NASA/CR—1998-208518

Performance Analysis of an Actor-Based Distributed Simulation

James D. Schoeffler
Cleveland State University, Cleveland, Ohio

Prepared under Grants NAG3–1441 and NCC3–461

National Aeronautics and
Space Administration

Lewis Research Center

October 1998

Available from

NASA Center for Aerospace Information
7121 Standard Drive
Hanover, MD 21076
Price Code: A04

National Technical Information Service
5287 Port Royal Road
Springfield, VA 22100
Price Code: A04

Performance Analysis of an Actor-Based Distributed Simulation

James D Schoeffler
Professor of Computer Science
Cleveland State University
Cleveland, Ohio

This work was supported under
NASA grants NAG 3-1441 and NCC 3-461

Summary

Object-oriented design of simulation programs appears to be very attractive because of the natural association of components in the simulated system with objects. There is great potential in distributing the simulation across several computers for the purpose of parallel computation and its consequent handling of larger problems in less elapsed time. One approach to such a design is to use "actors", that is, active objects with their own thread of control. Because these objects execute concurrently, communication is via messages. This is in contrast to an object-oriented design using passive objects where communication between objects is via method calls (direct calls when they are in the same address space and remote procedure calls when they are in different address spaces or different machines). This paper describes a performance analysis program for the evaluation of a design for distributed simulations based upon actors.

1. Introduction

The motivation for this research was distributed simulation of aircraft engines as part of an engine simulation environment developed by NASA Lewis Research Center for the

Simulation System". NPSS is a flexible object-oriented simulation of aircraft engines requiring high computing speed. Figure 1 shows the scope of the NPSS simulation environment.

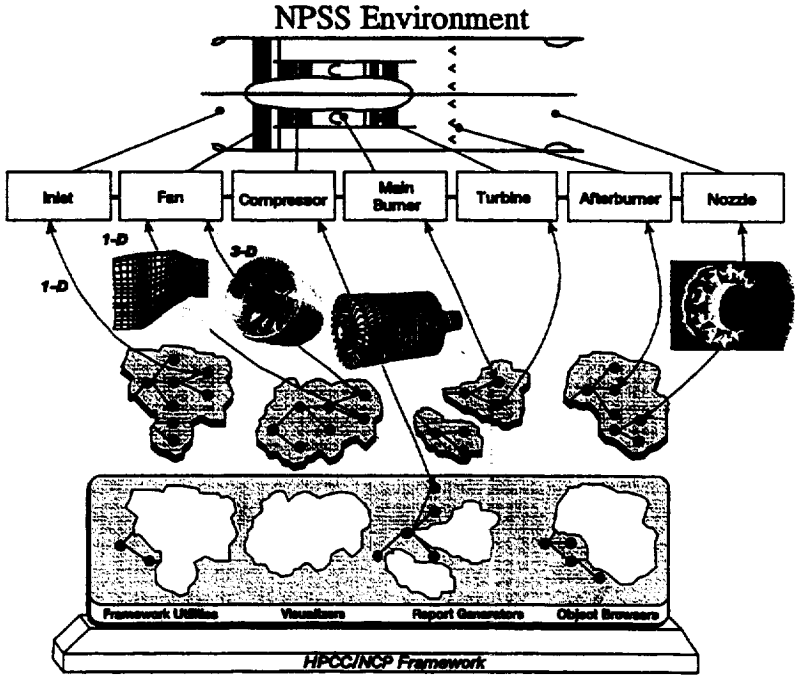


Figure 1
The NPSS Simulation Environment

It is desirable to run the simulation on a distributed computer system with multiple processors executing portions of the simulation in parallel. The purpose of this research was to investigate object-oriented structures such that individual objects could be distributed. The set of classes used in the simulation must be designed to facilitate parallel computation and not just remote computation. As a consequence, the object design is based upon the MIT "Actor" model of a concurrent object, that is, an object with its own thread of control [Schoeffler94]. The engine components, modeled as a set of actor objects, attempt to run concurrently on the various machines. Since the portions of the

simulation carried out in parallel are not independent of one another, there is the need for communication among the parallel executing processors which in turn implies need for their synchronization. Communication and synchronization can lead to decreased throughput as parallel processors wait for data or synchronization signals from other processors.

Simulations like this involve extensive iteration in order to match boundary conditions. In an engine simulation, for example, some flow through the engine is routed back and mixed with input flow. This feedback requires a special component called a "solver" to be used. Essentially the solver estimates the output for the purpose of calculating the net input data. Then the simulation calculates the data through the engine component modules, each of which permits calculating outputs given inputs. The solver then compares the calculated output data to the estimated data, revises its estimate, and iterates until adequate agreement is reached.

A set of C++ classes which carried out the required synchronization automatically and which allowed arbitrary distribution among computers has been designed. There is a need for performance analysis both to evaluate the design and to guide the distribution of the active objects for a given simulation [Lavenberg83, pp 1-10],[Schoeffler96].

Section 2 of this paper describes the active objects, the way they communicate, and the way they synchronize. Section 3 discusses the queuing network model of the actor objects and the buffer objects which queues the messages as they pass from actor to actor. Section

4 describes the organization of the performance analysis program including the ability to dynamically configure a given set of objects among different computers in different ways. Section 5 describes the results of this analysis and the limitations of the performance analysis program.

2. The actor objects, their communication, and their synchronization

An object representing a physical component to be simulated is modeled conceptually as shown in figure 2.1. The module is shown with distinct inputs and outputs, each of which represent ports or connection points so that objects can be interconnected. The figure uses the term "module" to differentiate it from other objects (such as connector objects). Data objects represent a group of data items which correspond to the variables at an interface between two components. Sharing of data is then taken to be sharing of data objects all of whose components represent variables calculated at a given time instant or iteration. By an input or output port is meant a path through which data objects can be requested and delivered via messages.

Similarly, outputs represent ports through which this module object can send data objects it has calculated at a given time instant or iteration. It is important to understand that the module itself does not know about the source of data objects it receives or the destinations of data objects it creates for these are dependent upon the particular simulation being carried out. It is equally important to understand that the modules must be capable of

executing on any processor in the network for load balancing purposes. In order to achieve these two requirements the relationship among input and output ports is defined by the interconnection of modules as shown in figure 2.1.

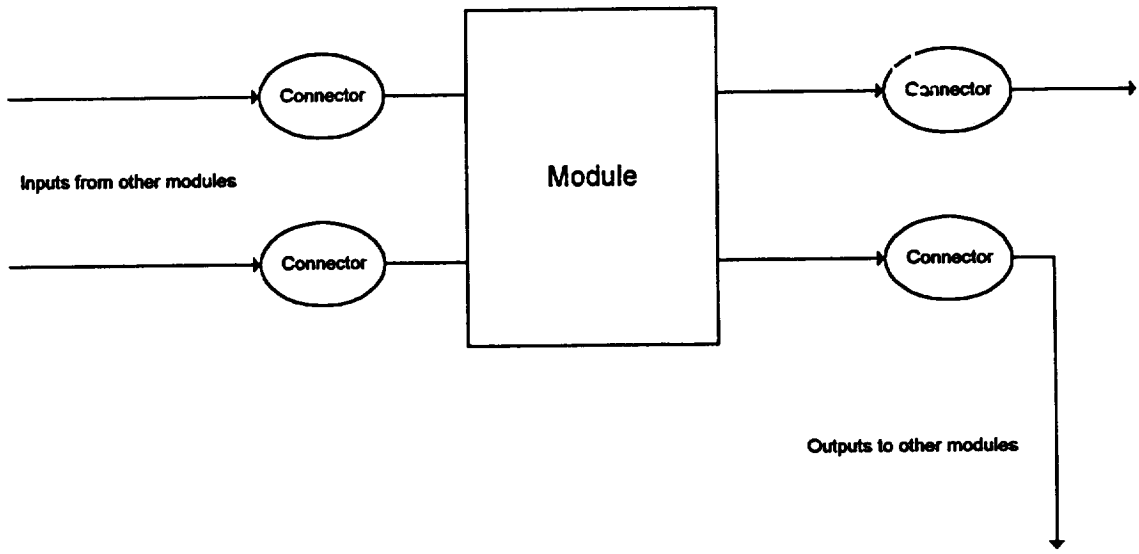


Figure 2.1
Modules with inputs, outputs and connectors

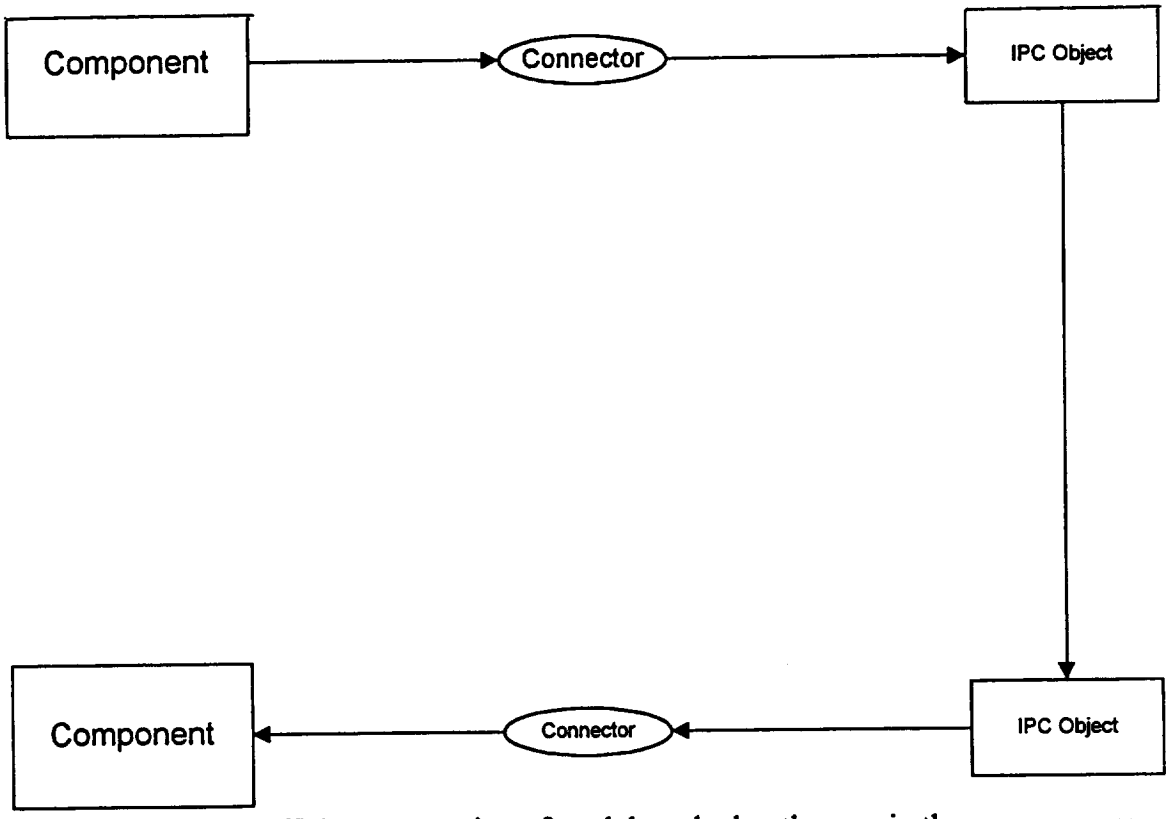
Objects represent engine modules and calculate pressure, temperatures, and flows given input conditions to the module. The module is modeled in single or multiple dimensions using either steady-state or time-evolving relationships. The model interconnects objects two ways:

1. objects whose inputs are actually outputs of another object are interconnected in the sense that an update of the former object at a given time step or iteration cannot take place before the latter object output has been obtained.
2. solver objects are used to break closed loops which occur in (1) by supplying inputs to one module object in the loop based upon previous values of the output

of the other module which must be the same as the supplied input when iteration converges.

Actor-object characteristics are:

1. Each actor acts as though it has its own thread of control.
2. Actors communicate by sending messages which are passed from the source actor, through a network-wide message passing system to the machine containing the destination actor. The message is delivered to the process containing the destination actor when that process becomes active (scheduled to execute in the destination machine). For two actors in the same machine, the message is passed the same way and differs only in that the message never leaves the machine to be transmitted to the other machine. Such messages still are handled by the message passing system.
3. Each actor acts as though it has its own queue of messages which it processes one at a time in a run-to-completion manner in the sense that a given actor-object does not start processing a second message until the processing of the first message has been completed. Note that this does not preclude the task in which the actor resides being blocked in favor of other tasks.
4. Processing is dependent upon the state of the actor-object and may include change of state.



In order to achieve efficient connection of modules whether they are in the same process (address space), in separate processes in the same machine, or in separate machines, connector objects use an "inter process communication" object which is specialized for either local or remote communication depending upon the location of the objects (Figure 2.2).

Figure 2.2
Components, connectors, and their inter-process-communication objects

At execution time, each component object and its connector is assigned to a process in a machine based upon load-balancing considerations. It is only at this time that each connector determines whether the connector with which it communicates is local or non-local. The connector then dynamically creates the appropriate local or remote IPC object whose behavior understands how to efficiently send local or remote messages.

If two modules are in the same process, their connectors use local IPC objects as shown in Figure 2.3.

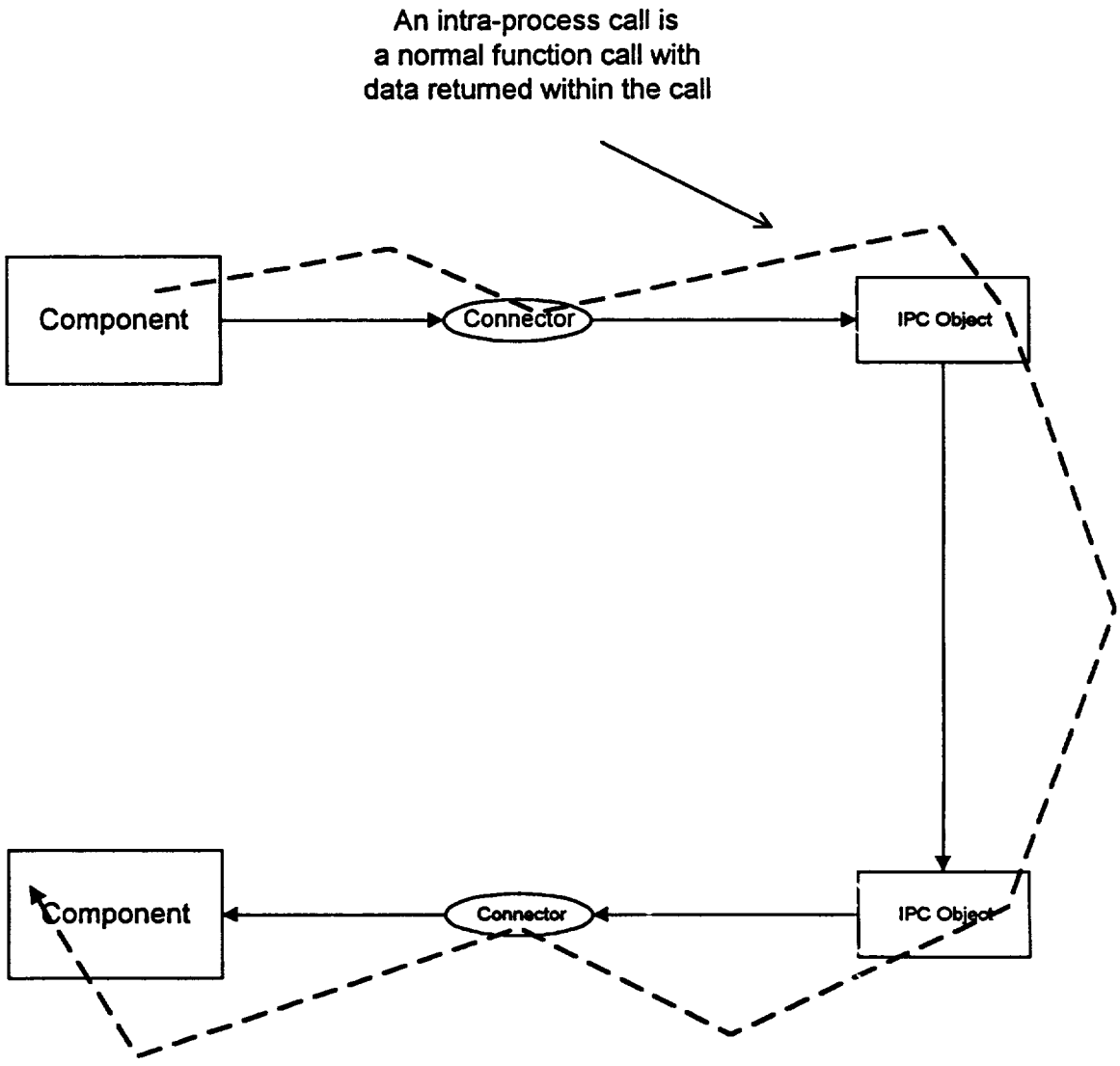
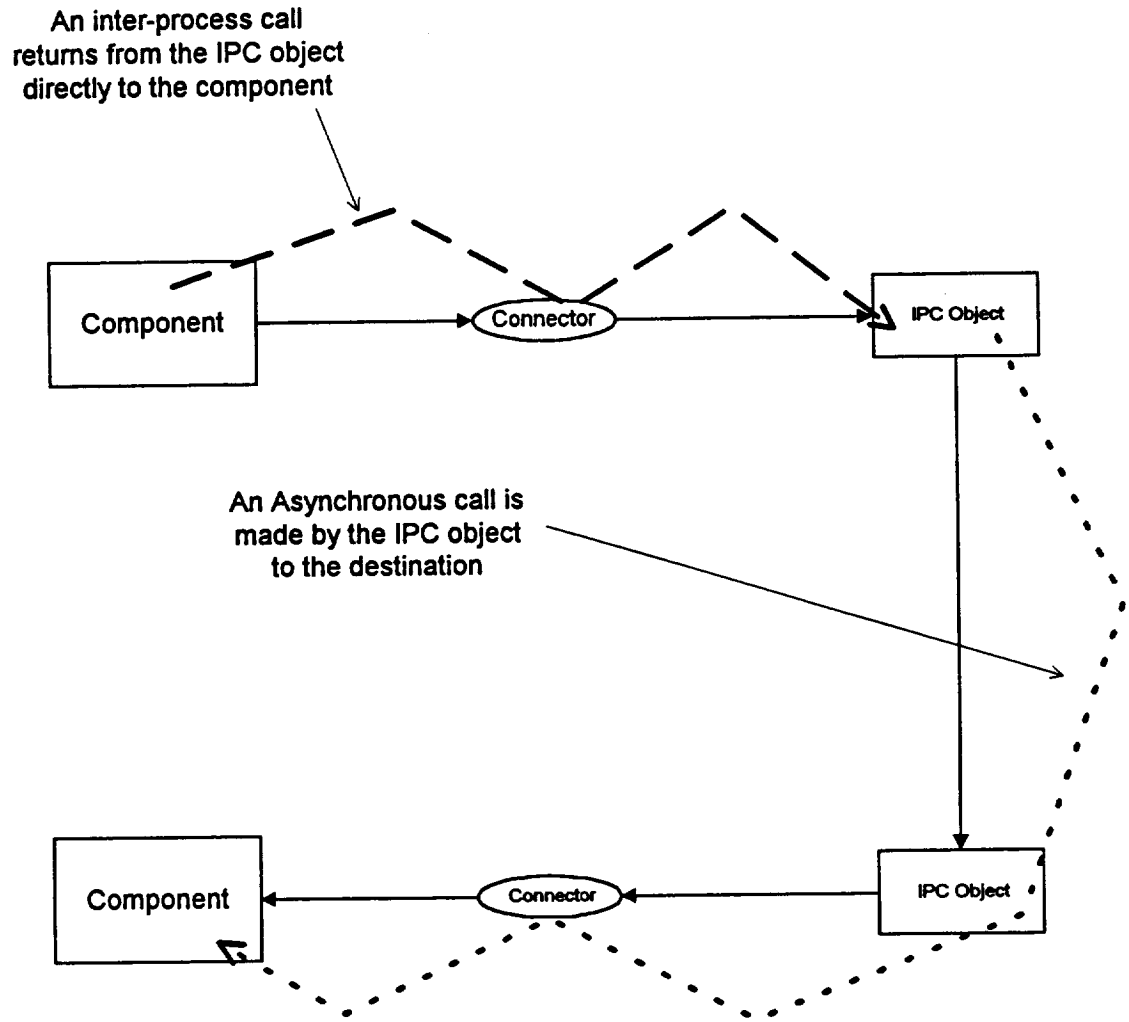


Figure 2.3
A local call is a normal function call with data directly returned

The component calls a connector function which in turn calls an (in-line) IPC function which for this specialized IPC object directly calls the next IPC object etc. so that the communication is equivalent to a function call instead of an actual message passing.

If the components are not in the same process, their connectors use IPC objects which can



send messages. In this case, it is not desirable to block the calling module so control returns directly after the call reaches its IPC objects. That IPC object initiates an asynchronous call to the remote component requesting data. When that data is returned, the IPC object then initiates a return call to the original module. This form of connection is shown in Figure 2.4.

Figure 2.4

A Remote call involves a non-blocking call and a remote message

This organization greatly simplifies the scheduling of module executions whether they are on the same machine or different machines which is a key requirement for this simulation. The actor model associates a single thread of execution with each object. As a result, each module (actor) may be assigned to a separate UNIX process in a given network of machines, or multiple actors may be assigned to a single process. In any case, a pseudo-control program must be present in each UNIX process which keeps track of the state of each actor in that process and turns control over to it depending upon its state. The network message passing system actually delivers messages to this pseudo control program which in turn passes the message to the actor in between activations of actors. Thus no concurrency problem can arise because of message delivery concurrent with an actor updating its state.

An overall control program uses the specification for how the objects are to be distributed to spawn the processes in the various machines and sets up "connector" objects to dynamically interconnect the modules into an arbitrary configuration.

The behavior of a module for one iteration is taken to be:

1. Request input data objects.
2. Wait (block) until the data objects have arrived.

3. Compute the data objects which represent outputs of the computation. This is the major computational work of the simulation.
4. Send output data objects in response to requests.
5. Advance time/iteration and repeat the sequence.

Synchronization is then automatic, because no module can begin a particular iteration until all its requested inputs have arrived from the modules generasting these data objects for that specific iteration number. After the last arrives, it may compute as required by that module to update during an iteration. It must then send output data objects to all other modules connected to it before starting the next iteration. Since each module requests its inputs, those messages arrive at their source module. There the messages must wait until the module has completed updating its outputs for that iteration can process them before the module can respond to those requests by sending the requested data object.

This behavior results in a complex state for the actor objects because they are often in a "blocked" or waiting state and a variety of events can cause transition out of these blocked states.

3. The queuing network model of a set of communicating actor objects

As is customary in performance analysis of computer systems, a queuing network model was created with a queuing model for each component actor and for each solver module.

The model is taken to be a Markov model with all exponential service times (module update times and message transmission times)[Klein75,p.21] . In addition, a single queuing center models the message handling system for the network. Each actor has a state as does the buffer. The state of the queuing network then is the collection of states of all the actors and the buffer. Unlike simple queuing networks (called separable networks [Sauer81, p. 86],[Lazowska84, p. 162]), the state of the actor is very complex as is the determination of a legal network state. These are discussed below.

Since each specific input to an actor is actually a specific output of another actor, it is convenient to consider each such pair a queuing network class (this is different from the C++ usage of the word class). The term "message class" is used to avoid confusion. Each actor is considered to have as many outputs as their are inputs of other actors connected to that actor and the output data of the actor is considered to be the same for all its outputs.

Since an actor sends a request message for each input to the specific output of the source actor which in turn sends a reply message back to that input, the message class can be taken to have a network population of 1 and its state at any instant is simply the location of the message: at the source actor, in the buffer, or at the destination actor. There are, then, as many message classes as inputs in all actors in the network. Unlike many queuing networks, the message classes visit only 3 centers in the network no matter how many modules there are: the source module for the data; the destination module for the data; and the buffer module representing the connecting network.

The state of an actor consists of the following:

1. its iteration number

2. the state of each of its inputs which is one of: present with a request for the current iteration input data not yet sent; present with current iteration input data received; or not present implying the actor has requested input data for the current iteration but has not yet received a reply.

3. the state of each of its outputs which is one of: a request message is present for the current iteration data but it has not yet been sent; the message requesting the current iteration data is not present and has not yet been received; the message requesting the current iteration data is not present because the data has already been sent; a request message for the next iteration's data is present (see below).

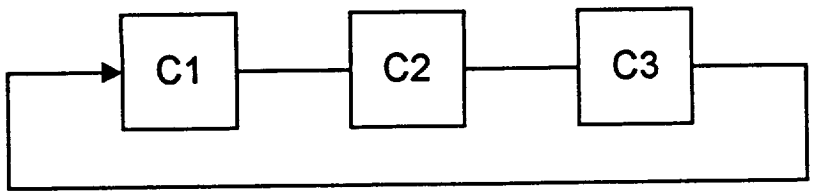
4. the actor update state which is one of: pre-computation (all input data for the current iteration is not yet present); computation (all input data for the current iteration is present and the module is in the process of doing its computational update for this iteration); post-computation (the module is in the process of sending its updated output to other actors connect to it and for which this actor has received a request or is waiting for such a request to arrive).

5. CPU ownership which is true if the CPU in which the actor resides is assigned to that actor and is false otherwise. CPU ownership can be true only if the actor can use the

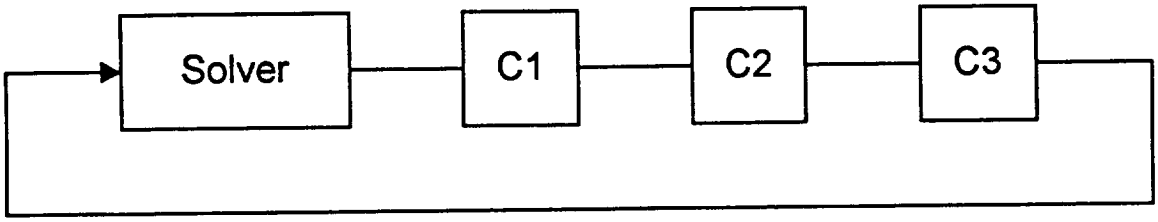
CPU which means it can send an input request message, send an output reply message, or perform update.

An actor moves from its pre-computation state to its computation state when the last reply to its input data requests arrives. It moves from computation state to post-computation state when it has completed its update computation for the current iteration. It moves from post-computation state to pre-computation state when it has completed sending its last output data. Notice that while waiting for input replies and output requests the actor modules are "blocked".

An interconnected set of modules may form a closed loop (Figure 3.1). In this case, each module needs the output of the previous module before it can update in a given iteration. Clearly no module will be able to execute. Hence a "solver" module is introduced to break the loop and allow the computation to proceed.



(a) Three components connected in a loop



Use of a solver module to "break" the loop

Figure 3.1
An example of three modules interconnected into a closed loop (a)
and the use of a solver to break the loop (b)

The only difference between a "component" module and a "solver" module is when the module changes from the current iteration number to the next. In the case of a component, changing from post-computation to pre-computation is accompanied by an increment of the iteration number (i.e., one iteration consists of requesting input, getting all input, updating, and sending all output data). The solver module is responsible for breaking closed loops of actor connections and hence already has output to send at the beginning of

its iteration (which is always connected to a component module) and then proceeds to request input data (which represents updated outputs of component modules), and finally perform update (which provides outputs to be available at the beginning of the next iteration). Hence it changes iteration number upon completing its update.

The iteration number as part of the state is an apparent complication because it would make the number of states arbitrarily large. It is easy to see, however, that along any closed path through the input of an actor and out through an output to the input of another actor etc. until the path closes implies that no two actors in that path can have iteration numbers different by more than 1. In fact, the only time iteration numbers anywhere in the network can differ by more than 1 is for a series of actors connected input to output with no closed path. For example, if actor A's single output is connected to actor B's single input and actor B's single output is connected to actor C's single input and C's single output is not connected to any other actor, then it is possible that actor C could be in iteration k, actor B in iteration k+1 (waiting for C to move to iteration k+1 and request B's output), and A could be in iteration k+2 (waiting for B to move to iteration k+2 and request A's output). Except for serial strings of actors such as these, no actors in the network may differ by more than one in iteration number. Hence the iteration state of all actors is taken to be either 0 or 1, an even or an odd iteration number with no error.

The message-buffer state is the state of each message class: not present; present in the direction of destination-to-source (meaning a request for output); or present in the direction of source-to-destination (meaning a message with the reply data). This buffer is

assumed to be independent of the CPU's in the network executing the actors and hence operates in parallel with all the actors. In effect, it has its own CPU and hence never blocks (although it can empty).

Strategy for determining legal network states

Solution of the queuing network problem requires generation of all legal network states. This is very complex because an actor's outputs may be connected to several actor's inputs which implies that there are constraints on the iteration numbers of the involved actors as well as the input and output portion of the states (the message in any class can be at only one place at any instant) as well as the actors' update states. This complexity was conveniently alleviated by defining three tests to determine a legal state.

Test 1: The "locally legal" test

First, a test of "locally legal" state of an actor was defined as any values for the states of that actor (independent of others) which can ever exist. For example, an actor's state cannot be local legal if it is in the computation state but with an input not present. Furthermore, the state of the actor must be consistent with the presence or lack of presence of the CPU at that CPU. For example, it is not locally legal to be in the pre-computation state with the CPU unless there is an input present which has not yet been requested because the CPU cannot be assigned to an actor unless it can use the CPU. Sharing of data is then taken to be sharing of data objects all of whose components

represent variables calculated at a given time instant or iteration. In addition, there is a constraint between the state of the module which is the source of a given message class (that is, outputs the data object) and the module which is the destination of that message class (is the input for that data object) and the message-buffer object. Appendix I shows the definition of the "combined state of the source and destination modules" which share a class (that is, a class is an output of the source module and an input of the destination module). These legal state combinations are specific to a message class. For example, the fifth legal combined state in Appendix I is:

{ 'C', PREC, INP_RQST_OUT, 'C', 0, PREC, NO_OUT_RQST,1,0}

which indicates that the destination module is a Component in the pre-compute state with the input not present in that module because a request has been sent to the destination module for data but not received yet; the source module is also a Component in the same iteration as the destination module and in the pre-compute state. The source module has not yet received a request for its output of this class in this iteration. The last two items indicate whether or not the message of this class is in the buffer (message set but not yet delivered) and the direction of the message (source to destination or destination to source). These items are redundant because the message must be either in the destination module or the source module or the buffer and depending upon the input and output states, the direction of the message is also clear. In the above example, the second last item indicates that the message of this class is in the buffer and this is consistent with the destination message state (request message has been sent but reply not yet received) and

source module output message state (no output request yet received). Furthermore, it is clear that the message is still enroute to the source module as the last item above indicates. These two redundant items are present for convenience of checking.

Test 2: the "network legal" test

Second, a "network legal" test is defined for each message class which ensures that the single message in each class is at only one location in the network and that the two actors it relates (one is the source and one is the destination) have compatible iteration numbers. This test is by far the most complex of the tests.

Test 3: the CPU test

Third, each CPU must be assigned to one and only one actor at any time.

Thus any trial network state generated is easily tested for legality by first applying the locally legal test to each message class, and then the network legal test followed by the CPU test. Two different strategies were used to generate states in the implemented performance analysis program to ensure that all legal states are correctly generated.

The first method, called "state increment", methodically generates all combinations of values in the network state (a trial state), and then determines whether it is a legal state by applying the locally legal test for all message classes and rejecting the state if any fails;

then rejecting the network state if the CPU test fails; and finally rejecting the network state if the network-legal test fails. If all tests pass, the state is retained.

The second method, called "state transition", starts by generating an obviously legal network state. Then each possible transition from this state is examined in sequence, and a trial state to which it transitions generated. The trial state is first tested to determine if it is a legal state (with the same series of tests as for the state increment method) and rejected if it is not legal. Then it is added to the list of generated states if it has not been generated previously and hence already present in the list.

The state transition method is significantly faster because many fewer trial states are generated but would not detect a set of legal states which are not reachable from the initial legal state for pathological networks with such states. The state increment method however generates all legal states. By generating state the two ways and showing that the same state set is generated, we are assured that the state generation program is in fact correct and that the network is indeed an irreducible Markov chain [Trivedi82, p. 319] as is apparent from the experimental implementation of the actors.

4. Design of the performance analysis program

The performance analysis program has two major parts: configuration generation and analysis. A complete listing is contained in the appendixes.

Configuration generation

Configuration generation creates a set of interconnected objects describing the set of actors to be analyzing and their interconnection. The defining data consists of:

1. The structure of the network of actors which includes for each actor its type (component or solver), the number of inputs to the actor, and the source module for that input. This data is independent of how the actors are assigned to computers for execution and also independent of the actual service times of the actors.

2. A list of the actors in the network along with their various service times. This includes the average time to initiate a message requesting input data, the average time to perform the update of the actor each iteration, and the average time to initiate an output message (the reply to the input data request message) containing the updated output data for the actor. Note that this data is independent of how the actor is interconnected with other actors and the distribution of actors to processors.

In addition, the buffer actor which represents the communication system is specified by two parameters: the average service time to process a message a local message and the average service time to process a remote message. A message arrives at the buffer as a "local" message if its source and destination actors are assigned to the same processor (CPU) and as a "remote" message if the source and destination actors are assigned to

separate processors. This behavior of the communication system is characteristic of the observed experimental system on which the actual actors were implemented.

3. Specification of the allocation of actors to processes in processors. All processors are assumed to be identical and the buffer is assumed to be separate from all processors containing actors. In the distributed actor implementation, a process is dispatched by the operating system of the machine in which it runs (hence in a round-robin manner within a UNIX workstation) and within a process, the actors are dispatched by a simple run-to-completion dispatcher which also responds to arriving messages and moves the data into the actor. In the analysis program, the same arrangement is assumed but no account for the overhead of dispatching is included. This is because the message passing overhead dominates the context switching and dispatching in the actual program. The separation of the input specification allows a given set of actors logically interconnected to solve a specific problem to be repeatedly analyzed for different distributions of the actors among various numbers of processors, thereby determining the efficiency gained from a given physical distribution.

C++ classes used to set the up the actors and their distribution in the configuration phase are the same classes used in the implementation of the distributed actor program itself, again for the purpose of ensuring that both the analysis program and the actual distributed actor implementation are consistent with one another:

1. class **group** encapsulates the set of actors assigned to a given process in a machine.
2. class **machine** encapsulates the individual processors and the groups of actors executing in each process there.
3. class **module** encapsulates an individual module, including its type(component/server), number of inputs and outputs, and connections to other modules.
4. class **modClass** encapsulates the types of data objects used as inputs and outputs for the actors.
5. class **npssrun** encapsulates the overall configuration to be analyzed including pointers to all modules, data types, groups, and machines.

The configuration section first instantiates the object of class npssrun followed by separate adding of machines, actors, and connections and then requests the distribution to be analyzed followed by creation of groups and allocating them to the machines. At this point, the complete data structure description of the system to be analyzed has been created in a form no different from that created in the distributed actor implementation.

Performance analysis

Once the configuration to be analyzed has been created, the set of objects used to generate the states of the queuing network and the global balance equations whose solution yields the steady state probabilities of states is generated.

The more important classes involved are the following:

1. class **net** which encapsulates the objects which define the state of each actor and the network buffer and the processors used in the simulation. class net also contains the methods which control the generation of network states, the generation of the global balance equations, the solution of the balance equations, and the generation of performance information from the state probabilities.

2. class **module** encapsulates a module from the point of view of analysis which chiefly means the module parameters object and the module state vector object. Since network state consists of the states of each module and the buffer, the network state is stored distributed among the modules, with each containing its own part of the network state. Hence a network state at index k is the collection of module and buffer states at index k.

3. class **modState** encapsulates a state of a module.

4. class **modStateVector** encapsulates the vector of module state objects.

5. class **modParams** encapsulates module parameters used in state generation. All of these parameters are found from the set of configuration objects previously generated.

Once the network state vector has been created as earlier described, then the definition of each state is available for the generation of global balance equations. There is one such equation for each network state and it has one term for the total flow leaving the state and one term for each state which may transition to this state. As a consequence, a global balance equation is a linear equation in the state probabilities, but only a small number of states have non-zero terms in the equation. Hence it pays to store the sparse balance equations as a set of terms each of which identifies the linear coefficient and the index of the associated state variable. This is carried out in a general purpose set of classes for storing and solving global balance equations.

The more important classes involved are:

1. class **joint** which encapsulates the vector of doubles representing the probabilities of the states.

2. class **eqTerm2** encapsulates one term in a balance equation consisting of the index of the state and the coefficient of the term.

3. class **power2** which encapsulates the storage of one balance equation in the form of an array of **eqTerm2** objects..

4. class **stgBlk** encapsulates storage for a group of global balance equations.

5. class **stg** encapsulates the multiple storage groups used for the global balance equations.

The latter two classes permit the generation of global balance equations without advanced knowledge of the amount of storage space required while heavily utilizing the storage space allocated. Blocks are allocated as needed but the entire storage area permits access to equations and terms by index without separate knowledge of the exact storage layout. The "2" indicator at the end of classes **eqTerm** and **power** simply indicate the second of two ways investigated to store terms either of which may be used in conjunction with classes stg and stgBlk and were the ways of choice for this analysis program.

The solution of the global balance equations is carried out by the "power method" which is effective for solving global balance equations of irreducible networks [Stewart78,p145]. It effectively modifies the coefficient matrix of the balance equations so that repeated multiplication of an initial trial solution vector by this matrix converges to the desired solution. The implementation in **power2** class carries out the matrix multiplication directly using the sparse storage scheme implemented without actually generating the matrix. Convergence has been observed to be quick with no numerical problems observed in the distributed actor problems studied here.

Appendix III contains the listings of performance analysis program class headers. Appendix IV contains the listings of the implementation files for preservation of the program as of the date of this report. It is important to note that this program has evolved over a 4 year period with much experimentation with various ways to model the actor networks and implement the resulting queuing network model. Hence much of the code exists for debugging purposes, detailed printing of intermediate results etc. and is not intended to be a user-friendly program for general use.

5. Results and conclusions

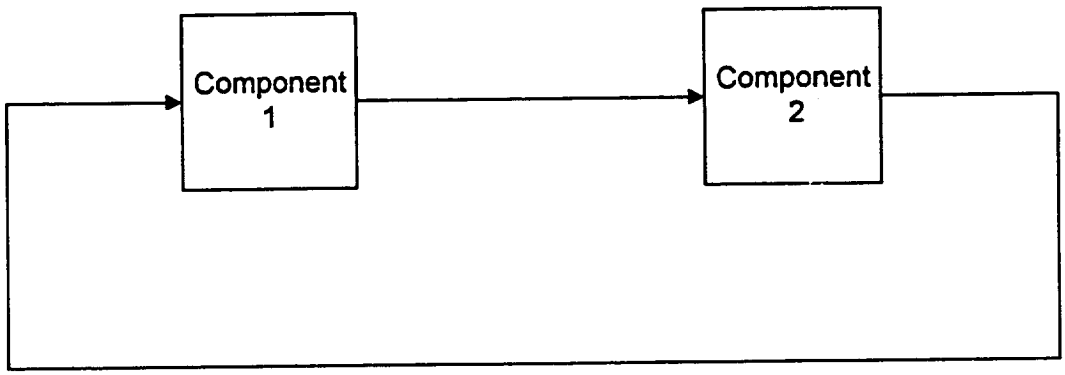
The resulting analysis program successfully uses the same data files input to the distributed actor implementation and generates not only the states and state probability vector, but detailed performance statistics on the individual actors and the network buffer. The results are shown by the following examples.

Example 1 -- an actor network with actor update times long compared to message times

The first examples use a simple set of actors in which performance is dominated by the update time of the actors and the message passing time causes an incidental overhead and the second consists of the same system but with actor update time of the same size as the message passing times. The first example emphasizes the calculation of performance for various choices of physical distribution.

Consider a simple example of two modules each with a single input and single output with the input of each connected to the output of the other (Figure 5.1). Because of the closed loop connection, it is necessary to insert a solver module which provides the input to each of the components each iteration and whose objective is to observe each modules output and to iterate until solver outputs converge to solver inputs.

This network then has two component modules each with one input and one output and one solver module with two inputs and two outputs.



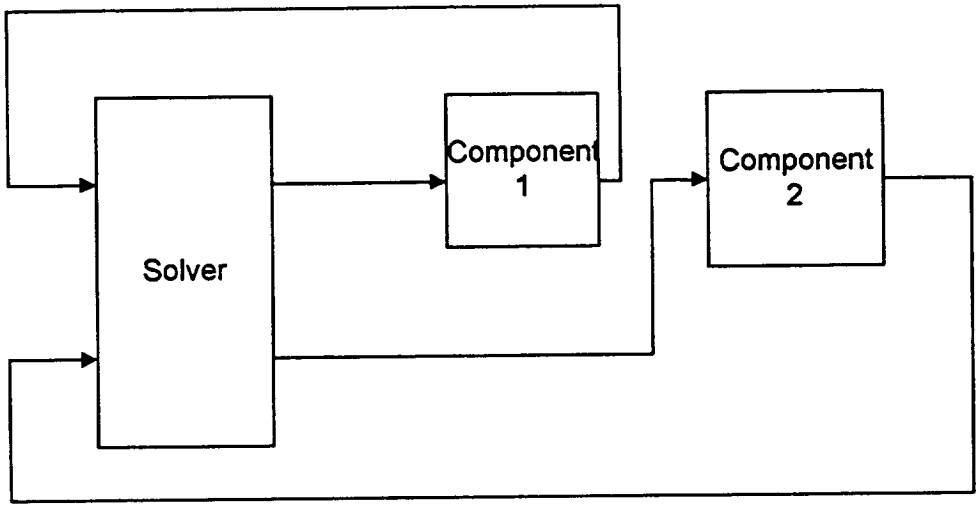


Figure 5.1
A two component application with solver

Example 1 -- update time long compared to message times

The module connection input describes the above example:

Module_S SOLVER 0 Module_C1 MODULE 0

Module_S	SOLVER	1	Module_C2	MODULE	0
Module_C1	MODULE	0	Module_S	SOLVER	0
Module_C2	MODULE	0	Module_S	SOLVER	1

Each line contains: the name of a module (arbitrary name), the type of module as defined in the input file "clsname.dat", the number of one of the modules inputs, the name of the source module, the type of the source module, and the output number of the source module which supplies that input. Note that the solver Module_S takes its first input from the single output 0 of component Module_C1 and its second from the single output 0 of component Module_C2. Module_C1 takes its single input from output 0 of the solver module Module_S and Module_C2 takes its single input from output 1 of the solver.

Message transmission times assumed for the buffer are 10 microseconds for remote messages and 1 microsecond for local messages. All modules have identical message transmission times (1 microsecond) and update times (100 microseconds). The analysis program is written assuming "microseconds" as the basic unit for service times. There would be no change if service times were uniformly changed to milliseconds or some other unit except that the printed report outputs would have to be scaled accordingly. It would be a simple change to allow different time scales for the service times.

Clearly the application update time dominates the message handling times. Notice that the solver has two send two input request messages and two output reply messages per iteration whereas each component sends one input and one output message.

The performance model generates 194 states for this example, requiring the solution of 194 linear equations in the same number of unknowns. This is a relatively small state space.

For all three actors in 1 process in a single CPU, the results are:

Module 0 update rate = 3230.5799 period: 0.0003 secs or 0.3095 millisecs

Module 1 update rate = 3230.9976 period: 0.0003 secs or 0.3095 millisecs

Module 2 update rate = 3230.2549 period: 0.0003 secs or 0.3096 millisecs

The module time per iteration are the same, 309.5 microsecs. Since all modules are in the same processor, the iteration time should be the 300 microseconds for the three update times, 6 microseconds for the messages requesting input and sending outputs for each class, plus the time for the concurrent buffer to pass the messages among the actors. Since each message passes through the buffer, the total buffer time per iteration is 6 messages at 1 microsecond each. However only 309.5-300-6 = 3.5 microseconds appear in the iteration time, implying overlap of buffer message transmission with actor work for the remainder of the time.

Module Number	Prob PREC	Prob C	Prob CDONE wait	Prob cpu wait	Prob msg	Prob busy update	Prob busy
0	0.5867	0.3231	0.0902	0.0000	0.6640	0.0129	0.3231
1	0.6636	0.3283	0.0081	0.0058	0.6646	0.0065	0.3231

2 0.4228 0.5736 0.0036 0.4934 0.1771 0.0065 0.3230

The advantage of a model as detailed as this one, is the information about where the system spends its time. Any performance item may be calculated by summing the performance function over the individual states weighted by the state probabilities.

Note above that the two components spend 59% and 66% of their time in the PreCompute mode waiting for the solver to reply with the data it has requested. These modules have just completed their updates in the previous iteration and have sent their output data to the solver. The server must handle this data transmission and then update itself before it can replay to these messages.

Notice also that module 2, the solver, could use the CPU but must wait for the CPU 49% of the time. This must be because the component modules are performing their updates in this time.

Buffer Performance

Prob Empty = 0.9742
Utilization = 0.0258
Num Lcl Msgs = 0.0375
Num Rem Msgs = 0.0000
Lcl Msg Rate = 25836.9678
Rem Msg Rate = 0.0000
Total Msg Rate = 25836.9678

Buffer performance indicates the buffer is usually empty (97% of the time). This is a consequence of the short message transmission time compared to the update times of the actors.

Overall queue length by population class are:

Class[0] avg pop = 1.0000 dest: 0.7449 src: 0.2455 bfr: 0.0096
Class[1] avg pop = 1.0000 dest: 0.4200 src: 0.5708 bfr: 0.0093
Class[2] avg pop = 1.0000 dest: 0.3398 src: 0.6513 bfr: 0.0089
Class[3] avg pop = 1.0000 dest: 0.8229 src: 0.1674 bfr: 0.0097

Contrast these results with the case where each module is in a separate CPU.

Now the results are the following:

Module 0 update rate = 3381.1667 period: 0.0003 secs or 0.2958 millisecs
Module 1 update rate = 3379.0812 period: 0.0003 secs or 0.2959 millisecs
Module 2 update rate = 3382.7768 period: 0.0003 secs or 0.2956 millisecs

The solver must execute serially, however, because it needs the results of both components before it can update. However this would only be 200 microseconds for the solver and the pair of components. But the iteration update time is 296 seconds. Because the actors are in separate machines, the assumed buffer message transmission times are now 10 microseconds each because they are machine-to-machine instead of process-to-process within one machine. There are 8 such messages sent each iteration. Furthermore, when the buffer sends input data to one component, it can begin its update. While this update proceeds, the buffer sends the input data to the second. Hence the components do not have their input data at the same time. Thus they cannot overlap 100% of the time. However this is not the whole answer.

The two components are in separate machines and can execute in parallel if they have their inputs at the same time. However, queuing network analysis assumes the service times are exponentially distributed. Even though the average time is 100 microseconds for each of the two components executing in parallel, the actual compute time is assumed to be exponentially distributed and both components must finish before the solver can begin execution. The expected time for two components with the same average service time (with exponential distributions) is actually 1.5 times the average time of each. This is because, the finish time is actually the longer of the two compute times selected from the independent distribution. Hence the expected time for both of them to finish is 150 microseconds, added to the solver time yields 250 microseconds. The additional 46 microseconds is due to the time remote 10 microsecond messages must pass between the solver and the components, delaying their start of execution.

Repeating this analysis with the remote message transmission time of the buffer set to 1 microseconds instead of 10 yields an iteration time of 255 microseconds, the 250 microseconds as expected plus 5 microseconds for the (now short 1 microsecond) messages.

Example 2 -- A more complex example

Complexity of the model is primarily based upon the number of states required to describe a configuration. This example generates between 2000 and 2500 states for each of the configurations, requiring solution of a set of as many 2500 simultaneous equations. Because these equations correspond to a Markov model with unique solution, the Power method used for the solution can readily handle this number of equations[Stewart78]. The program in this report has been tested with up to 5000 states. It will handle many more than this number, but has not been tested for larger numbers of states.

The example consists of three components connected end-to-end but with a solver with three inputs and three outputs which breaks the connections between the components so they can be computed in parallel (Figure 5.2).

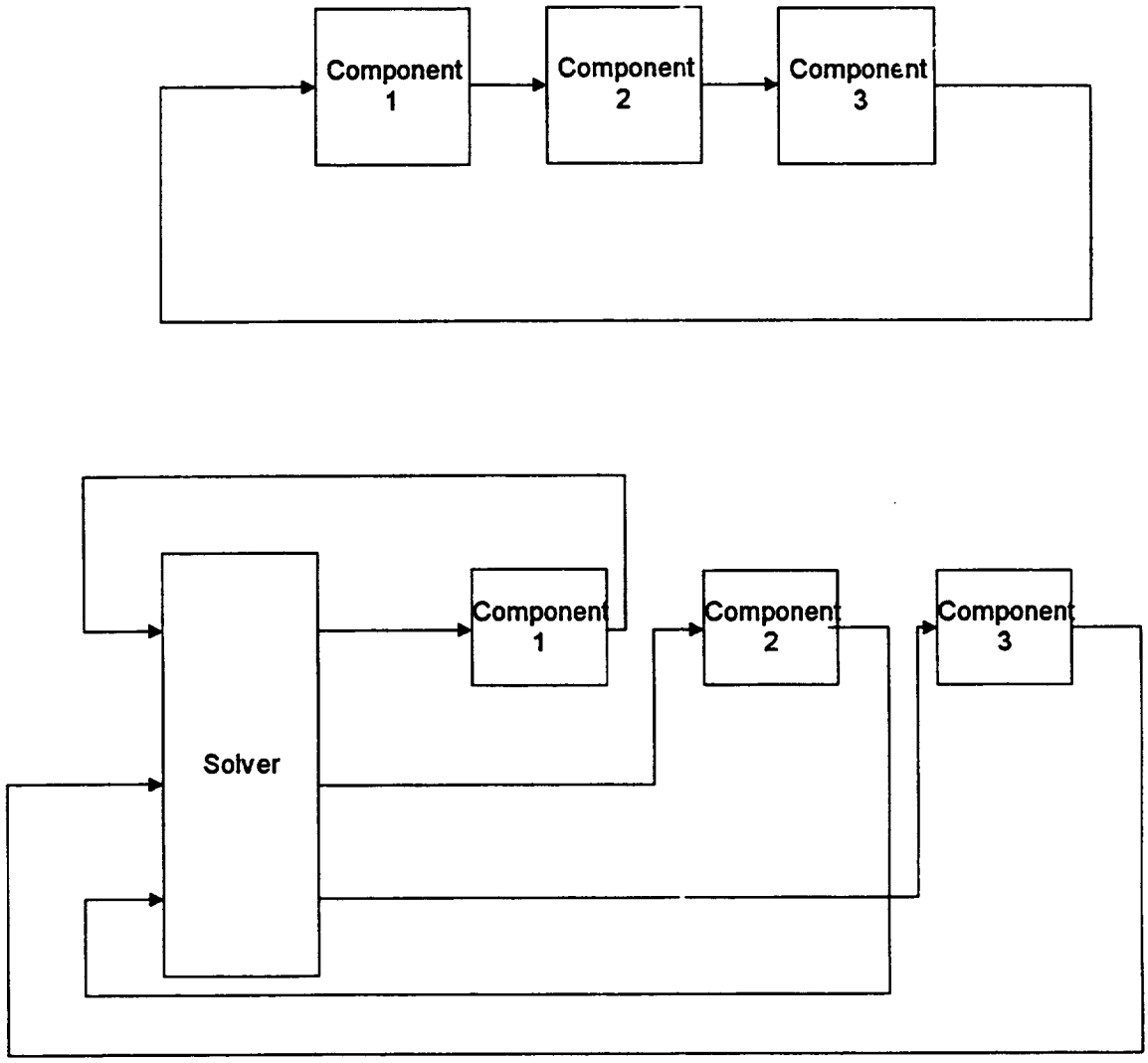


Figure 5.2
An example with a large number of states

Six different solutions for the model are given. In all six, the service times of the solver and component are identical: 100 microseconds to transmit a message (the request for input data or the sending of output data), and 200 microseconds to update the component or solver each iteration.

Three different configurations which allocate the four modules to a different number of processors (1, 2 or 4) are each given for two different values for the buffer message transmission times. Buffer message transmission times represent the speed of the interconnecting network.

The "slower" network cases correspond to buffer message transmission time of 100 microseconds for a remote message, and 50 microseconds for a local message. Thus message transmission time between actors is comparable to the modules' update time of 200 microseconds.

The "faster" network cases correspond to buffer message transmission time of 20 microseconds for as remote message and 5 microseconds for a local message. These message transmission times are small compared to the modules' update time of 200 microseconds.

The results of these runs are contained in Appendix II. The table below summarizes the resulting iteration times for the various configurations and buffer message times.

Number of Processors	Number of States	Iteration time (microsecs) Slow Network	Iteration time (microsecs) Fast Network
1	2083	2068	2005
2	2208	1680	1357
4	2484	1681	1100

Notice that a fast communication network allows a near 2-to-1 decrease in computation time (2 millisecs down to 1 millisec) using multiple processors. The slow network, on the other hand, experiences no such improvement because of the additional time modules must wait while remote messages are transmitted from the buffers. The times for 2 and 4 processors is not an error in the above table. With 2 processors, fewer remote messages are sent and the net result is an almost identical iteration time.

It is important to realize that the examples with the slow network are extreme with the message transmission times being comparable to module update time. These examples do become pertinent, however, if one starts dividing the computation among many actors such that the individual actors are responsible for less and less computation. Such a distribution of an application is said to be among "fine grain" objects rather than "coarse grain" objects in which the computation time of each actor is long compared to the message transmission time.

Conclusions

The object of this effort was to produce an efficient performance analysis program that could be used to guide the distribution of actor objects among processors. This is a realistic approach to determining the distribution because the effects of queuing delays and statistical affects due to concurrent execution of modules is difficult to estimate and not obvious intuitively as the examples here show. Most applications are run repeatedly so

that the effort involved in searching for a good distribution is well worth the effort

The program developed here is quite efficient even in models with several thousand states, running completely in less than a minute in most cases. However it does have the fundamental limitation that the number of states can become excessive as more complex configurations of actors are used.

As a consequence, another technique for performance analysis which is approximate (as opposed to the exact solution used in the program) has been developed and reported separately [Schoeffler97]. This program has been used to calibrate the approximate analysis. Some combination of the two approaches appear to be useful as part of the distributed actor computational system as a basis for choosing a good distribution of actors among processes and processors.

References

[Klein75] Kleinrock, Leonard, "Queueing Systems", Vol 1, John Wiley &Sons, NYC, 1975

[Lavenberg83] Lavenberg, Stephen S., "Computer Performance Modeling Handbook", Academic Press, 1983

[Lazowska, E. and J. Zahorjan, G. Graham, K. Sevcik, "Quantitative system performance", Prentice-Hall, 1984.

[Sauer81] Sauer, Charles and K. M. Chandy, "Computer Systems Performance Modeling", Prentice-Hall, 1981

[Schoeffler94] Schoeffler, J., "An object-oriented approach to distributed simulation", AIAA conference, Cleveland, Ohio, 1994.

[Schoeffler96] Schoeffler, J., "Design of object-oriented distributed simulation classes", final report, NASA grant NAG 3-1441, November, 1995.

[Schoeffler97] Schoeffler, J., "The nearest neighbor decomposition of queuing network performance models", report on NASA grant NCC 3-461, 1997.

[Stewart78] Stewart, William J., "A comparison of numerical techniques in Markov Modeling", CACM 21, 2 (February 1978), 144-152.

[Trivedi, K.] Trivedi, K, "Probability & Statistics With Reliability, Queuing, and Computer Science Applications", Prentice-Hall, 1982.

Appendix I: Locally Legal State Combinations

The legal combinations of state between the destination actor with a given class as input and the source actor which has the same class as n output are listed below. The legal combinations depend on whether the source and destination are a component ('C') or a solver ('S'); the module state which may be pre-compute (PREC), compute (C), or post-compute (CDONE); and the output state which is one of no request present (NO_OUT_RQST), request received but not sent (RQST_IT), request already sent (RQST_IT_SENT), or request for next iteration present (RQST_NEXT_IT).

The definition of the 9 elements in the combined state are:

0: destination module type (Solver or Component)
1: destination module execution state
2: input state
3: source module type (Solver or Component)
4: relative iteration (0, +1, -1) relative to destination module iteration
5: source module execution state
6: output state
7: message for this class is in buffer (1) else 0
8: direction for message (0: to source, 1: to destination, -1: no message)

All legal combinations for Component to Component are:

```
{ 'C', PREC, NO_INP_RQST, 'C', 0, PREC, NO_OUT_RQST,0,-1},
{ 'C', PREC, NO_INP_RQST, 'C', 0,    C, NO_OUT_RQST,0,-1},
{ 'C', PREC, NO_INP_RQST, 'C', 0,CDONE, NO_OUT_RQST,0,-1},
{ 'C', PREC, NO_INP_RQST, 'C',-1,CDONE,RQST_IT_SENT,0,-1},
{ 'C', PREC,INP_RQST_OUT, 'C', 0, PREC,NO_OUT_RQST ,1, 0},
{ 'C', PREC,INP_RQST_OUT, 'C', 0, PREC,    RQST_IT ,0,-1},
{ 'C', PREC,INP_RQST_OUT, 'C', 0,    C,NO_OUT_RQST ,1, 0},
{ 'C', PREC,INP_RQST_OUT, 'C', 0,    C,    RQST_IT ,0,-1},
{ 'C', PREC,INP_RQST_OUT, 'C', 0,CDONE,NO_OUT_RQST ,1, 0},
{ 'C', PREC,INP_RQST_OUT, 'C', 0,CDONE,    RQST_IT ,0,-1},
{ 'C', PREC,INP_RQST_OUT, 'C', 0,CDONE,RQST_IT_SENT,1, 1},
{ 'C', PREC,INP_RQST_OUT, 'C',-1,CDONE,RQST_IT_SENT,1, 0},
{ 'C', PREC,INP_RQST_OUT, 'C',-1,CDONE,RQST_NEXT_IT,0,-1},
{ 'C', PREC,INP_RQST_OUT, 'C', 1, PREC,NO_OUT_RQST,1, 1},
{ 'C', PREC,INP_RQST_OUT, 'C', 1,    C,NO_OUT_RQST,1, 1},
{ 'C', PREC,INP_RQST_OUT, 'C', 1,CDONE,NO_OUT_RQST,1, 1},
{ 'C', PREC,    INP_IN, 'C', 1, PREC,NO_OUT_RQST,0,-1},
{ 'C', PREC,    INP_IN, 'C', 1,    C,NO_OUT_RQST,0,-1},
{ 'C', PREC,    INP_IN, 'C', 1,CDONE,NO_OUT_RQST,0,-1},
{ 'C',    C,    INP_IN, 'C', 0,CDONE,RQST_IT_SENT,0,-1},
{ 'C',    C,    INP_IN, 'C', 1, PREC,NO_OUT_RQST ,0,-1},
{ 'C',    C,    INP_IN, 'C', 1,    C,NO_OUT_RQST ,0,-1},
{ 'C',    C,    INP_IN, 'C', 1,CDONE,NO_OUT_RQST ,0,-1},
{ 'C',CDONE,    INP_IN, 'C', 0,CDONE,RQST_IT_SENT,0,-1},
{ 'C',CDONE,    INP_IN, 'C', 1, PREC,NO_OUT_RQST ,0,-1},
{ 'C',CDONE,    INP_IN, 'C', 1,    C,NO_OUT_RQST ,0,-1},
{ 'C',CDONE,    INP_IN, 'C', 1,CDONE,NO_OUT_RQST ,0,-1},
```

All legal combinations for Component to Solver are:

```
{ 'S', PREC, NO_INP_RQST, 'C', 0, PREC, NO_OUT_RQST,0,-1},
{ 'S', PREC, NO_INP_RQST, 'C', 0,    C, NO_OUT_RQST,0,-1},
{ 'S', PREC, NO_INP_RQST, 'C', 0,CDONE, NO_OUT_RQST,0,-1},
{ 'S', PREC, NO_INP_RQST, 'C',-1,CDONE,RQST_IT_SENT,0,-1},
{ 'S', PREC,INP_RQST_OUT, 'C', 0, PREC,NO_OUT_RQST ,1, 0},
{ 'S', PREC,INP_RQST_OUT, 'C', 0, PREC,    RQST_IT ,0,-1},
{ 'S', PREC,INP_RQST_OUT, 'C', 0,    C,NO_OUT_RQST ,1, 0},
{ 'S', PREC,INP_RQST_OUT, 'C', 0,    C,    RQST_IT ,0,-1},
{ 'S', PREC,INP_RQST_OUT, 'C', 0,CDONE,NO_OUT_RQST ,1, 0},
{ 'S', PREC,INP_RQST_OUT, 'C', 0,CDONE,    RQST_IT ,0,-1},
{ 'S', PREC,INP_RQST_OUT, 'C', 0,CDONE,RQST_IT_SENT,1, 1},
{ 'S', PREC,INP_RQST_OUT, 'C',-1,CDONE,RQST_IT_SENT,1, 0},
{ 'S', PREC,INP_RQST_OUT, 'C',-1,CDONE,RQST_NEXT_IT,0,-1},
{ 'S', PREC,INP_RQST_OUT, 'C', 1, PREC,NO_OUT_RQST,1, 1},
{ 'S', PREC,INP_RQST_OUT, 'C', 1,    C,NO_OUT_RQST,1, 1},
{ 'S', PREC,INP_RQST_OUT, 'C', 1,CDONE,NO_OUT_RQST,1, 1},
{ 'S', PREC,    INP_IN, 'C', 0,CDONE,NO_OUT_RQST,0,-1},
{ 'S', PREC,    INP_IN, 'C', 1, PREC,NO_OUT_RQST,0,-1},
{ 'S', PREC,    INP_IN, 'C', 1,    C,NO_OUT_RQST,0,-1},
{ 'S', PREC,    INP_IN, 'C', 1,CDONE,NO_OUT_RQST,0,-1},
{ 'S',    C,   INP_IN, 'C', 0,CDONE,RQST_IT_SENT,0,-1},
{ 'S',    C,   INP_IN, 'C', 1, PREC,NO_OUT_RQST ,0,-1},
{ 'S',    C,   INP_IN, 'C', 1,    C,NO_OUT_RQST ,0,-1},
{ 'S',    C,   INP_IN, 'C', 1,CDONE,NO_OUT_RQST ,0,-1},
{ 'S',CDONE,   INP_IN, 'C', 0, PREC, NO_OUT_RQST,0,-1},
{ 'S',CDONE,   INP_IN, 'C', 0,    C, NO_OUT_RQST,0,-1},
{ 'S',CDONE,   INP_IN, 'C', 0,CDONE, NO_OUT_RQST,0,-1},
{ 'S',CDONE,   INP_IN, 'C',-1,CDONE,RQST_IT_SENT,0,-1},
```

All legal combinations for Component to Solver are:

```
{ 'C', PREC, NO_INP_RQST, 'S', 0,CDONE, NO_OUT_RQST,0,-1},
{ 'C', PREC, NO_INP_RQST, 'S',-1, PREC,RQST_IT_SENT,0,-1},
{ 'C', PREC, NO_INP_RQST, 'S',-1,    C,RQST_IT_SENT,0,-1},
{ 'C', PREC, NO_INP_RQST, 'S',-1,CDONE,RQST_IT_SENT,0,-1},
{ 'C', PREC,INP_RQST_OUT, 'S', 0, PREC,RQST_IT_SENT,1, 1},
{ 'C', PREC,INP_RQST_OUT, 'S', 0,    C,RQST_IT_SENT,1, 1},
{ 'C', PREC,INP_RQST_OUT, 'S', 0,CDONE,NO_OUT_RQST ,1, 0},
{ 'C', PREC,INP_RQST_OUT, 'S', 0,CDONE,    RQST_IT,0,-1},
{ 'C', PREC,INP_RQST_OUT, 'S', 0,CDONE,RQST_IT_SENT,1, 1},
{ 'C', PREC,INP_RQST_OUT, 'S',-1, PREC,RQST_IT_SENT,1, 0},
{ 'C', PREC,INP_RQST_OUT, 'S',-1, PREC,RQST_NEXT_IT,0,-1},
{ 'C', PREC,INP_RQST_OUT, 'S',-1,    C,RQST_IT_SENT,1, 0},
{ 'C', PREC,INP_RQST_OUT, 'S',-1,    C,RQST_NEXT_IT,0,-1},
{ 'C', PREC,INP_RQST_OUT, 'S',-1,CDONE,RQST_IT_SENT,1, 0},
{ 'C', PREC,INP_RQST_OUT, 'S',-1,CDONE,RQST_NEXT_IT,0,-1},
{ 'C', PREC,INP_RQST_OUT, 'S', 1, PREC, NO_OUT_RQST,1, 1},
{ 'C', PREC,INP_RQST_OUT, 'S', 1,    C, NO_OUT_RQST,1, 1},
{ 'C', PREC,INP_RQST_OUT, 'S', 1,CDONE, NO_OUT_RQST,1, 1},
{ 'C', PREC,     INP_IN, 'S', 0, PREC,RQST_IT_SENT,0,-1},
{ 'C', PREC,     INP_IN, 'S', 0,    C,RQST_IT_SENT,0,-1},
{ 'C', PREC,     INP_IN, 'S', 0,CDONE,RQST_IT_SENT,0,-1},
{ 'C', PREC,     INP_IN, 'S', 1,CDONE,NO_OUT_RQST ,0,-1},
{ 'C',    C,     INP_IN, 'S', 0, PREC,RQST_IT_SENT,0,-1},
{ 'C',    C,     INP_IN, 'S', 0,    C,RQST_IT_SENT,0,-1},
{ 'C',    C,     INP_IN, 'S', 0,CDONE,RQST_IT_SENT,0,-1},
{ 'C',    C,     INP_IN, 'S', 1,CDONE,NO_OUT_RQST ,0,-1},
{ 'C',CDONE,     INP_IN, 'S', 0, PREC,RQST_IT_SENT,0,-1},
{ 'C',CDONE,     INP_IN, 'S', 0,    C,RQST_IT_SENT,0,-1},
{ 'C',CDONE,     INP_IN, 'S', 0,CDONE,RQST_IT_SENT,0,-1},
{ 'C',CDONE,     INP_IN, 'S', 1, PREC,NO_OUT_RQST ,0,-1},
{ 'C',CDONE,     INP_IN, 'S', 1,    C,NO_OUT_RQST ,0,-1},
{ 'C',CDONE,     INP_IN, 'S', 1,CDONE,NO_OUT_RQST ,0,-1},
```

Appendix II Output of example runs

Run 1: Slow Network, Modules all in 1 CPU

```
Module Module_S1 of class SOLVER33 has 3 inputs and 3 outputs and is assigned to
group Group_1
Module Module_C1 of class MODULE has 1 inputs and 1 outputs and is assigned to
group Group_1
Module Module_C2 of class MODULE has 1 inputs and 1 outputs and is assigned to
group Group_1
Module Module_C3 of class MODULE has 1 inputs and 1 outputs and is assigned to
group Group_1
------- Summary Report -------
   1 machines used
   1 module groups (processes)
   4 modules (components)
---- display of file: 'connect.dat'----------
Module_S1 SOLVER33  0 Module_C1 MODULE   0
Module_S1 SOLVER33  1 Module_C2 MODULE   0
Module_S1 SOLVER33  2 Module_C3 MODULE   0
Module_C1 MODULE    0 Module_S1 SOLVER33  0
Module_C2 MODULE    0 Module_S1 SOLVER33  1
Module_C3 MODULE    0 Module_S1 SOLVER33  2
Solver module
      Module Class SOLVER
      Module 'Module_S1' parameters
     3 inputs
     0 input pop class number
     1 source module of input 0
     0 source module output number
     1 input pop class number
     2 source module of input 1
     0 source module output number
     2 input pop class number
     3 source module of input 2
     0 source module output number
```

Run 1: Slow Network, Modules all in 1 CPU

Component module
Module Class COMPONENT
Module 'Module_C1' parameters
1 inputs
3 input pop class number
0 source module of input 0
0 source module output number
Component module
Module Class COMPONENT
Module 'Module_C2' parameters
1 inputs
4 input pop class number
0 source module of input 0
1 source module output number
Component module
Module Class COMPONENT
Module 'Module_C3' parameters
1 inputs
5 input pop class number
0 source module of input 0
2 source module output number

Module Num	Input Req (us)	Output Send (us)	Compute Time (us)
0	100	100	200
1	100	100	200
2	100	100	200
3	100	100	200

Buffer Local and Remote Service Times are: 50 and 100
There are 1 cpu's executing 4 modules
Module 0 update rate = 483.3611 period: 0.0021 secs or 2.0688 millisecs
Module 1 update rate = 483.4976 period: 0.0021 secs or 2.0683 millisecs
Module 2 update rate = 483.4645 period: 0.0021 secs or 2.0684 millisecs
Module 3 update rate = 483.3133 period: 0.0021 secs or 2.0691 millisecs

Module Number	Prob PREC	Prob C	Prob CDONE wait	Prob CPU wait	Prob msg	Prob msg update	Prob busy	Prob busy
0	0.6031	0.0967	0.3002	0.0000	0.6133	0.2901	0.0967	
1	0.5374	0.2046	0.2580	0.1485	0.6581	0.0967	0.0967	
2	0.4524	0.2974	0.2502	0.2655	0.5411	0.0967	0.0967	
3	0.4642	0.4818	0.0540	0.6187	0.1879	0.0967	0.0967	

Run 1: Slow Network, Modules all in 1 CPU

```
Buffer Performance
   Prob Empty    = 0.7099
   Utilization   = 0.2901
   Num Lcl Msgs  = 0.4145
   Num Rem Msgs  = 0.0000
   TotalNum Msgs = 0.4145
   Lcl Msg Rate  = 5801.0144
   Rem Msg Rate  = 0.0000
   Total Msg Rate = 5801.0144
Overall queue length by population class
    Class[0] avg pop = 1.0000 dest: 0.7709 src: 0.1565 bfr: 0.0726
    Class[1] avg pop = 1.0000 dest: 0.6835 src: 0.2431 bfr: 0.0734
    Class[2] avg pop = 1.0000 dest: 0.5443 src: 0.3869 bfr: 0.0688
    Class[3] avg pop = 1.0000 dest: 0.5112 src: 0.4222 bfr: 0.0666
    Class[4] avg pop = 1.0000 dest: 0.5986 src: 0.3350 bfr: 0.0663
    Class[5] avg pop = 1.0000 dest: 0.8139 src: 0.1194 bfr: 0.0668
```

Run 2: Slow Network, Modules in 2 CPU

```
Module Module_S1 of class SOLVER33 has 3 inputs and 3 outputs and is assigned to
group Group_1
Module Module_C1 of class MODULE has 1 inputs and 1 outputs and is assigned to
group Group_1
Module Module_C2 of class MODULE has 1 inputs and 1 outputs and is assigned to
group Group_2
Module Module_C3 of class MODULE has 1 inputs and 1 outputs and is assigned to
group Group_2
---- display of file: 'connect.dat'----------
Module_S1 SOLVER33  0 Module_C1 MODULE   0
Module_S1 SOLVER33  1 Module_C2 MODULE   0
Module_S1 SOLVER33  2 Module_C3 MODULE   0
Module_C1 MODULE    0 Module_S1 SOLVER33  0
Module_C2 MODULE    0 Module_S1 SOLVER33  1
Module_C3 MODULE    0 Module_S1 SOLVER33  2
```

Run 2: Slow Network, Modules in 2 CPU

```
Solver module
      Module Class SOLVER
      Module 'Module_S1' parameters
     3 inputs
     0 input pop class number
     1 source module of input 0
     0 source module output number
     1 input pop class number
     2 source module of input 1
     0 source module output number
     2 input pop class number
     3 source module of input 2
     0 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C1' parameters
     1 inputs
     3 input pop class number
     0 source module of input 0
     0 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C2' parameters
     1 inputs
     4 input pop class number
     0 source module of input 0
     1 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C3' parameters
     1 inputs
     5 input pop class number
     0 source module of input 0
     2 source module output number
```

Run 2: Slow Network, Modules in 2 CPU

Module Num	Input Req (us)	Output Send (us)	Compute Time (us)
0	100	100	200
1	100	100	200
2	100	100	200
3	100	100	200

Buffer Local and Remote Service Times are: 50 and 100
There are 2 cpu's executing 4 modules

Number Modules — Probability that all are busy

Module 0 update rate = 595.4878 period: 0.0017 secs or 1.6793 millisecs
Module 1 update rate = 595.1323 period: 0.0017 secs or 1.6803 millisecs
Module 2 update rate = 595.2128 period: 0.0017 secs or 1.6801 millisecs
Module 3 update rate = 595.2093 period: 0.0017 secs or 1.6801 millisecs

0	0.1124
1	0.5848
2	0.3029

Module Number	Prob PREC	Prob C	Prob CDONE wait	Prob CPU wait	Prob msg	Prob busy msg	Prob busy update
0	0.6719	0.1191	0.2090	0.0000	0.5237	0.3572	0.1191
1	0.5620	0.3481	0.0899	0.2623	0.4996	0.1190	0.1190
2	0.6714	0.1190	0.2096	0.0000	0.7619	0.1190	0.1190
3	0.5708	0.2299	0.1993	0.1495	0.6124	0.1191	0.1190

Run 2: Slow Network, Modules in 2 CPU

Buffer Performance
Prob Empty = 0.4047
Utilization = 0.5953
Num Lcl Msgs = 0.3080
Num Rem Msgs = 0.8549
TotalNum Msgs = 1.1629
Lcl Msg Rate = 2381.0716
Rem Msg Rate = 4762.1423
Total Msg Rate = 7143.2139
Overall queue length by population class
Class[0] avg pop = 1.0000 dest: 0.5979 src: 0.2340 bfr: 0.1682
Class[1] avg pop = 1.0000 dest: 0.6778 src: 0.1052 bfr: 0.2170
Class[2] avg pop = 1.0000 dest: 0.5642 src: 0.2161 bfr: 0.2197
Class[3] avg pop = 1.0000 dest: 0.5278 src: 0.3324 bfr: 0.1398
Class[4] avg pop = 1.0000 dest: 0.3881 src: 0.4017 bfr: 0.2101
Class[5] avg pop = 1.0000 dest: 0.4953 src: 0.2965 bfr: 0.2082

Run 3: Slow Network, Modules in 4 CPU

```
Module Module_S1 of class SOLVER33 has 3 inputs and 3 outputs and is assigned to
group Group_1
Module Module_C1 of class MODULE has 1 inputs and 1 outputs and is assigned to
group Group_2
Module Module_C2 of class MODULE has 1 inputs and 1 outputs and is assigned to
group Group_3
Module Module_C3 of class MODULE has 1 inputs and 1 outputs and is assigned to
group Group_4
------- Summary Report -------
   4 machines used
   4 module groups (processes)
   4 modules (components)

---- display of file: 'connect.dat'----------
Module_S1 SOLVER33  0 Module_C1 MODULE   0
Module_S1 SOLVER33  1 Module_C2 MODULE   0
Module_S1 SOLVER33  2 Module_C3 MODULE   0
Module_C1 MODULE    0 Module_S1 SOLVER33  0
Module_C2 MODULE    0 Module_S1 SOLVER33  1
Module_C3 MODULE    0 Module_S1 SOLVER33  2
```

Run 3: Slow Network, Modules in 4 CPU

```
Solver module
      Module Class SOLVER
      Module 'Module_S1' parameters
     3 inputs
     0 input pop class number
     1 source module of input 0
     0 source module output number
     1 input pop class number
     2 source module of input 1
     0 source module output number
     2 input pop class number
     3 source module of input 2
     0 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C1' parameters
     1 inputs
     3 input pop class number
     0 source module of input 0
     0 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C2' parameters
     1 inputs
     4 input pop class number
     0 source module of input 0
     1 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C3' parameters
     1 inputs
     5 input pop class number
     0 source module of input 0
     2 source module output number
```

Run 3: Slow Network, Modules in 4 CPU

Module Num	Input Req (us)	Output Send (us)	Compute Time (us)
0	100	100	200
1	100	100	200
2	100	100	200
3	100	100	200

Buffer Local and Remote Service Times are: 50 and 100

There are 4 cpu's executing 4 modules

Module 0 update rate = 594.3243 period: 0.0017 secs or 1.6826 millisecs
Module 1 update rate = 594.6744 period: 0.0017 secs or 1.6816 millisecs
Module 2 update rate = 594.7934 period: 0.0017 secs or 1.6813 millisecs
Module 3 update rate = 594.8774 period: 0.0017 secs or 1.6810 millisecs

Probability that multiple modules are busy:

Number	Probability
0	0.2053
1	0.4896
2	0.2242
3	0.0722
4	0.0087

Module Number	Prob PREC	Prob C	Prob CDONE wait	Prob CPU wait	Prob msg	Prob busy msg	Prob busy update
0	0.6810	0.1189	0.2001	0.0000	0.5244	0.3568	0.1189
1	0.6705	0.1189	0.2105	0.0000	0.7621	0.1190	0.1189
2	0.6691	0.1190	0.2119	0.0000	0.7620	0.1190	0.1190
3	0.6719	0.1190	0.2092	0.0000	0.7621	0.1190	0.1190

Run 3: Slow Network, Modules in 4 CPU

Buffer Performance
Prob Empty = 0.2863
Utilization = 0.7137
Num Lcl Msgs = 0.0000
Num Rem Msgs = 1.6160
TotalNum Msgs = 1.6160
Lcl Msg Rate = 0.0000
Rem Msg Rate = 7136.9581
Total Msg Rate = 7136.9581
Overall queue length by population class
Class[0] avg pop = 1.0000 dest: 0.6089 src: 0.1108 bfr: 0.2803
Class[1] avg pop = 1.0000 dest: 0.6001 src: 0.1154 bfr: 0.2845
Class[2] avg pop = 1.0000 dest: 0.6037 src: 0.1191 bfr: 0.2772
Class[3] avg pop = 1.0000 dest: 0.3890 src: 0.3593 bfr: 0.2517
Class[4] avg pop = 1.0000 dest: 0.3904 src: 0.3495 bfr: 0.2601
Class[5] avg pop = 1.0000 dest: 0.3876 src: 0.3503 bfr: 0.2621

Run 4: Fast Network, Modules all in 1 CPU

Module Module_S1 of class SOLVER33 has 3 inputs and 3 outputs and is assigned to group Group_1
Module Module_C1 of class MODULE has 1 inputs and 1 outputs and is assigned to group Group_1
Module Module_C2 of class MODULE has 1 inputs and 1 outputs and is assigned to group Group_1
Module Module_C3 of class MODULE has 1 inputs and 1 outputs and is assigned to group Group_1

---- display of file: 'connect.dat'----------
Module_S1 SOLVER33 0 Module_C1 MODULE 0
Module_S1 SOLVER33 1 Module_C2 MODULE 0
Module_S1 SOLVER33 2 Module_C3 MODULE 0
Module_C1 MODULE 0 Module_S1 SOLVER33 0
Module_C2 MODULE 0 Module_S1 SOLVER33 1
Module_C3 MODULE 0 Module_S1 SOLVER33 2

Run 4: Fast Network, Modules all in 1 CPU

```
Solver module
      Module Class SOLVER
      Module 'Module_S1' parameters
     3 inputs
     0 input pop class number
     1 source module of input 0
     0 source module output number
     1 input pop class number
     2 source module of input 1
     0 source module output number
     2 input pop class number
     3 source module of input 2
     0 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C1' parameters
     1 inputs
     3 input pop class number
     0 source module of input 0
     0 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C2' parameters
     1 inputs
     4 input pop class number
     0 source module of input 0
     1 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C3' parameters
     1 inputs
     5 input pop class number
     0 source module of input 0
     2 source module output number
```

Run 4: Fast Network, Modules all in 1 CPU

Module Num	Input Req (us)	Output Send (us)	Compute Time (us)
0	100	100	200
1	100	100	200
2	100	100	200
3	100	100	200

Buffer Local and Remote Service Times are: 5 and 20

There are 1 cpu's executing 4 modules

Probability that multiple modules are busy

Number	Probability
0	0.0026
1	0.9974

Module 0 update rate = 499.0980 period: 0.0020 secs or 2.0036 millisecs
Module 1 update rate = 498.7390 period: 0.0020 secs or 2.0051 millisecs
Module 2 update rate = 498.5420 period: 0.0020 secs or 2.0058 millisecs
Module 3 update rate = 499.0213 period: 0.0020 secs or 2.0039 millisecs

Module Number	Prob PREC	Prob C	Prob CDONE wait	Prob CPU wait	Prob msg	Prob msg update	Prob busy	Prob busy
0	0.5113	0.0998	0.3889	0.0000	0.6010	0.2992	0.0998	
1	0.4592	0.1572	0.3836	0.1497	0.6509	0.0997	0.0997	
2	0.4043	0.2122	0.3834	0.2520	0.5486	0.0998	0.0997	
3	0.4938	0.4562	0.0499	0.7360	0.0645	0.0998	0.0998	

Run 4: Fast Network, Modules all in 1 CPU

Buffer Performance
Prob Empty = 0.9701
Utilization = 0.0299
Num Lcl Msgs = 0.0311
Num Rem Msgs = 0.0000
TotalNum Msgs = 0.0311
Lcl Msg Rate = 5983.6854
Rem Msg Rate = 0.0000
Total Msg Rate = 5983.6854

Overall queue length by population class
Class[0] avg pop = 1.0000 dest: 0.8429 src: 0.1519 bfr: 0.0052
Class[1] avg pop = 1.0000 dest: 0.7879 src: 0.2068 bfr: 0.0052
Class[2] avg pop = 1.0000 dest: 0.6382 src: 0.3566 bfr: 0.0052
Class[3] avg pop = 1.0000 dest: 0.5906 src: 0.4042 bfr: 0.0052
Class[4] avg pop = 1.0000 dest: 0.6456 src: 0.3493 bfr: 0.0051
Class[5] avg pop = 1.0000 dest: 0.9355 src: 0.0594 bfr: 0.0051

Run 5: Fast Network, Modules in 2 CPU

Module Module_S1 of class SOLVER33 has 3 inputs and 3 outputs and is assigned to group Group_1
Module Module_C1 of class MODULE has 1 inputs and 1 outputs and is assigned to group Group_1
Module Module_C2 of class MODULE has 1 inputs and 1 outputs and is assigned to group Group_2
Module Module_C3 of class MODULE has 1 inputs and 1 outputs and is assigned to group Group_2

---- display of file: 'connect.dat'----------
Module_S1 SOLVER33 0 Module_C1 MODULE 0
Module_S1 SOLVER33 1 Module_C2 MODULE 0
Module_S1 SOLVER33 2 Module_C3 MODULE 0
Module_C1 MODULE 0 Module_S1 SOLVER33 0
Module_C2 MODULE 0 Module_S1 SOLVER33 1
Module_C3 MODULE 0 Module_S1 SOLVER33 2

Run 5: Fast Network, Modules in 2 CPU

```
Solver module
      Module Class SOLVER
      Module 'Module_S1' parameters
     3 inputs
     0 input pop class number
     1 source module of input 0
     0 source module output number
     1 input pop class number
     2 source module of input 1
     0 source module output number
     2 input pop class number
     3 source module of input 2
     0 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C1' parameters
     1 inputs
     3 input pop class number
     0 source module of input 0
     0 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C2' parameters
     1 inputs
     4 input pop class number
     0 source module of input 0
     1 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C3' parameters
     1 inputs
     5 input pop class number
     0 source module of input 0
     2 source module output number
```

Run 5: Fast Network, Modules in 2 CPU

Module Num	Input Req (us)	Output Send (us)	Compute Time (us)
0	100	100	200
1	100	100	200
2	100	100	200
3	100	100	200

Buffer Local and Remote Service Times are: 5 and 20

There are 2 cpu's executing 4 modules

Number Modules	Probability that all are busy

Module 0 update rate = 737.7397 period: 0.0014 secs or 1.3555 millisecs
Module 1 update rate = 737.1089 period: 0.0014 secs or 1.3567 millisecs
Module 2 update rate = 736.6123 period: 0.0014 secs or 1.3576 millisecs
Module 3 update rate = 736.9040 period: 0.0014 secs or 1.3570 millisecs

Probability that multiple modules are computing

Number	Probability
0	0.0035
1	0.5190
2	0.4775

Module Number	Prob PREC	Prob C	Prob CDONE wait	Prob CPU wait	Prob msg	Prob busy msg	Prob busy update
0	0.5820	0.1475	0.2704	0.0000	0.4105	0.4420	0.1475
1	0.4787	0.4321	0.0892	0.4312	0.2739	0.1474	0.1474
2	0.5901	0.1473	0.2626	0.0000	0.7053	0.1474	0.1473
3	0.4469	0.3119	0.2412	0.2185	0.4866	0.1476	0.1474

Run 5: Fast Network, Modules in 2 CPU

Buffer Performance
Prob Empty = 0.8673
Utilization = 0.1327
Num Lcl Msgs = 0.0200
Num Rem Msgs = 0.1351
TotalNum Msgs = 0.1551
Lcl Msg Rate = 2947.5004
Rem Msg Rate = 5896.2358
Total Msg Rate = 8843.7362

Overall queue length by population class
Class[0] avg pop = 1.0000 dest: 0.6311 src: 0.3585 bfr: 0.0103
Class[1] avg pop = 1.0000 dest: 0.8626 src: 0.1038 bfr: 0.0336
Class[2] avg pop = 1.0000 dest: 0.7048 src: 0.2605 bfr: 0.0347
Class[3] avg pop = 1.0000 dest: 0.7369 src: 0.2534 bfr: 0.0097
Class[4] avg pop = 1.0000 dest: 0.4836 src: 0.4826 bfr: 0.0338
Class[5] avg pop = 1.0000 dest: 0.6358 src: 0.3312 bfr: 0.0330

Run 6: Fast Network, Modules in 4 CPU

Module Module_S1 of class SOLVER33 has 3 inputs and 3 outputs and is assigned to group Group_1
Module Module_C1 of class MODULE has 1 inputs and 1 outputs and is assigned to group Group_2
Module Module_C2 of class MODULE has 1 inputs and 1 outputs and is assigned to group Group_3
Module Module_C3 of class MODULE has 1 inputs and 1 outputs and is assigned to group Group_4

---- display of file: 'connect.dat'----------
Module_S1 SOLVER33 0 Module_C1 MODULE 0
Module_S1 SOLVER33 1 Module_C2 MODULE 0
Module_S1 SOLVER33 2 Module_C3 MODULE 0
Module_C1 MODULE 0 Module_S1 SOLVER33 0
Module_C2 MODULE 0 Module_S1 SOLVER33 1
Module_C3 MODULE 0 Module_S1 SOLVER33 2

Run 6: Fast Network, Modules in 4 CPU

```
Solver module
      Module Class SOLVER
      Module 'Module_S1' parameters
     3 inputs
     0 input pop class number
     1 source module of input 0
     0 source module output number
     1 input pop class number
     2 source module of input 1
     0 source module output number
     2 input pop class number
     3 source module of input 2
     0 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C1' parameters
     1 inputs
     3 input pop class number
     0 source module of input 0
     0 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C2' parameters
     1 inputs
     4 input pop class number
     0 source module of input 0
     1 source module output number
Component module
      Module Class COMPONENT
      Module 'Module_C3' parameters
     1 inputs
     5 input pop class number
     0 source module of input 0
     2 source module output number
```

Run 6: Fast Network, Modules in 4 CPU

Module Num	Input Req (us)	Output Send (us)	Compute Time (us)
0	100	100	200
1	100	100	200
2	100	100	200
3	100	100	200

Buffer Local and Remote Service Times are: 5 and 20

There are 4 cpu's executing 4 modules

Number Modules Probability that all are busy

Module 0 update rate = 909.0453 period: 0.0011 secs or 1.1001 millisecs
Module 1 update rate = 908.8122 period: 0.0011 secs or 1.1003 millisecs
Module 2 update rate = 908.7623 period: 0.0011 secs or 1.1004 millisecs
Module 3 update rate = 908.7547 period: 0.0011 secs or 1.1004 millisecs

Probability that multiple modules are computing

Number	Probability
0	0.0093
1	0.4217
2	0.3487
3	0.1825
4	0.0377

Module Number	Prob PREC	Prob C	Prob CDONE wait	Prob CPU wait	Prob msg	Prob busy msg	Prob busy update
0	0.5338	0.1818	0.2844	0.0000	0.2729	0.5453	0.1818
1	0.5773	0.1818	0.2409	0.0000	0.6365	0.1818	0.1818
2	0.5735	0.1818	0.2448	0.0000	0.6365	0.1818	0.1818
3	0.5717	0.1818	0.2466	0.0000	0.6365	0.1818	0.1818

Run 6: Fast Network, Modules in 4 CPU

Buffer Performance
Prob Empty = 0.7819
Utilization = 0.2181
Num Lcl Msgs = 0.0000
Num Rem Msgs = 0.2799
TotalNum Msgs = 0.2799
Lcl Msg Rate = 0.0000
Rem Msg Rate = 10906.1969
Total Msg Rate = 10906.1969

Overall queue length by population class
Class[0] avg pop = 1.0000 dest: 0.8081 src: 0.1448 bfr: 0.0471
Class[1] avg pop = 1.0000 dest: 0.8075 src: 0.1439 bfr: 0.0486
Class[2] avg pop = 1.0000 dest: 0.8091 src: 0.1435 bfr: 0.0475
Class[3] avg pop = 1.0000 dest: 0.5135 src: 0.4409 bfr: 0.0456
Class[4] avg pop = 1.0000 dest: 0.5174 src: 0.4366 bfr: 0.0460
Class[5] avg pop = 1.0000 dest: 0.5192 src: 0.4356 bfr: 0.0451

Appendix III

Header files for all classes

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)

2. REPORT DATE
October 1998

3. REPORT TYPE AND DATES COVERED
Final Contractor Report

4. TITLE AND SUBTITLE
Performance Analysis of an Actor-Based Distributed Simulation

5. FUNDING NUMBERS
WU–509–10–31–00
NAG3–1441 and NCC3–461

6. AUTHOR(S)
James D. Schoeffler

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Cleveland State University
1983 E. 24th Street
Cleveland, Ohio 44115–2403

8. PERFORMING ORGANIZATION REPORT NUMBER
E–11299

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135–3191

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CR—1998-208518

11. SUPPLEMENTARY NOTES
Project Manager, James D. Schoeffler, Computing and Interdisciplinary Systems Office, NASA Lewis Research Center, organization code 2900, (216) 433–5193.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Categories: 66 and 62 Distribution: Nonstandard
This publication is available from the NASA Center for AeroSpace Information, (301) 621–0390.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

Object-oriented design of simulation programs appears to be very attractive because of the natural association of components in the simulated system with objects. There is great potential in distributing the simulation across several computers for the purpose of parallel computation and its consequent handling of larger problems in less elapsed time. One approach to such a design is to use "actors", that is, active objects with their own thread of control. Because these objects execute concurrently, communication is via messages. This is in contrast to an object-oriented design using passive objects where communication between objects is via method calls (direct calls when they are in the same address space and remote procedure calls when they are in different address spaces or different machines). This paper describes a performance analysis program for the evaluation of a design for distributed simulations based upon actors.

14. SUBJECT TERMS
Computer system performance; Computer system simulation

15. NUMBER OF PAGES
75

16. PRICE CODE
A04

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	

Appendix IV

Implementation file listing for all classes