Time-Parallel Algorithms for Simulation of Multiple Access Protocols

Kevin G. Jones and Samir R. Das
Department of Electrical & Computer Engineering and Computer Science
University of Cincinnati
Cincinnati, OH 45221-0030
U.S.A.
Email: {kjones, sdas}@ececs.uc.edu

Abstract

We present time-parallel algorithms for parallel simulation of multiple access protocols for medium access – in particular, slotted Aloha and slotted p-persistent CSMA. Two mechanisms are presented — regeneration point-based and fix up-based. Aloha is simulated using both mechanisms and CSMA is simulated using only the first mechanism. An analytical technique is developed to predict speedup for the regeneration point-based scheme for slotted Aloha. Speedup values obtained from the analytical technique are found to be in good agreement with those obtained from simulations. In general, it is observed that any mechanism that reduces the number of backlogged packets has good parallel performance regardless of the protocol simulated or the mechanism used to parallelize the simulation.

1. Introduction

Large-scale simulations of wireless/mobile networks are complex and slow compared to their wired counterparts. The reason is threefold. First, intricate inter-layer interactions being common in wireless, there is a need for detailed simulations of all protocol layers together. The vagaries of radio channels and the shared nature of the medium affects the upper layer protocols in unpredictable and interesting ways. Second, mobility makes the network behavior very dynamic, increasing the number of events that need to be processed for effective modeling. Third, there is a new emphasis on sending multimedia over various forms of wireless networks, increasing the complexity in the application layers. Parallel discrete event simulation (PDES) mechanisms present an opportunity to speed up simulations of wireless networks. Indeed, some work in this direction has been reported in [2, 14, 10].

Traditional PDES techniques fall in two categories: conservative and optimistic. In both these mechanisms the simulation problem is usually partitioned *spatially* by partitioning the simulation state. Each partition is modeled by a logical process (LP). LPs are mapped onto physical processors. In conservative mechanisms [12], the LPs are synchronized by using a block-resume technique. An LP is prevented from making progress in simulation time if, after doing so, there is a chance of its receiving an event message at an older simulation time. Such an LP is blocked and is again allowed to make progress only when there is no such possibility. Conservative mechanisms require knowledge of *lookahead*, the ability of an LP to predict how long in future simulation time its execution will be unaffected by activities in other LPs. More lookahead generally means less blocking and hence more parallel performance.

In contrast, optimistic mechanisms use an aggressive style of operation. Here, an LP makes progress unless it receives an event message in a past simulation time. When this happens, the LP rolls back to an appropriate earlier state. To implement rollback, the LPs must checkpoint their state periodically. Optimistic methods do not need to determine lookahead, but checkpointing and rollback overheads affect performance. Also, checkpointing itself is a complex process unless the simulation kernel has complete access to the process states. This is hard to do for independently developed simulation models. Thus, it comes as no surprise that the tools that parallelize existing sequential simulators [13, 15, 8] chose to use conservative techniques.

Time-parallel simulation [11] partitions the simulation in the *temporal dimension.* Each processor simulates only a portion of the whole simulation time. The processor that starts from an intermediate simulation time "assumes" some starting state. If the ending state of the processor that simulates the previous portion of the simulation time turns out to be different from this assumed starting state, the simulation is wrong and must be corrected. Two techniques can be used to generate a correct simulation. In the *regeneration point-based* scheme [11], all starting states are equal to a state the simulator is known to return to occasionally. Each processor simulates until the regenerative state is encountered again. It is possible to achieve a coherent simulation by threading the simulations of all processors together. In the other, *fix up-based* [6] scheme, the simulation of a processor that started with a state not matching the previous processor's ending state is corrected. The success of the first scheme depends on how frequent regeneration points are (more frequent is better). The success of the second scheme depends on how long the corrections take relative to the original simulation. More details of such schemes are discussed in later sections.

In this paper we pursue the modest goal of evaluating the effectiveness of time parallel simulations for multiple-access protocols in a completely connected wireless network (every node can hear each other). We present and evaluate techniques for slotted simulations, in particular slotted Aloha and a variation of CSMA (carrier-sense multiple access) medium access protocol. The hope is that success in demonstrating effectiveness of time-parallel simulations for such protocols will indicate that similar mechanisms could be employed for faster simulations of a complete wireless network protocol stack (with routing, transport and application layers on top of medium access) and in more general scenarios (such as sparsely connected network, changing topology for mobility, diverse traffic). This study will also garner interest in exploiting combination mechanisms (both time- and space-parallel [4]) for more effective exploitation of parallelism.

2. Related work

A number of prior work used space-parallel, conservative or optimistic PDES techniques and tools to speed up various wireless network simulations. See for example [14, 2]. Here, we review in detail only the time-parallel mechanisms, as the rest of our work will concentrate on time-parallel mechanisms alone.

One of the early work was done by Chandy and Sherman [4]. They observe that simulation space-time can be partitioned arbitrarily, with updates being made when inputs to a partition change. Simulation continues until the whole system converges. A very general technique is proposed without any specific examples or indications of performance.

Heidelberger and Stone [6] simulate set associative cache accesses by partitioning the access trace in block fashion among the processors, resulting in a time-parallel partitioning. Each processor starts with an empty cache, and stores references for the first k misses of each set, so that fix-up can be performed later. When the cache accesses have been simulated, ending cache contents are compared to the next processor's first misses to correct for misses that were counted incorrectly. A miss was counted incorrectly if one of the first k references happened to be in the cache at the end of the previous neighbor's simulation. The speedup is very good because the amount of influence a processor has on its neighbor (in time domain) is limited by the size of each cache set. Also, initial cache state has a very limited influence on the performance of miss rate in general.

Lin and Lazowska [11] simulate a G/G/1 queue in time-parallel fashion by starting each processor with an empty queue at time zero. An empty queue defines a regeneration point. Each processor continues simulating, storing all departure times, until the queue again becomes empty, then informs its neighbor about its ending time so the neighbor can "fix" its departure times simply by incrementing them by the ending time.

Bagrodia, Chandy, and Liao [1] simulate feed-forward queueing networks using Chandy-Sherman's space-time technique, where different spatial regions are synchronized optimistically. Time-parallel regions are fixed by notification of remaining service times of servers that were busy at the end of a time partition. Fix-up is needed since each time region starts in the state where all servers are idle and all queues are empty. They evaluate performance for various space-time partitioning techniques. For purely time-parallel partitioning they observe that speedup decreases with increasing load (the ratio of arrival to service rates) in the network, since the amount of fix-up needed increases.

Greenberg, Lubachevsky, and Mitrani [5] use fast algorithms to solve certain recurrence relations to develop massively parallel SIMD algorithms for open and closed queueing networks and slotted Aloha simulations. In general, any simulations that can be expressed in terms of such recurrence relations can be parallelized by their technique.

3. Time-parallel slotted aloha

For simplicity we consider only the *slotted* versions of medium access control protocols. We describe two algorithms for time-parallel slotted Aloha. One is based on the occurrence of *regeneration points*, i.e. slots in which the simulation returns to a specific, commonly occurring state. The locations of regeneration points determine how the slots are partitioned among the processors. The other algorithm is based on a fixed partitioning of slots to processors but requires a *fix-up* phase to match simulation state across partition boundaries.

For the readers' benefit we describe briefly the operation of slotted Aloha [16]. The scenario is multiple nodes communicating via a single, shared, wireless channel, where each node can hear every other node in the network. It is assumed that the network has infinite number of nodes so that each packet arrival can be assumed to be on an otherwise idle node. This alleviates the complexities of handling buffering in the model. This assumption might underestimate performance of the parallel simulator compared to handling buffering, since some packets might get dropped in case of buffer overflow, decreasing the number of packets that actually reach the network. Under the infinite node assumption, all arriving packets are transmitted without the chance of being dropped, so the load is at least as high (for the same input parameters) as it would be using buffering.

In the model we use, zero or more new packets are generated in each slot according to a Poisson arrival process. Time is slotted, meaning that packet transmission begins only on a slot boundary. Each packet takes a single slot time to be transmitted. All new packets are immediately transmitted. If more than one packet is transmitted in a particular slot, a *collision* occurs and none of the transmitted packets are received correctly. Each colliding packet must be retransmitted in a later slot. The sender implicitly knows whether there has been a collision and retransmits the collided packet in each subsequent slot with a small retransmission probability, q, until the packet is transmitted successfully. This effectively constitutes a geometric backoff interval for the colliding packets. Retransmitted packets can again collide with other retransmitted or newly arriving packets. Thus a packet may require several attempts before transmission is successful.

Each time slot can end up in one of three states – *idle* (no packet transmission), *collision* (more than one packet transmission) or *successful* (exactly one packet transmission). The goal of the simulation is to determine the throughput of the network (i.e. , the rate of occurrence of successful slots) versus offered load. Two forms of load are of interest – rate of new packet arrivals (λ) and rate of transmissions into the network (G). G is λ plus rate of retransmissions. All rates are quoted on a packets per slot basis. Note that the network capacity is unity.

The total number of *backlogged* packets (i.e. , waiting to be retransmitted) constitutes the state of the simulation. If the evolution of the state space is memory-less (the case here), any value of the number of backlogged packets can be used as regeneration points. We use the value zero. The regeneration point-based scheme is presented below. The fix-up-based scheme will follow thereafter.

3.1. Regeneration points-based scheme

This scheme is somewhat similar to the idea explored by Lin and Lazowska in [11]. The number of slots, N, in a given simulation run is equal to the simulation endtime divided by the slot width. The slots are distributed evenly among P processors (PEs) in block fashion, so that each processor gets $S = N/P$ consecutive slots as shown in Figure 1. This slot distribution is only a first cut; each PE may end up simulating more or fewer slots, as described below.

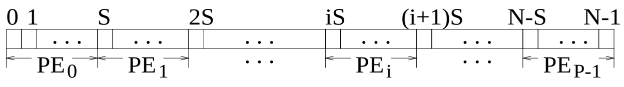


Figure 1. Distribution of N slots among P processors.

If the arrival rate of new packets is low enough, the number of backlogged packets may fall to zero in some slot. Such a slot is a *regeneration point*, since the simulation state has become the same as the starting state of the simulation. The time period from one regeneration point to the next is called a *regeneration cycle*. The simulation consists of a sequence of regeneration cycles. The idea is to distribute them evenly across the processors, but the problem is that the locations of the regeneration points are not known until the simulation is performed.

Our solution is to start each PE in a state equal to the start state and fit the results together when finished. Each PE simulates up to either the first regeneration point beyond S slots or to slot N, whichever comes first. At this point, each PE has simulated zero or more whole number of regeneration cycles, and possibly part of another one in case the PE went all the way to slot N. All that is left is to string the cycles together.

To accomplish this, the PEs communicate in a ring pattern at the end of the simulation. PE_0 initiates the communication by informing PE_1 of the number of slots it simulated, along with the number of successful transmissions counted. Each PE_i other than PE_0 reads the number of slots simulated so far (by PE_0 through PE_{i-1}), and the number of successes counted so far from its left neighbor, PE_{i-1}, updates the counts, and sends the updated counts to its right neighbor, $PE_{(i+1) mod P}$. As long as the number of slots so far is less than N, PE_i has something to contribute to the simulation. Otherwise, PE_i simply passes the counts to its right neighbor unchanged. If the slot count read from PE_{i-1} plus the number of slots simulated by PE_i is less than or equal to N, all of the work done by PE_i is useful, so PE_i increments the slot count by the number of slots it simulated and updates the success count similarly. Otherwise, only part of the work done by PE_i was useful. In other words, PE_i simulates the last regeneration cycle in the simulation. The number of useful slots simulated by PE_i in this case is just N minus the number of slots simulated so far. Call this difference k. PE_i simply looks up the cumulative success count stored in its kth slot and adds this value to the successes so far before passing a slot count of N and the total success count to its right neighbor. Finally, PE_0 reads the total slot count (which should be N) and the total success count from its left neighbor, PE_{P-1}, and outputs the results of the simulation. Pseudocode for

PE_i using this algorithm is shown below.

```
//Given N, P, and i
S := N/P;
backlog := 0;
slotsSimulated := 0;
repeat {
   simulate the slot (i.e. determine whether idle,
                      success, or collision)
   slotsSimulated++;
} until first regeneration point after S slots, or have
        simulated to slot N

if (i == 0) {
   send msg(slotsSimulated, number of successes)
        to PE i+1;
   receive msg (N, totalSuccesses) from PE P-1;
   output throughput of totalSuccesses/N;
}
else {
   receive msg (runningTotalSlots,
                runningTotalSuccesses) from PE i-1;
   if (runningTotalSlots < N) {
      k := N - runningTotalSlots;
      if (k >= slotsSimulated) {     //all work is useful
         runningTotalSlots += slotsSimulated;
         runningTotalSuccesses += number of Successes;
      }
      else { //only some of the work of PE i is useful
         runningTotalSlots := N;
         runningTotalSuccesses +=
            slot[k].(cumulative number of Successes);
      }
   }
   send msg(runningTotalSlots, runningTotalSuccesses)
        to PE (i+1) mod P;
}
```

In the worst case, no regeneration point is encountered before simulation endtime, so PE_0 does all the useful work, and there is no speedup over the sequential simulation. In the best case, each PE_i finds a regeneration point just after S slots, resulting in a speedup of N/S, or simply P.

We can think of the sequences of slots being shifted so they do not overlap. This is analogous to the state matching described in [11], where a processor that assumed a start time of 0 is later informed of the end time (t_1) of its previous neighbor, necessitating incrementing all computed departure times by t_1. This in effect shifts the range of time simulated from $[0, t)$ to $[0 + t_1, t + t_1)$. The processor does not know exactly where its time range falls in the simulated timeline until it is informed of the end time of its previous neighbor.

Note that there is no change needed to the simulation output computed in the regeneration cycle, since we are only interested in measuring system throughput. We need to know how many successful packets were transmitted in each cycle, not the actual times at which they were sent. We still need to do the shifting, however, so that the PE that simulates the last slot of the simulation (slot $N - 1$) knows which of its slots corresponds to slot $N - 1$, so the appropriate output can be presented.

Due to the time shifting, this algorithm assumes that each processor generates its own independent random arrivals during simulation runtime. This, however, can be a problem if the simulation must be run using a given random number stream for the entire arrival process or, equivalently, if the arrivals are trace-driven. Usually, this requirement is present if there is a desire for repeatability of the arrival process for debugging, comparisons across protocols, variance reduction in simulation output analysis, etc.

It is still possible, albeit with a little extra work, to use common random number streams for sequential and parallel execution. For instance, each PE_i in the parallel simulation can use a unique random number stream, (say, stream i). The sequential simulation can simply switch random number streams to match the stream used by the corresponding PE_i in the parallel simulation upon encountering a slot corresponding to the first slot of each PE_i. The sequential simulation can compute when to switch streams based on observing the regeneration cycles, and by knowing the value of N/P, the number of slots per processor in the parallel simulation. Of course, different sequential runs would be needed to match random number streams for different values of P in the parallel simulation. This technique, however, still cannot match random number streams among parallel simulations using different number of processors. These complications for using common random numbers are absent in the next protocol we study.

3.2. Fix-up-based scheme

This algorithm allocates slots to processors using the block partitioning described in the previous subsection and shown in Figure 1. Since no shifting is performed, this algorithm can be used for trace-driven simulations.

The simulation proceeds in two phases: the main phase and the fix-up phase. Each processor starts the main phase of the simulation with the initial state of zero backlogged packets. Each PE_i, i ranging from 0 through $P - 1$, simulates slot $i \times S$ through slot $(i+1) \times S - 1$. This ends the main phase. Then each PE_i, for i from 0 through $P - 2$, communicates the number of backlogged packets remaining at the beginning of slot $(i + 1) \times S$ to its right neighbor, PE_{i+1}. The number of remaining backlogged packets constitutes the state of the simulation after PE_i simulates its final slot in the main phase. Then each PE_i, for i from 1 through $P - 1$, enters the fix-up phase, which may require multiple iterations. PE_0 does not enter the fix-up phase since it does not have a left neighbor. Similarly, the left-most PE drops out of each subsequent fix-up iteration.

Fix-up may be necessary in case PE_i finished the main phase with a non-zero backlog, resulting in an ending state which does not match the starting state of PE_{i+1}, (zero backlog). In order to match the states, PE_{i+1} can resimulate the PE_i's remaining backlogs on top of what it already simulated in the main phase, provided it stored the results of each slot it simulated in the main phase.

The result of a slot is either *idle*, *success*, or *collision*,

Table 1. Fix-up computation.

Number of backlogged packets transmitted in the slot during fix-up	New result
0	same as main phase result
1	$\begin{cases} success, & \text{if main phase result was } idle \\ collision, & \text{otherwise} \end{cases}$
2 or more	*collision*

depending on whether 0, 1, or more than 1 packet was transmitted in the slot, respectively. Suppose there are k backlogged packets that need to be simulated by PE_i in the fix-up phase. PE_i continues re-simulating slots in the fix-up iteration until all k backlogged packets are transmitted successfully, or until PE_i's ending slot is reached, whichever comes first, and then communicates the ending backlog to its right neighbor.

During fix-up, 0, 1, or more than 1 backlogged packets are transmitted in each slot according to the retransmit probability, q. If 0 packets are transmitted in a particular slot, there is no change to the previous result of the slot. The backlog is actually higher than it was known to be in the main phase, but since no additional backlogged packets got transmitted during fix-up for the slot, it was simulated correctly during the main phase.

If exactly 1 packet is transmitted in the slot, the new result for the slot depends on the previous result. If the previous result was idle, the new result is success (since exactly 1 packet is transmitted in this slot). The success count is incremented and the backlog count is decremented. If the previous result was success, the new result is collision, since the currently transmitting backlog collides with the single packet that was transmitted in this slot during the main phase. The success count is decremented, and the backlog count is incremented to include the previous successful packet which has now collided with the new transmission. If the previous result was collision, the new result is a collision, with no change in backlog.

If more than 1 packet is transmitted in a particular slot, the new result is a collision, regardless of the result of the slot in the main phase. In case the previous result was a success, the success count is decremented and the backlog count is incremented due to the collision of the previously successful packet. The correction computation for fix-up is shown in Table 1.

The fix-up phase continues as long as some PE has backlogged packets to simulate. There can be as few as 0 or as many as $P-1$ iterations of the fix-up phase. In the best case, each PE ends the main phase with zero backlog, so no fix-up is needed. Each PE simulates exactly S slots, resulting in a speedup of $N/S = P$. In the worst case, each PE_i ends with a backlog so high that the each fix-up iteration takes S slots, and there is still a non-zero backlog upon reaching PE_{i+1}'s ending slot. This causes i fix-up iterations to be performed by each PE_i. PE_{P-1} simulates S slots in the main phase and $P-1$ fix-up iterations of S slots each. If each fix-up slot is counted with the same weight as each main phase slot, this results in a total of $P \times S = N$ slots, resulting in no speedup over sequential simulation.

4. Time-parallel slotted CSMA

We also used similar time-parallel mechanisms to simulate slotted CSMA (carrier-sense multiple access) protocols. The p-persistent variation of CSMA [9] is used for simulation, as this has a close resemblance to the industry standard unslotted CSMA protocols used in wireless ad hoc networks based on collision avoidance (such as IEEE standard 802.11 [7]). In CSMA, carrier is sensed before transmission of any packet. It is assumed that the carrier is always sensed for one full slot. If the carrier is idle (no other transmission in this slot) the packet is transmitted in the subsequent slots with probability p. This again presents a geometric backoff interval. On the other hand if the carrier is busy, the transmission is deferred until the carrier becomes idle when the above procedure is used. A packet in backoff is also transmitted only after carrier sensing. A busy carrier results in deferment until idle and another backoff following it. Collisions are now rarer than in slotted Aloha because of carrier sensing, but they may still happen. Colliding packets are retransmitted after a similar geometric backoff with probability parameter q.

Only the regeneration point-based scheme is explored for slotted CSMA. Determining regeneration points is now more involved, since the state of the simulation consists of more than just the number of backlogged packets. In addition to backlogged packets resulting from collision, there may be packets which arrived and found the carrier busy, so they continue sensing the channel, waiting for it to become idle. There may also be deferred packets, which failed to transmit upon sensing an idle channel; they are in backoff waiting to start sensing the channel again. There may also be some packets in the middle of transmission, since the size of a packet in CSMA can be more than 1 slot. A regeneration point occurs in time-parallel CSMA when the simulation enters the state where there are no sensing packets, no transmitting packets, and no packets in backoff, whether deferred or collided.

We do not present an algorithm based on fix-up computations for CSMA. The complication is due to the fact that packets can span multiple slots. Upon fix-up, a backlogged packet may start transmission in an idle channel, but the transmission may overlap with a previously simulated transmission that started in a later slot. The previous transmission needs to be undone, since it should have found the channel to be busy instead of idle, due to the new transmission. The problem is that undoing the previous transmission leaves the non-overlapping slots (those after the end of the new transmission up through the end of the previous transmission(s)) idle, whereas they used to be busy. Some other packets may have sensed the channel during these slots, taking some action based on the fact that the channel was busy. But now that the channel has become idle for those slots, those decisions become wrong and need to be fixed. Since our simulation does not trace the fate of individual packets, that fix-up cannot be done. We leave this study as a part of our future work.

5. Performance evaluation

Performance of the time-parallel algorithms is investigated using simulation of the time-parallel mechanism. This is essentially a sequential program that loops through the PEs in sequence, counting successes until N slots have been simulated (in the algorithm based on regeneration points) or until the fix-up phase is complete (in the algorithm using fix-ups). Speedup is calculated by dividing N by the *maximum* number of slots simulated by a single PE. A fix-up slot is given the same weight as a main phase slot in the fix-up algorithm, even though the amount of computation needed may be less. So we will at best underestimate the parallel performance.

We feel that a simulation gives sufficient insight into the parallel performance and a real multiprocessor implementation is not really necessary to estimate performance. This is because the amount of interprocessor communication is very minimal in the time-parallel mechanisms studied. The only communication needed is a single message to communicate the state between neighboring processors in the time domain at the end of the main simulation phase for time shifting, and one message for each processor in each fix-up iteration. Given that the main simulation phase is long enough on each processor, we can ignore the communication time.

Before we describe the simulation results, we note that the speedup for the regeneration point-based scheme for slotted Aloha can be analytically estimated.

5.1. Analytically estimating speedup using average size of a regeneration cycle

The average regeneration cycle size (the average number of slots in a single regeneration cycle) can be used to predict the number of slots simulated by each processor in the time-parallel slotted Aloha algorithm using regeneration points, which can then be used to estimate the speedup.

The slotted Aloha protocol can be modeled as a discrete-time Markov chain (see, for example, [3]), the state being the number of backlogged packets. Let us denote the steady-state probability for state i (i.e. , the state with i backlogged packets) by π_i and the state transition probability from state i to state j by $P_{i,j}$. For the benefit of the reader the Markov chain is shown in Figure 2. The transition probabilities can be determined by observing that in state i: (a) a binomially distributed number of backlogged packets (with parameters i and q) is retransmitted and (b) a Poisson distributed number of newly arrived packets (with rate λ) is transmitted. A combination of different possibilities in (a) and (b) may increase the backlog by one or more, may decrease backlogs by one, or the backlogs may stay the same. In [3] the transition probabilities $P_{i,j}$ have been determined for finite node assumptions, which can be easily extended for infinite node assumptions used here.

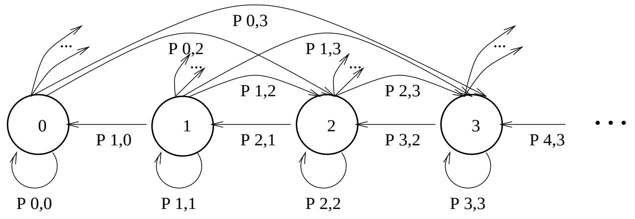


Figure 2. Markov chain for slotted Aloha. The state (i.e. , number of backlogged packets) can decrease by at most one in each slot, but can increase by an arbitrary amount.

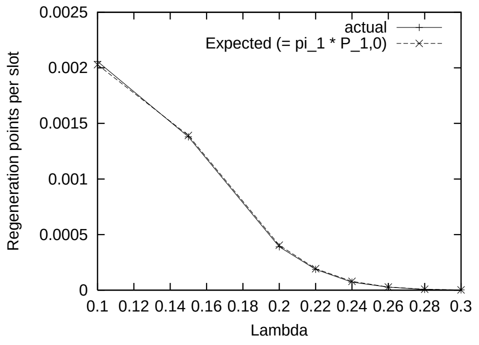


Figure 3. Regeneration points per slot, expected and actual; $q = 0.01$. Note that the curves are almost overlapping showing a very good agreement.

The transition from state 1 to 0 is used in the time-parallel algorithm to determine the regeneration point, as it defines the slot where a new regeneration cycle begins. The product $\pi_1 P_{1,0}$ gives the expected rate of occurrence of number of regeneration points (on a per slot basis). Extending the treatment in [3] for the infinite node case we get $P_{1,0} = e^{-\lambda} \times q$. This can also determined by simply observing that a state transition from 1 to 0 can only happen if both these conditions are true: there are zero new arrivals (probability = $e^{-\lambda}$) and the sole backlogged packet is transmitted (probability = q).

This still leaves for us to determine the steady state probability π_1. We calculate π_1 by numerically solving the steady state equations

$$\pi_i = \pi_{i+1}P_{i+1,i} + \sum_{j=0}^{i} \pi_j P_{j,i}$$

and

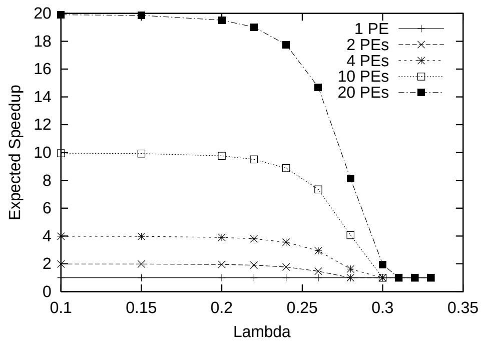
$$\sum_{i=0}^{\infty} \pi_i = 1.$$

Note that for the infinite node assumption, the Markov chain may not have a steady state distribution for large enough λ and q. For example, if $nq >> 1$ in state n, all retransmissions will collide with a very high probability and all newly arrived packets continue to add to backlogs, moving the state (see Figure 2) to the right indefinitely. See [3] and [9] for further treatment of this stability phenomenon.

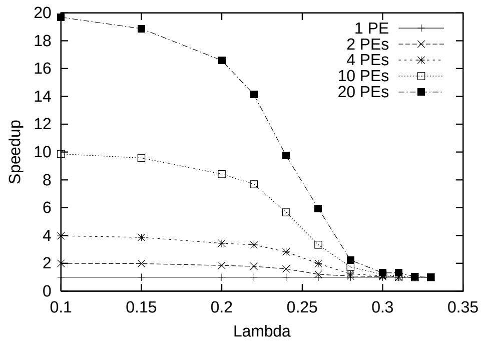
Figure 3 shows the analytically estimated number of regeneration points per slot for various values of λ and $q = 0.01$. λ was increased up to the extent steady state distributions are possible for this value of q. Actual values from simulations were found by counting the number of regeneration points that occurred in the simulation and dividing by the total number of slots, and are averaged over five different runs. The analytical and simulation curves are in excellent agreement.

The average regeneration cycle size is just the reciprocal of $\pi_1 P_{1,0}$. This can be used to determine the expected number of slots a processor actually simulates in the time-parallel simulation. A processor is expected to simulate more than one regeneration cycle if the average regeneration cycle size is less than the number of slots allocated per processor. The expected number of additional slots simulated beyond the processor's original stopping point is half the expected regeneration cycle size. In case the expected regeneration cycle size exceeds the number of slots per processor however, each processor is expected to simulate only one regeneration cycle. With this assumption, speedup can be estimated as the ratio of the total number of slots to be simulated and the expected number of slots actually simulated by each processor. The speedup predicted using this method for a simulation of 50,000 slots per processor and $q = 0.01$ is shown in Figure 4(a) for various loads. Once again, λ is varied up to the maximum extent possible. Beyond this, the slotted Aloha system is unstable.

Actual speedup from time-parallel simulation is shown in Figure 4(b). Note these set of plots are in good agreement with Figure 4(a). The slight overestimation in the expected speedup comes from fact that the *average* regeneration cycle size is used to predict speedup, whereas the *maximum* number of slots simulated by a single processor governs the actual speedup of the parallel simulation.



(a) Expected



(b) Actual

Figure 4. Expected and actual speedup of time-parallel slotted Aloha using regeneration points for increasing load; $q = 0.01$.

5.2. Simulation results

Simulations were run for 50,000 slots per processor for various loads and number of PEs in the parallel simulation. Each point in the plots presented here is an average of 10 to 15 runs using different random number seeds.

Figure 4(b) shows speedup versus λ for time-parallel slotted Aloha using regeneration points, where the retransmit probability q is 0.01. The curves correspond to different number of PEs used in parallel. As expected, speedup generally decreases as the load, λ, increases, since regeneration points become less frequent with increasing load due to the increased backlog. Note also that speedups are excellent at low values of load.

The value of q also affects the frequency of regeneration points, which in turn affects the speedup. The average number of slots a packet backs off after collision is $1/q$; so as q decreases, average backoff period increases, increasing the average regeneration cycle size, resulting in decreased speedup. Figure 5 shows speedup vs. q using 20 proces-

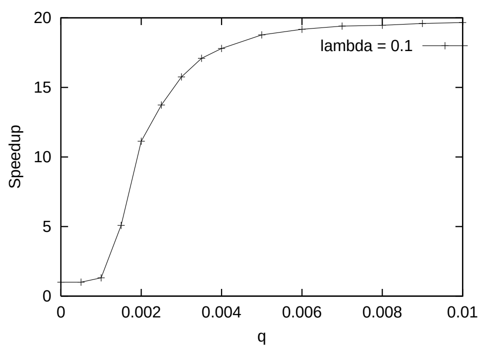
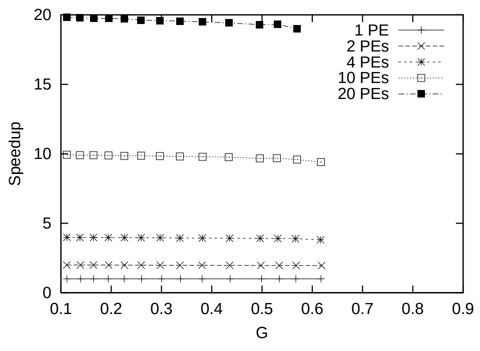


Figure 5. Speedup vs. q for time-parallel slotted Aloha using regeneration points; $\lambda = 0.1$, 20 processors.

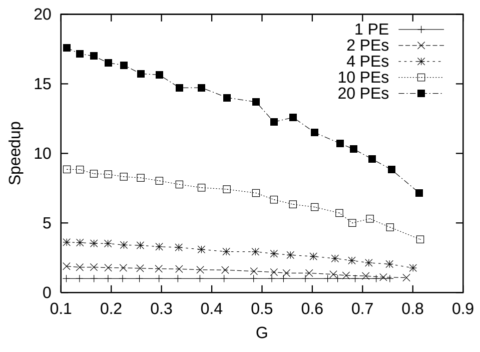
sors under relatively low $\lambda = 0.1$. Note that not only is large q good for speedup, but it also reduces packet delay in the simulated protocol, due to the smaller expected backoff period. However, large q drives the protocol to instability at a lower load as now more packets are retransmitted on average, increasing the possibility of collisions.

Speedup vs. G for time-parallel slotted Aloha with fix-ups is shown in Figure 6. Recall that G is λ plus the rate of retransmitted packets. The plots extend to different amounts on G axis as the protocol becomes unstable at different loads for different values of q. Note also lower speedup with lower value of q. This effect is similar to the regeneration point-based scheme. Larger number of packets backing off over longer intervals at processor boundaries generally means longer fix-ups. One noteworthy point is that the speedups here are higher compared to the regeneration point-based scheme. They fall comparatively much more slowly with load. This is because the fix up only involves successfully retransmitting all the packets that were backlogged at the starting slot of a processor. Finding the next regeneration point, on the other hand, involves finding a slot with zero current backlogs. Since new backlogs can always accumulate with newly arriving packets colliding, clearing out a specific set of old backlogs always happens earlier than getting to a point with zero current backlogs.

Speedup for time-parallel slotted p-persistent CSMA using regeneration points is shown in Figure 7 for two different packet sizes and two different values of q. The value p = 0.5 is used in all the plots. As expected, speedup falls with increasing load as regeneration points become rarer. Larger packet sizes usually gives better speedups as the efficiency of any CSMA protocol is higher with larger packet sizes. Efficiency goes down with increasing ratio of carrier sensing delay (one slot here) and packet transmission time [3]. Higher efficiency indicates fewer collisions, hence fewer backlogged packets to be retransmitted. As before, speedup is also sensitive to q. It decreases with lower q as



(a) $q = 0.01$



(b) $q = 0.001$

Figure 6. Speedup vs. G for time-parallel slotted Aloha using fix-up for different values of q and various numbers of processors.

the expected backoff period increases.

6. Conclusions

We presented time-parallel algorithms for parallel simulation of slotted multiple access protocols, *viz.*, slotted Aloha and slotted p-persistent CSMA. Two mechanisms were presented — regeneration point-based and fix up-based. Slotted Aloha was simulated using both mechanisms and CSMA was simulated using only the first mechanism. In addition, an analytical technique was developed to predict speedup for the regeneration point-based scheme for slotted Aloha. Speedup values obtained from the analytical technique were found to be in good agreement with those obtained from simulations. In general, it was observed that any mechanism that reduced the number of backlogged packets had a good parallel performance regardless of the protocol simulated and the mechanism used to parallelize the simulation — the reason being that our techniques used the number of backlogged packets as the simulation state.

Thus speedup is higher at lower loads, higher retransmit probabilities or larger packet sizes (for CSMA). Almost perfect speedups at low loads indicate that time-parallel techniques can be quite viable for speeding up simulations for which efficient space-parallel techniques may be hard to develop. Our future work will involve extending this work to develop combined time- and space-parallel techniques for parallel simulation of multi-hop (ad hoc) networks.

Acknowledgment

This work is partially supported by NSF CAREER grant ACI-0096186, NSF networking research grant ANI-0096264, and Ohio Board of Regents computing research enhancement funds in University of Cincinnati.

7. References

[1] R. Bagrodia, K. M. Chandy, and W.-T. Liao. An experimental study on the performance of the space-time simulation algorithm. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, pages 159–168, 1992.

[2] R. Bagrodia and X. Zeng. Glomosim: A library for the parallel simulation of large wireless networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS '98)*, pages 154–161, 1998.

[3] D. Bertsekas and R. Gallager. *Data Networks, Second Edition*. Prentice-Hall, 1992.

[4] K. M. Chandy and R. Sherman. Space-time and simulation. *Proceedings of the SCS Multiconference on Distributed Simulation*, 21(2):53–57, March 1989.

[5] A. G. Greenberg, B. D. Lubachevsky, and I. Mitrani. Superfast parallel discrete event simulations. *ACM Transactions on Modeling and Computer Simulation*, 6(2):107–136, April 1996.

[6] P. Heidelberger and H. S. Stone. Parallel trace-driven cache simulation by time partitioning. In *1990 Winter Simulation Conference Proceedings*, pages 734–737, December 1990.

[7] IEEE. Wireless LAN medium access control (MAC) and physical layer (PHY) specifications, IEEE standard 802.11–1997, 1997.

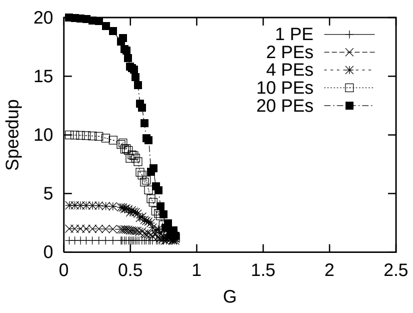
[8] K. G. Jones and S. R. Das. Parallel execution of a sequential network simulator. In *2000 Winter Simulation Conference Proceedings*, pages 418–424, December 2000.

[9] L. Kleinrock and F. A. Tobagi. Packet switching in radio channels: Part-I - carrier sense multiple access modes and their throughput-dely characteristics. *IEEE Transactions in Communications*, COM-23(12):1400–1416, 1975.

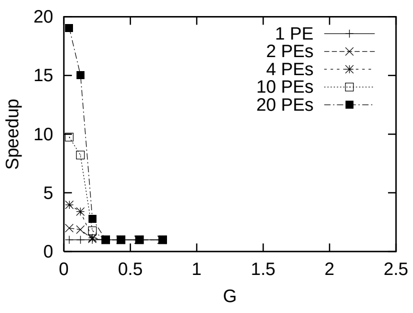
[10] M. Liljenstam, R. Rönngren, and R. Ayani. Mobsim++: Parallel simulation of personal communication networks. *IEEE Distributed Systems Online*, 2(2), February 2001.

[11] Y.-B. Lin and E. D. Lazowska. A time-division algorithm for parallel simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(1):73–83, January 1991.

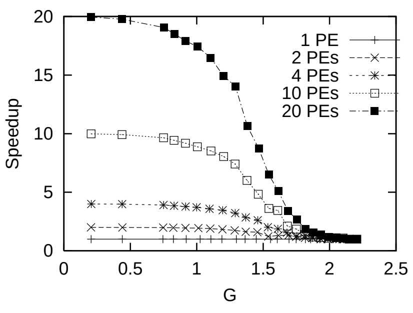
[12] D. Nicol. Principles of conservative paralllel simulation. In *Proc. of the 1996 Winter Simulation Conference*, pages 128–135, Dec 1996.



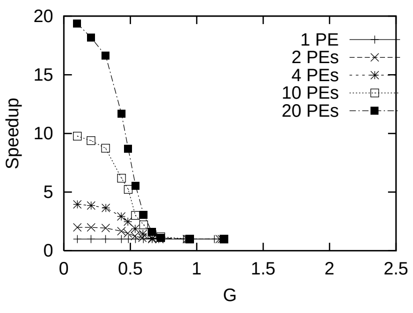
(a) Packet size = 4 slots, $q = 0.01$



(b) Packet size = 4 slots, $q = 0.001$



(c) Packet size = 20 slots, $q = 0.01$



(d) Packet size = 20 slots, $q = 0.001$

Figure 7. Speedup vs. G for slotted CSMA using regeneration points for various packet sizes and different values of q.

[13] D. Nicol and P. Heidelberger. On extending parallelism to serial simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 60–67, June 1995.

[14] J. Panchal, O. Kelly, J. Lai, N. Mandayam, A. T. Ogielski, and R. Yates. WiPPET, a virtual testbed for parallel simulations of wireless networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation (PADS)*, pages 162–169, 1998.

[15] G. F. Riley, R. Fujimoto, and M. H. Ammar. A generic framework for parallelization of network simulations. In *Proceedings of the 7th International Conference on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'99)*, pages 128–135, 1999.

[16] L. G. Roberts. Dynamic allocation of satellite capacity through packet reservation. In *Proc. of AFIPS NCC*, volume 42, pages 711–716, 1973.