Distributed Simulation, No Special Tools Required*

Frank Paterra, C. Michael Overstreet, and Kurt Maly

March 30, 1990

Abstract

In this paper the authors present a toolkit of C language functions that can be linked with SIMSCRIPT programs to provide the data communication primitives necessary for distributed simulation. The authors' test case is discussed and some timing data are presented. Additionally some metrics, developed to determine the applicability of the server model decomposition for particular simulations, are discussed.

1 Why Distribute Simulations

Computer simulations are often computationally intensive tasks requiring long runs in order to obtain useful results. The runtime requirements of a simulation model can be a problem both during model development and validation and while performing production runs of the simulation.

The development of computer models to simulate a real world objects is a well understood problem and number of tools exist to aid the model developer [1]. Often times, the initial runs of a simulation model provide more questions than answers and the focus of study is changed. This results in an evolutionary process for model development, with refinements directed at different attributes as the object or its environment becomes better understood. Often the complexity of the model also increases during this process.

As the model is evolving, many runs may be needed to better understand the object and to verify the correctness of the simulation. The runtime requirements of complex models can greatly increase the time needed to

*This work was supported in part by CIT under grant INF-89-002-01, by NASA under grant NAG-1-908, and Sun Microsystems under RF596043.

Once a model has evolved to the point that production runs are being made, the runtime requirements again come into play. Often the output from each run may only consist of a single data point for a graph. In this case multiple runs of the same simulation with different inputs are needed. This can force the investigator to reduce the number of data points collected in order to reduce the time needed to generate a graph, resulting in incorrect conclusions about the simulated object.

Complementary to the problem of long, computationally intensive, runtimes is the fact that many times other computers are sitting idle and can provide basically free processor cycles to the simulation. In an effort to utilize some of these free cycles, much research has gone into developing algorithms for performing a single simulation on a number of losely coupled, cooperating processors.

The idea of distributing a simulation model among cooperating processors involves difficult problems. Principal among these are the identification of an effective decomposition of the simulation model, and the maintenance of processor synchronization to insure that the program is being executed in the correct order. The use of very tightly coupled functions and dependence on shared data, common in simulation programs, makes these problems are very acute to distributed simulation.

Significant research has gone into these two problems and the results are promising depending on the model being simulated. It is not our intention to address these problems in this paper, but rather to select an effective decomposition and synchronization scheme that will be used to demonstrate distributed simulation using our communication toolkit and standard simulation and operating system tools. A comprehensive treatment of the processor synchronization and model decomposition problems can be found in [2,3,4]. The problem of processor synchronization is more easily solved in very tightly coupled processors that support very high speed communication [5,6].

2 Model Decomposition

The model decomposition most easily supported by the tools here is to distribute some special types of model components, here called servers and receivers, on different machines. The term server is borrowed from object oriented design: a component is a server submodel if it can be represented as only sending to other model components.

This decomposition can be thought of as a collection of data servers and and receivers with no cycles. With no cycles synchronization becomes easier and the problems with deadlock such as that described in [7] is avoided. This is an easy and usually useful decomposition for complex, tightly coupled models, because the extensive data interaction among the model's parts are not interfered with. Other, potential more effective decompositions are outside the scope of this paper.

High performance scientific workstations sharing a LAN are becoming common. Since shared memory is not available and message passing among workstations in the network is slow, a decomposition of the simulation task is only likely to to be effective if messages are infrequently passed among workstations. This toolkit has been developed with these constraints in mind.

The tools support a "warehouse" approach. Information to be sent to receivers is "batched" and sent periodically as a single large message rather that as several smaller messages. In addition, the receiver workstation maintains inventories of data from servers, and based on anticipated consumption "orders" additional data *periodically* so that *new* data *should* arrive *before* current supplies are exhausted.

If data provided by the servers requires significant computation and data sent to the receivers also requires computing time, than significant parallelism can result since the required computation is off loaded from the receiver workstation. No possibility of deadlock exists in this approach and synchronization is particularly simple.

3 Distributed Simulation with Standard Tools

In this paper we present a toolkit of functions that allows distributed simulation to be carried out in a loosely coupled, general purpose, workstation environment without the use of special purpose operations systems, programming languages, or hardware.

The environment for which this software was developed contains a collection of Sun workstation computers connected via an ethernet LAN. These are very loosely coupled UNIX workstations with no shared memory and only communicate via a shared bus (the ethernet LAN). SIMSCRIPT was selected as the simulation language because of it wide use in the simulation community. All processors cooperating in the simulation run programs written in SIMSCRIPT and call external functions for processor communication.

The processor communication functions are provided via the UNIX Interprocess Communication (IPC) functions [8]. These functions are standard with the BSD UNIX operating system and allow communication among processes both within the same computer and those residing on separate computers. Because the IPC functions are designed to provide communication among a large number of varying processor types, a significant amount of overhead is inherent with data communication. This could be reduced by writing replacement functions that only provide for the needs of this simulation, however a design goal was to use as little custom software as possible.

4 The Toolkit

The toolkit consists of a collection of functions, written in the C language and linkable with SIMSCRIPT programs. The basic functions provided by the toolkit are interprocess data communications and sufficient processor synchronization to allow a simulation to be broken into a collection of data servers and receivers.

To use these tools, one must first determine what in their model can be thought of as a source or generator of precomputable objects. In order for an object to be precomputed, no information about current simulation time or access to local variables can be required. The most obvious precomputable object is random numbers, however, more complex objects may be precomputed based on the simulation model at hand. Once the data sources have been identified, the simulation is written as usual, except that the identified source objects are written as a separate SIMSCRIPT program. This results in the simulation being implemented as a data generator program and a simulation program. Two C language functions must be called by both the simulation program and the generator program to install the communications interrupt handler and to identify each of the generators participating in the simulation. Each of the SIMSCRIPT programs must also contain a function that the C routine will call to transfer generated data to and from SIMSCRIPT variables. Each of these functions are described below.

C Functions

- inst_int(host,mode)
  - char *host - The name of the host running the receiver program

- char *mode - Must equal "server" or "receiver" for the server and receiver programs respectively.

This is a C routine that is called by both the receiver and server programs. Called only once, and before any the link server function described below, this function opens a socket for reading, installs the communications interrupt handler, and initializes the variables used to maintain the server information.

- link_server(server,host,mode) -
  - char *service - The name of the service being identified
  - char *host - The name of the host where the server resides
  - char *mode - Must equal "server" or "receiver" for the server and receiver programs respectively.

This C routine is called by both the receiver and server processes to identify the services that are being used in this simulation. The function creates a record of the identified server's information, opens a sending socket for the server, initializes the list of messages to that server as null, and adds the server to the list of participating servers.

- request(service,command) -
  - char *service - The name of the service being requested
  - int command A command to be sent to the server. This command integer is not examined by the toolkit: it is completely definable by the model and server developers.

This C routine is called by the simulation module to request more data items from a server. After the request is sent, control is return to the simulation software. When the requested data are received, the simulation code will be interrupted and the toolkit will make a call to user provided accept_data routine, described below.

SIMSCRIPT Functions

- accept_data given service, data, length
  - service - text variable containing the name of the service supplying the data. This field is used to route data to the correct inventory.

- data - memory for objects created. This memory space will be formatted by the server to hold the data in the correct SIMSCRIPT format.
- length - This is the length in bytes of the data area.

The accept_data function is called by the C routine that performs the socket reads when new data arrives. Because the arrival of data causes and interrupt to be serviced and this function is called during that interrupt, the code may be executed at any time. This will have an impact on the simulator's use of pointers or indices to the inventory of data.

- fill_request given service, data, yielding length
  - service - Text variable containing the name of the service being requested.
  - data - memory for objects being created. This is unformatted memory and can be interpreted and filled according to the objects being requested.
  - length - Integer variable returning the length in bytes of the data to be supplied.

  This SIMSCRIPT function is the server complement to the accept data function. When a request for objects is received by the communications handler, this function is called to fill the request. As before, because the communications are interrupt driven, this function can be called at any time.

After the receiver code has been moved to a separate program, additional SIMSCRIPT code will have to be added to the receiver program to manage the remotely generated data. This additional code keeps track of the available inventories of remotely generated data, placing requests for additional data when the local inventory falls below some threshold. How this threshold is calculated is discussed in a later section.

The receiver program may itself be a data source, for example generating simulation data that are sent to additional programs that provide statisical analysis and summary reports or to other servers for graphical display.

5 Timing Data and Decomposition Considerations

Message passing overhead must be considered when designing any type of distributed processing. To decide if any speedups can be expected for a simulation using the server model decomposition, some analysis for message passing times verse local computational costs should be performed. The following definitions are used to perform this analysis.

- CIO - Overhead induced by servicing a communications interrupt. This includes the time required to transfer data from the communications buffer to a SIMSCRIPT variable.
- CSO - Overhead induced by actively sending a message to another server.
- CTT - Time for a command message to travel between two hosts.
- DTT - Time for a data message to travel between two hosts.
- MST - Minimum time required to supply objects.
- OS - Order size. The number of items shipped in each order.
- ECR - Expected consumption rate for generated items.
- DCC - Distributed computation cost.
- LCC - Local computational cost for generating one object.
- RGT - Remote generation time. The time required by the server to generate the values. This value is determined by OS and LCC.
- TBO - Time between orders.

Assume that the generators have precomputed more of the objects than will be requested so that the time that the generator will spend processing a request is 0. With this assumption we can define

$$DCC = CSO + CTT \tag{1}$$

$$LCC = CIO + DTT \tag{2}$$

$$MST = DCC + LCC \tag{3}$$

Clearly the formulae below must hold or it will always be faster to compute the objects locally.

$$OS \geq MST + ECR \tag{4}$$

$$OS * LCC > CIO + CSO \tag{5}$$

In many cases, unless LCC (the cost of computing the data locally) is significant, OS will have to be large to make this approach feasible. Practically speaking, since for most simulations the actual consumption rate can vary, $OS * LCC$ should be significantly larger than $CIO + CSO$.

In order to assist in determining the potential effectiveness of using this approach for distributed simulation, some timing data was collected for LCC, CIO, CSO, and CT. The variable ECR is model dependent and, with the other variables fixed, OS can be determined.

Timing data for message passing among Sun workstations connected via ethernet networks and bridges was collected. The times required for message passing are not significantly affected by message length as long as are less than the maximum packet length for the ethernet (1500 bytes). Messages were passed between processors that resided on the same physical network and those on separate, bridged networks. As can be seen below, messages that had to go across bridges took twice as long as those that stayed on a single network. All data was collected when the network was lightly loaded.

Packet Size: 100 bytes

Number of Packets	Single Net (seconds)	Bridged Nets (seconds)
100	1	3
500	5	9
1,000	9	23
5,000	45	112
10,000	87	206
50,000	435	944

Packet Size: 500 bytes

Number of Packets	Single Net (seconds)	Bridged Nets (seconds)
100	1	3
500	6	18
1,000	14	26
5,000	68	137
10,000	118	256
50,000	513	1,113

Packet Size: 1000 bytes

Number of Packets	Single Net (seconds)	Bridged Nets (seconds)
100	2	3
500	8	20
1000	16	33
5,000	79	171
10,000	156	332
50,000	793	1,721

Analysis of the above data gives the following table of average times and throughput rates. Times are given in seconds/byte and throughput is given in bytes/second.

Packet Size	Average Time (sec.)		Throughput (bytes/sec)	
	Intra-net	Inter-net	Intra-net	Inter-net
100	0.000087	0.000194	11,400	5,100
500	0.000022	0.000047	46,000	21,500
1,000	0.000015	0.000034	63,000	29,000

To obtain values for the variables LCC, CSO, and CIO, the UNIX prof command was used. This is a standard UNIX tool that profiles executable code and generates reports on number of times each function is called, time spent during each call, and total time spent in the function during program execution. For more information on the prof command see [9].

6 Example

As an example, consider the generation of normally distributed random numbers. The machines used are Sun 3/60 workstations with 8 megabytes of memory. LCC was found to be 0.1 ms; the CSO and CIO were both 0.025 ms. The table below shows values for OS with corresponding ECR values.

ECR/minute	OS	Orders	RGT	TBO	MST
10,000	50	200	0.005	0.30	0.002
	100	100	0.010	0.60	0.013
	500	20	0.050	3.00	0.061
	1,000	10	0.100	6.00	0.121
	5,000	2	0.500	30.00	0.601
100,000	50	2,000	0.005	0.03	0.002
	100	1,000	0.010	0.06	0.013
	500	200	0.050	0.30	0.061
	1,000	100	0.100	0.60	0.121
	5,000	20	0.500	3.00	0.601
500,000	50	10,000	0.005	0.006	0.002
	100	5,000	0.010	0.012	0.013
	500	1,000	0.050	0.060	0.061
	1,000	500	0.100	0.120	0.121
	5,000	100	0.500	0.600	0.601
1,000,000	50	20,000	0.005	0.003	0.002
	100	10,000	0.010	0.006	0.013
	500	2,000	0.050	0.030	0.061
	1,000	1,000	0.100	0.060	0.121
	5,000	200	0.500	0.300	0.601

Two points can be made from this table. First, when the order size becomes large, the communication time for transferring the numbers from the server to the simulator becomes larger than the time required to compute the values locally. As long as the time needed to consume the numbers is greater than the MST, a speedup is possible with the server decomposition.

Secondly, when the ECR becomes very large, the remote server cannot keep up with the ECR, the receiver will be forced to wait for the server to generate numbers. If the time spent waiting is significantly less than what is required to compute the values locally, then the server decomposition still provides speedup.

7 Conclusion

The toolkit that we have developed can be use to develop distributed simulation applications without having to invest in new environments or training. SIMSCRIPT and the UNIX operating system are widely available, allowing easy access to these tools. The toolkit is composed of 650 lines of C code and requires about approximately 50 lines of additional SIMSCRIPT code per server/receiver pair to be added to the simulation model. The user of these tools need only be concerned with three C function calls and two SIMSCRIPT routines, so the complexity of the simulation program is not significantly affected.

Use of these tools requires decomposing a model into components in which information flow is unidirectional such as traffic arrival generators, statistical analysis procedures, or graphical displays.

References

[1] Richard L. Gimarc. Distributed simulation using hierarchical rollback. In *1989 Winter Simulation Conference Proceedings*, pages 621–629, 1989.

[2] Richard M. Fujimoto. Parallel discrete event simulation. In *Proceedings of the 1989 Winter Simulation Conference*, pages 19–28, 1989.

[3] David Jefferson. Distributed simulation and the time warp operating system. *ACM SIGOPS*, 77–93, Nov. 1987.

[4] B. A. Cota and R. G. Sargent. *Concurrent Programming in Discrete Event Simulation: A Survey*. Technical Report, Syacuse University, Dec. 1986.

[5] Fred J. Kandel. *A literature Survey on Distributed Discrete Event Simulation*. 1987.

[6] Douglas W. Jones, Chien-Chun Chou, Debra Renk, and Steven C. Bruell. Experience with concurrent simulation. In *Proceedings of the 1989 Winter Simulation Conference*, pages 756–763, 1989.

[7] Rajive L. Bagrodia, K. Mani Chandy, and Jayadev Misra. A message-based approach to discrete-event simulation. *IEEE Transactions on Software Engineering*, SE-13(6):654–665, June 1987.

[8] *Unix Programmer's Manual, Supplementary Documents 1.*

[9] *Unix User's Manual, Reference Guide.*