteration of which generates an action. The program keeps track of which processes have terminated, that is, for which processes terminate actions have been generated. The main loop terminates when all processes have terminated. There are a number of ways to control when processes terminate, which are discussed below.

In generating an action, GENACTIONS first chooses which process the action will be for. It then generates the type of the action (execute, io, send, receive, or terminate), and any needed action parameters. After each action is generated, the program checks whether there are still unterminated processes; if so, it generates another action.

GENACTIONS keeps track of a number of conditions, for instance, which processes have terminated, that is, for which processes terminate actions have been generated. It also keeps track of the number of messages available in each process' incoming message queue (mailbox). GENACTIONS has no notion of time, nor does it know which processes will be collocated and which remote, so it does not know when a process will take an action, how long the action will take, nor how long it will take a message to travel from the sending process to the destination process' mailbox. Actions are considered taken as soon as they are generated, and messages are considered to have arrived in their destination process' mailbox as soon as the send-message action is generated. GENACTIONS counts the number of messages sent to each process and the number of receives done by each process, so it knows approximately how many messages are available in each process' mailbox. Thus it can tell, approximately, whether a new receive action will cause the receiving process to be suspended waiting for a message to arrive.

In discussing GENACTIONS we will often speak of ''a process terminating'' or ''a process sending a message,'' and so on. The reader should understand this to mean ''the GENACTIONS program has generated a terminate action for the process'' or ''GENACTIONS has generated a send-message action for the process,'' and so on.

In the following sections, we describe in greater detail how GENACTIONS operates to choose the process an action will be for and how the type of an action is selected. Terminate actions are generated using GENACTIONS' *termSync* feature, which is discussed in Section 5-3.6. The generation of action parameters is discussed in Section 5-3.7, and in Section 5-3.8 we go into the way in which random numbers are generated.

5-3.4. Choosing the Process

In generating an action, GENACTIONS first chooses which process the action will be for. The number of processes to be generated, *numprocs*, is read from the parameter file; processes are numbered from zero to $numprocs-1$. A random integer is generated, with equal probability of selecting any integer between zero and $numprocs-1$, inclusive. If, however, the process chosen has terminated, or if it is suspended waiting for a message which has not yet been sent, then that process number is thrown away and another process number is randomly generated. This continues until a process is selected which is neither terminated nor waiting.

5-3.5. Selecting the Action Type

Having chosen the process, GENACTIONS next generates the action type, which is one of ''execute,'' ''io,'' ''send,'' ''receive,'' or ''terminate.''

The parameter file contains two sets of probabilities for action types, one for use when a process has messages in its incoming mailbox ready to be received, called the ''message-ready state,'' and one for when it has no messages available, called the ''regular state.'' The states are distinguished because we assume that processes usually don't send useless messages, and we want to make the probability of generating a receive action higher for processes that have messages ready. (Of course, it is possible to supply the same probabilities for both states.)

The parameter file contains two sets of probabilities, one for the regular state and one for the message-ready state. Each set specifies the probability of generating each type of action; the probabilities in each set should add up to one. Each set of probabilities is transformed into a set of ranges between zero and one. For example, if the probabilities shown in Figure 5-4(a) appeared in the parameter file, they would be transformed into the ranges shown in Figure 5-4(b).

probability of generating execute action:	0.4
probability of generating io action:	0.1
probability of generating send action:	0.2
probability of generating receive action:	0.2
probability of generating terminate action:	0.1

Figure 5-4(a):
Example of Probabilities of Generating Actions

range for execute:	0.0 - 0.4
range for I/O:	0.4 - 0.5
range for send:	0.5 - 0.7
range for receive:	0.7 - 0.9
range for terminate:	0.9 - 1.0

Figure 5-4(b):
Example of Probability Ranges for Generating Actions

To select an action type, a random fraction between zero and one (inclusive) is generated, and the range into which the random fraction falls determines the type of the current action.

There are a number of situations in which a particular action is not acceptable for the selected process:

- If the process is the last process still in existence (the only one not terminated) then a send is not acceptable because processes do not send to themselves and do not send to terminated processes, so there would be no process to which to send a message.
- If the process is the last process running (the last process neither terminated nor suspended in message wait), and there is not already a message ready for it, then a receive is not acceptable because there is no other process which could send the needed message, and the processes would be deadlocked. Carnival would detect this and abort the simulation.
- If the process is the last process running and there are other processes waiting for messages, then a terminate is not acceptable because there would then be no process to send the messages the other processes are waiting for.

As with process selection, if an unacceptable action type is generated is it discarded and another is randomly generated, and this continues until an acceptable one is generated.

5-3.6. The TermSync Feature

Using the method of selecting action types described in the previous section, it is difficult to control how many actions will be generated. In order to get a large number of actions, the probability of terminating must be set very low. In that case, however, the processes in a trace tend to have widely differing numbers of actions. The result is not a trace with *numprocs* number of processes, but rather one with a decreasing number of processes. To avoid this problem, GENACTIONS has a ‘‘termination synchronization’’ feature, called ‘‘termSync,’’ which allows us to control much more finely the number of actions generated. If the parameter *termSyncSwitch* is off (set to zero in the parameter file), then action types are generated as described above, leading to the problems just described.

If the termSync feature is turned on (by setting the *termSyncSwitch* to one in the parameter file), the probabilities of generating terminate actions for both the regular and message states should be set to zero in the parameter file. In fact, GENACTIONS enforces this by adding the probability of terminate to the probability of execute and setting the probability of terminate to zero; this is done for both the regular and message-ready states, separately. This eliminates the possibility of a terminate action being generated ‘‘prematurely’’ for some process; when termSync is on, terminates can be generated only by the termSync mechanism.

The minimum number of actions desired in the trace is given by the parameter *min_actions,* which is used only with the termSync feature. While the total number of actions generated is less than *min_actions,* the type of action is selected by probability ranges, as described in the previous section, but the probability of terminate is zero.

Note that *min_actions* is compared to the total number of actions generated, not the number of actions per process. Different processes can, and usually do, have different numbers of actions, but because there are equal probabilities of generating each process number, the number of actions per process usually does not vary too much within a trace. In our experiments, the termSync feature was always used, and *min_actions* was set to 400 times the number of processes (800 for two-process traces, 1,200 for three-process traces, and 1,600 for four-process traces), except in experiment sets S and T, as described in Sections 6-9 and 6-10.

After *min_actions* actions have been generated, there is a certain probability that each action will be a terminate action. This probability is given by the parameter *termSync_prob*, which should be a fraction less than one. Each time an action type is to be selected, a random number between zero and one (inclusive) is drawn and compared to *termSync_prob*. If the action should be a terminate, (because the random number is less than *termSync_prob*), and if a terminate action is acceptable for that process, then the process terminates (that is, a terminate action is generated for the process). If the action is not a terminate, then another random number is drawn and an action type is selected according to probability ranges, as described above. The larger *termSync_prob* is, the faster the processes will terminate after the minimum number of actions has been generated, and the less variation there will be in the number of actions per process. On the other hand, if *termSync_prob* is too large, it will be difficult to generate other action types which may be needed to clear certain situations; for example, if two processes remain, one of which is waiting for a message, the second process must generate a send to the first before they can terminate. GENACTIONS checks that the *termSync_prob* parameter is less than one. In all of our experiments, the termSync feature has been used, and *termSync_prob* has had the value 0.80.

5-3.7. Generating Action Parameters

The *action parameters* required for an action depend on the type of the action. In this section, we describe the data required for each action type, and how it is generated.

Execute actions require a service time, called the execution time, which is the number of milliseconds of service time for which the process will use the CPU. The execution time is generated from a uniform distribution of real numbers between *exec_time_low* and *exec_time_high* (inclusive). *Exec_time_low* and *exec_time_high* are real number parameters; in our experiments, they had the values 2.0 ms and 200.0 ms, respectively.

The action parameters required for an I/O action are the type of I/O, which is either ‘‘read’’ or ‘‘write’’, and the size of the I/O, which is the number of bytes of data to be read or written. The probability of generating a read is *io_read_prob*; an I/O is either a read or a write, so the probability of generating a write is 1−*io_read_prob*. *Io_read_prob* is a real number parameter; in our experiments, it always had the value 0.66, so that approximately two thirds of I/Os were reads and one third were writes. The I/O size is an integer between *io_size_low* and *io_size_high* (inclusive), and there is an equal probability of generating each integer. In our experiments, *io_size_low* and *io_size_high* had the values 1 and 8,192 (8K), respectively.

A send action requires a destination process number and a message size, which is the number of bytes of data to be sent. GENACTIONS has a clustering feature, which, when turned on, creates clusters or groups of processes which communicate among themselves but seldom communicate outside their group; the clustering feature was not used in our experiments. With the clustering feature turned off, the destination process is an integer chosen with equal probability from all process numbers; destinations that are self-sends or sends to terminated processes are rejected and another destination generated. The message size is generated from a uniform distribution of integers between *send_size_low* and *send_size_high* (inclusive). In our experiments, *send_size_low* and *send_size_high* had the values 1 and 8,192 (8K), respectively.

Receive actions and terminate actions do not have any action parameters. (Recall that a receive action receives the first-arrived message in the incoming message queue; there is no way to select a particular message in the queue.)

Once generated, the action is written to the trace file. If the action generated was a send, the number of messages ready for the destination process is incremented. Recall that GENACTIONS does not take account of the amount of time it takes to carry out actions or for messages to reach their destinations; if a message has been sent, it is considered ready to be received. If the destination process was suspended waiting for a message, then it is unsuspended. If the action generated was a receive, then the number of messages ready for the receiving process is decremented. If there were no messages ready, then the receiving process is marked as suspended. If the action was a terminate, the process is marked as terminated. GENACTIONS then returns to the top of its main loop and decides whether to generate another action or to stop.

5-3.8. Random Number Streams

5-3.8.1. Introduction

GENACTIONS uses multiple streams of random numbers by calling the routines in the ‘‘msrandom’’ package we have written; msrandom is in turn based on ‘‘random,’’ ‘‘setstate’’ and other associated UNIX† routines.

5-3.8.2. GENACTIONS’ Use of Random Numbers

GENACTIONS generates actions for *numprocs* processes; the processes are numbered zero through $numprocs - 1$. Random number streams zero through $numprocs - 1$ are used for generating action types and action parameters for each process; thus the actions of one process have no effect on the actions of other processes. A separate stream (stream number *numprocs*) is used for choosing the number of the process to which the next action will belong. If the communication clustering feature is used, stream number $numprocs + 1$ is used to select the sizes and members of the clusters.

† UNIX is a registered trademark of UNIX System Laboratories, Inc.

5-3.8.3. Generating Multiple Streams of Random Numbers

The UNIX ‘‘random’’ package of routines [KLJ84] allows setting up blocks of data which constitute the state of the random number generator. These state arrays can be of a variety of sizes, and the manual says that the larger the state array, the better the random numbers; we use state arrays of 256 bytes, the largest size allowed. The ‘‘setstate’’ routine tells the random number package which state array to use when the ‘‘random’’ routine is called to generate random numbers. Each state is in effect a different random number generator; generating a sequence of random numbers using one state does not effect the sequences generated using other state arrays.

To get multiple, independent streams of random numbers, GENACTIONS calls an msrandom initialization routine which sets up and initializes a state array for each stream. Thereafter, calls to msrandom random number generating routines include the stream number from which the number is to be drawn; the msrandom routines ‘‘swap in’’ the appropriate state array and call the underlying ‘‘random’’ routine to generate random numbers.

Each state array is initialized with a ‘‘seed,’’ which must be provided as an argument to the ‘‘initstate’’ routine. To obtain different sequences of random numbers from each stream, the seed for each stream should be different. The msrandom package includes an array of ‘‘seed bases,’’ 1,024 32-bit integers, all different, which are used as the basis of the seeds given to ‘‘initstate.’’ The seed base array was obtained by reading 4,096 bytes of data from an arbitrarily selected location in the main memory of one of our local computers.‡ Dividing the 4,096 bytes into 1,024 32-bit integers resulted in 120 pairs of duplicates. The second member of each pair of duplicates was incremented by one; this eliminated all duplicates.

If the seed bases were used as seeds, the msrandom routines would produce the same sequences of random numbers every time they were used. To avoid this, the msrandom initialization routine requires a seed parameter; this parameter is added to the seed base for each stream to obtain the actual seed used to initialize each stream. GENACTIONS uses the UNIX ‘‘time’’ routine to obtain the overall seed parameter; ‘‘time’’ returns the number of seconds since midnight, January 1, 1970. Thus, in any run of GENACTIONS, each stream should provide a different and independent sequence of random numbers from every other stream, and each run of GENACTIONS should have entirely different streams.

5-4. GGAP: Grid Coordinates and Action Probabilities

5-4.1. Introduction

The GGAP program reads grid coordinates and a template file, and produces a GENACTIONS parameter file which will cause GENACTIONS to generate a trace falling into the desired grid cell. GGAP stands for Generate GenActions Parameters. In this section, we discuss the need for GGAP, how the program operates, and the policies embodied in it.

5-4.2. Purpose of GGAP

Experiment sets are based on the grids and grid coordinates, however GENACTIONS parameter files contain probabilities of generating actions, as shown in Figure 5-4(a). Furthermore, the grids have only one communication coordinate, whereas GENACTIONS parameter files need separate probabilities for send and receive actions. Also, the GENACTIONS parameter files contain two sets of probabilities, one for the ‘‘regular’’ state and one for the ‘‘message-ready’’ state, whereas the grid coordinates do not. At the same

‡ We read 4,096 bytes at offset 32,648 from file /dev/mem on vangogh.Berkeley.EDU at 13:57:56 on Friday, August 19, 1988. The address 32,648 was based on some of my family members’ birthdays.

time, the GENACTIONS parameter files contain many parameters which have nothing to do with the grid coordinates. In any experiment set, the grid coordinates, and hence the probabilities of generating each type of action, will be different in every cell, while the other GENACTIONS parameters will be identical. GGAP generates GENACTIONS parameter files based on a template parameter file and a file containing grid coordinates. The GENACTIONS parameter file produced contains action-type probabilities for the regular and message-ready states based on the grid coordinates; all other parameters in the file are copied from the template file. Thus, a single template GENACTIONS parameter file can be prepared for each experiment set, and a file containing grid coordinates can be prepared for each grid cell in the experiment set. Then GGAP can be run on each grid cell, to produce a GENACTIONS parameter file for each cell.

The following section explains how GGAP computes action-type probabilities from grid coordinates, how it separates the single COMM coordinate into send and receive probabilities, and how it generates different probabilities for the regular and message-ready states.

5-4.3. How GGAP Operates

The probability of an execute action is the CPU coordinate, given as a percentage of all actions, converted to a decimal fraction, according to the formula

$$prob.execute = grid.cpu/100$$

The probability of an io action is the IO coordinate fraction of the non-execute actions. The fraction of non-execute actions is (100–*grid.cpu*), and the probability of an io is

$$prob.io = [(100 - grid.cpu) \cdot grid.io]/100$$

Similarly, the probability of a communication action is the comm coordinate percentage of the non-execute actions:

$$prob.comm = [(100 - grid.cpu) \cdot grid.comm]/100$$

The division between send actions and receive actions is always 70% to 30%. For the regular state, the division is 70% sends to 30% receives; for the message-ready state it is 30% sends to 70% receives. As discussed above, we assume that the processes which make up distributed programs are written in such a way that the processes are more or less coordinated, so processes are significantly more likely to do receives when some other process has sent them a message or will soon send them a message, and they are less likely to do a receive if no one has sent them a message or will soon do so. Since GENACTIONS ignores the time delays of performing actions and transmitting messages, its notion of when a process has a message ready is likely to anticipate the point at which the message is really available in the simulation; this covers the ‘‘soon will send a message’’ aspect.

For regular state, the probability of a send action is computed by the formula

$$prob.send = [(100 - grid.cpu) \cdot grid.comm \cdot 70]/100$$

and the probability of a receive is computed by the formula

$$prob.receive = [(100 - grid.cpu) \cdot grid.comm \cdot 30]/100$$

For the message-ready state, the probabilities are reversed.

As an example, Figure 5-5 shows the regular state and message-ready state probabilities corresponding to the grid coordinates 60% CPU, 25% I/O, and 75% COMM, which are the coordinates of cell **q** of the full grid, corresponding to cells **nq**, **uq**, and **aq** of the appropriate experiment sets.

Action-Type	Regular State	Message State
execute	0.60	0.60
io	0.10	0.10
send	0.21	0.09
receive	0.09	0.21
terminate	0.00	0.00

Figure 5-5:
Regular State and Message State Probabilities
Corresponding to the Grid Coordinates
60% CPU, 25% I/O, and 75% COMM

5-4.4. Summary

In this section, we have discussed the conversion of grid coordinates to action-type probabilities, the division of the COMM grid coordinate into send and receive probabilities, and the way send and receive probabilities are raised or lowered depending on whether a process has messages waiting in its incoming message queue. The GGAP program reads a set of grid coordinates and converts them to two sets of action-type probabilities, one for processes in the message-ready state and one for processes in the regular, no-messages-ready state. GGAP produces a GENACTIONS parameter file containing the two sets of action-type probabilities. The rest of the parameters in the output file, in fact, the rest of the text of the output file, is taken from a template parameter file.

We have described the preparation of GENACTIONS parameter files as being done by the GGAP program. In fact, we did this by hand for the first several experiment sets, and only later automated the process with GGAP. The process is essentially the same, whether done by hand or automatically.

5-5. The SUM Program: Summarizing Traces

The SUM program reads a trace file produced by GENACTIONS, and summarizes the characteristics of the processes in the trace. Summaries can be used to check that traces actually have the characteristics they are supposed to, and data from trace summaries are used in a variety of ways in the investigations described in Chapters 6, 7, and 8.

An example summary is shown in Figure 5-6; it is a summary of run **nq-2**, which has two processes per trace, the grid coordinates 60% CPU, 25% I/O, and 75% COMM, and the action-type probabilities shown in Figure 5-5.

number of processes:	2

PROCESS 0:			
number of actions:	404		
number of executes:	239	sum of execution times:	22,633.164
number of io's:	40	sum of io sizes:	197,086
number of reads:	22	sum of read sizes:	107,966
number of writes:	18	sum of write sizes:	89,120
number of sends:	73	sum of send sizes:	306,731
number of receives:	51		
number of terminates:	1		

Send actions from this process:

Destination Process	Number of Sends to Dest. Proc.	Sum of Sizes of Messages to Dest. Proc.
0	0	0
1	73	306,731

proportion of executes:	0.591584
proportion of I/Os:	0.099010
proportion of communication actions:	0.306931
proportion of terminates:	0.002475
sum of proportions:	1.000000

Division between I/O and communication (IO-COMM Division):

proportion I/O:	0.242424
proportion communication:	0.751515

PROCESS 1:			
number of actions:	399		
number of executes:	236	sum of execution times:	23,303.231
number of io's:	38	sum of io sizes:	174,865
number of reads:	27	sum of read sizes:	116,481
number of writes:	11	sum of write sizes:	58,384
number of sends:	51	sum of send sizes:	218,686
number of receives:	73		
number of terminates:	1		

Send actions from this process:

Destination Process	Number of Sends to Dest. Proc.	Sum of Sizes of Messages to Dest. Proc.
0	51	218,686
1	0	0

proportion of executes:	0.591479
proportion of I/Os:	0.095238
proportion of communication actions:	0.310777
proportion of terminates:	0.002506
sum of proportions:	1.000000

Division between I/O and communication (IO-COMM Division):

proportion I/O:	0.233129
proportion communication:	0.760736

Figure 5-6:
An Example Trace Summary
Produced by SUM

As may be seen in Figure 5-6, there is a separate summary for each process in the trace. Reads and writes are sub-types of io's, so the number of reads plus the number of writes should equal the number of io's, and the sum of the read sizes plus the sum of the write sizes should equal the sum of the io sizes. The total number of actions should equal the sum of the number of executes, io's, sends, receives, and terminates. There should be exactly one terminate action per process. The summary of send actions by destination shows that there are no self-sends.

The proportion of executes is computed by dividing the number of execute actions by the total number of actions; the proportions of io's and terminates are computed similarly. The proportion of communication actions is the number of sends plus the number of receives divided by the total number of actions. The sum of the proportions should be one, of course.

In the GENACTIONS parameter files used in our experiments, the probability of a terminate action was always zero because we used GENACTIONS' termSync feature. Real traces, however, always have one terminate action per process. Because of this, and because of the randomness used in generating traces, the actual proportion of each type of action as reported in the summary never exactly matches the GENACTIONS parameters.

The IO-COMM proportions are computed using the total number of actions minus the number of executes as the divisor:

$$propIO = ioCount/(actionCount - executeCount)$$

$$propComm = (sendCount + receiveCount)/(actionCount - executeCount)$$

The use of information from SUM is described throughout Chapters 6, 7, and 8.

5-6. Consecutive Execute Actions

5-6.1. Introduction

The GENACTIONS program which generates the program traces we use operates by randomly chosing the type of the next action according to the proportion of each type of action that is to appear in the trace. Choosing the action type is a memoryless procedure; the type of the previous action is not taken directly into account in selecting the next action type. This means that a trace could contain two or more consecutive execute actions for the same process. (Actions for the different processes in the trace are interleaved, so being physically consecutive is not important; we are discussing sequences of actions for a particular process.)

It is more common to think of a CPU burst (which is what an execute action represents) as a use of the CPU continuing until some other type of activity occurs. The question then arises whether the presence of consecutive executes might have an effect on our results. This is an important question, because one of the grid coordinates is the proportion of execute actions. Furthermore, for our results to be generally useful, others must be able to tell where on the grid their programs fall. This may not be possible if a CPU burst in a program can be modelled as an arbitrary number of CPU bursts in a trace; the sum of the CPU time used would be the same, but the number, and hence the proportion, of CPU bursts or execute actions would be very different.

To see whether consecutive executes are, in fact, a problem, we first checked whether there were in fact many instances of consecutive execute actions, and if so, how long the sequences of execute actions were. We did this for experiment sets U and V (three-process full-grid and zoomed-in grid, respectively; see Chapter 7). Second, for set U, we developed software which replaced strings of consecutive executes with a single execute action, the execution-time (service time) parameter of which is the sum of the execution-times of the whole string, and compared the results of simulating the original traces with the results of simulating the altered traces.

5-6.2. Terminology

Any number of consecutive execute actions for the same process is a *string* of executes. (The execute actions in a string may not be physically consecutive in the trace file, because actions for different processes are interleaved in the file.) The number of consecutive executes in a string is the string's length or size. Consider, for example, the trace shown in Figure 5-7.

#	proc	action	action parameters	
	0	send	1	5831
	2	exec	8	
	2	io	read	5617
	1	exec	1	
	0	exec	2	
	1	exec	3	
	2	exec	8	
	0	exec	4	
	1	io	read	4093
	0	exec	5	
	1	exec	6	
	0	terminate		
	2	terminate		
	1	terminate		

Figure 5-7:
Action Trace Showing Consecutive Execute Actions

In this trace, process 0 has a string of three executes. Process 1 has two strings: first, one of length two, which is broken by an I/O action, then one of length one. Process 2 has two strings, each of length one.

Strings whose length is one are called 1-strings; these are just single execute actions. Strings of length two are 2-strings, strings of length three are 3-strings, and so on. It is these latter strings, whose length is greater than one, that we wish to find; we call such strings *g-strings* (for length-Greater-than-one).

5-6.3. Number and Size of Consecutive Execute Strings

In the following sections we examine the number and length of consecutive execute strings, first in set V, then in set U. For each set, we first examine the number of strings of length greater than one (g-strings); we then examine the distribution of string lengths.

5-6.3.1. Set V

In set V, there are 92 processes which have one or more g-strings, out of 432 processes total (there are 36 cases in the set, four runs per case, and three processes per run), so only 92/432 = 21.3% of the processes have strings of length greater than one. Figure 5-8 shows the distribution of numbers of g-strings per process.

Number of G-Strings	Number of Processes Having That Many G-Strings	Percent of 92	Percent of 432
1	67	72.8%	15.5%
2	24	26.1%	5.6%
3	1	1.1%	0.2%
total	92	100.0%	21.3%

Figure 5-8:
Distribution of Numbers of G-Strings in Set V

No process in set V has more than three g-strings. There are a total of $(1 \cdot 67) + (2 \cdot 24) + (3 \cdot 1) = 118$ g-strings in set V.

Generally, the lower in set V's full grid one looks, the more g-strings one finds. This is not surprising, since there are more execute actions in the lower rows of the grid. In the first row, CPU is zero, so there are no execute actions at all. Figure 5-9 shows the number of processes with g-strings in each row of set V. There are 72 processes in each row of the set V zoomed-in grid (six cases per row, four runs per case, and three processes per run), and Figure 5-9 also shows the percentage of processes in each row that have strings longer than length one.

Row	Number of Processes	Percent of Processes
1	0	0.0%
2	3	4.2%
3	6	8.3%
4	18	25.0%
5	36	50.0%
6	29	40.3%
	92	

Figure 5-9:
Number of Processes with G-Strings by Row in Set V

Overall, only about 20% of the processes have strings longer than one, but in the lower two rows a large proportion of processes have such strings.

Let *maxlen* be the length of the longest string of consecutive executes in a process. In the first row of the grid, where there are no execute actions, the maximum length is zero. In most processes below row 1, maxlen is one. There are 92 processes which have a maxlen greater than one; in 90 of those processes, maxlen is two; in the other two processes, maxlen is three. There are no strings longer than length three.

The average length of strings of consecutive executes is a measure which incorporates both the number of g-strings and their length. If the average length of the strings in a process is close to one, then strings of consecutive executes are not very important in that process. If the average length is significantly greater than one, then strings of consecutive execute actions are an important factor for that process, in which case the average length gives a good indication of how important they are.

From Figures 5-8 and 5-9 we know that there are 92 processes with average string lengths greater than one. Sorting the average lengths and grouping them into ranges we get the results shown in Figure 5-10.

Range of Average Lengths	Number of Processes in Range	
1.037 - 1.095	58	
1.100 - 1.182	30	
1.200 - 1.286	3	
1.500	1	(run **vaj-4**, process 0)
	92	

Figure 5-10:
Distribution of Average Lengths of Strings of Consecutive Executes
in Set V

Most of the average lengths are very close to one. Only four of the 432 processes (0.9%) have average lengths of 1.2 or greater. Even the largest average length is only 1.5, and that figure comes from a process with two strings, one of length one and the other of length two.

In summary, there are not many strings of consecutive execute actions in set V, and the lengths of the strings is quite short. We conclude that consecutive execute actions are not very significant, and are not a problem in set V, and by extension, in the zoomed-in grid.

5-6.3.2. Set U

The results described here were obtained from the subset of set U shown in Figure 5-11.

	IO-COMM									
CPU	IO 0	COMM 100	IO 25	COMM 75	IO 50	COMM 50	IO 75	COMM 25	IO 100	COMM 0
0	**ua**				**uc**				**ue**	
20										
40	**uk**				**um**				**uo**	
60										
80	**uu**				**uw**				**uy**	
100	**uz**									

Figure 5-11:
Subset of Set U Used for Analysis Consecutive Executes

In set U, as in set V, the first row of the grid represents zero percent execute actions, so there are no execute actions and no strings. In set U, the last row, consisting only of case **uz**, is also an anomaly. Case **uz** has 100% execute actions, so the processes in the runs in cell **uz** each consist of exactly one string of consecutive execute actions, whose length is equal to the length of the trace (except for the terminate actions).

There are ten cases in this subset of set U; each case has two repetitions, and each repetition has three processes, so the total number of processes is $10 \cdot 2 \cdot 3 = 60$. As noted, there are no strings in the first row; excluding the first row, there are seven cases and $7 \cdot 2 \cdot 3 = 42$ processes.

Every process in this subset of set U, except those in row 1, has g-strings. The number of g-strings per process in each row of the subset is shown in Figure 5-12.

Row	Number of G-Strings
1	0
3	28 - 49
5	36 - 67
6	1

Figure 5-12:
Number of Strings of Consecutive Executes by Rows in Subset of Set U

As in set V, the lower in the grid one looks the more g-strings per process one finds.

The maxlen, the length of the longest string of consecutive executes in a process, also varies by row, as shown in Figure 5-13.

Row	Maximum String Lengths
1	0
3	4 - 9
5	15 - 34
6	390 - 406

Figure 5-13:
Maximum Lengths of Consecutive Execute Strings by Row in Set U

Figure 5-14 shows the number of strings of each length for rows 3 and 5 of the subset of set U.

Row 3		Row 5	
Length of String	Number of Strings of that Length	Length of String	Number of Strings of that Length
1	1,093	1	220
2	428	2	188
3	135	3	123
4	63	4	106
5	32	5	89
6	14	6	72
7	3	7	58
8	1	8	52
9	1	9	39
		10	30
		11	39
		12	19
		13	21
		14	12
		15	14
		16	7
		17	3
		18	6
		19	3
		20	8
		21	1
		22	3
		23	2
		24	1
		25	0
		26	1
		27	1
		28	0
		29	0
		30	0
		31	0
		32	0
		33	0
		34	1

Figure 5-14:
Number of Strings of Consecutive Executes of Each Length
by Row in Set U

There are many more strings of shorter lengths, and the number of strings of each length falls off rapidly with increasing length. There are far more 1-strings than strings of any other length. Although the maximum string lengths are substantial in rows 3 and 5, most of the strings are fairly short.

The average string length weights the string lengths by the number of strings of each length. The average length of strings of consecutive execute actions in individual processes is between 1.45 and 1.91 in row 3, and between 4.35 and 6.38 in row 5. The average over all processes in row 3 is 1.63, and in row 5 it is 5.16; the average string length over all processes in rows 3 and 5 is 3.00.

Considering that the proportion of execute actions in the third row is 40%, and the fifth row 80%, these average lengths are surprisingly small. Row six is, of course, a special case.

In summary, in the subset of set U we studied, there are moderately large numbers of g-strings and the maximum lengths of the strings are fairly long; these factors would seem to indicate that strings of consecutive execute actions are significant. On the other hand, most of the strings are quite short, and the average lengths are only a few actions; this indicates that strings of consecutive executes are not an important factor. We believe the latter factors are more significant, and that strings of consecutive executes are of little importance in the subset of set U, and, by extension, in the full grid.

5-6.4. Comparing Simulation Outcomes

The real test of whether the presence of consecutive execute actions is important, and whether it is a problem, is if they alter the simulation outcomes. To test whether the presence of consecutive execute strings might change the outcomes, we developed software which finds strings of consecutive execute actions in a process and replaces the multiple execute actions with a single execute action, with the execution-time of the single execute action being the sum of the execution-times of the multiple executes. We then simulated these altered traces in the usual manner.

We carried out this experiment on the subset of set U shown in Figure 5-11. The subset contains ten cases, with two repetitions per case. With three processes, there are five configurations possible, for a total of 100 configurations to be simulated in the subset. In all simulations in the first row, the response time of the altered trace we the same as that of the original trace, as expected.

In no case did the order of which configuration is best, second best, and so on, change. This is the most important consideration.

The change in the simulated response times was quite small. In only six of the 100 configurations was the change in response time greater than 0.1 milliseconds. These six are shown in Figure 5-15.

Run	Configuration	Response Time of Altered Trace (ms)	Difference between Response Times (ms)
uy-2	LONER1	64,328	-0.1468
uy-1	LONER1	67,526	-0.1525
uy-2	TOGETHER	97,891	-0.2277
uy-2	LONER0	66,995	0.2711
uo-2	TOGETHER	51,649	-0.2795

Figure 5-15:
Six Largest Differences in Response Time
between Original and Altered Traces

The differences are computed by subtracting the response time of the altered trace from the response time of the original trace.

The response times of these six configurations are from about 52,000 to about 98,000 milliseconds. The largest change in response time was approximately 0.28 milliseconds, where the response time is approximately 52,000 milliseconds. These changes are quite small in absolute terms, and very small indeed relative to the response times.

Furthermore, the response times of the altered traces are sometimes faster and sometimes slower than the response times of the original traces. The 30 simulations in the top row are neither faster nor slower, in 60 simulations the altered trace is slightly faster, and in ten simulations the original trace is slightly faster. This adds further evidence that the change in response time due to the alteration does not have a systematic trend, but are more in the nature of noise.

Since the ordering of the response times of the configurations did not change, and the changes in response times are very small (especially relative to the response times), we conclude that the presence of strings of consecutive execute action does not seem to be important.

5-6.5. Conclusions

In experiment set V, which is based on the zoomed-in grid, there are not many strings of consecutive execute actions, and the strings which do exist are very short, no more than three execute actions long. We conclude that consecutive executes are not significant in the zoomed-in grid.

In a subset of experiment set U, based on the full grid, rows 1 and 6 are anomalies; row 1 because there are no execute actions in the processes at all, and row 6 because the processes consist of nothing but execute actions (except for the final terminate action), so that the processes consist of one long string. In the remaining part of the subset, rows 3 and 5, there are moderately large numbers of strings of consecutive execute actions, and some of the strings are fairly long. However, most of the strings are fairly short, and the average lengths are short. These results are not conclusive, but we believe that strings of consecutive executes are not a problem in the full grid.

To further explore consecutive executes in the full grid, we replaced strings of consecutive executes with a single equivalent execute action, simulated the resulting traces, and compared the results to the results of simulating the original traces. There were no changes in the order of the response times of the configurations, the changes in response times were very small. Thus, in the most important test, consecutive executes have no significant effect.

We conclude that the existence of strings of consecutive executes in the process traces is not significant.

5-7. Traces and Trace Lengths

The length of traces in number of actions is an important issue. In our experiments, traces contained approximately 400 actions per process, that is, two-process traces contain approximately 800 actions, three-process traces contain approximately 1,200 actions, and four-process traces contain approximately 1,600 actions. Experiment sets S and T were used to investigate whether these are appropriate trace lengths; experiment sets S and T are discussed in Sections 6-9 and 6-10, respectively.

5-8. Summary

In this chapter, we have described the software used in preparing and running our experiments. Specifically, we have described the Carnival simulator; GENACTIONS and GGAP, which together are used to generate traces from grid coordinates and a template parameter file; and SUM, a program which summarizes the characteristics of traces. We have also discussed a set of experiments with which we investigated the effect of consecutive execute actions in traces; we found that consecutive executes are not a problem. Finally, we mentioned the issue of trace length and referred to Sections 6-9 and 6-10, in which trace length is discussed.

This concludes the introductory material, background, and description of our experimental approach and software apparatus. In Chapters 6, 7, and 8 we describe the outcomes of the experiments and what they mean.