IN-61
116444
p- 29

NASA Contractor Report 189680
ICASE Report No. 92-30

ICASE

A SWEEP ALGORITHM FOR MASSIVELY PARALLEL SIMULATION OF CIRCUIT-SWITCHED NETWORKS

Bruno Gaujal
Albert G. Greenberg
David M. Nicol

Contract Nos. NAS1-18605 and NAS1-19480
July 1992

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23665-5225

(NASA-CR-189680) A SWEEP ALGORITHM FOR MASSIVELY PARALLEL SIMULATION OF CIRCUIT-SWITCHED NETWORKS Final Report (ICASE) 29 p

N92-32157

Unclas

G3/61 0116444

A SWEEP ALGORITHM FOR MASSIVELY PARALLEL SIMULATION OF CIRCUIT-SWITCHED NETWORKS

Bruno Gaujal
Rutgers University, Dimacs Center
and AT&T Bell Laboratories
Albert G. Greenberg
AT&T Bell Laboratories
and
David M. Nicol[1]
College of William and Mary

ABSTRACT

A new massively parallel algorithm is presented for simulating large asymmetric circuit-switched networks, controlled by a randomized-routing policy that includes trunk-reservation. A single instruction multiple data (SIMD) implementation is described and corresponding experiments on a 16384 processor MasPar parallel computer are reported. A multiple instruction multiple data (MIMD) implementation is also described and corresponding experiments on an Intel IPSC/860 parallel computer, using 16 processors, are reported. By exploiting parallelism, our algorithm increases the possible execution rate of such complex simulations by as much as an order of magnitude.

[1]Research was supported by the National Aeronautics and Space Administration under NASA Contract Nos. NAS1-18605 and NAS1-19480 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665. Research was also supported in part by NASA grants NAG-1-1060, NAG-1-995, NSF Grants ASC 8819373 and CCR-9201195, and was initiated during a visit to AT&T Bell Laboratories.

1 Introduction

Discrete event simulation is an indispensable tool for the design and analysis of large telecommunication systems [12]. Unfortunately, such simulations present a very large computational burden; the execution duration of a typical simulation run is often measured in hours. In this paper we consider the problem of call by call simulation of large circuit-switched networks controlled by a simple state dependent, randomized-routing policy. We present a new massively parallel simulation method for such networks, and discuss the algorithm's implementation and performance on SIMD (single instruction multiple data) and MIMD (multiple instruction multiple data) parallel machines. Our algorithm executes an order of magnitude faster on these machines than can be expected from an optimized serial simulation, on a workstation with a tremendously large memory.

Without loss of generality, we consider completely connected circuit-switched networks having N nodes and $N(N-1)/2$ bi-directional links. A call between a node-pair is either accepted and routed along a path connecting the node-pair, or blocked (i.e., rejected and lost). A link's capacity is counted in *trunks*, equal to the number of calls that the link can simultaneously carry. If accepted, the call simultaneously seizes a single *trunk* from each link of its route at the time that the call arrives, and simultaneously releases these trunks at the time that the call finishes. Typical parameters for a large network, such as the AT&T circuit-switched network, are $N \approx 100$, with almost all of the ≈ 5000 links having non-zero capacity, and a total of ≈ 1 million trunks. (We will use the AT&T network as a guide for constructing realistic simulation scenarios. However, the routing policy we consider is different, being far simpler, than the policy used in the AT&T network.)

Call routing involves *alternate-routing* and *trunk-reservation* mechanisms [7]. Alternate-routing allows for the sharing of excess capacity:

- A call between a node-pair $\{i, j\}$ is accepted on link $\{i, j\}$ if that link is not full to capacity.
- Otherwise, a third node v, termed the *via*, is selected, and the call is accepted on the two link path $\{i, v\}$, $\{v, j\}$, if both links are not reserved.
- Otherwise, the call is blocked.

Under *randomized-routing*, the choice of via v is made by independent sampling from a probability distribution over the $N-2$ possibilities, which may depend on the parameters of node-pair $\{i, j\}$, but not on the network state describing the calls in progress. Thus, whether or not the call is offered to an alternate two-link path depends on the network state, but the choice of path does not. As mentioned in Section 2, randomized-routing can be adapted to approximate more complex routing policies where the choice of via node is state dependent.

Roughly, the function of trunk-reservation is to put a link into a "reserved" state when the number of calls holding on the link nears the link's capacity. While in the reserved state the link can be used only to carry only calls between its endpoints. This simple control is remarkably effective [7] in steering the network away from scenarios where the network blocking becomes unreasonably high because calls routed on multilink paths consume capacity that might otherwise be used to carry a larger number of calls routed on single link paths.

A key difficulty in the design of a massively parallel simulation of the network is coping with asymmetries. In realistic networks, the call arrival rates may vary by three orders of magnitude over the node-pairs. Similarly, the link capacities may vary widely. On the other hand, general purpose parallel computers are typically quite regular. Identical processors with identical memory capacities are linked in a symmetric interconnection network.

We cope with this mismatch as follows. The computation is decomposed into separate (but coupled) computations for each node-pair. Each node-pair computation is simple, regular, and highly parallel. All the node-pair computations can be carried out together in a manner well-suited for SIMD architectures, which are characterized by large numbers of processors, each with moderate speed and memory capacity. As illustrated in the top half of Figure 1, we may dedicate a larger number of processors to a node-pair whose parameters indicate the likely receipt of a larger number of events. The experiments of Section 6 show that this mapping of node-pairs to processors leads to performance that scales with the aggregate capacity of the network (i.e., the total number of trunks), and so uncovers massive parallelism.

It is not hard to adapt the algorithm for MIMD architectures, characterized by a moderate number of high speed processors, each with a large memory capacity. In this case, we map each node-pair to only one processor. In large networks, involving 100 or more nodes, the computational load can be effectively balanced by assigning a group of node-pairs to each processor. This is illustrated in Figure 1 as the MIMD mapping.

Related Work

Our approach has much in common with an approach based on *synchronous relaxation*, proposed in [2, 3], and applied and implemented as a SIMD circuit-switched simulation method in [11]. In the synchronous

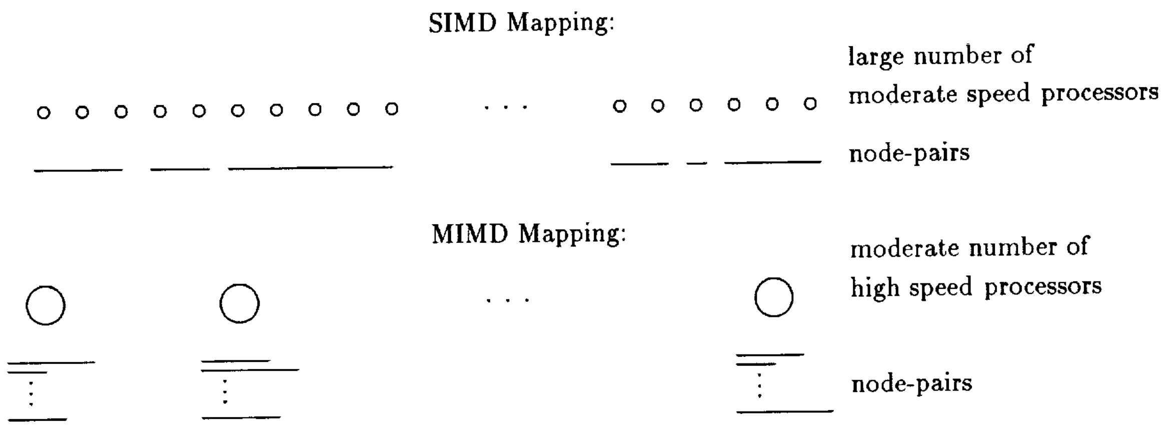
SIMD Mapping:
large number of moderate speed processors
node-pairs
MIMD Mapping:
moderate number of high speed processors
node-pairs

Figure 1: A node-pair is represented as a line segment whose length is proportional to the rate of events at the node-pair. Under the SIMD mapping a group of processors is assigned to each node-pair, using a heuristic that attempts to give each processor the same event rate. Under the MIMD mapping a group of node-pairs is assigned to each processor, again using a heuristic that attempts to give each processor the same event rate.

relaxation approach, the computation is also decomposed into separate computations for each node-pair, and these computations are mapped into the machine in much the same way as described here (Section 5). However, the node-pair computations are completely different. As described in Section 5, in the sweep algorithm, the node-pair computations require that time be partitioned into intervals so that each call arriving within an interval departs after the interval. In [11], a method related to that described in [4], is used for these computations, which does not require time to be partitioned in this way. The advantage of the sweep algorithm is its simplicity.

Typically, parallel simulation methods are classified as either "conservative" or "optimistic" [5]. Conservative methods are characterized by the property that no event e is computed before all earlier events on which e depends are computed. Optimistic methods allow dependent events to be computed out of order. This may lead to temporary errors, which are corrected later by some form of rollback or relaxation. On applications where conservative methods work well, they typically incur less overhead than optimistic methods. On the other hand, optimistic methods have the potential for exploiting a higher degree of parallelism. Our sweep algorithm is a hybrid with some of the advantages of both conservative and optimistic methods. To uncover massive parallelism the method allows dependent events to be computed out of order, like an optimistic method. However, unlike an optimistic method, no mistakes are made. Instead, we generate a superset of all possible events (some of which are mutually exclusive) within a small time window, and then use fast parallel operations to identify the correct subset of real events. At most a bounded number of messages are generated for each real event.

Plan

In Section 2, the network model is fully specified. In the following section, the particular network scenarios used to evaluate the performance of the sweep algorithm are described. In Section 4, the sweep algorithm is described at a high level, and in Section 5 the details are provided. In Section 5.3, we briefly discuss adapting the algorithm for a MIMD implementation. In Section 6, we report on performance of the SIMD and MIMD codes.

2 Network Model

A circuit-switched network consists of N nodes, where the link from any pair of nodes i and j has finite capacity $C_{i,j} \geq 0$ counted in trunks. A trunk represents the resources needed to carry a single call. A call between nodes i and j is either accepted and routed on a path in the network between i and j, or blocked (i.e., rejected and lost). An accepted call makes exclusive use of one trunk on each of the links of its route for the duration of the call. We assume here that all paths are of length one (using one link) or two (using two links). As the number of trunks used to carry a call equals its route length, allowing routes of lengths greater than two typically adds nothing to network performance, and the restriction to lengths ≤ 2 is almost always made in practice.

We assume that, for each node-pair $\{i, j\}$, call arrival times are described by a Poisson process with fixed rate $\lambda_{i,j}$. We assume that each call's holding time is an independent, identically distributed random variable:

$$C + E,$$

where C ($0 \leq C \leq 1$) is a fixed constant and E is exponentially distributed with mean $1 - C$. Thus, the average holding time is one, and the units of the arrival rate are "erlangs" [7]. Typically, in switched network simulation studies one assumes purely exponential holding times ($C = 0$). However, it is more realistic to allow for an initial constant delay C. Furthermore, we will see later that including this delay actually improves the efficiency of the parallel simulation of the system. As shown in Section 6, system performance measures such as blocking, turn out to be rather insensitive to C.

The routing scheme we consider belongs to the class of schemes that use state dependent *alternate-routing* to share idle capacity and *trunk-reservation* to ensure that the network does not become loaded inefficiently with calls routed on multilink paths. For each link $\{i, j\}$ there is a trunk reservation parameter, $r_{i,j}$, $0 \leq r_{i,j} \leq C_{i,j}$. Suppose a new call is offered between nodes i and j at time t. Let $n_{u,v}$ denote the number of calls holding on any link $\{u, v\}$ at this instant. If the number of trunks in use on link $\{i, j\}$ is less than the link capacity ($n_{i,j} < C_{i,j}$) then the call is accepted on the direct one link path from i to j. Otherwise, an intermediate node v, called the ***via***, is selected and the call is offered to the two-link path: $\{i, v\}$, $\{v, j\}$. The call is accepted on this path if neither link is reserved; that is, if $C_{i,v} - n_{i,v} > r_{i,v}$ and $C_{v,j} - n_{v,j} > r_{v,j}$. Otherwise, the call is blocked; i.e., rejected and forever lost.

Under randomized-routing, for each call blocked on its direct path $\{i, j\}$, the choice of via node is made by independent random selection of one of the $N-2$ possible nodes $v \neq i, j$, from a distribution that depends on $\{i, j\}$, but not on the network state describing the calls currently in progress. We note that randomized-routing can be adapted to approximate some routing policies where the choice of via node depends on the network state by the simple device of biasing the random selection in accordance with a recent sample of the network state. For example, under the aggregated least busy alternative (ALBA) routing policy, each via is assigned a load state in a small, bounded range, $0, \ldots, K$, where the lower values indicate roughly a greater number of free trunks on the two-link path determined by the via. To approximate ALBA, the load states can be periodically sampled, and randomized-routing adapted to choose uniformly at random from those vias in the minimal load state.

3 Simulation Scenarios

To evaluate the performance of the parallel simulation, we consider two types of scenarios:

- symmetric networks where each of the $N * (N-1)/2$ links have identical capacity, and
- a 114 node network, modeled after a realistic fiber cut scenario for the AT&T switched network.

To completely specify the randomized-routing policy, in the symmetric network, we assume each choice of via is made uniformly at random from the $N-2$ possibilities. In the asymmetric network scenarios, we assume the choices are biased in accordance with the end to end capacity of each two-link alternate path. Specifically, consider a node-pair $\{i, j\}$, and let $cap_{i,v}$ and $cap_{v,j}$ denote the capacities of links $\{i, v\}$ and $\{v, j\}$. Via v is chosen with probability proportional to $\min\{cap_{i,v}, cap_{v,j}\}$.

In Section 6, the symmetric network scenarios are used as a simple means of testing how performance of the simulation scales with the network size, measured as its total capacity.

In practice, the main role of simulation is in evaluating the performance of the network under stress: typically traffic surges and equipment failures, such as fiber cuts. In practice networks are asymmetric, and become more so after equipment failures. In the 114 node network we consider, the total capacity is about 740,000 trunks, but the link capacities vary widely from 0 to 4000 with a mean of ≈ 100. Similarly, the total arrival rate is about 530,000 erlangs, but the arrival rates vary over the node-pairs from 1 to 2500 with a mean of ≈ 80. As a result of the fiber cut several hundred node-pairs have arrival rates significantly greater than the corresponding direct link capacities. It turns out that call blocking is highly focused on about 500 of the more than 6000 links.

4 The Sweep Algorithm

In this section, we give a high level description of the sweep algorithm, leaving some of the details to Section 5.

Arrival events at a given node-pair $\{i,j\}$ are of one of two types:

- *direct-arrival* (A), marking the starting time of a call between i and j offered on link $\{i,j\}$,
- *via-arrival* (V), marking the starting time of a call between another node-pair ($\{i,k\}$ or $\{k,j\}$) offered on an alternate two-link path that includes link $\{i,j\}$.

By the nature of the routing algorithm, a *via-arrival* for a given call is not offered unless the corresponding *direct-arrival* is blocked. However, let us take the view that all possible arrival events are *tentatively* offered. That is, for every call generated between a node-pair $\{i,j\}$ all three possible events are offered: a *direct-arrival* at $\{i,j\}$ and a *via-arrival* at each of the two links of the alternate path the call would take if blocked on the direct path. This is possible because the choice of via node and the duration of the call are independent of the network state. In the parallel simulation method, a state is associated with each arrival event. The method is iterative. Each iteration sweeps through the offered events, updating associated state information, and possibly rejecting and removing some offered events. On termination, the remaining offered events are exactly those events that actually occur.

Towards this end, we partition time into consecutive intervals, termed *windows*, which are simulated serially. We construct the windows in a way that allows us to apply simple, massively parallel algorithms (described below) to simulate them. Specifically, let s denote the start of a window; initially $s = 0$. Let t denote the greatest time $> s$ such that no call arrival offered anywhere in the network after time s has a finishing time less than t. Thus, each call that arrives in the interval $[s,t)$ departs after it. A window starting at time s can be chosen as any enclosed interval $[s,u)$; $u \leq t$.

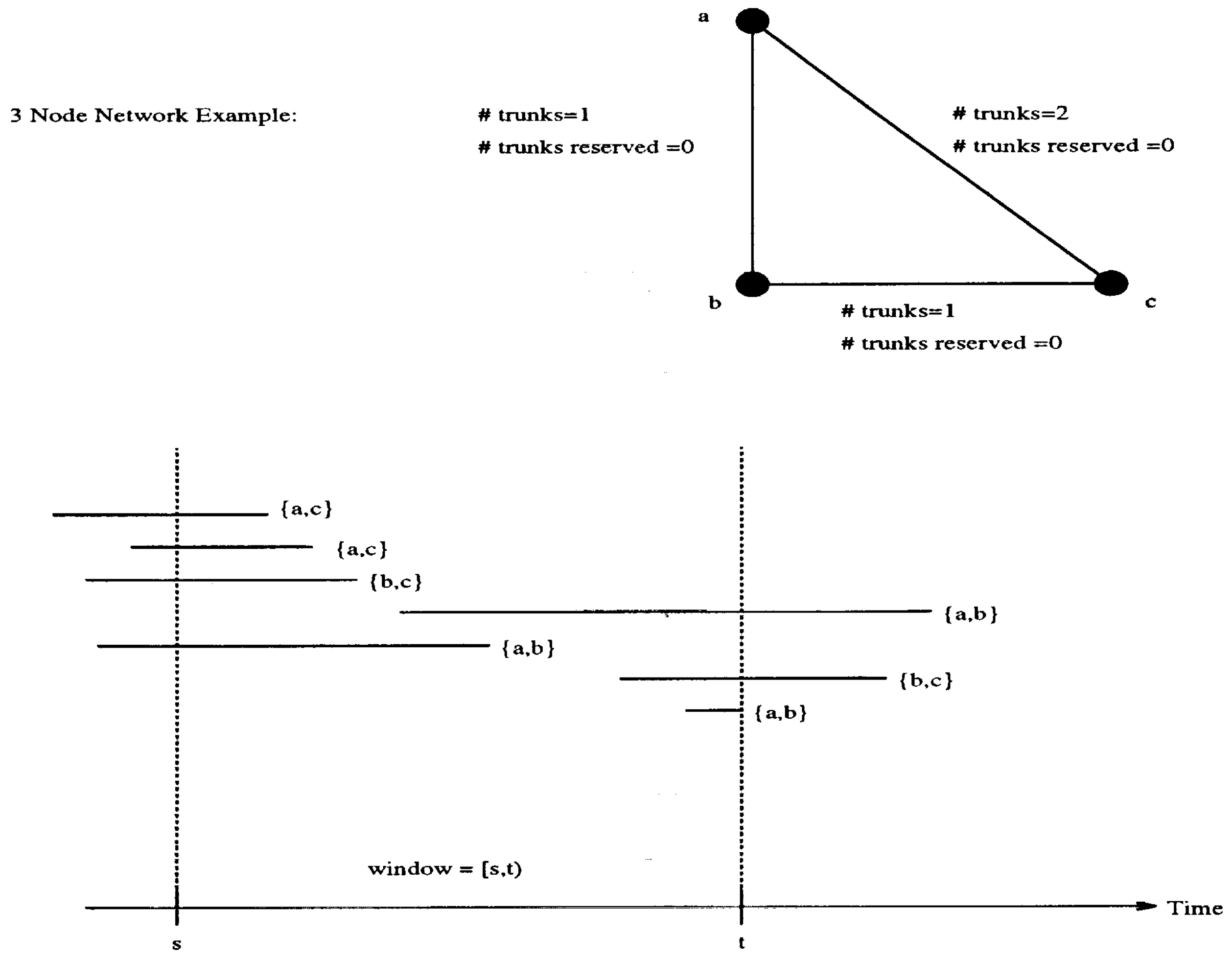

Figure 2: A line segment represents an offered call, with its left endpoint marking the arrival time and the right endpoint the departure time. As there are just three nodes in the network, the via associated with each of the calls is forced; for example the via associated with the two calls arriving to link $\{a, b\}$ within the window $[s, t)$ must be c.

Figure 2 depicts a simple example for a three node network. The first three events within the window $[s, t)$ are departures of calls that arrived earlier, and so do not enter into the calculation of the extent of the window. The window has maximal extent t, terminating with the departure of the second call offered to a, b within the window.

It is natural to ask how many arrivals fall into the maximal window $[s, t)$. Recall that a call's minimum holding time is C. Decompose $[s, t)$ into an initial part $[s, s + C)$ and a final part $[s + C, t)$. The distribution of arrivals within the initial part is Poisson with mean λC, where

$$\lambda = \sum \lambda_{i,j}$$

is the aggregate call arrival rate in erlangs. The analysis of the final part $[s+C,t)$ requires a bit more work. In brief, the idea is to consider an absorbing Markov process describing the number of calls present that have completed the deterministic C delay in their holding times: when k are present, one of the k finishes (thereby stopping the process) at rate $k(1-C)$, whereas another such call arrives at rate λ. Analysis of this process provides the expected number of events within $[s+C,t)$ as

$$\textstyle\sum_{k=0}^{\infty} B(\lambda(1-C)+1,k+1)[\lambda(1-C)]^{k+1}/k! \quad = \quad \sqrt{\lambda(1-C)\pi/2}+O(1/\sqrt{\lambda}) \quad \text{as } \lambda \to \infty,$$

where $B(\cdot,\cdot)$ is the Beta-function. Summarizing, the expected number of events within the window is

$$\lambda C + \sqrt{\lambda(1-C)\pi/2} + O(1/\sqrt{\lambda}).$$

For large networks, such as the AT&T network, the aggregate call arrival rate λ is on the order of 1 million erlangs, and the number of events in the window will be large, even for small C, as will be seen in Section 6.

An iterative method is used to simulate the window. As alluded to above, an iteration operates on the offered events, updating associated state information. An event rejected in the course of an iteration is not offered at the next. An iteration involves a separate computation for each node-pair, addressing the *feasibility* of each arrival event on the corresponding link. We say a *direct-arrival* offered to node-pair $\{i,j\}$ is feasible if at least 1 trunk is free (i.e., unused) on link $\{i,j\}$ at the time of the arrival, and is not feasible otherwise. Similarly, a *via-arrival* offered to node-pair $\{i,j\}$ is feasible if at least $r_{i,j}$ trunks are free on the link at the time of arrival, and is not feasible otherwise.

Feasibility decisions are combined so as to implement the logic of the routing policy. A call arrival at node-pair $\{i,j\}$ offers three events: one *direct-arrival* at $\{i,j\}$ and two *via-arrivals* at some node-pairs $\{i,v\}$ and $\{v,j\}$. If the *direct-arrival* is feasible then this event should be accepted and the two *via-arrivals* rejected: the call should be routed direct. If not and both *via-arrivals* are feasible then the *direct-arrival* should be rejected and the two *via-arrivals* accepted: the call should be routed on the alternate two-link path. Otherwise, all of the events should be rejected: the call should be blocked.

In the computation for node-pair $\{i,j\}$, the events offered to link $\{i,j\}$ are scanned in chronological order. On scanning each arrival event, an associated *state* may be updated, summarizing information collected thus far on the feasibility of the three events of the associated call arrival. Specifically, for a *direct-arrival* at node-pair $\{i,j\}$, the state is a 2-tuple whose first component is one of $\{\text{yes}, \text{no}, ?\}$, according to whether the call is feasible on link $\{i,j\}$, is not feasible on the link, or is not yet decided. Similarly, the second component is one of $\{\text{yes}, \text{no}, ?\}$, according to whether the call is feasible on both links of the two-link alternate path, is not feasible on at least one of these two links, or is not yet decided. The state of a *via-arrival* is a 3-tuple, where each component is of the same form, describing the feasibility information for the corresponding *direct-arrival*, the event itself, and the other corresponding *via-arrival*.

All state components are initially blank; i.e., set to "?". An event's state is *final* if the feasibility information determines whether or not the event should be accepted or rejected. For a *direct-arrival*, these accepted states are (yes, X), for $X \epsilon \{\text{yes}, \text{no}, ?\}$ (meaning the call is feasible on the direct path), and the single rejected state is (no, no) (meaning the call is not feasible on the direct nor on the alternate path). For a *via-arrival*, the single accepted state is $(\text{no}, \text{yes}, \text{yes})$ (meaning the call is not feasible on the direct path, but is feasible on the alternate path), and the rejected states are (yes, X, Y), $(\text{no}, \text{no}, X)$, and $(\text{no}, X, \text{no})$, for $X, Y \epsilon \{\text{yes}, \text{no}, ?\}$.

The correctness of the simulation follows from the correctness of each iteration and the fact that each iteration makes progress. In particular, the earliest event not in a final state at the start of an iteration is guaranteed to be driven into a final state during the iteration.

5 Implementation

An implementation of the sweep algorithm must address:

- the mapping of possibly unbalanced node-pair computations into the parallel computer, and
- the arrival event feasibility and state computations associated with the node-pairs.

We discuss these two issues in turn. Adaptations for a MIMD implementation are described in Section 5.3.

5.1 The Mapping

Impose an order on the processors in the parallel computer, and an order on the $N(N-1)/2$ node-pairs. We dedicate a fixed number $P_{i,j} \geq 1$ of consecutive processors to node-pair $\{i, j\}$; the index of the first of these is obtained by summing the values $P_{u,v}$ for node-pairs $\{u, v\}$ earlier than $\{i, j\}$ in the node-pair order. To simplify the discussion, consider one node-pair $\{i, j\}$. Suppose the current simulation window is $[s, t)$. To map the events of node-pair $\{i, j\}$ into the $P = P_{i,j}$ processors, assign the k^{th} in the processor order to store and manage all events that fall into the k^{th} subwindow $[s + (k-1)(t-s)/P, s + k(t-s)/P)$, for $k = 1$, $\ldots$, P. In this way, each event, identified by a node-pair $\{i, j\}$ and a time u within the window $s \leq u \leq t$, maps into a unique processor.

In our implementation, we further restrict the mapping by setting the extent of each window to C, the extent of the constant portion of the call holding time, which we assume is non-zero. Thus, the j^{th} window is the interval $[(j-1)C, jC)$. Under this restriction, the window spans an average of λC call arrivals, where λ is the aggregate call arrival rate; without the restriction the span would include $O(\sqrt{\lambda(1-C)})$ additional arrivals. This disadvantage is offset by removing the need to compute the greatest lower bound of the window boundary. In general, the current window contains *departure* events (D) corresponding to calls accepted at previous windows. Generating the random call arrivals and departures is straightforward. Initializing the set of events offered within the window:

1. Each processor independently generates the departure times for *direct-arrivals* and *via-arrivals* accepted at its link at the previous window, and creates a *departure* event at the appropriate processor. Those *departure* events that fall within the window are included in the computations to follow.

2. Each processor independently generates the call arrivals within its subwindow, at the appropriate Poisson rate, and creates a *direct-arrival* event locally, and two *via-arrival* events remotely at the appropriate processors.

To ideally balance the computational and communications load, $P_{i,j}$ should be chosen proportional to the rate at which events are offered at link $\{i,j\}$. At the outset of the simulation, this rate is unknown because we do not know which calls are accepted, and so do not know which *departure* events are offered. However, we do know the rate at which *direct-arrivals* and *via-arrivals* are offered to link $\{i,j\}$:

$$\alpha_{i,j} = \lambda_{i,j} + \sum \lambda_{i,k} P(\text{selecting via } j \text{ at link } \{i,k\}) + \sum \lambda_{k,j} P(\text{selecting via } i \text{ at link } \{k,j\}),$$

and this is within a factor of two of the total event arrival rate. In our implementation, we take $P_{i,j}$ proportional to $\alpha_{i,j}$, with an adjustment to ensure $P_{i,j} \geq 1$. We found that heuristics determining $P_{i,j}$ as linear functions of the link capacity $cap_{i,j}$ and the offered call arrival rate $\lambda_{i,j}$ performed nearly as well.

It turns out that the node-pair computations involve parallel prefix [9] computations, which can exploit locality within the interconnection network of the parallel processor. All other communications patterns are essentially balanced, random patterns.

5.2 A Single Iteration

In this section, we describe the individual node-pair computations that make up a single iteration of the sweep algorithm.

At the start of each iteration, the set of offered events consists of

- all *departures* that fall in the window (corresponding to calls accepted before the start of the window), and
- all *direct-arrivals* and *via-arrivals* that (i) fall within the window and (ii) were not rejected at an earlier iteration.

The structure of the computation is simple. As described in the previous section, the events offered to each node-pair $\{i,j\}$ are distributed across $P_{i,j}$ processors, so that each processor is responsible for a unique node-pair over a unique subwindow. At each iteration, each processor takes as input

- a local lower bound $\underline{f}_1$ on the number of free trunks available at the start of its subwindow,
- a local upper bound $\overline{f}_1$ on the number of free trunks available at the at the start of its subwindow,
- and its list of events offered within the subwindow.

In the course of the iteration, each processor scans its list of events in chronological order. On scanning its k^{th} event, the processor computes local lower and upper bounds $\underline{f}_{k+1}$ and $\overline{f}_{k+1}$ on the actual number of free trunks f_{k+1} available just before the next event. In addition, if the k^{th} event is a *direct-arrival* or *via-arrival* then the processor may locally update the event's state and may remotely update the states of the two other arrival events associated with the same call, where the updates depend in part on $\underline{f}_k$ and $\overline{f}_k$. At the end of the iteration, a parallel prefix computation [9] is carried out that determines new local lower and upper bounds $\underline{f}_1$ and $\overline{f}_1$ for the next iteration. The events offered at the next iteration are those that have not been rejected at this or any earlier iteration.

First, let us consider the computation of the bounds $\underline{f}_k$ and $\overline{f}_k$. Consider a processor dedicated to node-pair $\{i,j\}$, having trunk reservation parameter $r = r_{i,j}$. A *via-arrival* is *needed* if the corresponding call cannot be carried on the direct path. By construction, each accepted *via-arrival* is needed, but a *via-arrival* in any other state may or may not be needed—at this point we don't know. As a result, in general we cannot compute the f_k exactly, as the k^{th} event is scanned. However, by assuming that all *via-arrivals* are needed we obtain a lower bound $\underline{f}_k \leq f_k$, and by assuming all that have not been accepted are not needed we obtain an upper bound $\overline{f}_k \geq f_k$:

$$\underline{f}_k = (\underline{f}_{k-1} + \underline{x}_{k-1})^+ \quad (1)$$

$$\overline{f}_k = (\overline{f}_{k-1} + \overline{x}_{k-1})^+, \quad (2)$$

where $(z)^+$ denotes $\max\{z, 0\}$ and

$$\underline{x}_k = \begin{cases} 1 & \text{if the } k^{th} \text{ event is a } \textit{departure} \\ -1 & \text{if } (\underline{f}_k \geq 1 \text{ and the } k^{th} \text{ event is a } \textit{direct-arrival}) \\ & \text{or } (\underline{f}_k \geq r \text{ and the } k^{th} \text{ event is a } \textit{via-arrival}) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$$\overline{x}_k = \begin{cases} 1 & \text{if the } k^{th} \text{ event is a } \textit{departure} \\ -1 & \text{if } \underline{f}_k \geq 1 \text{ and the } k^{th} \text{ event is a } \textit{direct-arrival} \\ & \text{or an accepted } \textit{via-arrival} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Consider the three node network example of Figure 2. For link $\{b, c\}$ (having reservation parameter $r = 0$), the events initially offered within the window and the corresponding lower and upper bounds are

index k	1	2	3	4
event type	D	V	A	V
$\overline{f}_k$	0	1	1	0
$\underline{f}_k$	0	1	0	0

The two *via-arrivals* (V) are associated with the two calls offered to link $\{a, b\}$ within the window. The uncertainty as to whether the first *via-arrival* is needed leads to the gap between $\underline{f}_3$ and $\overline{f}_3$.

Using the bounds, we obtain the following rule for the feasibility of the k^{th} event:

	feasible	not feasible
direct-arrival	$\underline{f}_k > 0$	$\overline{f}_k = 0$
via-arrival	$\underline{f}_k > r$	$\overline{f}_k \leq r$

On scanning its k^{th} event the processor may locally update the event's state and the states of the two other arrival events associated with the same call. Figures 4 and 5 describe these updates in complete detail. (To obtain a simple and regular layout of the state transition diagrams, we include transitions out of final states; these do not occur in the implementation.) The rules are rather transparent. An event's state is just a finite memory that keeps track of the feasibility (yes, no, or ?) of the event on the link at which it is offered, and the feasibility of the two other events associated with the same call. For *direct-arrivals*, two components of this memory are collapsed to one, by an "and;" we need only determine whether both of the associated *via-arrivals* are feasible.

Figure 3 describes the operation of the sweep algorithm on the three node network example of Figure 2, assuming that each of the three node-pairs is assigned to a different processor. In this case, nothing is needed following an iteration to reinitialize the local bounds $\underline{f}_1$ and $\overline{f}_1$; these retain their initial values, namely, the number of calls in progress at the start of the window. Note that in this network each link's reservation parameter r is 0. We assume that the processors operate in lockstep, as in a SIMD architecture. At the n^{th}

List of events on each node-pair:

node-pair	Events						
{a,b}				A_4	D_5	V_6^1	A_7
{b,c}			D_3	V_4^1		A_6	V_7^2
{a,c}	D_1	D_2		V_4^2		V_6^2	V_7^1

Description of the iterations:

	iteration 1				iteration 2			
Step 1	$\overline{f}$	$\underline{f}$	Local Updates	Remote Updates	$\overline{f}$	$\underline{f}$	Local Updates	Remote Updates
{a,b}	0	0	A_4 (no,?)	$V_4^1(no,?,?)$ $V_4^2(no,?,?)$	0	0	A_4 (no,yes)	
{b,c}	0	0	D_3		0	0	D_3	
{a,c}	0	0	D_1		0	0	D_1	
Step 2								
{a,b}	0	0	D_5		0	0	D_5	
{b,c}	1	1	V_4^1 (no,yes,?)	V_4^2 (no,?,yes)	1	1	V_4^1 (no,yes,yes)	
{a,c}	1	1	D_2		1	1	D_2	
Step 3								
{a,b}	1	1	V_6^1 (?,yes,?)	V_6^2 (?,?,yes)	1	1	V_6^1 (?,yes,yes)	
{b,c}	1	0	A_6 (?,?)		0	0	A_6 (no,yes)	V_6^1 (no,yes,yes) V_6^2 (no,yes,yes)
{a,c}	2	2	V_4^2 (no,yes,yes)	V_4^1 (no,yes,yes) A_4 (no,yes)	1	1	V_4^2 (no,yes,yes)	
Step 4								
{a,b}	1	0	A_7 (?,?)		0	0	A_7 (no,no)	V_7^2 (no,no,?) V_7^1 (no,?,no)
{b,c}	0	0	V_7^2 (?,no,?)	A_7 (?,no)	0	0	V_7^2 (no,no,?)	
{a,c}	1	1	V_6^2 (?,yes,yes)	V_6^1 (?,yes,yes) A_6 (?,yes)	1	1	V_6^2 (no,yes,yes)	
Step 5								
{a,b}								
{b,c}								
{a,c}	1	0	V_7^1 (?,?,no)		0	0	V_7^1 (no,no,no)	

Figure 3: Sweep algorithm applied to the 3 node network example discussed earlier. Calls are numbered from 1 to 7 in chronological order of their arrival times. A, D, and V represent *direct-arrival*, *via-arrival*, and *departure* events, respectively. Subscripts identify the calls. Superscripts 1 and 2 distinguish the two *via-arrivals* associated with the same call. One processor is assigned to each node-pair. At step n ,the n^{th} event on each node-pair is processed, as functions of the bounds $\overline{f}_n$ and $\underline{f}_n$, triggering local and remote updates. In the local update column, at each step, the state of a *direct-arrival* or *via-arrival* is shown, even if no change is made.

step of each iteration, each scans its n^{th} event. A rejected event is left in the list, and skipped during the sweep; that is, a processor scanning such an event just drops out for the current step. It turns out that the simulation of the window $[s,t)$ converges after two iterations.

To be sure this example is clear, let us walk through the first three steps of the first iteration. We will only describe actions taken on processing *direct-arrivals* and *via-arrivals*. *Departures* trigger increments to the $\underline{f}$ and $\overline{f}$ but no state updates. Let $P(a,b)$, $P(a,c)$, and $P(b,c)$ denote the processors assigned to node-pairs $\{a,b\}$, $\{a,c\}$, and $\{b,c\}$, respectively.

At the first step, processor $P(a,b)$ scans the *direct-arrival* event A_4 (and simultaneously, $P(b,c)$ scans D_3 and $P(a,c)$ scans D_1). As the upper bound on the number of free trunks $\overline{f}_1 \leq r = 0$, A_4 is found to be infeasible. Accordingly, the state of A_4 is updated from its initial value (?,?) to (no,?), and the states of the corresponding *via-arrivals* V_4^1 and V_4^2 on node-pairs $\{b,c\}$ and $\{a,c\}$ are both updated to (no,?,?). At the second step, processor $P(b,c)$ scans *via-arrival* V_4^1. As $\underline{f}_2 > r = 0$, we find that V_4^1 is feasible. Its state is updated from (no,?,?) to (no,yes,?) and the state of its counterpart V_4^2 is updated to (no,?,yes).

Last, consider the third step. Processor $P(a,b)$ scans *via-arrival* V_6^1. As $\underline{f}_3 > r = 0$, we know the event is feasible, and so the processor updates its state from (?,?,?) to (?,yes,?), and updates the state of its counterpart V_6^2 to (?,?,yes). Processor $P(b,c)$ scans the *direct-arrival* A_6. The bounds $\underline{f}_3 = 0$ and $\overline{f}_3 > 0$ do not decide feasibility, so no updates are made. Processor $P(a,c)$ scans the *via-arrival* V_4^2. Since $\overline{f}_3 > r = 0$ the event is feasible. As a result, processor $P(a,c)$ updates the event's state from (no,?,yes) to (no,yes,yes), and updates the state of A_4 to (no,yes), and V_4^1 to (no,yes,yes). At this point, the three events associated with the arrival of call 4 are in final states: A_4 is rejected, and both V_4^1 and V_4^2 are accepted, meaning the call is carried on its alternate route.

It remains only to describe how to initialize the local lower and upper bound computations for the next iteration, when more than one processor is assigned to a node-pair. Let us consider the lower bound; the upper bound is handled analogously. Focus on node-pair $\{i,j\}$, with events distributed across $P = P_{i,j}$ processors, which we number $1, 2, \ldots, P$. Let $n(k)$ denote the number of events that map to the k^{th} processor. As consecutive processors hold the events of consecutive subwindows, we can think of all the events offered to the node-pair as distributed across the processors in chronological order in a single list of $n = \sum n(k)$ events. To compute the lower bound on the number of free trunks before the t^{th} event, for any $t = 1$ to n, we need only solve recurrence

$$\underline{f}_t = (\underline{f}_{t-1} + \underline{x}_{t-1})^+ \tag{5}$$

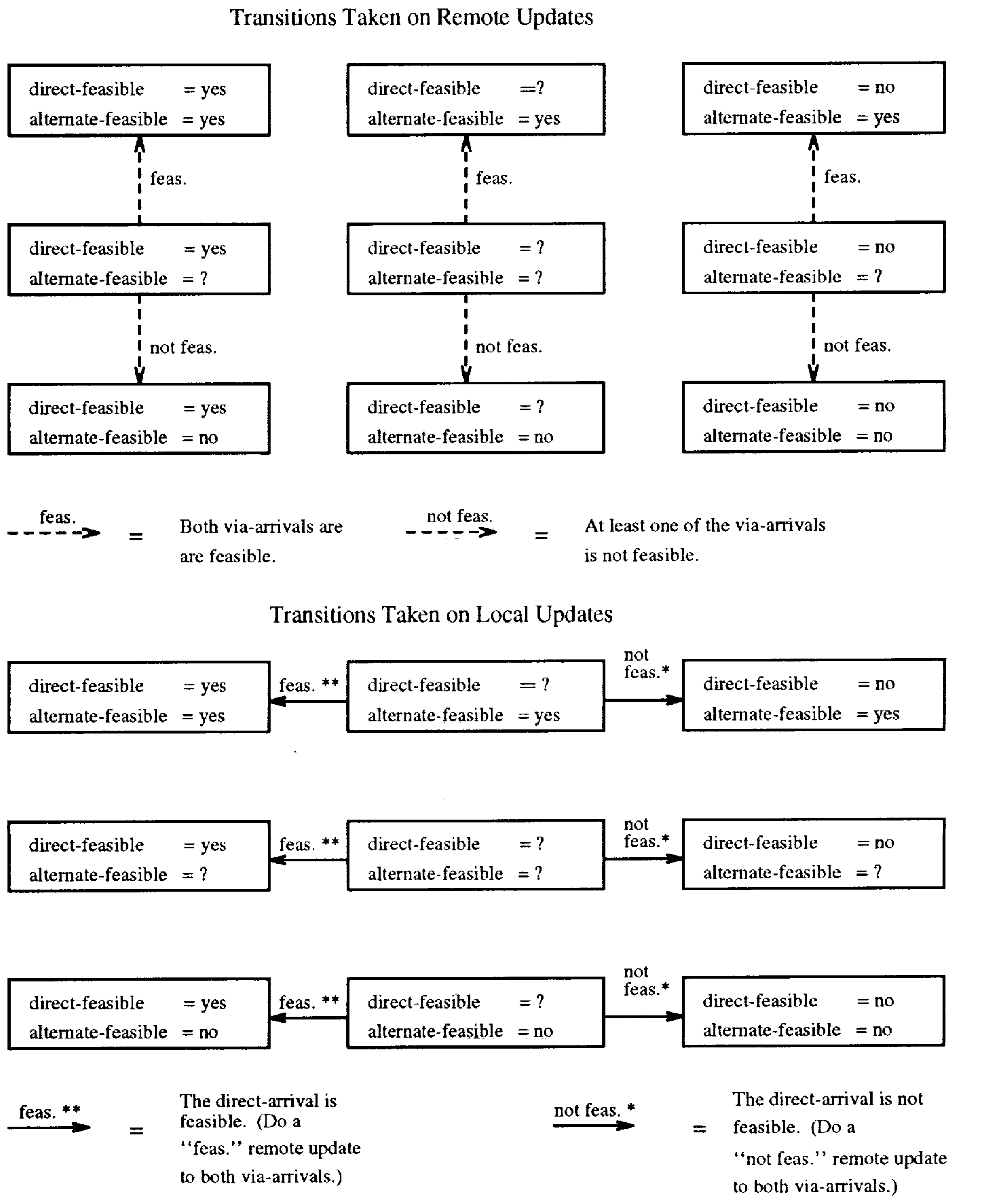

Figure 4: State transition diagram for a *direct-arrival* event. The *via-arrivals* mentioned in the figure are the two associated with the same call.

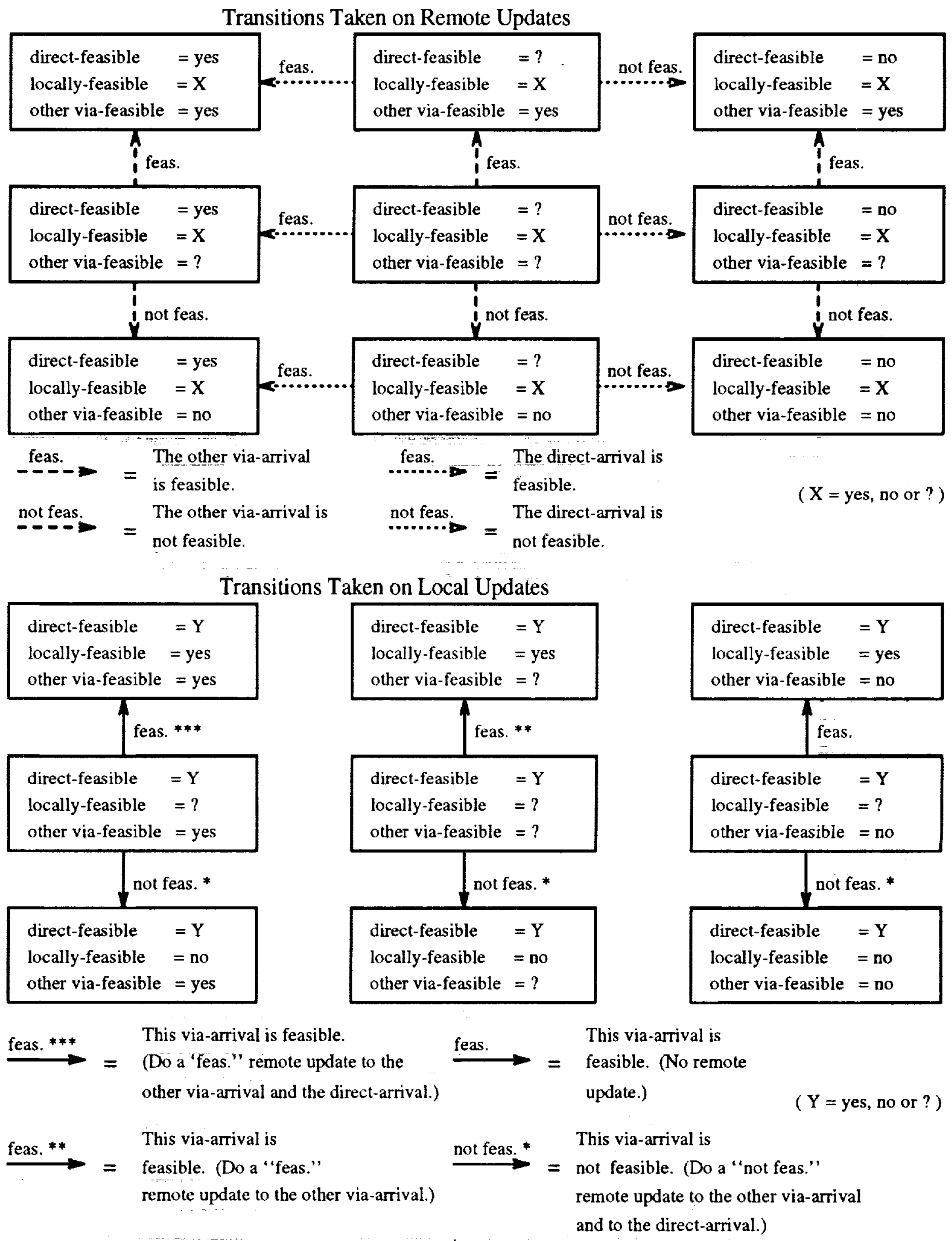

Figure 5: State transition diagram for a *via-arrival* event. Each half of the Figure encodes three pictures, obtained by varying X or Y through {yes, no, ?}. The other *via-arrival* and the *direct-arrival* mentioned in the figure are the two other arrival events associated with the same call.

for $t = 1$ to n, using the values $\underline{x}$ computed as described earlier, considered in this new order. The value needed to initialize the next iteration for the k^{th} processor is $\underline{f}_{s(k)}$, where $s(k) = \sum_{t=1}^{k-1} n(k)$, that is, the lower bound on the number of free trunks available after the last event of the previous processor.

Solving (5) reduces to parallel prefix computation [9]. Given inputs $z_1, \ldots, z_n$, and an associative operator $\circ$, the *parallel prefix* problem is to compute the n partial products: z_1, $z_1 \circ z_2$, $\ldots$, $z_1 \circ z_2 \circ \ldots \circ z_n$. To put (5) in this form, we recast it as a matrix recurrence in the semiring where max is the addition operator with identity $-\infty$ and $+$ is the multiplication operator with identity 0. Under this interpretation, the distributive law is $a + \max\{b, c\} = \max\{a + b, a + c\}$, and (5) is expressible as

$$v_t = M_t v_{t-1} \tag{6}$$

where

$$v_t = \begin{bmatrix} \underline{f}_t \\ 0 \end{bmatrix}, M_t = \begin{bmatrix} \underline{x}_t & 0 \\ -\infty & 0 \end{bmatrix},$$

and the usual rules of vector and matrix multiplication apply but with scalar addition and multiplication taken to be max and $+$, respectively. Telescoping (6) we obtain

$$v_t = M_t M_{t-1} \ldots M_2 v_1.$$

Hence to compute the $\underline{f}_t$, it suffices to:

1. solve the parallel prefix problem of computing the partial matrix products $M_2' = M_2$, $M_3' = M_3 M_2$, $\ldots$, $M_n' = M_n M_{n-1} \ldots M_2$
2. compute $v_t = M_t' v_1$, for $t = 1$ to n.

The first step dominates the computational cost. Kruskal et al. [8] show that on a shared memory model, it is possible to solve the parallel prefix problem in $O(\log n)$ time using $O(n/\log n)$ processors. Their algorithm is easily adapted to the situation at hand, where the n inputs are distributed across P processors. Taking into account that the distribution of events is random, it can be shown that the computational cost is $O(\alpha_{i,j}/P + \log P)$ with high probability where $\alpha_{i,j}$ (defined in Section 5.1) is the rate at which node-pair $\{i, j\}$ receives events. If $P = P_{i,j}$ is taken proportional to $\alpha_{i,j}$ the time becomes $O(\log P)$.

5.3 Adaptations for a MIMD Implementation

The implementation just described is well-suited for SIMD architectures, and we refer to it as the SIMD implementation. The sweep algorithm has also been implemented on a MIMD architecture, the Intel iPSC/860 (which is identical to the iPSC/2[1], except it is based on the i860 processor). There are only two significant differences between the MIMD and SIMD implementations: the mapping of node-pairs to processors, and the handling of interprocessor communication. Each of these is described in turn below.

The MIMD version maps multiple node-pairs to each processor. Thus, a given node-pair's events are always all on the same processor. We accept a node-pair's call arrival rate as a reasonable estimate of the node-pair's workload, and then view the mapping problem as identical to a multiprocessor scheduling problem where we seek to minimize the makespan of a set of independent, non-preemptable tasks. Our implementation uses a minor variation of the well-known *longest processing time first* list scheduling algorithm, first analyzed in [6]. We order the node-pairs in decreasing order of arrival rate, then step through the list assigning the next node-pair to the most lightly loaded processor. Our variation (included to balance memory utilization) limits a processor to no more than $2N/P$ node-pairs. On the scenarios studied with 16 and 32 processors, the processors received very nearly identical numbers of node-pairs. Observe that this algorithm makes no explicit attempt to balance communication, an issue that could become important on larger MIMD machines such as the Intel Touchstone Delta[10].

To initialize the simulation of a window starting at time s, the processors first determine the maximal span of the window $[s,t)$. A simple iterative procedure whose cost is negligible suffices. Next, each processor builds an event list representing all call arrivals and departures within the window $[s,t)$, for each of its assigned node-pairs.

In the SIMD version, one processor communicates with the other by directly modifying the other's state information. Our MIMD version assumes no such capability. Node-pairs communicate with each other using messages, even if the communicating node-pairs are assigned to the same processor. An identifier is associated with every new call arrival. The identifier is passed to the associated via node-pairs, and serves to uniquely identify the call arrival on its node-pair, or a via arrival on its node-pair. During window initialization a binary search tree is constructed for each node-pair, recording the identifiers for all its call and via arrival events. Presented a with message, a processor probes the appropriate search tree to find the event receiving the communication.

Our message-passing strategy attempts to minimize the number of message startup costs, by amortizing a startup cost over as long a message as possible. Towards this end, any time our MIMD algorithm generates a message between node-pairs on different processors, that message is actually buffered internally until all node-pairs have been swept over. All messages destined for a given processor i are stored in a contiguous buffer as they are generated. A message sent by one node-pair to another on the same processor is not buffered, rather it is "received" immediately. Interprocessor communication is performed in two steps. Processors notify each other of the lengths of messages about to be sent, which allows each processor to pre-allocate the buffer space into which the messages will be read (without this step an unanticipated message must be accepted in system space, and then copied to user space when requested). Following a global synchronization, the aggregate messages are sent, one per processor, to each processor. While this strategy does amortize startups and minimize memory-to-memory copying, it does not utilize the communication network bandwidth particularly well. Our strategy is fine when the number of processors is moderate (≈ 16), so that the computation/communication ratio is high. For larger numbers of processors one ought to adopt a strategy of sending smaller messages more frequently, in an effect to reduce contention and better use the communication network.

6 Experiments

Next we present the results of experiments performed on symmetric and asymmetric networks, on both the MasPar MP-1 and Intel iPSC/860.

6.1 SIMD

There are a number of different performance issues we might examine in the SIMD version. The first of these is scalability—does overall performance increase as the problem size and architecture size grows? The table below records the estimated number of calls processed per minute, as a function of the number of processors used, and the number of nodes in a symmetric network.[1] Each link is assumed to have 200 trunks, an arrival rate of 210 erlangs, and 5 reserved trunks. We have observed that performance is weakest in this situation where the ratio of arrival rate to capacity is close to 1. The table shows that the problem is naturally massively parallel. Nearly three million calls are processed each minute on the largest problem, on the largest architecture.

[1] To convert to events processed per minute—the actual measurement, multiply each number by 4 (one arrival, 2 via arrivals, 1 departure).

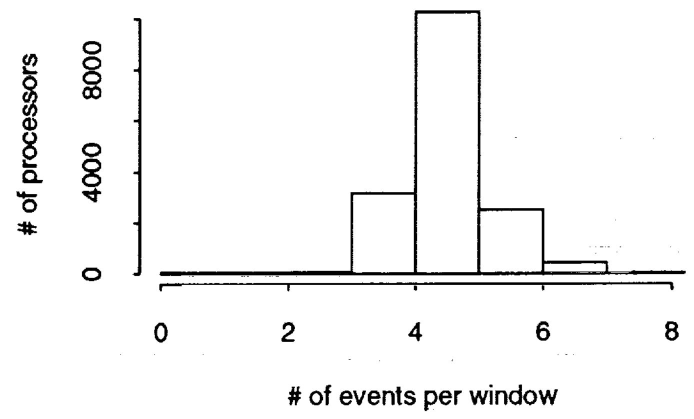

Figure 6: Histogram plotting the number of events per window per processor for a 50 node subnetwork of the 114 node network of the fiber cut scenario, with C = 0.2. Low variance proves the quality of the partitioning.

Number of Calls Simulated per Minute			
# Processors	10 nodes (≈ 10,000 trunks)	32 nodes (≈ 100,000 trunks)	100 nodes (≈ 1,000,000 trunks)
1,000	147,720	306,840	
2,000	213,720	458,880	
5,000	284,880	779,340	1,114,560
10,000	330,300	1,124,400	2,348,520
15,000	337,680	1,368,600	2,964,600

The partitioning of processors among the node-pairs is done using the arrival rate and routing parameters as described in Section 5.1. This choice is validated by the histogram in figure(6.1), which depicts the observed number of events per window per processor. The quality of the partitioning is inferred from the low variance in the histogram.

A key ingredient for top performance is the use of a small constant C in the call holding time. However, does the constant appreciably alter network performance statistics? It appears that the answer is no. The next table gives network and performance statistics as a function of C, on the realistic fiber cut scenario where the parameters are based on the AT&T network. First, observe that network statistics (average percentage calls rerouted, blocked) are relatively insensitive to C (the $C = 0.05$ statistics are nearly identical to $C = 0$ statistics measured on a different implementation). Now consider performance. An "iteration" comprises one scan through the offered events, as described in Section 5.2. Two natural metrics are the average numbers of events processed per window, and the average iterations required to bring a window to converge. However, note that as C increases, the latter average increases. This comes as no surprise, as the total number of events in the window increases in C. A normalized metric is to measure the iterations per event; the lower that number, the lower the per-event cost of simulation. Finally, we are also interested in the raw number of calls processed per minute.

C	% rerouted	% blocked	events/window	iterations /window	iterations / event	calls/min
0.05	3.36	7.46	104,367	7.24	$7.01\ 10^{-5}$	1.22M
0.1	3.37	7.48	215,944	8.81	$4.15\ 10^{-5}$	1.51M
0.2	3.39	7.50	413,316	10.62	$2.57\ 10^{-5}$	2.0M
0.3	3.42	7.62	620,005	12.35	$1.99\ 10^{-5}$	1.98M
0.4	3.43	7.70	826,148	13.16	$1.59\ 10^{-5}$	1.97M

We see that the percentage of blocked calls varies in a range of 2 % while the speed of the simulation varies in a range of 40 %. For $C = 0.2$, where the performance peaks, the variation in the percentages of blocked and rerouted calls over $C = .05$ is very small. There is much to be gained performance-wise from a substantial C, with little cost to accuracy. As further evidence of insensitivity to C, Figure (6.1) compares the blocking statistics for a $C = 0.05$ run, and a $C = 0.4$ run. For each node-pair with arrival rate > 100 erlangs, we plot a point (x, y) where x is the node-pair's measured blocking frequency for $C = 0.05$, and y is the corresponding statistic for $C = 0.4$. (Limiting the data to node-pairs with arrival rates of at least 100 erlangs removes the Monte Carlo error; node-pairs with fewer erlangs may have received few arrivals in the runs from which the data was collected.) Insensitivity to C is observed by noting that most points lie on the diagonal, i.e., $x \approx y$.

Another important performance factor is the ratio of a node-pair's arrival rate to its capacity. When there is imbalance between the two, the computation for the node-pair quickly discovers it is able to accept, or able to reject its calls. This is reflected in relatively smaller numbers of iterations required per window for the computation to converge. As shown in Figure 8 more iterations are required when the arrival rate and capacity are close to each other.

We found no one part of the computation crops up as a bottleneck. Only about 25% of the time is spent on interprocessor communications. The number of events processed per window is large and the number of iterations needed per window to converge is small. In absolute terms, on the realistic fiber cut scenario, using $C = 0.2$ we achieve eight million events/minute, equivalently, two million calls / minute. A useful run requires about 20 million calls to be simulated, which we an achieve in a few minutes. As discussed below,

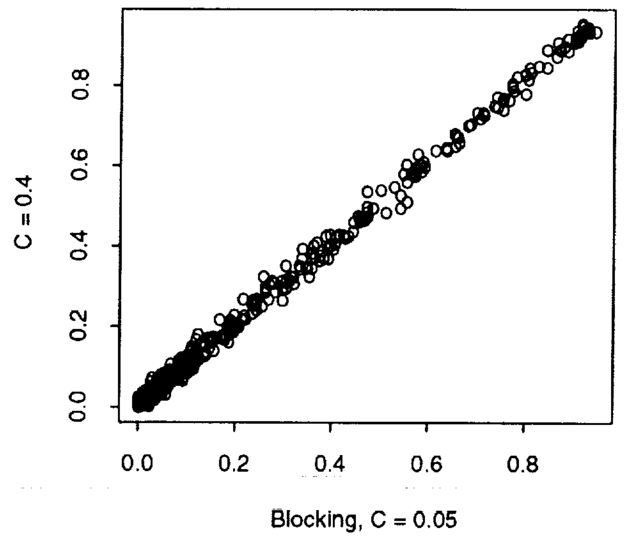

Figure 7: 114 node fiber cut scenario; measured blocking frequencies for node-pairs with arrival rates at least 100 erlangs.

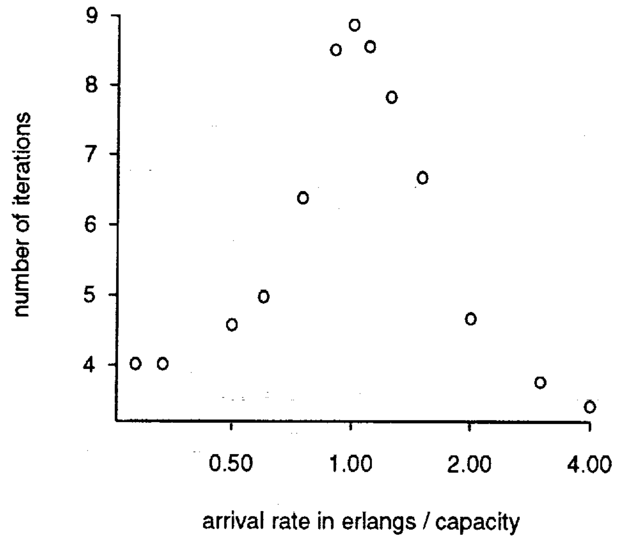

Figure 8: Variation of the number of iterations with the ratio rate of arrival / capacity, measured in the simulation of a 50 node symmetric network.

it appears that this is about ten times faster than an optimized serial simulation running on a powerful workstation with memory large enough to retain the simulation in core.

6.2 MIMD Performance

Many of the performance metrics examined in the SIMD case reflect event densities and sweep counts. Such metrics are (for the most part) not architecture dependent, and so are very close in both SIMD and MIMD versions. The principle difference between the versions is the event processing rate, which we examine below.

In the SIMD case we observe that raw performance increases on symmetric networks as the size of the problem and the number of processors changes. The corresponding data for the MIMD version (on the same simulations) is given below. Empty entries could not be filled owing to memory exhaustion.

Number of Calls Simulated per Second			
	10 nodes	32 nodes	100 nodes
# Processors	(≈ 10,000 trunks)	(≈ 100,000 trunks)	(≈ 1,000,000 trunks)
1	96,935		
2	163,495	172,312	
4	244,909	324,755	
8	281,703	570,376	
16	228,999	953,646	1,570,680

Like the SIMD version, we observe increasing call processing rates as both problem size and number of processors increases.

We also considered the performance of the asymmetric AT&T network example. Here we measure various metrics as a function of C. Utilization generally measures the fraction of time a processor spends doing "useful" work, which is measured in this case as the time **not** spent in inter-processor synchronization and message passing.

C	events/window	iterations/window	iterations/event	calls/min	utilization
0.0	3,966	4.56	1.2510^{-3}	0.63M	75%
0.01	25,000	6.58	2.510^{-4}	1.20M	83%
0.05	110,715	12.03	1.110^{-4}	1.22M	87%
0.1	217,695	22.93	1.0510^{-4}	0.98M	88%
0.2	431,892	46.34	1.0510^{-4}	0.73M	89%

The table above exhibits a curious anomaly. We see that the event processing rate increases in C for a time, then drops off. At the same time the processor utilization does not diminish. We hypothesize that the phenomenon is due to the behavior of the binary search tree routines used in message passing. These trees are probed both for interprocessor messages, and intraprocessor messages; as C changes the relative proportion of interprocessor to intraprocessor messages remains constant. The time spent handling an interprocessor message is counted as overhead, whereas the time spent handling a intraprocessor message is not (the coarse resolution of the timer disallows a more uniform treatment). However, since the window size grows, the number of messages in the search trees grows, thereby increasing the cost of the probe. This may explain why raw performance turns down, while the relative fraction of on-processor to off-processor related work does not.

The data presented here shows that a 16 processor iPSC/860 can deliver over a million calls per minute on the AT&T example. Compared to an ordinary workstation, this is excellent. An optimized serial simulation of the AT&T network, running on an ordinary Sparc workstation, required over 8 hours running time to simulate 10M calls . Most of this was due to handling page faults; the problem needs a very large memory (and of course, one of the advantages of parallel architectures is the enlarged memory space). However, one does wonder how fast an optimized serial implementation would execute on an i860 based processor with sufficient memory. While we have not answered that particular question, we have measured the execution rate of a large optimized network simulation on one iPSC/860 node to be 0.2M calls/min. One expects that simulation to process calls faster than on the full AT&T problem, as its event list management costs are somewhat lower. If we use this rate as a serial baseline, we see that our parallel AT&T code achieves a speedup of at least six. In reviewing this data, one should remember two things. First, that our algorithm provides a way to exploit the larger memory of a distributed memory machine; on the order of 128 Mbytes are needed to simulate our largest example. Secondly, our algorithm requires more computation than an optimized serial version. This is the price we pay to exploit parallelism.

7 Concluding Remarks

The massive parallelism of SIMD architectures offers significant computational advantages for large problems. However, one of the real challenges of SIMD parallelism is to find algorithms that effectively exploit parallelism within the SIMD paradigm. This paper develops a new SIMD algorithm for the discrete-event simulation of circuit switched networks. Our algorithm introduces two innovations. Using the notion of sweeps, we straddle the gap between optimistic and conservative parallel simulation methods, showing how one can compute events out-of-order, and yet never commit to an event in error. The second innovation is in showing how to effectively distribute the workload of a highly heterogeneous network model.

An SIMD version algorithm we propose was implemented on a MasPar MP-1, and a MIMD version on the Intel iPSC/860. The former architecture using 16K processors is able to simulate as many as three million calls per minute, the latter (using 16 processors) can simulate at half that rate. These processing rates are an order of magnitude faster than what one could expect from an optimized serial implementation running on a i860-based processor with huge memory. Given that running times of realistic simulations are often measured in hours, our algorithms offer the ability to simulate larger models, faster, than is now the practice.

References

[1] L. Bomans and D. Roose. Benchmarking the iPSC/2 hypercube multiprocessor. *Concurrency: Practice and Experience*, 1(1):3–18, September 1989.

[2] S. Eick, A.G. Greenberg, B.D. Lubachevsky, and A. Weiss. Synchronous relaxation for parallel simulations with applications to circuit-switched networks. In *Proceedings of the 1991 SCS Multiconference, Simulation Series; vol. 23, no. 1*, pages 151–162. SCS, 1991.

[3] S. Eick, A.G. Greenberg, B.D. Lubachevsky, and A. Weiss. Synchronous relaxation for parallel simulations with applications to circuit-switched networks. To appear in *ACM Transactions on Modeling and Computer Simulation*, 1992.

[4] T. Feder, A.G. Greenberg, M. Rausch, V. Ramachandran, and L.-C. Wang. Complexity of the telephone connect problem. To be submitted to *Information and Computation.*

[5] R.M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):31–53, 1991.

[6] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–419, March 1969.

[7] F.P. Kelly. Loss networks. *The Annals of Applied Probability*, 1(3):319–378, August 1991.

[8] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, C-34(10), October 1985.

[9] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, 1980.

[10] Sigurd L. Lillevik. The Touchstone 30 gigaflop DELTA prototype. In *Distributed Memory Computer Conference 91*, pages 671–677. IEEE PRESS, April 1991. The Paragon is not discussed.

[11] A.G. Greenberg B.D. Lubachevsky and L.-C. Wang. A synchronous relaxation method for massively parallel discrete simulation of circuit-switched networks. To be submitted to *ACM Transactions on Modeling and Computer Simulation.*

[12] R.B. Wolf. Advanced techniques for managing telecommunications networks. *IEEE Communications Magazine*, 28(2):76–81, October 1990.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE July 1992	3. REPORT TYPE AND DATES COVERED Contractor Report

4. TITLE AND SUBTITLE A SWEEP ALGORITHM FOR MASSIVELY PARALLEL SIMULATION OF CIRCUIT-SWITCHED NETWORKS	5. FUNDING NUMBERS C NAS1-18605 NAS1-19480 WU 505-90-52-01
6. AUTHOR(S) Bruno Gaujal Albert G. Greenberg David M. Nicol	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	8. PERFORMING ORGANIZATION REPORT NUMBER ICASE Report No. 92-30
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	10. SPONSORING/MONITORING AGENCY REPORT NUMBER NASA CR-189680 ICASE Report No. 92-30

11. SUPPLEMENTARY NOTES

Langley Technical Monitor: Michael F. Card
Final Report

Submitted to Journal of Parallel and Distributed Computing

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified - Unlimited Subject Category 61	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A new massively parallel algorithm is presented for simulating large asymmetric circuit-switched networks, controlled by a randomized-routing policy that includes trunk-reservation. A single instruction multiple data (SIMD) implementation is described and corresponding experiments on a 16384 processor MasPar parallel computer are reported. A multiple instruction multiple data (MIMD) implementation is also described and corresponding experiments on an Intel IPSC/860 parallel computer, using 16 processors, are reported. By exploiting parallelism, our algorithm increases the possible execution rate of such complex simulations by as much as an order of magnitude.

14. SUBJECT TERMS simulation; networks; parallel processing; SIMD algorithms	15. NUMBER OF PAGES 28
	16. PRICE CODE A03

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