Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A UNIFYING FRAMEWORK FOR DISTRIBUTED SIMULATION

R. Bagrodia
K. M. Chandy
W.-T. Liao

August 1989
CSD-910071

A Unifying Framework for Distributed Simulation

R. Bagrodia
Computer Science Dept
UCLA, Los Angeles, CA 90024

K.M.Chandy
Computer Science Dept
Caltech, Pasadena, CA 91125

Wen-toh Liao
Computer Science Dept
UCLA, Los Angeles, CA 90024

August 23, 1991

Abstract

This paper presents a theory of distributed simulation, which is applicable to both discrete-event and continuous simulation. It derives many existing simulation algorithms from the theory, and describes an implementation of a new algorithm derived from the theory. A high-level discrete-event simulation language has been implemented, using the new algorithm, on parallel computers; performance results of the implementation are presented.

1 Introduction

The goals of this paper are to:

1. Present a simple unifying theory of simulation that encompasses many different methods and which treats both discrete-event and continuous simulation within the same framework.
2. present new algorithms derived from the theory.
3. Demonstrate efficiency of the new algorithm in the simulation of deterministic and stochastic applications.

We begin by giving a specification of the simulation problem. Next, we describe a nondeterministic simulation algorithm, called the **space-time** algorithm [CS89b]. The nondeterministic algorithm is, in effect, a class of algorithms; different deterministic algorithms can be derived from the nondeterministic one by making different deterministic choices in place of nondeterministic choices. We derive several distributed algorithms from the space-time algorithm including a new algorithm that forms the basis for a distributed implementation of a high-level simulation language. Performance results of the implementation are presented.

The development of algorithms in this paper follows the UNITY methodology [CM88b]: We derive a nondeterministic algorithm from the specification and then restrict the nondeterminism to obtain efficient implementations for different architectures and problem domains.

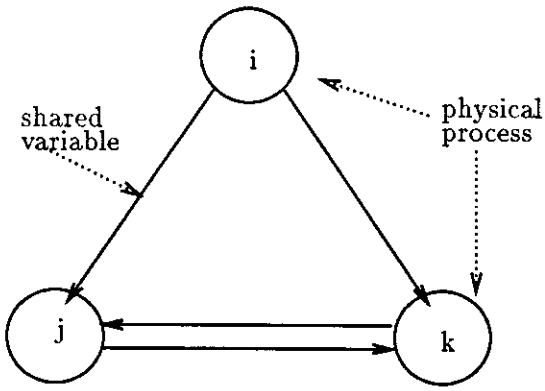

Figure 1: Physical System

2 Specification of Simulation

2.1 Physical System

We are required to simulate a **physical system** in a continuous time interval $[0, H]$.

A physical system is a (nonempty) directed graph where the vertices are **physical processes** or **PPs**, and a directed edge from PP i to PP j is a variable that is modified by PP i and read by PP j, see figure 1.

Notation We use the symbols t,u and v for time, where $0 \leq t, u, v \leq H$. We use symbols i and j for PPs.

Let $S(j,t)$ be the state of j at t. For edge (i,j) in the graph, let $M(i,j,t)$ be the value, at t, of the variable modified by i and read by j. We refer to $M(i,j,t)$ as an output of i and an input of j at time t.

For any given j, t, let $M(*,j,t)$ be the vector of elements $M(i,j,t)$ for all edges (i,j). Thus $M(*,j,t)$ is the vector of inputs of j at t. Let $M(i,*,t)$, be the vector of outputs of i at t, and let $M(*,*,t)$ have the analogous meaning. For any times u, v where $u \leq v$, let $M(i,j,[u,v])$ be the values $M(i,j,t)$ for all t in the interval $[u,v]$. Let $S(j,[u,v])$ and $S(*,t)$ have analogous meanings.

2.2 Physical Processes

A PP i is defined by

1. a set of states, an initial state $S(i,0)$, and a state-transition function f_i, and
2. a shared-variable-modification function g_i.

The state-history and outputs of a PP in a time interval, are functions of its state at the beginning of the interval and its inputs in the interval.

$$S(i,[u,v]) = f_i(S(i,u), M(*,i,[u,v])) \tag{1}$$

$$M(i,*,[u,v]) = g_i(S(i,u), M(*,i,[u,v])) \tag{2}$$

2.3 The Lookahead Restriction

A Lookahead Physical Process A PP is a **lookahead** PP [CM88a] if and only if there is some positive constant ϵ such that its inputs at any time t do not affect its outputs until time $t + \epsilon$ or later. Therefore, i is a lookahead PP if and only if there exists a positive constant ϵ and function e_i such that the output of i in any time interval $[u, u + \epsilon]$ is a function e_i of its state $S(i, u)$ at time u, and is independent of its input in that interval.

$$M(i, *, [u, u + \epsilon]) = e_i(S(i, u)) \tag{3}$$

Therefore, the outputs of a lookahead PP i in an interval $[u, v]$ depend only on its inputs in $[u, v - \epsilon]$, and hence there exists function h_i, such that:

$$M(i, *, [u, v]) = h_i(S(i, u), M(*, i, [u, v - \epsilon])) \tag{4}$$

All PPs that have no inputs are lookahead PPs.

Restriction We restrict attention to physical systems in which every cycle contains at least one lookahead PP, and in which functions f_i, g_i, e_i and h_i are total functions (i.e., they are defined for all possible values of their arguments).

The restriction of at least one lookahead PP in every cycle is to rule out systems such as a queueing network in which a job traverses a cycle in the network, an infinite number of times, instantaneously.

The restriction that the functions defining a PP be total is to avoid PPs whose behaviors cannot be simulated for certain inputs.

2.4 The Problem

Given the set of PPs of a physical system, compute $S(*, [0, H])$, the states of all PPs for all t, and $M(*, *, [0, H])$, the values of all variables for all t that satisfy equations (1) and (2).

2.5 Extensions to the Specification

Varying Numbers of Physical Processes The number of PPs in a physical system may vary, but for our purposes it suffices to assume that there is an arbitrarily large fixed number of PPs. We can represent the creation and termination of PPs as follows. Each PP has two special states *dormant* and *finished* in addition to other states. A PP is in the *dormant* state until it is created, and it enters the *finished* state when it terminates execution.

Multiple Readers A shared variable is written by at most one PP and read by at most one PP. A shared variable that has many readers can be represented by many pairwise shared variables.

Lookahead Every cycle in the network need not have a lookahead PP. It is sufficient if for any interval $[u, u + \epsilon]$, at least one PP in the cycle satisfies equation (3); different PPs in the cycle can satisfy (3) for different values of u. For simplicity, we assume that every cycle has a single PP which satisfies (3) for all u.

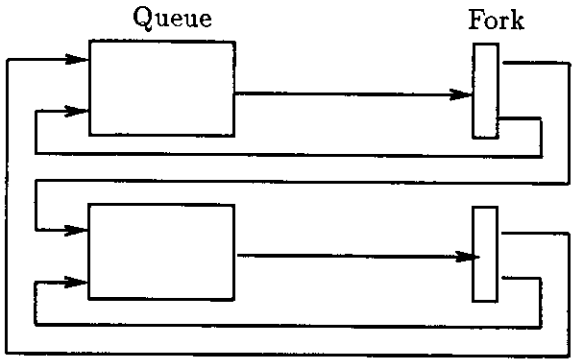

Figure 2: Example: Queuing Network

2.6 Examples

An Air Pollution Simulation In an air-pollution model, PPs are grid points in a 3-dimensional mesh. There are edges between PPs responsible for simulating neighboring grid points. An example of shared variable written by (the PP responsible for simulating) one grid point and read by a neighboring grid point is the concentration of ozone. The lookahead restriction limits air-pollution models that we consider to those in which the behavior of a grid point in an arbitrarily small positive time step is independent of the behavior of other grid points in the same time step.

Simulation of a Queueing Network In a queueing network, PPs are queues, forks (at which jobs are routed in different directions) or joins (at which jobs coming from different directions are merged). The edges of the physical system correspond to edges of the network. An example of a shared variable value $M(i, j, t)$ is the descriptor of the job traversing an edge from i to j at t; a special symbol, say *null* is used if there is no job on the edge at t. The lookahead restriction limits networks to those in which a job cannot traverse a loop instantly.

2.7 Running Example

In the rest of the paper, we will use the following closed queuing network as a running example: the network consists of N queues, each of which is a first-in-first-out ($fifo$) server. An incoming job is served by the server and forwarded to a fork process. The fork process routes incoming jobs, with equal probability, to one of the N queues. Each queue is initially assigned J jobs. Consider an instance of this network with N=2, J=1 and service time 2 for each server. The physical processes in the system are the two server ($s1$, $s2$) and two fork ($f1$ and $f2$) processes connected as shown in figure 2. The state of the server is represented by the remaining service time of jobs in the queue. The output of a server at time t, $M(s1, f1, u)$ is 1, if a job departs the server at time t and 0 (or *null* otherwise). The state of a fork is empty. Outputs of a fork are similar to the output of a server; for instance $M(f1, s1, u)$ is 1 if a job departs $f1$ for $s1$ at time u, and 0 otherwise. Each server is a lookahead PP with $\epsilon = 2$.

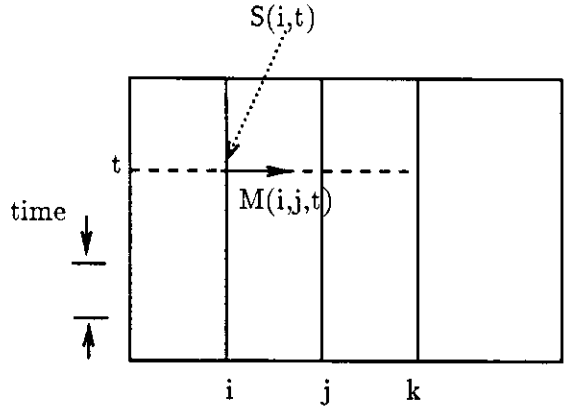

Figure 3: Time-Line Diagram

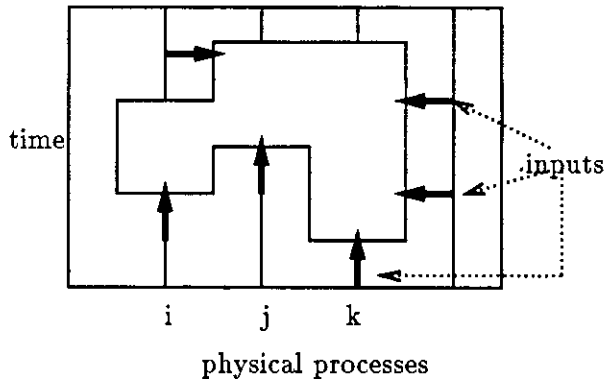

Figure 4: Space-time region

3 Space-Time

3.1 Space-Time Regions

Graphical Representation A physical system can be represented by a time-line diagram with a vertical line for each process, and the vertical axis — the y-axis — representing time, see figure 3.

Let point (i,t) be the point at height t on the time-line for PP i. Associate with each point (i,t), the outputs $M(i,*,t)$, the inputs $M(*,i,t)$ and the state $S(i,t)$ of PP i at time t. We represent $M(i,j,t)$ by a horizontal directed line from point (i,t) to point (j,t). Point (i,t) represents the behavior of PP i at time t.

The time-line diagram is called space-time, with PPs occupying the space dimension.

Regions Define a region w of space-time as a set of points (i,t) in space-time. A region can have arbitrary boundaries, and need not be contiguous, see figure 4.

Inputs of a Region The inputs of a space-time region w are defined as follows (see figure 4):

1. For an edge (i,j): $M(i,j,t)$ is an input of w if (i,t) is not in w and (j,t) is in w.
2. $S(i,t)$ is an input of w if $(i,t-)$ is not in w and (i,t) is in w, where $t-$ is the instant before t.

Outputs of a Region The outputs of a space-time region w are $M(i, *, t)$ and $S(i, t)$ for all (i, t) in w.

3.2 The Space-Time Algorithm

The algorithm is described in terms of a shared-memory, concurrent computation. A computation of a set of concurrent processes is a fair interleaving of the process computations.

Our goal is to propose a correct nondeterministic simulation algorithm in which we specify as little as possible. The algorithm we describe may seem outrageously nondeterministic; but there is a reason for the nondeterminism. We want to start with a very general algorithm, so that we can derive specific algorithms from it by restricting nondeterminacy.

Variables Our goal is to compute $M(i, j, t)$ and $S(i, t)$ for all i, j, and t. We shall use lower case letters for variables in the simulation, and upper case letters for the corresponding values of the physical system; thus we use variables $m(i, j, t)$ and $s(i, t)$ in the simulation to estimate $M(i, j, t)$ and $S(i, t)$ respectively. We refer to m as an estimate of M, and to s as an estimate of S. We require that at termination of the program, the estimates are correct: $m = M$, and $s = S$. The data structures employed to represent m and s are not discussed in this paper.

Simulation of a Space-Time Region Define the simulation of a space-time region w as follows: Given the input estimates to w, compute its output estimates that satisfy equations (1) and (2) for all intervals $(i, [u, v])$ in w, with m and s substituted for M and S respectively. Thus, simulation of a space-time region sets the output estimates of the region to values that are correct for the given input estimates.

Simulation Processes We call processes employed in the simulation, **simulation processes** or **SPs** to distinguish them from PPs. (We use SPs rather than the traditional term of LPs, for logical processes, because each LP is associated with a single PP for all time, whereas an SP is associated with an arbitrary region of space-time.) There can be an arbitrary number of concurrently executing SPs. Each SP simulates an arbitrary space-time region.

The inputs and outputs of the SP are the estimates corresponding to the inputs and outputs of the region that the SP simulates. For example, consider the region w consisting of the set of points (j, t) for all t in the interval $[0, 1]$. The inputs to w are $S(j, 0)$ and $M(*, j, [0, 1])$. The inputs to an SP simulating w are $s(j, 0)$ and $m(*, j, [0, 1])$.

The variables shared by concurrent SPs are the estimates m and s. Each SP maintains local copies of shared variables. An SP reads an input shared-variable once into a local copy, and uses the local copy thereafter. A local copy of an input shared-variable of an SP is undefined until the value of the shared variable is written into it. Likewise, an SP writes its local copy once into an output shared-variable. Reading and writing of shared variables are atomic operations.

An input shared-variable of an SP may be modified after the SP copies its value into a local copy. Likewise an output shared-variable of an SP may be modified after an SP copies its local copy into it. Therefore, the value of a shared variable and its local copy in an SP can be different.

Space-time regions can be assigned to SPs in a completely arbitrary fashion subject to the fairness rule given later. For example, in figure 5, at some point some region A can be simulated by one SP, and at a later point in the computation, while simulation of A is continuing, simulation

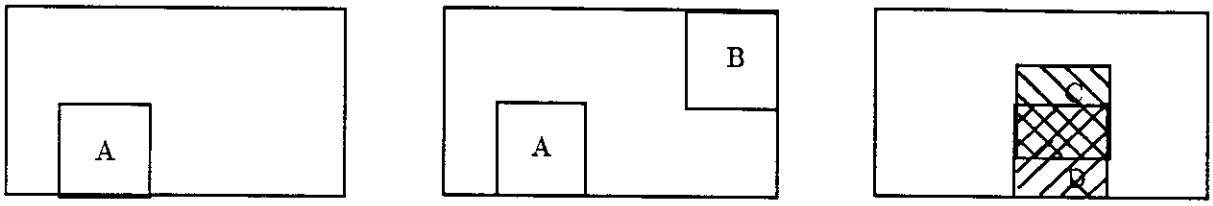

Figure 5: Assigning SPs to Compute the Space-Time Region

of some region B can be initiated. Later simulation of regions C and D can be initiated, where C and D overlap.

The Algorithm At arbitrary points in the computation, until the computation reaches a **fixed point**, initiate execution of an SP subject to the **fairness rule**.

Initialization The initial value of m is arbitrary. The initial value s is arbitrary for all times t other than $t = 0$. For $t = 0$, the initial value of s is correct.

$$s(i, 0) = S(i, 0)$$

The Fixed Point For any region w, we define a predicate *correct*(w) as follows: *correct*(w) holds if and only if the output estimates are correct for given input estimates: for all intervals $(i, [u, v])$ in w, equations (1) and (2) hold, with m and s substituted for M and S, respectively.

A **computation is at a fixed point** if and only if:

- *correct*(W) holds where W is all of space-time i.e., W is the region consisting of (i, t) for all i and all t.
- the inputs $s(i, t)$ and $m(*, i, t)$ of executing SPs, are equal to the local copies of the inputs.

Fairness Let w be the interval $(i, [u, u + \epsilon])$, for any i, an arbitrarily small positive constant ϵ, and any interval $[u, u + \epsilon]$ in $[0, H]$. Informally, the fairness condition is: There is no point in the computation after which *correct*(w) never holds and w is never simulated. We define the fairness condition as follows:

1. Eventually, *correct*(w) holds, or,
2. a finite number of SPs are eventually executed where the union of the regions simulated by the SPs includes w.

3.3 Correctness

Lemma 1 Let G be the directed graph obtained by deleting all incoming edges to lookahead PPs. Then G is acyclic, and the only PPs in G without incoming edges are lookahead PPs.
Proof Outline: The proof is based on two facts:

1. All cycles have lookahead PPs; hence deleting inputs to lookahead PPs removes all cycles.
2. All PPs without inputs are lookahead PPs.

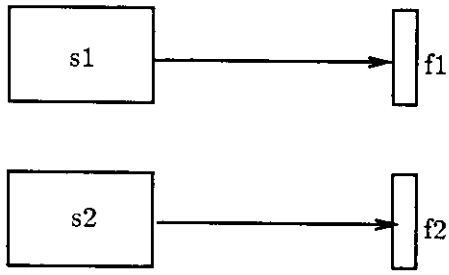

Figure 6: Acyclic Graph for Queuing Network

Running Example The graph G for the example queuing network of figure 2 is shown in figure 6.

Level of a PP Let the **level** of a PP j be the maximum length (measured in number of edges) of all paths from lookahead PPs to j in G. Therefore,

1. the level of a lookahead PP is 0, and
2. the level of a non-lookahead PP is 1 greater than the maximum level of its immediate predecessors in graph G.

For the example network, servers $s1$ and $s2$ are at level 0 and the fork processes are at level 1. We use the notation $level(i)$ for the level of i. Let the maximum level of any PP be K.

Horizontal Strips in Space-Time Define ϵ to be a positive constant such that **all** lookahead PPs have a lookahead value that is at least ϵ. (We can define ϵ to be the minimum lookahead value for all lookahead PPs.) For convenience, choose an ϵ such that H/ϵ is an integer, say N.

Partition space-time into N horizontal strips, each of size ϵ, see figure 7. The n-th strip, for $0 \leq n < N$, is the set of all points (i,t) for all i, and all t in $[n.\epsilon,(n+1).\epsilon]$.

Vertical Strips in Space-Time For convenience, order PPs in increasing order of level. Thus, PPs with lower levels are to the left of PPs with higher levels in the space-time diagram. The leftmost PPs are lookahead PPs because PPs at level 0 are lookahead PPs.

Partition space-time into $K+1$ vertical strips, where all PPs at the same level are in the same vertical strip. The k-th strip, for $0 \leq k \leq K$, is the region of all points (i,t), for all t, and all i where $level(i) = k$. (Figure 7)

Space-time Cells Define a **cell** (n,k) as the region in horizontal strip n and vertical strip k; thus it is the set of points (i,t) where $n.\epsilon \leq t \leq (n+1).\epsilon$, and $level(i) = k$. (Figure 7)

The following observations are central to the correctness of the algorithm:

1. **For each cell in the leftmost vertical strip: the output M of the cell is a function of S on the cell's lower boundary.** For all points (i,t) in cell $(n,0)$: the outputs $M(i,*,t)$ of i are a function of its state $S(i,n.\epsilon)$ at the lower boundary of the cell. This is because PPs at level 0 are lookahead PPs and, from equation (3):

$$M(i,*,[n.\epsilon,(n+1).\epsilon]) = e_i(S(i,n.\epsilon)) \tag{5}$$

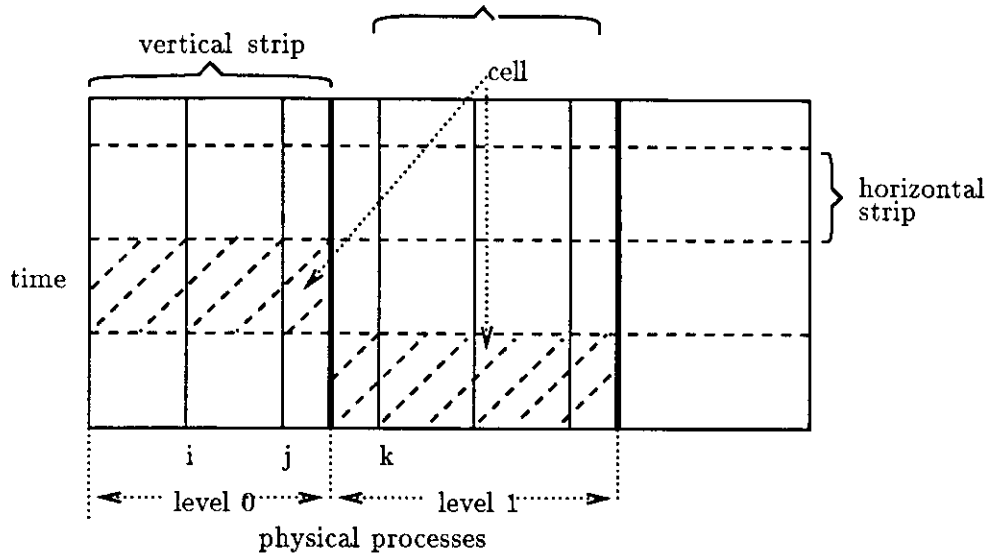

Figure 7: Cells in the Space-Time Region

Equivalently, due to equation 4, the outputs of a cell in this strip may be expressed in terms of its state and input at the lower boundary

$$M(i, *, [n.\epsilon, (n+1).\epsilon]) = h_i(S(i, n.\epsilon), M(*, i, n.\epsilon)) \tag{6}$$

2. **For cells not in the leftmost vertical strip: the output M is a function of S on the cell's lower boundary, and the outputs M of cells to its left.** For all points (i, t) in cell (n, k), where $k > 0$: the outputs $M(i, *, t)$ are functions of its state $S(i, n.\epsilon)$ at the lower boundary of the cell, and the outputs $M(j, *, t)$ of cells to its left, because the edges directed towards a PP i at level k are from PPs j with levels less than k:

$$M(i, *, [n.\epsilon, (n+1).\epsilon]) = g_i(S(i, n.\epsilon), M(*, i, [n.\epsilon, (n+1).\epsilon])) \tag{7}$$

where $M(i, i', t)$ exists only if $level(i) < level(i')$.

3. **For any horizontal strip the states $S(i, t)$ for points (i, t) in the strip are functions of the states at the lower boundary of the strip and outputs M of the strip.**

$$S(i, [n.\epsilon, (n+1).\epsilon]) = f_i(S(i, n.\epsilon), M(*, i, [n.\epsilon, (n+1).\epsilon])) \tag{8}$$

Theorem 1 Given $S(*, 0)$, equations (1) and (2) have a unique solution for M and S.
Proof Outline:

1. If there is a unique solution for M and S in the region consisting of the first n horizontal strips, $n < N - 1$, then there is a unique solution for outputs M of the first (i.e., leftmost) cell of the $(n+1)$-th horizontal strip, from equation (5).

2. If there is a unique solution for M and S in the region consisting of the first n horizontal strips, $n < N - 1$, and there is a unique solution for outputs M of the first k cells of the $(n+1)$-th horizontal strip, $k < K$, then there is a unique solution for outputs M of the $(k+1)$-th cell of the $(n+1)$-th horizontal strip, from equation (7)

3. If there is a unique solution for M and S in the region consisting of the first n horizontal strips, $n < N-1$, and there is a unique solution for M in the $(n+1)$-th horizontal strip, then there is a unique solution for S in the $(n+1)$-th horizontal strip, from equation (8).

The theorem follows by induction on horizontal strip number n, and within horizontal strip n by induction on vertical strip number k.

Indeed, one way of computing M and S is as follows. Given $S(*,0)$, compute output M for the leftmost cell of the first horizontal strip, i.e., cell $(0,0)$ using equation (5), and then compute output M for each succeeding cell to the right, using equation (7) until M is computed for the entire strip, and then compute S for the entire strip, using equation (8). Now proceed to the next horizontal strip, and repeat the procedure.

Theorem 2 Let w be the region consisting of the first n horizontal strips. If *correct*(w) holds at some point in the computation, then there is a point in the computation at which *correct*(w) holds and output $m(i,*,t)$ of the leftmost cell in the $(n+1)$-th horizontal strip is correct.
Proof: If *correct*(w) holds at any point in a computation, then it continues to hold forever thereafter; this is because if the input and output estimates are correct for all t in $[0, n.\epsilon]$ then the estimates for any t in this interval remain unchanged by the execution of any SP.

At some point in the computation, after *correct*(w) holds, the cell $(n+1,0)$ is assigned to a finite number of SPs, from the fairness rule; after termination of the SPs, $m(i,*,t)$ will be correct for all (i,t) in cell $(n+1,0)$, from equation(5).

Theorem 3 Let w be as in the previous theorem. If *correct*(w) holds and output estimates m are correct for the k leftmost cells in the $(n+1)$-th horizontal strip, $k < K$, at any point in the computation, then there is a point in the computation at which *correct*(w) holds and output estimates m are correct for the $k+1$ cells in the $(n+1)$-th horizontal strip.
The proof is similar to that of the previous theorem.

Theorem 4 Let w be as in the previous theorem. If *correct*(w) holds and output estimates m are correct for all cells in the $(n+1)$-th horizontal strip, at any point in a computation, then there is a point in the computation at which *correct*(w) holds and output estimates m and s are correct for all cells in the $(n+1)$-th horizontal strip.
Proof: Follows from equation (1).

Theorem 5 The space-time algorithm reaches fixed point in a finite number of steps.
Proof: Follows from the last three theorems by induction on horizontal strip number n, and for each n by induction on vertical strip number k.

Theorem 6 The space-time algorithm is correct: It terminates in a finite number of steps, and $m = M$ and $s = S$ at termination.
Proof: Follows from theorems 1 and 5.

4 Deriving Simulation Algorithms

In this section we show that the well-known distributed simulation algorithms are special cases of the space-time algorithm, and can be obtained by specifying the regions into which space-time is

partitioned, and the manner in which fairness is guaranteed and convergence detected. We will use the queuing network example to illustrate each algorithm discussed in this section.

4.1 Running Example

We begin with a definition of the state transition and output generation functions for both server and fork processes.

Assume time is integer. The state of a server is represented by a sequence variable *outq* and integer *loutq*. We use $outq(u)$ to refer to its value at time u, and $outq[u]$ to refer to the $u-th$ element in the sequence; $outq[u,v]$ refers to the subsequence $outq[u]..outq[v]$. Element $outq[u]$ is 1, if a job departs the server at time u and 0 (or *null*) otherwise; $loutq(u)$ is the length of sequence *outq* at time u. Initially, each server has one job in service, with remaining service time 2. As this job will depart at time 2, $soutq(0) = outq[0,3] = 0010$ and $loutq(0) = 3$.

Each shared variable is implemented by a sequence variable J; J($f1$,$s1$,t) refers to the data item at position t in the variable shared between $f1$ and $s1$. J($f1$,$s1$,t) is 1 if a job arrives at $s1$ from $f1$ at time t; similarly for other shared variables. Initially, $J(*,*,0) = 0$.

The output for a server, which is a lookahead PP, is defined using equation (3):

$$\text{J}(s1,f1,[u,u+\epsilon]) = sout(outq(u), loutq(u), u, u+\epsilon)$$

Function *sout* simply reproduces the sequence *outq* over the appropriate interval:

sout(*outq*,*l*,*u*,*v*) ≡
  if $(u > v)$ **then** *sout*:=NIL
  else *sout*:=$outq[u,v]$

The state transition function for a server process returns a 2-tuple (*loutq*,*outq*). As a server is a lookahead PP, for a given u, $loutq(u)$ is at least $[u, u+2]$. The state of a server is given by the solution to the following equation, adapted from equation (1):

$$((loutq(v), outq(v)) = sstate(outq(u), loutq(u), \text{J}(f1,s1,[u,v]), \text{J}(f2,s1,[u,v]), u, v)$$

Function *sstate* computes the departure time for incoming jobs from $f1$ or $f2$ in the interval $[u,v]$ and updates *outq* and *loutq* appropriately. Using & to represent concatenation, function *sstate* is defined as follows:

```
sstate(ina,inb,outq,l,u,v) ≡
    for t in u to v do
    {   if (ina[t]=1) then
        { outq:= outq&10; l := l + 2;}
        if (inb[t]=1) then
        { outq:= outq&10; l := l + 2;}
        if (t + ε > l) then
        { outq:= outq&0; l := l + 1;}
    }
    sstate:=(outq,l);
```

The outputs for a fork process are determined entirely by its current input and thus its state transition function is empty. Its outputs are determined by the following equations adapted from equation (2):

$$(J(f1,s1,u),J(f1,s1,u)) = fout(J(s1,f1,u),u)$$

Function $fout$ routes each incoming job randomly to one of its outputs:

$fout(ina,t) \equiv$
  if $(ina[t]=0)$ **then** $fout:=(0,0)$
  else if $(rand() \geq 0.5)$ **then** $fout:=(0,1)$
    else $fout:=(1,0)$;

Strictly speaking, $rand()$ is a pseudo-random number generator with an input parameter *seed*, which must be included in the state of the fork PP. We use the simpler definitions for brevity.

4.2 Time-Driven Simulation

Time-driven simulation computes M and S in horizontal strips of size ϵ, following the same order as the proof of space-time, as follows:

1. After computing M and S for the first n horizontal strips, compute M for the first cell in the $(n+1)$-th strip, using equation (5) or (6),
2. After computing M for the k leftmost cells in the $(n+1)$-th horizontal strip, compute M for the $(k+1)$-th cell in the $(n+1)$-th horizontal strip using equation (7).
3. After computing M for all cells in the $(n+1)$-th horizontal strip, compute S for all cells in the $(n+1)$-th horizontal strip using equation (8).

Running Example Consider simulation of the example network using this algorithm. The space-time region of this system is subdivided into horizontal strips, each having a width of 2 time units, which is the minimum lookahead for a PP in this system. The two server processes are at level 0 and the forks are at level 1. The computation for the $0-th$ strip proceeds as follows:

1. For n=0, compute outputs for the server using equation (5) on the output generation function ($sout$) for the servers. For $s1$, the output is given by:

   $$J(s1,f1,[0,2]) = sout(outq(0), loutq(0), 0, 2)$$

   From the initial conditions and the definition of function *sout*, this yields J($s1$,$f1$,[0,2])=001. A similar computation for $s2$ will yield J($s2$,$f2$,[0,2]) to also be 001.

2. For $n = 0$, compute M for the fork processes using equation (7) on the output function for the forks. $M(f1, *, [0,2])$ is determined by the following function call:

   $$(J(f1,s1,[0,2]),J(f1,s2,[0,2])) = fout(J(s1,f1,[0,2]),0,2)$$

   From the preceding step, J($s1$,$f1$,[0,2])=001, which yields the following output for $f1$ (assuming an appropriate value for $randf()$):

   $$(J(f1,s1,[0,2]),J(f1,s2,[0,2])) = (001,000).$$

   The output for $f2$ is computed similarly.

3. Finally, the state of the server processes in the interval [0,2] is computed by applying equation (8) to update the state for a server. The computation for $s1$ is as follows:

$$((loutq([0,2]), outq([0,2])) = sstate(outq(0), loutq(0), \mathrm{J}(f1,s1,[0,2]), \mathrm{J}(f2,s1,[0,2]), 0, 2)$$

Using the initial conditions and the outputs from the preceding step, this yields $loutq(2)$=5 and $outq(2)$=001010.

The outputs and states for subsequent strips may be computed similarly.

4.3 Sequential Simulation

Assign one SP to each PP. Set the initial values of $m(i,j,t)$ to a special default symbol, say *null*, for all i, j, and t. Compute $s(i,t)$ and new estimates $m'(i,*,t)$ for each i up to the earliest time t_i at which some $m'(i,j,t) \neq m(i,j,t)$; therefore, t_i is the earliest time at which some $m'(i,j,t)$ is not *null*. Let $tmin$ be defined as:

$$tmin = min_i\ t_i$$

If $tmin$ is unique, the corresponding output estimate is generated as the next output for the simulation:

$$M(i,j,tmin) = m'(i,j,tmin)$$

and execution is continued by treating $tmin$ as time 0 and repeating the previous steps.

If $tmin$ is not unique, a unique next output must be chosen using additional criterion[Mis86]. For simplicity, we assume that the next output is selected using tuple comparison for all $m'(i,j,t)$ that have $t = tmin$, and proceed as before.

Values of m and s are computed in horizontal strips of different sizes in sequential simulation. The sizes of the strips varies from step to step. If the initial assumption that m is *null* is correct for large horizontal strips, then the sequential algorithm converges to a fixed point quickly; if the assumption is correct for only small strips, then the algorithm converges slowly.

Running Example Based on the initial conditions, the earliest non-null estimates are generated by $s1$ and $s2$: $m'(s1,f1,2)$ and $m'(s2,f2,2)$. Using tuple comparison, generate $J(s1,f1,2) = m'(s1,f1,2)$. At time 2, the earliest non-null events are: $m'(s2,f2,2)$ and $m'(f1,s1,2)$, which generates $J(f1,s1,2) = m'(f1,s1,2)$. By applying these steps repeatedly, the estimates are transformed into outputs in the sequential order of their time value, such that each output is written only *once*.

4.4 Time-Warp

Each PP is simulated by one SP [Jef85]. As in sequential simulation, initially $m(i,j,t)$ is *null* for all i, j and t. Each PP i computes new estimates $m'(i,*,t)$. If $m'(i,j,t) \neq m(i,j,t)$ for some j and t, then PP i sends a message to PP j indicating that $m(i,j,t)$ has been set to $m'(i,j,t)$; the message is of the form $+m'(i,j,t) - m(i,j,t)$, a *negative* message canceling the old value and a *positive* message inserting the new value. This form of cancellation has been called lazy cancellation[Gaf88]. For Time-Warp with aggressive cancellation[Jef85], on computing an estimate $m'(i,j,t) \neq m(i,j,t)$, PP i must additionally cancel $m(i,j,t')$, for all $t' \geq t$. (In the interests of efficiency $+null$ and $-null$ messages need not be sent.) If an SP gets a message with timestamp t it repeats the simulation of its PP, starting at a state earlier than t.

If $m'(i,j,t) = m(i,j,t)$, for all $t \leq T$, and there are no messages, in transit, with time-stamp t where $t \leq T$, then m is at a fixed point for the region consisting of all times $t \leq T$.

In Time-Warp, space-time is partitioned into vertical strips, one for each PP, and each vertical strip is estimated from the bottom up. Convergence also proceeds from the bottom upwards.

Running Example We use $s1, s2 \ldots$ to mean the SP that simulates the corresponding PP. Based on the initial conditions, $s1$ transmits $+m(s1, f1, 2)$ to fork $f1$; similarly for $s2$. On receiving their inputs, assume both $f1$ and $f2$ route incoming jobs to $s1$; that is, they generate $m(f1, s1, 2) = 1$ and $m(f2, s1, 2) = 1$ respectively. Note that as outputs to $s2$ are *null*, they are not sent. Also assume that the output from $f1$ arrives at $s1$ before that of $f2$. On receiving this estimate, $s1$ computes its output $m(s1, f1, 4)$ assuming $m(f2, s1, [0, 2]) = null$. Assume that the next few jobs received by $f1$ are routed back to $s1$. Thus, $s1$ will receive $m(f1, s1, 4)$, generate $m(s1, f1, 6)$, receive $m(f1, s1, 6)$, generate $m(s1, f1, 8)$ and so on. When $s1$ eventually receives $+m(f2, s1, 2)$ from $f2$, it must repeat its simulation starting from time 0, and cancel the generated outputs. In the specific case of this example, lazy cancellation will be more efficient.

4.5 Conservative Simulation

Conservative simulation is an extension of time-driven simulation. The space-time region in conservative simulation is not filled in horizontal strips as in time-driven simulation because some PPs can be simulated further in time than others.

Each PP is simulated by one SP [Mis86]. For each PP i, correct values of $m(i,j,t)$ are computed for $t \leq T_{i,j}$ for some nonnegative $T_{i,j}$. Given correct values for $m(j,i,t)$ for all $t \leq T_{j,i}$, correct values for $m(i,*,t)$ are computed at least up to $t = min_j\ T_{j,i}$ for a non-lookahead PP i, and at least up to $t = \epsilon + min_j\ T_{j,i}$ for a lookahead PP i.

Here too, space-time is partitioned into vertical strips, one for each PP, such that SP $x(i, [0, H])$ simulates PP i. Each strip is filled from the bottom up, but only with *correct* values. Because of the interaction between PPs, the region can be filled in more slowly than for Time-Warp. An advantage is that each shared variable $m(i,j,t)$ is modified *at most once*, from its initial value (which is usually *null*) to its final correct value; and this makes for straightforward implementation of these shared variables as messages.

Running Example We use the conservative algorithm to simulate the same instance of the example as was used in the preceding section. Based on its correct initial conditions, $s1$ and $s2$ compute $M(s1, f1, 2)$ and $M(s2, f2, 2)$ in parallel. This enables $f1$ and $f2$ to compute their correct outputs ($M(f1, s1, 2) = 1$, $M(f1, s2, 2) = 0$) and ($M(f2, s1, 2) = 1$, $M(f2, s2, 2) = 0$) respectively. Whereas an estimate for $M(s1, f1, 4)$ was computed in Time-Warp using only input from $f1$, in this case, $s1$ waits to receive both $M(f1, s1, [0, 2])$ and $M(f1, s1, [0, 2])$ before computing $M(s1, f1, 4) = 1$ and $M(s1, f1, 6) = 1$. Meanwhile $s2$ waits to receive inputs from both forks and computes $M(s2, f2, 4) = 0$. Unlike optimistic algorithms, $s2$ must send its *null* message to $f2$ to allow $f2$ to generate its own (*null*) output at time 4.

4.6 Different Partitions of Space-Time

Traditional methods partition space-time into vertical strips, one strip per PP. We have shown that space-time can be allocated to processes in arbitrary and dynamic ways (subject to the fairness

condition). For example, suppose we are simulating traffic patterns in a city in which most of the activity at midday is downtown and most of the activity in the early mornings and late evenings is in the suburbs. Rather than allocate one simulation process to each city block over the entire day, we may choose to assign a few tightly-coupled highly-active downtown blocks during the midday period to a single SP, and assign large sections of downtown in the late evening to another SP.

Another decomposition is to partition space-time into horizontal strips, where each strip is assigned to an SP and all SPs are executed in parallel. This decomposition, called time-parallel simulation is particularly effective in simulating regenerative systems, where some state occurs frequently. For such systems, if the input estimates to an SP (which includes a regenerative state) are found to be incorrect, the outputs may be corrected without resorting to the message cancellations and recomputations associated with existing optimistic techniques. Techniques to exploit this type of decomposition for simulation of queuing networks have been described in [GML90, LL91]. A performance study on the use of time-parallel decomposition in the simulation of a simple feed-forward network has been described in [BCL91].

Space-time can integrate conservative and optimistic methods. Partition space-time arbitrarily into regions. Give higher priority to regions lower in the space-time diagram, i.e., regions with earlier time. Simulate a region only if higher priority regions have been simulated, and their inputs have remained unchanged since they were last simulated. Thus low-priority regions are simulated only if there is no work to be done on higher-priority regions. In the case of conservative simulation, low-priority regions are not simulated until higher-priority regions have reached fixed points. In the case of optimistic simulation, arbitrary scheduling of regions can be used. By suitably adjusting the scheduling decision of when to simulate low-priority regions, we can make a space-time simulation behave either as a conservative simulation or as an optimistic one.

5 A New Algorithm

In this section, we describe a new algorithm derived from space-time: partition space-time arbitrarily into a finite number of regions. Assign an SP to each region. Set m to some arbitrary value initially. (The better the initial estimate of m, the faster will the algorithm converge; so it is not necessarily efficient to set $m(i,j,t)$ to *null* for all i, j and t. Indeed, *null* plays no role in space-time, though we may choose to introduce it in the interests of efficiency. We should set the initial values of m to the best estimates we have from theory or from previous simulations of similar physical systems.) First, we present a synchronous version of the algorithm, and then we present an asynchronous version.

Each iteration of the synchronous algorithm is as follows: Each SP computes new estimates $m(i,*,t)$ for all points (i,t) in its region and waits until all SP have finished their computation. If, for every SP, the new estimates for every shared variable is equal to its local copy, computation terminates. Otherwise, each SP updates its local copy for each shared variable. (note that for the sake of efficiency, an SP is executed only if the value of some input shared variable is changed.) The fairness condition is trivially satisfied and it is easy to deduce that at termination, m is correct for all $t \leq T$.

The asynchronous algorithm is as follows. Initially, each SP computes $m(i,*,t)$ and $s(i,t)$ for all points (i,t) in its region; these values are recomputed each time the input estimates to the region change. If all SPs are idle, and no input estimate to an SP has changed since the SP last executed, the algorithm is at a fixed point.

The primary difference between the two algorithms is in detecting convergence: for the synchronous algorithm, each SP simply tracks $t.max$, where $t.max$=largest t such that for all i, j in its region, its estimate in an iteration is the same as in the previous iteration. The computation terminates when for all SP, $t.max$ is equal to the upper bound on its time-region. For the asynchronous algorithm, the implementation must additionally guarantee that all SP are idle. The simplicity of detecting convergence for the synchronous algorithm must be counterbalanced by the overhead of synchronizing the SP which may potentially execute on a large number of processors.

6 Implementation

Many experimental studies have been devised to evaluate the performance of various simulation algorithms on both shared memory and message-passing architectures[RMM87, Fuj88a, SS89, Fuj88b, JBWea87, CS89a]. In this section, we discuss different implementations of the new algorithm and present experimental results on the efficiency of the implementations.

Any implementation of the new algorithm must perform the following primary tasks:

- subdivision of the space-time region among multiple SP.
- updating the state and output variables of an SP when its inputs are modified,
- implementing the fairness rule for the given decomposition, and
- detecting fixed-point using synchronous or asynchronous algorithms.

A high-level simulation language called Maisie[BL90b] has been implemented on a Symult S2010 multicomputer using the new algorithm. The primary design goal of Maisie was to maintain a clean separation between the simulation language and the specific simulation algorithm, sequential or parallel, that is used to execute Maisie models. This allows an analyst to design a model and subsequently execute it on either sequential or parallel architectures with minimal modifications. This section describes distributed execution of Maisie programs using a synchronous implementation of the new algorithm. It also presents the results of a case study to measure the speedup obtained from this implementation.

6.1 Maisie

Maisie enhances C with a few primitives to model physical processes and their interactions. Its central construct is an entity-type to describe objects of a given type. An entity-instance, henceforth referred to simply as an entity, represents a specific object and may be created and destroyed dynamically. Entities communicate with each other using buffered message-passing. Every entity is associated with a unique message-buffer. An entity deposits a message in the message-buffer of a destination entity using a non-blocking send command. The message is deposited in the destination buffer at the same simulation time as it is sent. If required, an appropriate delay statement may be executed to model message transmission times or a separate entity may be defined to simulate the transmission medium. An entity accepts messages from its message-buffer by executing a blocking receive command. The receive command allows the entity to either accept successive messages from its buffer (or be blocked if the buffer is empty) or selectively accept only specific messages. Additional constructs are provided to allow an entity to delay itself in simulation time to simulate

actions of objects in the physical system. The interested reader is referred to [BL90a, BL90b] for a description of the language features.

A physical system may be easily modeled as a Maisie program. Each PP in the system is modeled by an entity. For instance, entity-types *queue* and *fork* may be used to model the queue and fork PPs in the example network. Two instances of the *queue* entity-type are created to represent the specific PPs; similarly for forks. Inputs and outputs of a PP are naturally expressed using messages. For instance, updating output $M(s1, f1, t)$ is modeled by a message being sent by the entity (that corresponds to PP) $s1$ to entity $f1$ with timestamp t. An entity may execute a non-selective receive to accept successive inputs, or it may execute a selective receive to accept inputs from specific entities.

6.2 Maisie: Parallel Implementation

The primary challenge in the parallel Maisie implementation was to keep details of the simulation algorithm transparent from the programmer. Beginning with the decomposition of the space-time region, we consider each task outlined earlier and briefly describe its implementation for a Maisie program.

Although each entity in a Maisie program models a PP, from the perspective of the simulation algorithm, each entity *is* a PP. In this section, we use the term entity to be synonymous with PP and use i, j, k to refer to an entity. Consider an entity that executes over the simulation interval [0,H]. The states and the outputs of entity i in the interval $[u, v]$ are computed by SP $x(i, [u, v])$. We use $x(i, *)$ to refer to the set of SP that simulate the entity i over all t in [0,H]; at most one SP from the set executes at any given time in the computation. (the space-time regions used to execute a Maisie program are non-overlapping.) If multiple entities are created on a processor, the corresponding SPs are executed using the sequential simulation algorithm of the previous section. SPs mapped to different processors are executed using the synchronous implementation of the new distributed algorithm. For simplicity, we henceforth assume that each entity is mapped to a unique processor.

The length of the interval [u,v] assigned to an SP is an important factor in determining overall efficiency of the implementation. The optimal duration is typically determined by a number of factors. As the SPs mapped to different processors are executed in parallel, an SP $x(i, [u, v])$ that was executed based on an estimate of input $M(*, i, [u, v])$, must be recomputed if the estimate is subsequently discovered to be inaccurate at any point in interval $[u, v]$. Let $[u, w]$, $u \leq w \leq v$ represent the largest sub-interval in which the estimated inputs were *correct* in the previous execution of $x(i, [u, v])$. It follows that the SP needs to be recomputed only in the interval $[w,v]$. Thus the probability of recomputation decreases as interval $[u, v]$ is shortened, and is least when the lifetime of each SP is equal to the minimum lookahead in the application. However, as this interval is reduced, synchronization costs increase, and in the extreme case would effectively imply that the simulation is executed using a time-driven algorithm. A compromise solution is to define the lifetime of an SP based on the desired frequency of synchronizations and to reduce recomputation by checkpointing intermediate states of the SP. For the transparent Maisie implementation, the lifetime of an SP is determined by the average round-trip communication time between two nodes in the parallel architecture, and an SP is checkpointed after executing every event. The next section describes alternative strategies.

The SPs are executed synchronously, where a typical iteration is as follows: each $x(i, [u, v])$ is executed based on an estimate $m(*, i, [u, v])$ of its inputs; the outputs (messages) generated by the

SP are written to a local copy. After completing its execution, the outputs are transmitted in a message to the appropriate destination SPs and the node executes SP $x(i,[v,w])$ again based on estimated inputs. The lifetimes of the SPs are chosen such that the remote inputs for $x(i,[u,v])$ arrive approximately when the execution of $x(i,[v,w])$ completes (if not, the processor waits until inputs from all other processors have been received). If the inputs for $x(i,[u,v])$ are different from the estimates used in its execution, $x(i,[u,v])$ is recomputed from an appropriately checkpointed state.

The part of the space-time region that has reached its fixed-point is detected using the following synchronous algorithm: Every SP saves a local copy of the outputs that are generated during its computation. If the SP is subsequently (partially) recomputed, it compares the outputs generated in its execution with its previous copy. The SP computes its *lvt*, the maximum time to which the outputs generated in the subsequent iteration are identical. An SP piggybacks the *lvt* with its output message. (even if an SP does not need to send output messages to another SP, it must send its *lvt*). Except for the initial iteration, subsequent iterations at a processor are initiated only after the *lvt* has been received from all other processors. The convergence time for the computation is simply the minimum over the *lvt* of all processors and can be computed individually by each node.

Let δ refer to the lifetime of an SP. The fairness rule is obeyed by this scheduler, as execution of $SP(i,n.\delta)$, $n \geq 2$, is executed only after $SP(i,(n-2).\delta)$ has reached its fixed-point.

The next section describes a Maisie model for a simple predator-victim simulation together with the relevant speedup results.

6.3 Sharks World Problem

Physical System The physical system is a 2-dimensional toroidal ocean that contains two types of creatures: sharks and fish. Each creature moves with a constant velocity. The lifetime of a fish is simple: it continually swims in the ocean until it gets eaten by some shark. A shark's life is slightly more interesting: Whenever it is within a specific radius a of some fish, it eats the fish. Initially, the sharks and fish are assumed to be uniformly distributed in the ocean.

Maisie Model For the Maisie model, the ocean is divided into a number of rectangular sectors, each of which is modeled by an entity. (As discussed subsequently, the division of the ocean into sectors is desirable, even if the simulation is to be executed sequentially.) The movement of creatures from one sector to another is modeled by message communication among the corresponding entities. Arrival and exit of sharks to a sector is handled in a straightforward manner. However, due to the non-zero attack range of a shark, a fish in sector i may be visible to a shark in a neighboring sector j before it actually enters j. For this reason, the boundary of each sector is extended by a in all directions and the region within the extended boundary is referred to as the sphere of influence or *soi* of a sector. Thus the major tasks processed by each sector are: death of a fish in its *soi*, arrival and exit of a shark, and arrival and exit of a fish to and from its *soi*. Note that as a fish may be simultaneously within the *soi* of multiple (upto 3) sectors, when a shark kills a fish that is not in its own sector, it must inform all sectors within which the fish may be visible. The following message-types are defined for each sector entity:

- *c_enter*{*c_type*,*c_rec*}: *c_type* may be either s or f; *c_rec* is the data structure containing relevant information about the creature. Receipt of this message may cause the also cause the entity to recompute its earliest event.

- *c_exit*{*c_type*,*c_rec*}: A sector sends this message to itself when a shark leaves the sector, or a fish its *soi*. This message simply causes the sector entity to update its creature-list.
- *kill*{f,S}: If sector S kills a fish f, this message is sent to all sectors that have fish f within their *soi*. Receipt of this message may also cause the entity to recompute its earliest event.

Every sector entity is initialized with an equal number of creatures. The entity computes the kill and exit events for all its creatures, stores them in an ordered list and schedules its earliest event. Receipt of any of the preceding messages causes it to update its state and may cause it to reschedule its earliest event as indicated. The model tracks the total number of fish that are killed and records the time-of-death and shark id for each kill.

Results The experiments were conducted for a square, toroidal ocean which is 65K units on each side. The attack range of a shark was fixed at 50 units. The creature speeds were chosen uniformly at random from the interval [50,200] and the direction chosen uniformly at random from 0 to 360-ϵ, for a sufficiently small, positive ϵ. Successive experiments were executed by gradually increasing the total number of creatures from 64 to 1024 while maintaining an equal number of sharks and fish. Each configuration was executed on both sequential and parallel architectures. A detailed description of the model and its performance may be found in [BL90c]. A capsule summary of the results is presented in this paper to illustrate the effectiveness of the transparent parallel implementation in reducing completion times of simulations.

The sequential implementations were executed on a single node of the Symult S2010 and the parallel configurations used a unique processor for every sector entity. For a given system configuration, the number of sectors used in the experiments affect the overall performance of the simulation for both sequential and parallel implementations. As such for each configuration, the number of sectors were varied to obtain the best sequential and parallel times. For different creature counts, figure 8 shows the completion time of the sequential implementation as a function of the number of sectors. As expected, in general, the execution time of the simulation initially decreases and then begins to increase as the number of sectors is increased. A similar behavior is reported for the parallel implementations in figure 9. Figure 10 compares the optimum sequential simulation time with the optimum parallel execution time for each creature configuration studied in these experiments. The number in brackets indicates the number of sectors (which is also equal to the number of nodes for the parallel implementation) that yielded the optimum time for the corresponding configuration. As seen from the figure, the slope of the curve for the parallel implementation is significantly flatter than that of the sequential implementation. This data is replotted in figure 11 as a speedup curve, where speedup is the ration of the best parallel time (t_{par}) to the best sequential time (t_{seq}). Note that the Maisie program for the sequential and parallel implementations were identical except for the allocation of sector entities among the parallel processors. This allowed consistent measurements for both sequential and parallel implementations.

6.4 Alternative Implementations

This section examines the performance of the new algorithm using an asynchronous implementation.

Physical System The experiments used a closed queuing network (henceforth referred to as CQNF) similar to the network considered in the running example: the network consists of N nodes, where each node is a merge process that feeds a tandem queue of Q $first-in-first-out$ servers.

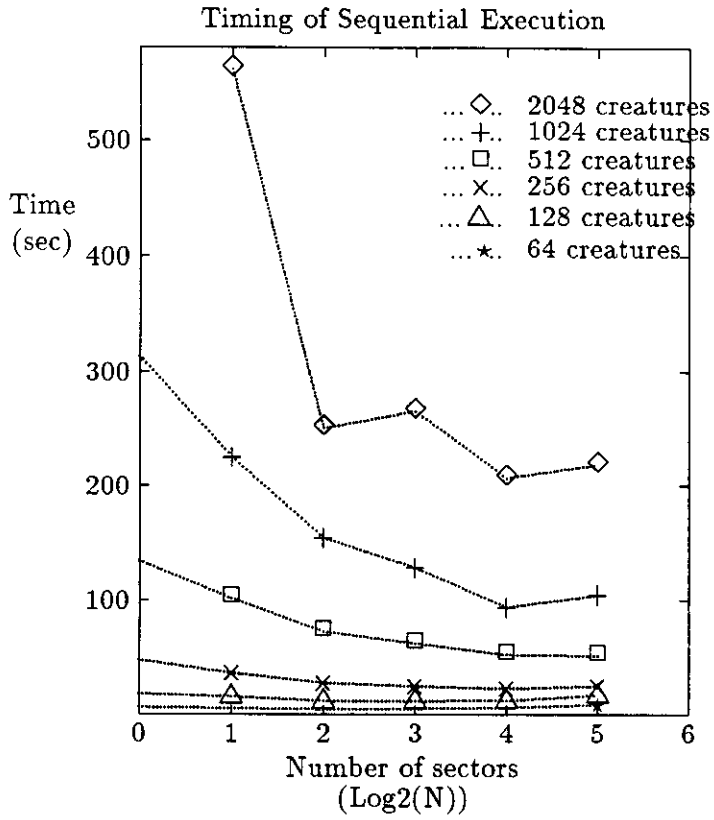

Figure 8: Shark's World: Sequential Implementation

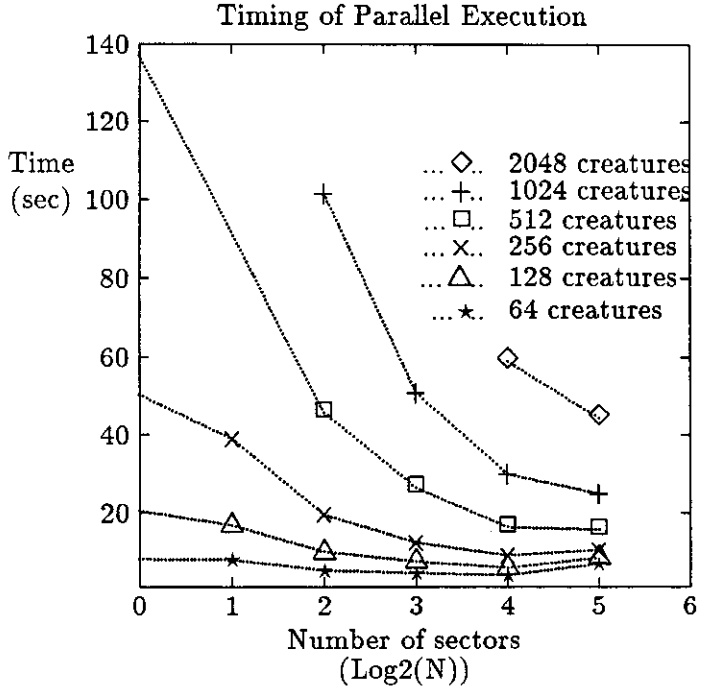

Figure 9: Shark's World: Parallel Implementation

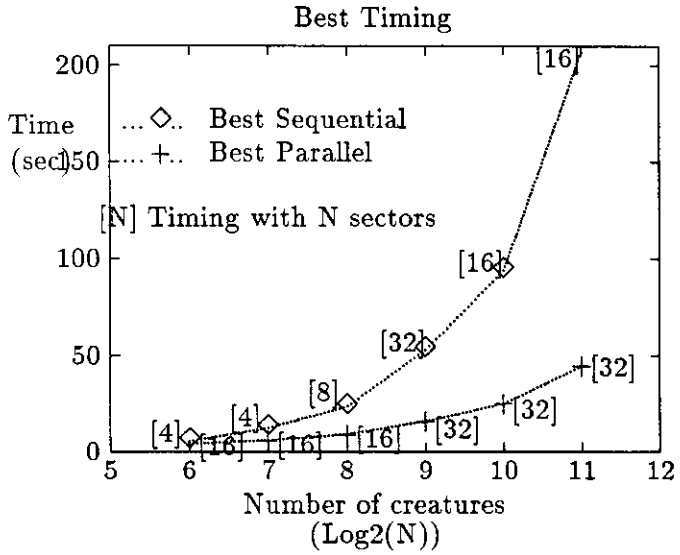

Figure 10: Shark's World: Best Configurations

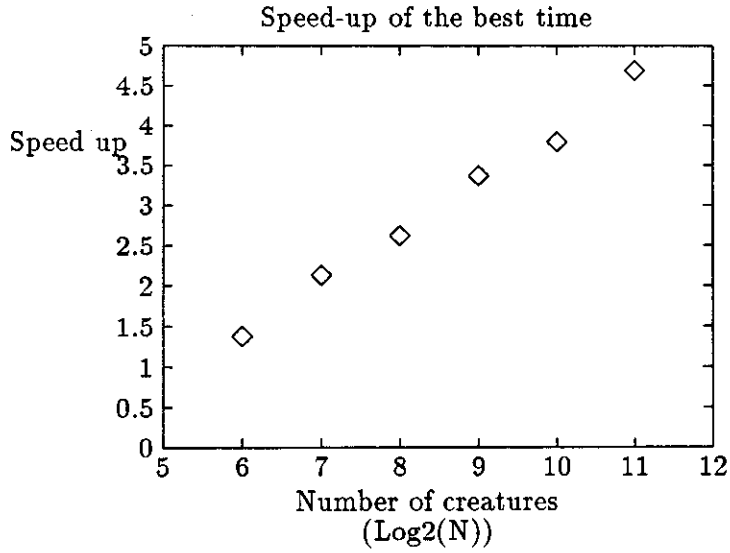

Figure 11: Shark's World: Speedup

An incoming job at the queue is served sequentially by the Q servers and is subsequently routed to one of the merge processes selected at random. Each merge process is initially assigned J jobs.

Implementation The CQNF network was programmed using the entity and message-passing facilities provided by Maisie; the simulation facilities of Maisie *were not used.* Thus, Maisie was used simply as a parallel programming language, and all simulation activities (checkpointing, recomputation, convergence detection ...) were explicitly coded into the application itself.

Each node of the network is modeled by two entities: a *merge* entity that models the merge process and a *queue* entity that models the queue of servers. The *merge* and *queue* entities that correspond to a single node of the queuing network are created on a unique processor. A *merge* entity continuously executes the following: for a suitably chosen k, merge the next k incoming jobs (or less, if the message-buffer of the entity does not contain k jobs), checkpoint the final state and forward the jobs to the *queue* entity. The *queue* entity successively simulates service of the k jobs at each of its servers, and buffers the outputs locally. After k jobs have been simulated, its state is checkpointed, and buffered messages transmitted to the appropriate destination entity. Incoming messages that have arrived during the previous iteration, are accepted by the *merge* entity and the process repeats. If an incoming message invalidates the estimated input to the *merge* entity for a preceding interval, the *merge* and *queue* entities are recomputed over the appropriate interval.

The preceding implementation decomposes the space-time region into a number of SPs, such that each SP simulates service of k jobs at a node. Let SP $x(i,n)$ simulate the $n-th$ lot, $n \geq 0$, of k jobs at queue (and merge process) i. The input to the SP is the arrival time of the jobs at the corresponding merge process. The output of each SP consists of the departure times of the jobs serviced in that lot and its lvt (as defined in the previous section). All SPs that simulate service of the jobs at a given node are executed on the same processor. At any given time, the next SP to be executed is the SP with the smallest lot number that is not at its fixed-point. The SP is executed using its most recent inputs and the outputs are sent to appropriate destinations. Once again, we require that a node send its lvt to all nodes in the network. Note that unlike the synchronous implementation considered in the preceding section, a merge entity does not wait to receive incoming messages from all neighbors; it accepts whatever messages are available in its buffer assuming that its estimate matches the correct input for others.

The convergence time is computed periodically using the following asynchronous technique:

1. determine $maxk$=max k, such that a message with lot number k has been received from all processors.
2. $lvt_i = lvt$ from the most recent message with lot number $maxk$ received from processor i.
3. convergence time = min lvt_i

Results Different configurations of the model were executed on sequential and parallel architectures to measure speedup as a function of number of processors. For a given configuration, speedup is defined as S = T_{seq}/T_{par}, where T_{seq} is the completion time for the simulation when executed using a sequential simulation algorithm and T_{par} is the completion time using a parallel implementation. Except where noted, each sequential run was executed on a single node of the Symult S2010 multicomputer whereas the parallel implementations used multiple nodes that were identical to the one used for the sequential executions. A detailed report on the performance of this network

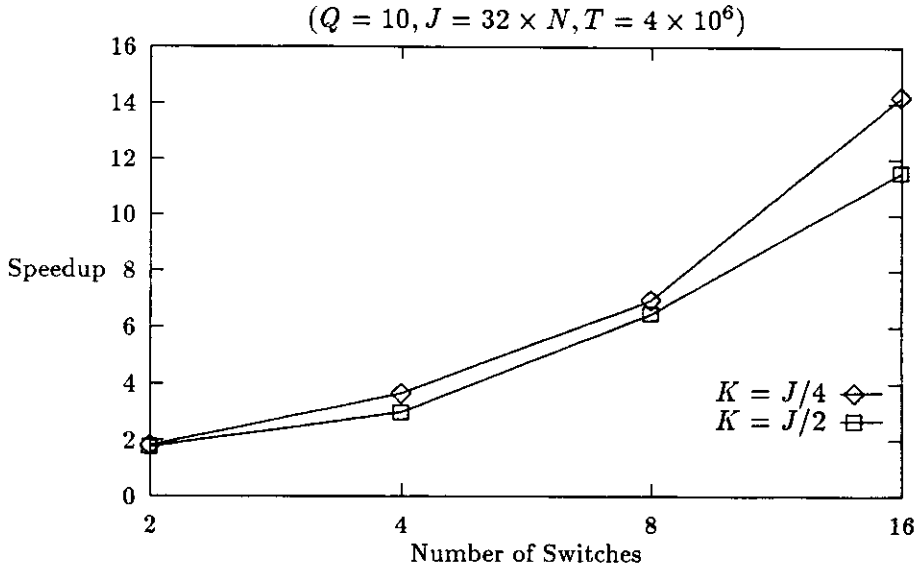

Figure 12: CQNF Speedup: number of switches

as well as other queueing networks, using the asynchronous algorithm, may be found in [BCL91]. This section presents a summary of the results for the CQNF experiments.

The first graph (figure 12) measures the speedup obtained in the parallel implementations as a function of N (number of nodes). The remaining parameters were fixed at Q (number of servers)=10 and J(number of jobs initially assigned to each node)=32. As seen from the figure, the speedup increases almost linearly with N for $N \leq 16$. Larger values of N are not included in this graph, as the available memory of each multicomputer node restricted the largest sequential implementations that could be executed. Modified configurations that were executed for larger values of N are discussed subsequently.

For a CQNF network of N nodes, the amount of computation is determined by the number of jobs and number of servers in each queue. Figure 13 presents the speedups obtained for a 16 node CQNF as a function of the number of jobs in the system, for Q=5 and 10 respectively. Note that initially the speedup increases with J and then levels off to reach a peak speedup of about 14 for J=32. This behavior is expected as the initial increase in J offsets the message communication time in the network and once every node is fully utilized, further increases in J have no affect on the speedup. Note also that whereas for smaller values of J the speedup is greater for larger Q, the value of Q is less relevant as J is increased. Once again this behavior is expected: for smaller J, the larger Q will cause each processor to be better utilized. As J increases, a processor is fully utilized even for smaller values of Q and so shows less dependence on J. The experiments that are reported in this graph assumed a lot size (k) of J/4. The behavior of the network as a function of k is discussed subsequently.

To permit networks with larger N to be executed, memory requirements of the model were reduced by changing the topology: each CQNF node was assumed to be connected to N/2 other nodes in the system. Furthermore, the number of jobs were reduced from J=32 (which yielded the optimum speedups for the experiments in figure 13) to only 4. This allowed us to execute sequential

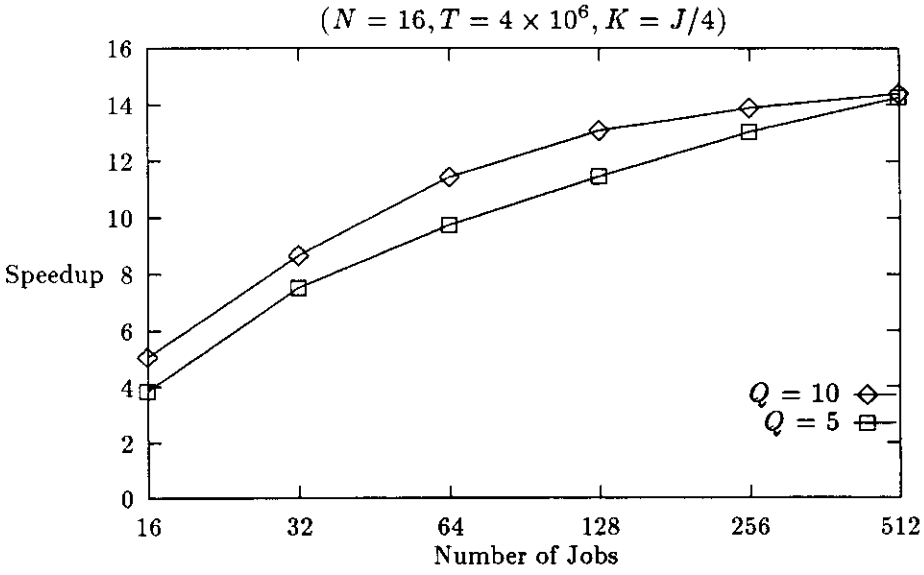

Figure 13: CQNF Speedup: jobs per switch

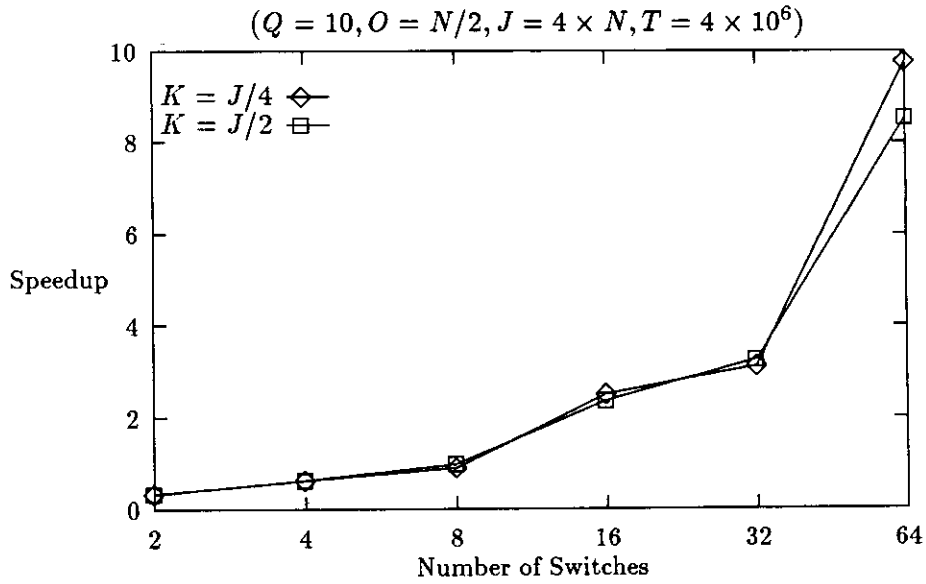

Figure 14: CQNF: Speedup over Sun Sparc-IPC

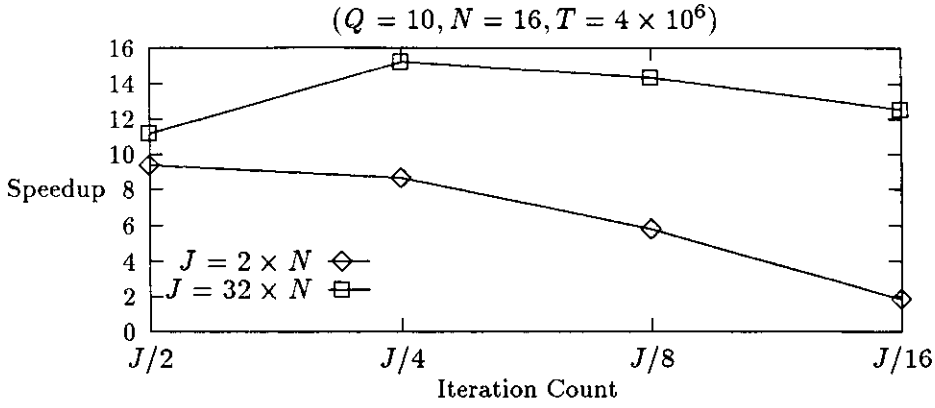

Figure 15: CQNF Speedup: iteration count

simulations of upto 64 nodes, and although the speedup is sub-optimal, it was still close to 35 for N=64. Figure 14 gives the speedup as a function of N, *where the sequential implementations were executed on a Sparcstation IPC*, a configuration that is considerably more powerful than the single node of a Symult. As seen from the figure, significant speedups were obtained (upto a factor of 10) even when compared against a superior sequential implementation.

The final set of experiments examined the variations in speedup as a function of the lot size k. As this parameter refers to the lifetime of an SP, it has a significant impact on the frequency of checkpointing, convergence detection and other overheads of the simulation algorithm. Figures 15 present the speedup as a function of k for two configurations that differ only in the number of jobs in the network. As seen from the figure, for J=32, the speedup increases initially and then decreases. The intuitive explanation for this behavior is that as k decreases, it initially causes the SPs on different processors to communicate more frequently, causing the computation to converge more rapidly. However, as k is decreased further, the simulation overheads (including the state saving overheads) increase causing a net decrease in speedup. This explanation was supported by separate measurements to determine the total number of times each SP is executed, checkpointing costs and other costs like message communication, convergence detection, and garbage collection. For J=2, the speedup decreases monotonically, dropping significantly for lower values of k. Measurements of the overhead costs showed that this was accompanied by a sharp increase in the overhead costs and also in the total number of times each SP is executed. In examining the contributants to the overhead, it was found that although all costs increase, the major increase is due to higher message delivery costs. Note that for this configuration, for the lowest value of k, each queue processed exactly 2 jobs before communicating with its neighbors. This caused a significant increase in the total number of iterations needed for the simulation, and increased the overhead costs. As the amount of useful work done per iteration is relatively small with respect to the overtheads, this configuration yielded a minimal speedup of 2.

7 Conclusion

This paper developed a unifying theory for distributed simulation which encompasses both continuous and discrete-event simulation. A number of existing algorithms, using both conservative and optimistic synchronization strategies, were derived from the theory. A new distributed simulation algorithm was also derived which may be implemented using synchronous or asynchronous parallel processes.

Performance results for the new algorithm were presented for both stochastic and deterministic applications. The deterministic application was a simulation of a sharks world problem[CCU90]. The simulation model was programmed in Maisie, a high-level simulation language that has been implemented on multicomputers using the synchronous implementation of the new algorithm. The stochastic application was the simulation of a closed queuing network which was programmed using the asynchronous implementation. The parallel implementation for both applications yielded significant speedups.

References

[BCL91] R. Bagrodia, K.M. Chandy, and W-L. Liao. An experimental study on the performance of the space-time algorithm. Technical report, Computer Science Dept, UCLA, Los Angeles, CA 90024, August 1991.

[BL90a] R. Bagrodia and Wen-toh Liao. *Maisie User Manual.* Computer Science Department, University of California at Los Angeles, 1990.

[BL90b] R.L. Bagrodia and Wen-toh Liao. Maisie: A language and optimizing environment for distributed simulation. In *1990 Simulation Multiconference: Distributed Simulation*, San Diego, California, January 1990.

[BL90c] R.L. Bagrodia and Wen-toh Liao. Parallel simulation of the sharks world problem. In *1990 Winter Simulation Conference*, New Orleans, December 1990.

[CCU90] D. Conklin, J. Cleary, and B. Unger. The sharks world: A study in distributed simulation. In *1990 Simulation Multiconference: Distributed Simulation*, San Diego, California, January 1990.

[CM88a] K.M. Chandy and J. Misra. Conditional knowledge as a basis for distributed simulations. Technical report, Dept. of Computer Sciences, California Institute of Technology, Los Angeles., January 1988.

[CM88b] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.

[CS89a] K.M. Chandy and R. Sherman. The conditional event approach to distributed simulation. In *Distributed Simulation Conference*, Miami, 1989.

[CS89b] K.M. Chandy and R. Sherman. Space-time and simulation. In *Distributed Simulation Conference*, Miami, 1989.

[Fuj88a] R. Fujimoto. Lookahead in parallel discrete event simulation. In *International Conference on Parallel Processing*, August 1988.

[Fuj88b] R. Fujimoto. Time warp on a shared memory multiprocessor. Technical report no. uucs-88-021a, Computer Science Dept., University of Utah, 1988.

[Gaf88] A. Gafni. Rollback mechanisms for optimistic distributed simulation systems. In *Distributed Simulation Conference*, 1988.

[GML90] I. Greenberg, A.G. Mitrani, and B. Lubachevsky. Unbounded parallel simulations via recurrence relations. In *Proceedings of the 1987 ACM SIGMETRICS Conference*, pages 1–12, May 1990.

[JBWea87] D. Jefferson, B. Beckman, and F. Wieland et al. Distributed simulation and the time warp operating system. In *Symposium on Operating Systems Principles*, Austin, Texas, October 1987.

[Jef85] D. Jefferson. Virtual time. *ACM TOPLAS*, 7(3):404–425, July 1985.

[LL91] Y.B. Lin and E.D. Lazowska. A time-division algorithm for parallel siulation. *ACM TOMACS*, 1(1), January 1991.

[Mis86] J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1), March 1986.

[Rey82] P. Reynolds. A shared resource algorithm for distributed systems. In *9th International Symposium on Computer Architecture*, pages 259–266, Austin, Texas, 1982.

[RMM87] D.A. Reed, A.D. Malony, and B.D. McCredie. Parallel discrete event simulation: A shared memory approach. In *Proceedings of the 1987 ACM SIGMETRICS Conference*, pages 36–39, May 1987.

[SS89] Wen-king Su and C.L. Seitz. Variants of the chandy-misra-bryant distributed simulation algorithm. In *1989 Simulation Multiconference: Distributed Simulation*, Miami, Florida, March 1989.