Simulation Modeling and Statistical Computing
N. Adam Guest Editor

Asynchronous Distributed Simulation via a Sequence of Parallel Computations

K. M. Chandy and J. Misra
University of Texas, Austin

An approach to carrying out asynchronous, distributed simulation on multiprocessor message-passing architectures is presented. This scheme differs from other distributed simulation schemes because (1) the amount of memory required by all processors together is bounded and is no more than the amount required in sequential simulation and (2) the multiprocessor network is allowed to deadlock, the deadlock is detected, and then the deadlock is broken. Proofs for the correctness of this approach are outlined.

Key Words and Phrases: discrete event simulation, distributed systems, message-passing systems, communicating sequential processes, deadlock, recovery, parallel algorithms
CR Categories: 3.8, 8.1

1. A Scheme for Distributed Programming

The design of parallel programs is a vitally important issue as increasingly common parallel architectures are developed. One approach to constructing parallel programs is to recognize parallelism in existing sequential programs. This approach does not seem to be generally successful. It is particularly poor for simulation because of the frequent manipulation of a single data structure (the event-list). A different approach to the problem of simulation is proposed here. A number of problems admit the following solution structure: Phases of the problem may be solved in parallel where the phases are required to follow one another in a sequential manner (Figure 1). This solution structure is called a *sequence of parallel computations*. It allows asynchrony within phases and requires synchronism at phase interfaces. In this paper the sequence of parallel computations approach is applied to the problem of simulation.

Because we do not want to have a centralized process which oversees the network, it is critically important that the termination of a phase be detected in a *distributed* manner by the network. Dijkstra and Scholten [6] were the first to propose a distributed solution to this problem. In our simulation algorithm the termination of a phase is manifested as a deadlock, which is detected in a distributed manner using a modification of the Dijkstra–Scholten scheme. The start of the next phase corresponds to recovery from deadlock. It seems more efficient to run the computation until deadlock, recover from the deadlock, and then resume the computation than to avoid deadlock altogether in distributed simulation.

The time required to obtain statistically valid results from simulations may be prohibitively large for some complex systems. The advent of multiprocessor architectures offers the possibility of reducing run-times by concurrently carrying out the simulation on several processors. Unfortunately, multiprocessor architectures cannot be used with conventional event-driven simulation techniques because of the inherently sequential nature of event-list manipulation and the very high frequency with which event-lists are manipulated; this makes it difficult to run parts of the simulation concurrently. Significant parallelism can be achieved only by doing away with the event-list in its *usual* form.

One approach to distributed simulation is to have a single global clock that drives every processor in the network. This approach is used with time-driven simulation. We do not wish to use any global variables nor do we want to use a single process to drive the simulation because it will prove to be a bottleneck. Our approach is totally asynchronous; every process maintains its own local clock and there is no global synchronization mechanism such as a global clock.

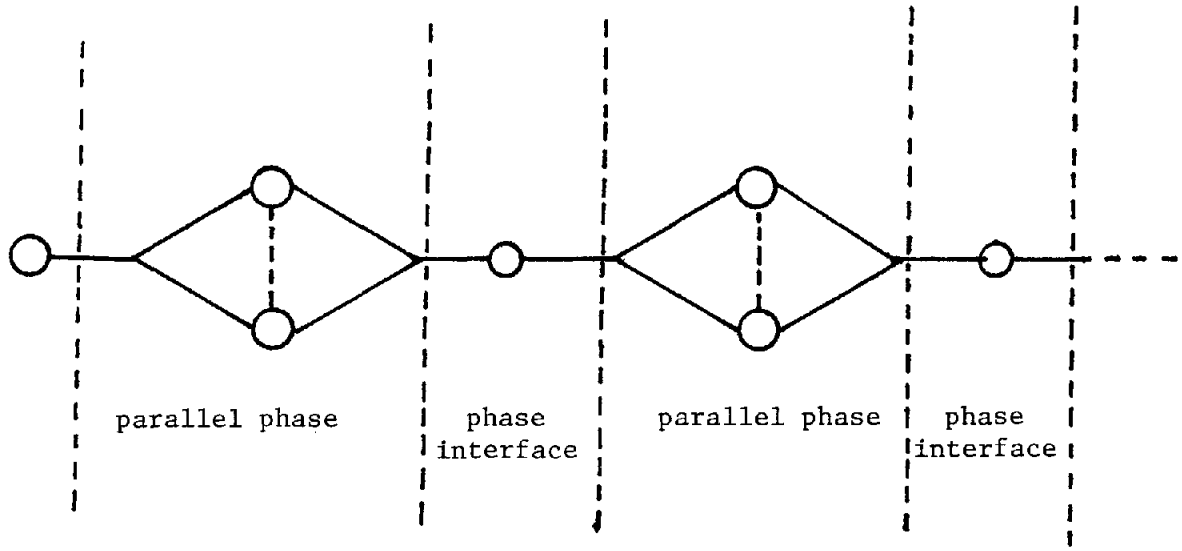
2. Overview of Distributed Simulation Algorithms

Any system where component entities interact at discrete times can be represented by a network of processes that communicate via messages (where processes correspond to entities). Any interaction between entities i and j can be modeled as a message between processes i and j. Physical systems will be simulated that can be partitioned into physical processes (PPs) that communicate with one another exclusively via messages. The behavior of a PP *at* time t cannot be influenced by messages transmitted to it *after* t. This is called the *realizability condition.*

Authors' present addresses: K.M. Chandy and J. Misra, Department of Computer Science, University of Texas, Austin, TX 78712

This work is partially supported by NSF grants MCS77-09812 and MCS 79-25383 and AFOSR 77-3409.

Fig. 1. A Sequence of Parallel Phrases.



All asynchronous distributed simulation schemes have some common characteristics. There is a logical process (LP) corresponding to every PP. Each LP simulates the corresponding PP. A message m from PPi to PPj at time t is simulated by LPi sending LPj a tuple: (t, m). The encoding of time in logical messages results in synchronization without a global clock.

An LP simulates the corresponding PP in the following manner. Let the sequence of messages sent by LPi to LPj be (t_1, m_1), (t_2, m_2), (t_3, m_3), . . . We require that

(1) $0 \leq t_1 \leq t_2 \leq t_3 \ldots$, (monotonicity) and
(2) PPi must have sent message m_k to PPj at time t_k, $k = 1, 2, 3, \ldots$ and
(3) PPi must have sent no other messages to PPj besides $m_1, m_2, \ldots, m_k, \ldots$, i.e., the sequence of messages sent by an LP must correspond exactly to the actual sequence of messages sent by the corresponding PP. During the course of the simulation, if LPi sends LPj a message (t_k, m_k) it implies that all messages from PPi to PPj have been simulated up to time t_k.

The protocol used for message communication in the LP network is designed to ensure that the total amount of memory used by all the processes in the network is approximately the same as that used in a sequential simulation. This is achieved by using bounded (rather than infinite) buffers for communication. In the following discussion buffers of size 0 (zero) are assumed for simplicity of exposition; our algorithm works for buffers of arbitrary size. Thus, a message is transmitted from LPi to LPj only if LPi is waiting to send a message to LPj and LPj is waiting to receive a message from LPi. This protocol was proposed by Hoare [8]. (If we have nonzero size buffers between LPi and LPj, then LPi may transmit messages until the buffer is full).

Example 1

Example 1 (see Figure 2 below)

In Example 1 assume that queues 1 and 2 have a First-Cme First-Served (FCFS) discipline. The source, sink, and each queue are simulated by distinct LP's. Assume that the source produces the first customer at time 3 and the second at time 5. The generation of the first customer at the source results in a message (3, 1) from the LP simulating the source (called LP0) to the LP simulating queue 1 (called LP1); the first component of the message indicates the time (i.e., 3) customers transit from source to queue 1 and the second component denotes the number of customers (i.e., 1). Upon receipt of this message LP1 can determine that this customer will depart at time 13 (arrival time + service time). Hence LP1 can now send the message (13, 1) to LP2. Upon receipt of this message LP2 can determine that the customer will depart queue 2 at time 33. It will then send the message (33, 1) to LP3 simulating the sink. Meanwhile LP0 would have sent the message (5, 1) to LP1 to indicate the time of arrival of the next customer. LP1 will then send the message (23, 1) to LP2 (where 23 = last departure + service time). Upon receipt of this message LP2 will send (53, 1) to LP3.

Note that all the LP's could be working in parallel.

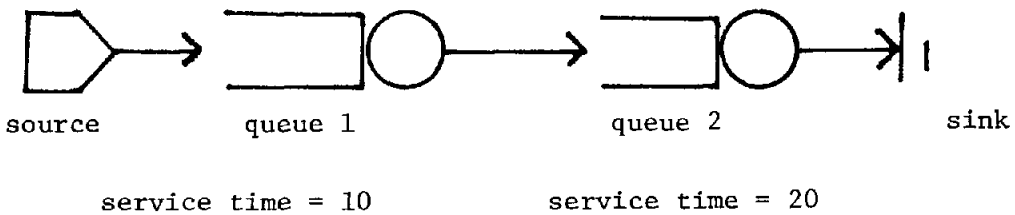


Fig. 2. Example Illustrating Message Transmission.

3. LP Operation and the Problem of Deadlock

The clock value T_i of LPi at any point in simulation is defined as the maximum time satisfying the following requirement:

All subsequent messages (t, m) sent or received by LPi *must have* $t > T_i$.

This means that

(*i*) LPi must have received a message along each input line, where the t-component of the message is greater than or equal to T_i. (Thus LPi can guarantee that all subsequent input messages will have t-components no less than T_i, from the monotonicity requirement of Sec. 2.)

(*ii*) LPi must have deduced that it will not send any message (t, m) on any output line where $t \leq T_i$. (LPi can make this deduction based on its simulation of PPi, it is not necessary for LPi to have actually sent messages (t, m); $t \geq T_i$ on every output line provided it can guarantee that t-components of all subsequent output messages will be greater than T_i).

A process i computes the *clock-value* o a line (i, j), which is a lower bound on the t-component of the next message transmitted by i along that line. Process j computes the clock-value of line (i, j) as the t-component of the last message received along (i, j). LP's i and j, in general, will compute different clock-values for line (i, j); it should be clear from the context which value is meant. Initially clock-value is zero for every line.

The operation of an LP is as follows. It alternates between computing and waiting to communicate. Whenever it is waiting, it follows these waiting rules.

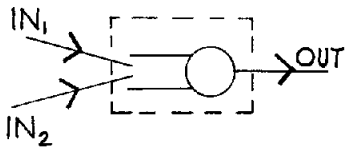
3.1 Waiting Rules for LP's

(1) An LP waits to receive messages on all input lines whose clock values equal the LP clock value.

(2) An LP waits on all output lines on which there is a message to be sent.

Upon receipt or transmission of a message, the LP enters the computational phase, which is of finite duration, to determine which set of lines it should wait on next according to the waiting rules, after which it waits to communicate once again according to the above rules.

Fig. 3. A FCFS Queue with Two Input Lines.



Example 2

Consider the PP shown in Figure 3. The PP consists of a First-Come First-Served queue fed by 2 input lines in_1, in_2 and a single output, *out*. Assume a constant service time of 8 units for every job. Initially the clock values for all lines are 0 and the LP clock value = 0. Hence, the LP waits for input on in_1 and in_2 and does not wait to output since there is no message to be output. Assume $(10, m_1)$ is received on in_1. The LP cannot compute any output with a message on in_1 alone. Now the LP waits only on in_2. Suppose $(5, m_2)$ is received on in_2. The LP can then guarantee that (1) no other message will arrive at PP before time 5 and (2) the next output will occur at $5 + 8 = 13$ corresponding to m_2. The LP clock value is now 5. The LP waits to input on line in_2, since t_2 = LP clock-value and waits to output $(13, m_2)$, since it has something to output. A possible sequence of message transmissions is displayed in Table I.

3.2 Deadlock

Deadlock can occur in a simulation. (See the following example in Figure 4.)

Example 3

Assume for this example that all jobs are of the same class. Each edge (i, j) is labeled with the pair {time-of-next-message, time-of-last-message along the line}. If LPi is waiting to send a message (t^*, m) to LPj, then the time-of-next-message along edge (i, j) is t^*. If LPi is *not* waiting to send a message to LPj, then the time-of-next-message along edge (i, j) is unknown and is represented by a question mark (?). In Figure 4 the messages are

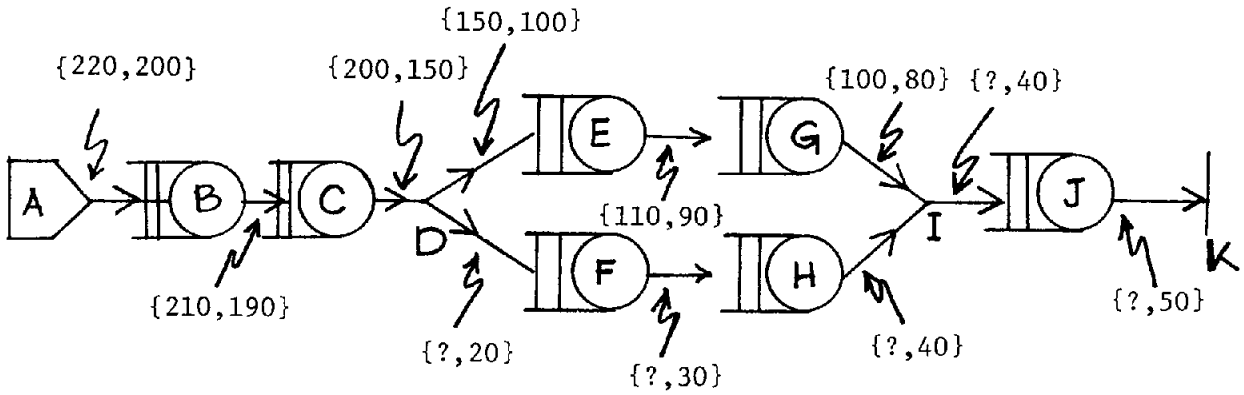


Fig. 4. Example of a System That May Deadlock. A: Source of jobs. B, C, E, F, G, H, J: First-Come First-Serve Queues. D: Probabilistic Branch Process Sends Jobs to E or F as Appropriate. I: Merge Process as in Example 3, but Without an Associated Queue. K: Sink.

Table I. Sequence of Messages in Example 2.

LP clock value	Set of lines on which the LP is waiting	Message, if any, that LP is waiting to output	Next event
5	$\{in_2, out\}$	$(13, m_2)$	$(13, m_2)$ is sent on *out*
5	$\{in_2\}$	...	$(7, m_3)$ is received on in_2
7	$\{in_2, out\}$	$(21, m_3)$[a]	$(12, m_4)$ is received on in_2
10	$\{in_1, out\}$	$(21, m_3)$	$(21, m_3)$ is sent on *out*
10	$\{in_1, out\}$	$(29, m_1)$	$(18, m_4)$ is received on in_1

[a] Service for the job m_3 cannot begin until the departure of m_2 (at 13). Hence it will depart at 13 + 8 = 21.

tuples (t, m) where m is an integer representing the number of jobs traversing the line at time t. Normally m will be 1 (one), though the possibility of batch arrivals is allowed. In the example, the source, (process A) has sent a message (200, 1) to queue B representing the entry of a single job at time 200 into queue B. Process A has determined that the next job to depart from the source will do so at time 220. Thus, process A is *waiting* to output message (220, 1). However, process B is not waiting to receive a message from A so this message cannot be sent; we therefore label the edge (A, B) {220, 200}.

The situation shown in Figure 4 is the result of the following events: Jobs are generated in the physical system at times 0, 40, 60, 80, 130, 180, 200 and 220. The jobs generated at times 0 and 180 take the DF branch at branch point D and all others take the DE branch. All service times at B, C, E, G, J are a constant 10 units and at F, H a constant 7 units.

Table II shows the times when each job crosses the corresponding line of the physical system. All entries on the upper left of the jagged line have already been simulated. Entries on the lower right cannot be simulated because of a deadlock.

The deadlock centers around the branch point D; D has already sent a message (100, 1) to E but now wishes to send a message (150, 1) to E. However, E is not waiting to receive a message. E is waiting to output (110, 1) to G and G is waiting to output (100, 1) to I. Meanwhile F is waiting to receive a message from D, having processed and sent the last message received from D at 20. D is not waiting to send a message to F. It cannot compute the time of the next job going to F since it has

Table II. Times at Which Jobs Cross Lines in Example 3.

Lines \ Job #	1	2	3	4	5	6	7	8
$A \rightarrow B$	0	40	60	80	130	180	200	220
$B \rightarrow C$	10	50	70	90	140	190	210	230
$C \rightarrow D$	20	60	80	100	150	200	220	240
$D \rightarrow E$	—	60	80	100	150	—	220	240
$D \rightarrow F$	20	—	—	—	—	200	—	—
$E \rightarrow G$	—	70	90	110	160	—	230	250
$G \rightarrow I$	—	80	100	120	170	—	240	260
$F \rightarrow H$	27	—	—	—	—	207	—	—
$H \rightarrow I$	34	—	—	—	—	214	—	—
$I \rightarrow J$	34	80	100	120	170	214	240	260
$J \rightarrow K$	44	90	110	130	180	224	250	270

not received the time of its arrival from C. Thus, the time-of-next-message from D to F is represented by a question mark. Similarly, H is waiting to receive from F and I is waiting to receive from H. The waiting relationships are shown in Figure 5, where W represents waiting and N represents not waiting.

Deadlock occurs when there is a cycle of W → N arcs that are assumed to go from W to N. In Figure 5, the cycle consists of D E G I H F D.

4. Deadlock Avoidance in Previous Asynchronous Distributed Algorithms

One scheme [3] for breaking deadlock is for LPi to send a message of the form (t, NULL) to LPj at some point in the simulation denoting that PPi does not send PPj any message in the time interval between the last message along line (i, j) and t. For instance, in Example 3, LP D could send a message (t, NULL) along an outgoing line every time it sends a message (t, m) on the other outgoing line. The NULL message does not correspond to any real message in the physical system. It

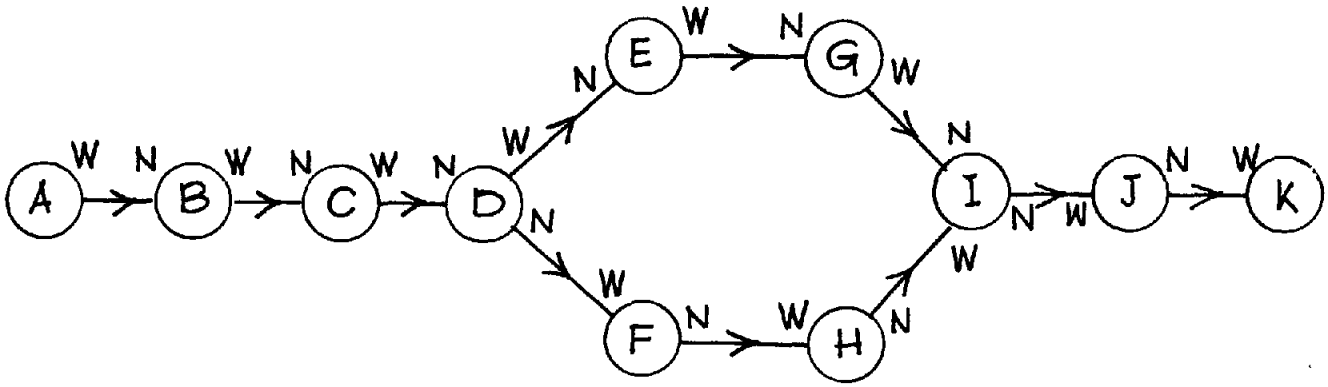


Fig. 5. Waiting Conditions at Deadlock in Example 3.

will be shown in some detail how null messages are used to avoid deadlock in Example 3. The entries corresponding to null messages are circled in Table III. For example, the message (20, NULL) is sent from D to E; LP E can then deduce that its next output to G will not occur before time 30 and hence it sends a message (30, NULL) to G, which then sends (40, NULL) to I. Note that the merge process I will output a stream of messages (34, 1), (40, NULL), (74, NULL), (80, 1), (94, NULL), etc. Thus deadlock is broken. See [3, 4] for a formal proof of absence of deadlock.

Empirical evidence [15] suggests that this approach to deadlock avoidance is expensive because a large fraction of the messages transmitted are NULL messages. Note that the presence of NULL messages causes LPJ to process about twice the number of messages that A produces. If there is a feedback path from the output of I to the input of D, a large number of "NULL jobs" will be created at D for every "real job" entering D. Every message entering D will cause a NULL message to be sent along at least one of the two outgoing edges but there is no mechanism to annihilate NULL jobs.

Peacock et al. [13] and Holmes [9] have suggested a method of detecting deadlock by using "probes". Probes are messages periodically sent out to collect status information of processes. Our approach [5], which is based on the work of Dijkstra and Scholten [6], avoids the use of probes for deadlock detection; it is possible to specify bounds on the number of message transmissions for deadlock detection in this scheme.

5. Overview of a Distributed Scheme to Detect and Recover from Deadlock

The entire algorithm works using the following sequence of computations:

(*i*) Parallel phase: Run simulation until deadlock.
(*ii*) Phase interface: Initiate a computation whereby the various LP's can advance their LP clock values.

A special process, called *controller*, is proposed which synchronizes these actions. The controller detects deadlock with a distributed scheme based on the termination detection algorithm of Dijkstra and Scholten [6] which is discussed in detail in [5]. The controller then orders the various LP's to start the phase interface computations. Upon termination of phase interface computation by any LP, the LP informs the controller of the termination of this computation and then resumes the parallel phase. The controller is a central process; however, it is not expected to be a bottleneck since the only message transmissions involving the controller take place at the terminations of the parallel phase and the phase interface computations. The sole function of the controller is to detect the termination of one phase and to initiate the next one; the controller does not carry out any computation.

Table III. Deadlock Avoidance Using Null Messages.

Lines \ job #	1	2	3	4	5	6	7	8
$A \rightarrow B$	0	40	60	80	130	180	200	220
$B \rightarrow C$	10	50	70	90	140	190	210	230
$C \rightarrow D$	20	60	80	100	150	200	220	240
$D \rightarrow E$	(20)	60	80	100	150	(200)	220	240
$D \rightarrow F$	20	(60)	(80)	(100)	(150)	200	(220)	(240)
$E \rightarrow G$	(30)	70	90	110	160	(210)	230	250
$G \rightarrow I$	(40)	80	100	120	170	(220)	240	260
$F \rightarrow H$	27	(67)	(87)	(107)	(157)	207	(227)	(247)
$H \rightarrow I$	34	(74)	(94)	(114)	(164)	214	(234)	(254)

5.1 Implementation of the Phase Interface

Upon detection of deadlock, two quantities, U_{ij} and W_{ij} for each edge (i, j), are computed that are used to advance the computations of LP's. For any edge (i, j), U_{ij} is defined to be the time-of-next-message output by LPi along edge (i, j), assuming the next message received by LPi along every input edge corresponds to time ∞ (or equivalently assuming no further input is received by LPi). Let M_{ij} be the content of the message. Thus if no further message is received along any input line, LPi should send (U_{ij}, M_{ij}) along line (i, j). U_{ij} can be computed locally at LPi; other processes, in particular LPj, may not know the value of U_{ij}.

Example 4:

Consider the system in Figure 4.

$U_{AB} = 220$, since process A is waiting to output at 220.
$U_{BC} = 210$, $U_{CD} = 200$, $U_{DE} = 150$, $U_{EG} = 110$, $U_{GI} = 100$.
$U_{DF} = \infty$, $U_{FH} = \infty$, $U_{HI} = \infty$, $U_{JK} = \infty$, for similar reasons.
$U_{IJ} =$ 80, since if (∞, m) is received along U_{HI}, then the next output along (I, J) will be at 80.

The definition of U_{ij} is derived from sequential simulation concepts. In a sequential simulation, each process posts into the event-list the time t of its next output assuming it receives no further inputs. The event with the smallest time in the event-list is guaranteed to be the next event in the physical system. The U_{ij} are used for a similar purpose. However, unlike sequential simulation, many events can be concurrently initiated when deadlock is broken.

Let U_{kr} be the minimum of all U_{ij}. It is guaranteed that LP k will not receive any further messages with time components less than U_{kr}, for the same reason as in sequential simulation. Therefore, LP k can send out its next message (U_{kr}, M_{kr}). One scheme to break deadlock

is to compute $U_{kr} = \min_{all\ i,j} \{U_{ij}\}$ in a distributed manner and then signal LP k to resume computation and send message (U_{kr}, M_{kr}).

The performance of the simulation can be improved by starting up *many* LP's. However, it must be guaranteed that if LPi is started up to send a message (t^*, m^*) to LPj, then all succeeding messages *to* LPi must have time-components greater than t^* so they cannot impact the time or the content of the next message sent by LPi. Therefore, the goal is to find the best possible lower bound on the time-component of the next message to be transmitted along (k, r) for every (k, r). Obviously the minimum over all U_{ij} is a lower bound, but it can be improved on.

If there is a directed path from (i, j) to (k, r), then edge (i, j) is said to be an *ancestor* of (k, r).

Example 5

In Figure 6, (2, 4) is an ancestor of (6, 8) whereas (1, 3) is not. Obviously only those messages along ancestor lines can affect messages along any given line. Hence, in the above example, messages along (1, 3) cannot impact messages along (6, 8) whereas messages along (2, 4) may. Thus, it it is obvious that the time-component t^* of the next message transmitted along (k, r) must be greater than or equal to min $\{U_{ij}\}$ where the minimum is taken only over ancestors of (k, r) and not over all edges of the network. In this example, a lower bound for t^*, the time component of the next message on (6, 8), is the minimum of $U_{2,4}$, $U_{4,6}$, and $U_{6,8}$.

An even tighter bound on t^* can be obtained as follows. Consider any path $(i, j), \ldots, (x, y)$. Suppose LPx is waiting to output $(\hat{t}, \hat{m})$ to LPy. No message transmitted along (i, j) can alter the fact that the next transmission along (x, y) is $(\hat{t}, \hat{m})$. Therefore, the best lower bound W_{ij} on the time of the next message on each line (i, j) must satisfy the following equation:

$$W_{ij} = \begin{cases} \hat{t}, \text{ if LP}i \text{ is waiting to output a message } (\hat{t}, \hat{m}) \text{ to LP}j \\ \max\left(t_{ij}, \min[U_{ij}, \min_r \{W_{ri}\}]\right), \text{otherwise} \end{cases}$$

The first case in the above equation follows from the previous paragraph. The argument for the second case is as follows. The earliest time component of the next message received by LPi is $W_i^* = \min_r\{W_{ri}\}$. Now consider three cases.

Case (a) $W_i^* < t_{ij} \leq U_{ij}$

The next message on line (i, j) must have a time component greater than the last message on the line. Hence we set $W_{ij} = t_{ij}$. Note that (in general) LPi can send a message with thetime component arbitrarily close to t_{ij}. Thus, a better bound cannot be obtained.

Case (b) $t_{ij} \leq W_i^* \leq U_{ij}$

LPi may send a message with time component W_i^* as a consequence of receiving a message with time component W_i^*. Hence we set $W_{ij} = W_i^*$.

Case (c) $t_{ij} \leq U_{ij} < W_i^*$

In this case LPi will definitely send a message with time component U_{ij}. Because future inputs to a PP cannot affect its past output, it follows that PPi will send a message to PPj at time U_{ij} if it receives its next input (if any) after U_{ij}. In this case, we set $W_{ij} = U_{ij}$.

An algorithm for computing the lower bounds W_{ij}'s is found later in this section; it is also shown that every LP can determine for itself that its computation of W_{ij}'s is complete.

Example 6

Consider the system in Figure 4.

Using the equations describing W_{ij}, we have,

$$W_{AB} = 220,\ W_{BC} = 210,\ W_{CD} = 200,\ W_{DE} = 150,$$
$$W_{EG} = 110,\ W_{GI} = 100,$$
$$W_D^* = W_{CD} = 200$$
$$W_{DF} = \max(t_{DF}, \min(W_D^*, U_{DF})) = \max(20, \min(200, \infty)) = 200$$

Similarly,

$$W_{FH} = W_{HI} = 200$$
$$W_I^* = \min(W_{HI}, W_{GI}) = 80$$
$$W_{IJ} = \max(40, \min(100, 80)) = 80$$
$$W_{JK} = 80$$

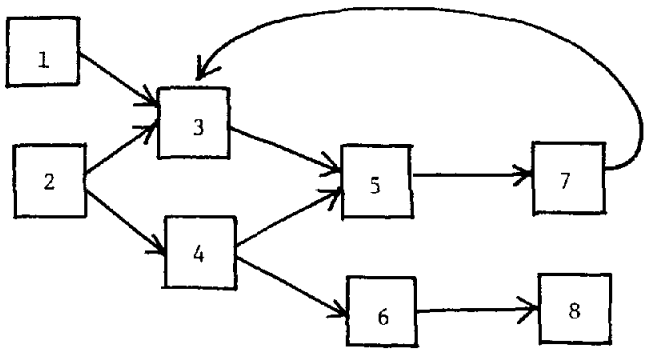


Fig. 6. A Possible LP Network.

5.2 Computation of W: Computing Lower Bounds on Time of Next Events

The following computation procedure for W_{ij} is motivated by the definitions given earlier. A proof of correctness of this procedure appears in Sec. 6.

Let $W_{ij}^{(1)} = U_{ij}$

$$W_{ij}^{(k+1)} = \begin{cases} \hat{t}_{ij}, \text{ if LP}i \text{ is waiting to send } (\hat{t}_{ij}, \hat{m}_{ij}) \text{ to LP}j \\ \max(t_{ij}, \min[U_{ij}, \min_r \{W_{ri}^{(k)}\}]), \text{otherwise} \end{cases}$$

Then $W_{ij} = W_{ij}^{(n)}$.

The computation of $W_{ij}^{(k)}$, for all lines i, j and $1 \leq k \leq n$ can be carried out in a distributed manner as follows.

Table IV. Message Transmissions After the Deadlock is Broken.

$\langle (I, J), (80, 1) \rangle$	$\langle (G, I), (100, 1) \rangle$	$\langle (I, J), (100, 1) \rangle$	$\langle (D, E), (150, 1) \rangle$
	$\langle (J, K), (90, 1) \rangle$	$\langle (E, G), (110, 1) \rangle$	$\langle (G, I), (120, 1) \rangle$
			$\langle (J, K), (110, 1) \rangle$

The computation at each LP consists of n cycles, 1 through n, wherein the kth cycle LPi computes $W_{ij}^{(k)}$ and sends it to LPj, for every outgoing line (i, j). LPi also receives $W_{ri}^{(k)}$ along every incoming line (r, i) during the kth cycle, which enables LPi to carry out computation in the $(k + 1)$th cycle. Note that there is no possibility of deadlock since every LP is waiting to communicate along every incident line in every cycle.

5.3 Restarting LP's—How the Network Resumes after Deadlock

Upon completion of the computation of W, LPi can be assured that it will receive no message with time component less than W_{ri} on input line (r, i), nor will it send out a message with time component less than W_{ij} on line (i, j). Thus, the clock value of every line incident on LPi is updated to reflect this fact. LPi also updates its LP clock values and then starts waiting to input or output according to the waiting rules given in Sec. 3.1. LPi is defined to be *resumable* if the set of lines it waits on at this point are different from the set of lines it was waiting on at the point of deadlock. Thus, at the end of the W-computation, each LP can determine if it is resumable. Each LP then sends a signal to the controller stating whether or not it is resumable. An LP is free to send and receive messages for the next computation phase (i.e., continuing the simulation) after sending this signal to the controller. It is shown in Sec. 6 that there is at least one resumable process and that at least one message will be transmitted before the next deadlock.

Example 7

Consider the system in Figure 4 and U, W as given in Examples 4 and 6. For LPI, $W_{HI} = 200$ and hence node I may increase its clock value to 80 which makes it wait on input line G, I. It also waits on output line I, J, since it has (80, 1) to output along I, J. Since LPI was previously waiting only on H, I, LPI is thus resumable. Table IV shows a possible scenario of message transmissions from this point.

The entry $\langle (i, j), (t, m) \rangle$ in column k of Table IV denotes that message (t, m) is sent along (i, j) in the kth simulation step. Note that many processes may transmit messages in parallel.

5.4 Summary of the Algorithm

5.4.1 *Algorithm for Each* LP

```
initially: clock value is 0 for every line
        (hence LP clock value is 0 for every LP);
        Define this LP to be resumable if it is
        waiting to output along some line;
loop:
   communicate with controller:
      {this phase is entered initially and upon completion of W-com-
      putation}
      send a signal to controller stating whether or not this LP is
      resumable;
   simulate:
      use waiting rules of Sec. 3.1 to send and receive tuples (t, m);
   compute W:
      {this phase is entered upon detection of deadlock. Controller sends
      a signal to each LP to enter this phase.}
      Use algorithm of Sec. 5.2 to compute U and W.
end-loop
end-algorithm-for-LP.
```

5.4.2 *Algorithm for the controller*

```
loop:
   receive signals from all LP's as to whether or not they are resumable;
   receive signals denoting deadlock;
   send signals to all LP's to initiate W-computation
end-loop
end-algorithm-for-controller.
```

Note: It is not immediately obvious why LP's communicate their resumable status to the controller. The LP's do so because of the way the deadlock detection algorithm is structured in [5]. The controller detects deadlock only when it receives signals from *all* previously resumable processes. Therefore, at the end of W-computation the controller needs to keep a list of all resumable processes for the next deadlock detection.

6. Proofs of Properties of the Algorithm

The following properties of the algorithm are proved in this section.

(1) No LP will terminate in an infinite horizon simulation, i.e., every LP must either be computing or waiting to communicate.
(2) The W_{ij}'s computed in Sec. 5.2 satisfy the definition in Sec. 5.1.
(3) Following the W-computation, there exists at least one resumable process and at least one message must be transmitted before the next deadlock.

It follows therefore that the algorithm can continue to simulate up to any point in physical time.

LEMMA 1. *According to the waiting rules of Sec. 3.1, an* LP *must wait to communicate along at least one incident line.*

PROOF: Let t_{xy} denote the clock value of line (x, y) at any point in simulation. If LPi does not wait on line (i, j) because it has nothing to output, then $t_{ij} \geq \min_r \{t_{ri}\}$; this is because all outputs up to $\min_r \{t_{ri}\}$ can be predicted for every output line from the realizability condition of Sec. 1. Therefore, if LPi waits on no output line, then $t_{ij} \geq \min_r \{t_{ri}\}$ for every output line (i, j) and hence $\min_j \{t_{ij}\} \geq \min_r \{t_{ri}\}$. Recall that the clock value of LPi

$$= \min_{r, j} \{t_{ri}, t_{ij}\}$$

$$= \min_r \{t_{ri}\}, \quad \text{if} \quad \min_j \{t_{ij}\} \geq \min_r \{t_{ri}\}$$

Hence, if LPi is waiting on no output line, it must wait on all input lines having the minimum clock value.

THEOREM 1: *The W_{ij}'s computed in Sec. 5.2 satisfy the definition in Sec. 5.1.*

PROOF: We show that $W_{ij}^{(k)}$ is a lower bound on the time component of the next message transmitted along line (i, j) *assuming that no further message is transmitted on lines of distance k or more*[1] *from line* (i, j), i.e., assuming $U_{xy} = \infty$, if (x, y) is at distance k or more from (i, j). It then follows that $W_{ij}^{(n)}$ meets the definition of W_{ij} in Sec. 5.1, where n is the number of LP's.

The proof is by induction on k. For $k = 1$, every line (x, y), $(x, y) \neq (i, j)$, is at distance k or more from (i, j); hence $W_{ij}^{(1)} = U_{ij}$, if all $U_{xy} = \infty$.

Assume that the claim holds for all j, $1 \leq j \leq k$. Any line (x, y) at a distance of $(k + 1)$ or more from line (i, j) must be at a distance of k or more from any input line (r, i) to process i. Hence, $\min_r \{W_{ri}^{(k)}\}$ is a lower bound on the time component of the next message transmitted to LPi provided $U_{xy} = \infty$ if line (x, y) is at a distance of $(k + 1)$ or more from (i, j). The next message transmitted along line (i, j) must have a time component of at least t_{ij} (from monotonicity condition) and at most U_{ij} (from the definition of U_{ij}). Combining this with the meaning of $\min_r \{W_{ri}^{(k)}\}$ given above, the theorem is proven.

LEMMA 2: *There is a line (y, z) such that,*

$$U_{yz} = W_{yz} = \min_{(i,j)} \{W_{ij}\}$$

PROOF: If there are several lines with the same minimum value of W, then pick that line (y, z) with the minimum value of k such that $W_{yz}^{(k)} = W_{yz}$, in the computation procedure of Sec. 5.2. It is shown that $W_{yz} = U_{yz}$. According to the procedure of Sec. 5.2, $W_{yz} \leq U_{yz}$. Let W_y^* denote $\min_x \{W_{x,y}\}$. If $W_{yz} < U_{yz}$, then $W_y^* \leq W_{yz}$. If $W_y^* < W_{yz}$, then $\min_{(i,j)} \{W_{ij}\} \neq W_{yz}$. If $W_y^* = W_{yz}$, then for some line (x, y), $W_{xy} = W_{yz}$ and line (x, y) has lower k such that $W_{xy}^{(k)} = W_{yz}$. Contradiction, therefore $W_{yz} = U_{yz}$.

THEOREM 2: *Following the W-computation, there exists at least one resumable process and at least one message must be transmitted before the next deadlock.*

PROOF: Any LPy can move its clock up to $\min_{x,z} \{W_{xy}, W_{yz}\}$, following the W-computation. Consider the line (y, z) as defined in LEMMA 2. Clearly both LPy and LPz can move their LP clocks up to W_{yz}. LPz will start waiting on line (y, z) since it now has the minimum clock value. LPy will start waiting on line (y, z) since, from LEMMA 2, $W_{yz} = U_{yz}$ and therefore LPy must have something to send on this line.

Not both LPy and LPz were waiting on line (y, z) at the time of deadlock (otherwise there would not have been a deadlock). Since at the end of W-computation, they are both waiting on line (y, z), at least one of these LP's must have changed the set of lines it is waiting on and hence is resumable. Furthermore, there cannot be another deadlock until at least either LPy or LPz changes the set of lines it is waiting on—this can happen only after at least one message gets transmitted.

7. Discussion

Sequential simulation and its associated event-list mechanism is the point of departure for this paper. One possible way to view the proposed algorithm in terms of the event-list mechanism is that this scheme allows several events in the event-list to take place simultaneously when it can be guaranteed that the events so chosen cannot be impacted by any event occurring in the future. This causes parts of the simulator to be far ahead of others. However, the asynchronous approach causes deadlock. Deadlock is detected in a distributed manner. Then a scheme derived from sequential simulation is used to determine the processes at which the next set of events are guaranteed to occur. These processes are given control and the procedure repeats.

Our algorithm may be profitably used with queueing network simulations where processes are queues or routing nodes and messages correspond to jobs or "tokens" [14]. The topology of process interaction (and consequently, of paths between processors in the simulation) is described in the network model. An elegant scheme for describing the topology is found in RESQ [14]. Processes in RESQ are described either by defining he process type (First-Come First-Served, preemptive, etc) from a system-defined set of process types or by writing simulation procedures for user defined types. It is conceptually straightforward to compile a RESQ program for a distributed simulation architecture. Information control nets [5] are also suitable for distributed simulation. It is more difficult to compile simulations written in other languages, though the problems are implementation-related rather than fundamental.

Work on infinite buffer simulation models was carried out by Bryant [1] and Peacock, Wong, and Manning [11–13]. Peacock et al. [11] provide a very comprehensive study of distributed simulation in general. The major difference between these approaches and ours is that ours does not have infinite buffers. Thus, deadlock is possible bcause processes cannot output (due to buffer size limitations) which cannot happen in the other schemes. In the infinite buffer case, deadlocks can occur only if all processes are waiting for input; the action of breaking deadlock is relatively straightforward in this case.

Nutt [10] presents an interesting distributed fixed time increment simulator. His model uses information

[1] A line (x, y) is at distance $k \geq 1$ from (i, j) where $(i, j) \neq (x, y)$ if the shortest path from (x, y) to (i, j) is $(x_0, x_1, \ldots, x_k, x_{k+1})$ where $x_0 = x$, $x_1 = y$, $x_k = i$, $x_{k+1} = j$. The distance of a line to itself is assumed to be 0 (zero).

control nets. He has described process, message types which implement his algorithm in detail. Our approach departs from Nutt's in its basis in discrete-event simulation. As in sequential simulation there are cases where discrete-event approaches are preferable to time driven simulations and there are cases where the reverse is true.

The running time of the distributed algorithm depends upon the model being simulated. It is known empirically [15] that the distributed scheme approaches ideal performance when there are no multiple loops in the network. Extensive experimentation with various models is necessary in order to predict the performance of the proposed algorithm.

Received 2/80; revised 9/80; accepted 12/80

References

1. Bryant, R. E. Simulation of packet communication architecture computer systems. M.I.T. Lab. Comptr. Sci., M.S. Thesis, Nov. 1977.
2. Chandy, K. M., Holmes, V., and Misra, J. Distributed simulation of networks. *Comptr. Networks 3,* 1 (Feb. 1979), 105–113.
3. Chandy, K. M. and Misra, J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering, SE-5,* 5 (Sept. 1979), 440–452.
4. Chandy, K. M. and Misra, J. Deadlock absence proofs for networks of communicating processes. *Information Processing Lett. 9,* 4, (Nov. 1979), 185–189.
5. Chandy, K. M. and Misra, J. Termination detection of diffusing computations in communicating sequential processes. Dept. of Comptr. Sci., Tech. Rept, TR-144, 1980, University of Texas, Austin, TX.
6. Dijkstra, E. W. and Scholten, C. S. Termination detection for diffusing computations. EWD687a, 5671 AL Nuenen, The Netherlands.
7. Ellis, C. A. Information control nets: A methematical model of office information flow. *Proc 1979 Conf. on Simulation, Measurement and Modeling of Computer Systems.* (Aug. 1979), 225–239.
8. Hoare, C. A. R. Communicating sequential processes. *Comm. ACM 21,* 8, (Aug. 1978) 666–677.
9. Holmes, V. Parallel algorithms for multiple processor architectures. Ph.D. Dissertation, Comptr. Sci. Dept. Univ. of Texas, Austin, TX, 1978.
10. Nutt, G. J. An experimental distributed modeling system. Tech. Rept, Jan. 1980, Xerox Palo Alto Research Center, Palo Alto, CA 94305.
11. Peacock, J. K., Wong, J. W., and Manning, E. G. Distributed simulation using a network of processors. *Comptr Networks, 3,* 1 (Feb. 1979), 44–56.
12. Peacock, J. K., Wong, J. W., and Manning, E. G. Synchronization of distributed simulation using broadcast algorithms. *Proc of the Winter Simulation Conference,* December, 1979.
13. Peacock, J. K., Wong, J. W., and Manning, E. G. A distributed approach to queueing network simulation. *Proc. 4th Berkeley Conf. on Distributed Data Management and Computer Networks,* Berkeley, CA, August, 1979, 237–259.
14. Sauer, C. H. Characterization and simulation of generalized queueing networks. RC-6057, IBM Research, Yorktown Heights, NY, May, 1978.
15. Seethalakshmi, M. Performance analysis of distributed simulation. M.S. Rept, 1978, Comptr. Sci. Dept., Univ. of Texas, Austin, TX.

Simulation Modeling and Statistical Computing
N. Adam Guest Editor

Use of Polya Distributions in Approximate Solutions to Nonstationary $M/M/s$ Queues

Gordon M. Clark
The Ohio State University

Delays are important processes represented by continuous simulation models; however, representing queueing delays efficiently within continuous simulations merits the development of new methodology. Rothkopf and Oren introduced the concept of using a surrogate distribution, viz., the negative-binomial, as a closure approximation to the infinite set of Chapman-Kolmogorov equations representing a nonstationary $M/M/s$ queue. The method presented in this paper uses the Polya-Eggenberger distribution as a surrogate for the true distribution of the number in the queueing system at a particular time and only requires the numerical integration of five differential equations. The paper presents numerical results comparing the Polya surrogate and Rothkopf and Oren's approximation for a number of diverse cases, and these results indicate that the Polya surrogate is, in general, more accurate, although exceptions were encountered. Moreover, queueing delays represented by a closure approximation involving a surrogate distribution, in particular, the Polya, are suitable for use within a larger continuous simulation.

Key Words and Phrases: continuous simulation, queueing delays, $M/M/s$ queue, queueing approximation, system dynamics.

CR Categories: 5.5, 8.1,

Working Paper Series Number 1980-004

Author's Present Address: Gordon M. Clark, Industrial and Systems Engineering, The Ohio State University, Columbus, OH 43210.