TIME–PARALLEL GENERATION OF SELF-SIMILAR ATM TRAFFIC

Ioannis Nikolaidis

Computing Science Department
University of Alberta
Edmonton, Alberta
CANADA T6G 2H1

C. Anthony Cooper

Room 2K-241
Lucent Technologies
480 Red Hill Road
Middletown, NJ 07748, U.S.A.

Kalyan S. Perumalla
Richard M. Fujimoto

College of Computing
801 Atlantic Drive
Georgia Institute of Technology
Atlanta, GA 30332-0280, U.S.A.

ABSTRACT

We present a time–parallel technique for the fast generation of self–similar traffic which is suitable for performance studies of Asynchronous Transfer Mode (ATM) networks. The technique is based on the well known result according to which the aggregation of a large number of heavy–tailed ON/OFF–type renewal/reward processes asymptotically approximates a Fractional Gaussian Noise (FGN) process and, therefore, it possesses the characteristics of self–similarity and long–range dependence. The technique parallelizes both the generation of the individual renewal/reward processes as well as the merging of these processes in a per–time–slice manner. Results obtained from a message–passing implementation on a cluster of workstations confirm that it is possible to generate self–similar ATM traffic in real–time for 155 Mbps (or even faster) links and that, furthermore, the technique achieves an almost linear speedup with respect to the number of available workstations.

1 INTRODUCTION

It has been shown that the traffic resulting from Variable Bit Rate (VBR) video coding (Garrett and Willinger 1994) as well as the data traffic on an Ethernet (Leland et al. 1994) resemble self–similar processes. Subsequently, the interest has increased for generating synthetic self–similar traffic for use in the simulations of networks. The self–similar traffic generation presented herein relies on the observation (Willinger et al. 1995) that the aggregation of renewal/reward processes, with lengths derived from a heavy tailed distribution, asymptotically approximates a Fractional Gaussian Noise (FGN) process which is a self–similar process. The asymptote is taken for an increasingly large number of superposed renewal/reward processes, N, and for an increasingly large size of observation interval for the count process. In the following, this technique will be called the *aggregation* technique. Note that contrary to the aggregation of Poisson processes, the aggregation of heavy–tailed renewal reward processes leads to a remarkably different behavior (with respect to the long–term correlation) than that of the constituting processes.

The renewal/reward processes are perceived as processes alternating between an ON and an OFF period. As long as the ON/OFF period is heavy tailed (i.e., with infinite variance) and with a finite mean, the source model fits the context of the theorem on aggregation (Willinger et al. 1995). Although some initial work on the parallel generation of self–similar traffic traces has been performed (Willinger et al. 1995), it suffers from two problems. First it is specific to MasPar's Single Instruction Multiple Data (SIMD) architecture which is not a widely–used platform as opposed to networked scientific workstations environments. Second, the simulation on the MasPar proceeds in a lock–step fashion, while, as demonstrated in this paper, significant performance gains are available by treating as a single event an entire period during which the source activity remains the same.

In essence, this paper presents a Multiple Instruction Multiple Data (MIMD) algorithm that can be used in a message–passing environment. The reported performance results are for a cluster of various types of Sun Sparcstations connected to a 10 Mbps Ethernet and using PVM 3.3.10. The fact that no shared memory was available in the examples also illustrates the performance penalty that communication latency brings into the computation without harming, as will be demonstrated, the linear speedup with respect to the number of available processors.

The main drawback of the aggregation technique is its large computational requirements per sample produced (because of the large number of aggregated sources). This is the main reason why parallelism is introduced. However, even though the computational requirements are large, its actual computational *complexity* is $O(n)$ for n produced samples. All the other self–similar traffic generation techniques exhibit complexity worse than $O(n)$ and they eventually result in a slowdown of the simulation as the simulated time interval increases. The two frequently used techniques with execution time worse than $O(n)$ are Hosking's method (Hoskings 1984) and the Fast Fourier Transform (FFT) based method (Paxson 1995). Typically, no a–priori limit can be put on how long a simulation will run and it is preferable to use a technique which can continue producing samples *ad infinitum* with the same computation cost per produced sample.

Moreover, the FFT–based and Hosking's methods produce samples of the count process. Hence, post-processing is required to scale the count process, eliminate negative values and eventually to produce packet/cell arrivals consistent with the network model (i.e., the link speed). In contrast, the aggregation technique does not suffer from such problems. One way to contrast the aggregation to previous techniques is that the former operates in a bottom–up fashion (from individual arrivals to asymptotic self–similarity of the count process) while the latter operate in a top–down fashion (from a self–similar count process to individual arrivals).

Finally, certain techniques attempt to, in addition to long–range dependence (LRD), also capture the short–range dependence (SRD) as depicted, e.g., by the short term autocorrelation (Huang et al. 1995). We view these techniques as orthogonal to the task of finding an arbitrarily scalable parallel generation technique for the generation of LRD traffic. Inclusion of specific SRD components is left for future study. Instead, the current paper bases the generation of the self–similar, LRD, traffic on only three parameters, the desired Hurst parameter, H, the utilization of the link bearing the self–similar traffic, U, and the average burst length, B.

The rest of the paper is organized as follows: The details of the simulation model are illustrated in Section 2. The current implementation of the model on a message-passing network is described in Section 3. Section 4 presents and analyzes the performance results. Finally, Section 5 summarizes the conclusions.

2 THE SIMULATION MODEL

Cell arrivals will be represented by Run–Length Encoded (RLE) tuples. An RLE tuple t_i includes two attributes, $s(t_i)$, the state of the tuple, and $d(t_i)$, the duration of the tuple. The two attributes represent, the discrete time duration $d(t_i)$ over which the state $s(t_i)$ stays the same. The state is either an indication of whether the source is in the ON or OFF state (e.g., 0 for OFF and 1 for ON in a strictly alternating fashion), or the aggregate number of sources active (in the ON state) for the specified duration, that is, for N sources, $s(t_i) \in \{0, 1, \cdots, N\}$. Thus a sequence of t_i's is sufficient for representing the arrival process from an ON/OFF source or from any arbitrary superposition of such sources. The benefits of such representation is that the activity of the source over several time slots can be encoded as a single RLE tuple.

Similar representations have been used in the past for simulations for the generation of cell loss statistics in ATM multiplexers (Nikolaidis, Fujimoto and Cooper 1994). In the current context, it is not possible to avoid the fixups inherent in the time–parallel simulation. Instead, the traditional time–parallel technique of performing fixups of the state trajectory is followed (Lin and Lazowska 1991). The fixups are fast and, as it turns out, they do not alter the state in a way that several fixups are necessary. That is, due to the length of the slices, (12.5 and 25 seconds of 155 Mbps link activity per slice in the given examples), the transient due to a fixup does not cause subsequent fixups.

In summary, the algorithm proceeds by generating a large number, N, of individual source traces in RLE form. The utilization of each one of these sources is set to U/N, such that the aggregation of all N of them results in the desired link utilization to U. Each logical process (LP) of the simulation merges and generates the combined arrival trace for a separate non–overlapping segment of time, that we call a *slice*. Thus, each LP is responsible for the generation of the self–similar traffic trace in the form of RLE tuples over a separate segment (slice) of time. In logical terms, the concatenation of the slices produced by each LP in the proper time succession is the desired self–similar process. The LPs continue looping generating a different slice each time. The LP performs the generation of the self–similar traffic trace by going through the following three steps at each simulated time–slice:

1. It generates the merge of the RLE tuple traces of the N individual sources.

2. It aggregates the merged traffic into a link speed equal to the desired access link speed.

3. It corrects (fixup) the produced RLE trace by incorporating any residual cell counts.

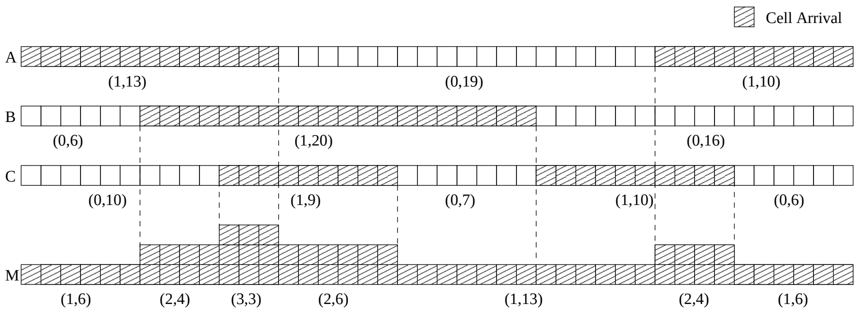

Figure 1: Example of the Merge Operation

2.1 Heavy–Tailed ON/OFF RLE Trace Generation

The generation of the individual RLE source traces is also performed in parallel. Each LP generates all the slices of a subset of sources that will be necessary to the P LPs during the generation of the current slices. That is, if P LPs are participating in the simulation, each LP generates the RLE tuples of P subsequent slices for the sources that it has been assigned to generate. It then sends the $P-1$ of them to the other LPs for each source it simulates. The individual ON/OFF sources are parameterized accordingly to fit the desired self–similar traffic. Namely:

- The shape value, α, of the Pareto distribution used for the ON period is set according to $H = (3-\alpha)/2$ (Willinger et al. 1995), where H is the desired Hurst value.
- Since N ON/OFF sources are aggregated, the per–source utilization of U/N is determined by the ratio of the ON and OFF periods of the individual processes. That is, the average OFF period E[OFF] is set to E[OFF] = E[ON] $(1-U/N)$
- The average ON period E[ON] is set to E[ON] $= B$, the average burst length, which can be derived from traffic measurements. E[ON] does not have any impact on the self–similarity, and it can be considered a free variable.

2.2 RLE Trace Merging and Aggregation

The merging of the N RLE tuple traces is performed as a merge–sort operation, where the key of the sort is the implicit position of the starting time of RLE tuples within a length of one slice. The value of the merge at any point (slot time) is the number of sources (out of the N) that are in their ON state. Figure 1 illustrates how the produced merged RLE tuple trace is related to the constituting ON/OFF RLE tuples. In Figure 1, the sequence of RLE tuples, t_i, is represented as a sequence of the pairs of its attributes, $(s(t_i), d(t_i))$. It is worthwhile to note that the priority list used to produce the merge–sort maintains, at all times, N keys. As the performance results show, apart from the generation of source RLE tuple traces, a significant portion of the execution time is spent at this step.

The sequence of merged arrivals has to be aggregated into a single ON/OFF RLE tuple trace. To accomplish this task, the merged RLE tuple trace passes through a server (which can be viewed as a multiplexer) with infinite buffer capacity and with an output link rate equal to the link rate of the desired self–similar traffic access link rate. The RLE encoded departure sequence of the multiplexer, is the desired ON /OFF traffic stream. Due to the queueing, SRD artifacts develop but do not harm the LRD features of the process. Figure 2 depicts the operation of such a multiplexer where the constituting source streams, before they are merged, are shown to the left, and the corresponding produced ON/OFF merged stream is shown to the right of the multiplexer.

The correctness of the simulation depends on the state of the multiplexer buffer. The dynamics of such a buffer are trivially represented by the following discrete–time recursion on the number of cells, Q stored at the infinite multiplexer buffer at the time just after an RLE tuple, t_i:

$$Q = \max\{Q + d(t_i)(s(t_i) - 1), 0\} \qquad (1)$$

The initial value of Q is not known to $P-1$ of the P LPs. That is, for the P parallel LPs, only the one assigned to simulate the first slice (with respect to temporal order, and indexed by 0) knows the initial state of the multiplexer Q. The remaining LPs, simply use an initial value of zero for Q. The final value of Q is sent from an LP to the LP simulating the next (in temporal order) slice. Since the assumption of $Q = 0$ may prove to be incorrect, depending on the state of the queue left over by the previous slice, a fixup phase is necessary.

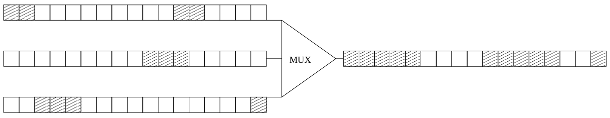

Figure 2: Example of the Aggregation Operation.

2.3 The Fixup Operation

The ON/OFF RLE tuple trace produced at the output of the multiplexer is in fact an encoded sequence of the busy and idle periods of the server. If a final state of the previous (in temporal order) slice indicates that Q should have been a value larger than zero, the corrected departure sequence can be constructed by coalescing the idle periods in order to fit the new value of Q in the idle periods of the prefix of the original departure trace of RLE tuples. Figure 3 illustrates an example of the fixup operation where the original departure RLE tuple trace from the multiplexer S is transformed by the fixup to include $Q = 9$ cells from the queue residual of the previous slice. The nine cells occupy the first idle period and part of the second, thus expanding the first ON period, as can be seen in S'. The result of the fixup is the trace at the bottom. (Note that the description of the RLE tuples is in the form $(s(t_i), d(t_i))$ for each tuple t_i.)

The overhead of the fixup is very small since it involves iterating over the first couple of RLE tuples of the departure trace until the residual cells are accommodated in the idle periods between the previously calculated departures. Typically, the fixup advances over just a few RLE tuples before it terminates. Hence, compared to all the other operations which are performed in a tuple–wise fashion (generation and merging), the fixup is the least expensive operation. In the experiments, it was verified that the large size of the slice (relative to the residual Q) does not cause further changes to the final state. That is, the transient due to the incorporation of Q additional cells is absorbed well within the length of the slice and no subsequent fixups are necessary (through a possible change in the final state of the slice). For this reason, the algorithm presented in the next section assumes that the transient due to the fixup terminates within the length of the slice.

3 A MESSAGE–PASSING IMPLEMENTATION

Figure 4 presents a message–passing implementation of the presented algorithm. There exist P LPs, LP_i, $i = 0, \cdots, P-1$. The time slices simulated by the LPs are in the order implied by the index i of the LP_i, i.e., LP_{i+1} simulates the slice of arrivals following immediately after the slice of arrivals of LP_i. Any residual multiplexer queue contents are propagated from LP_i to LP_{i+1} in order for LP_{i+1} to perform the fixup. LP_0 does not require any fixup since it has always perfect knowledge of the initial multiplexer queue state (set equal to zero before the first loop of the algorithm). The algorithm proceeds by simulating P slices in parallel at a time. Once the simulation of the P slices is completed, the final state of LP_{P-1} is sent to LP_0 so that the next set of P slices can be generated in parallel.

Each of the P LPs is responsible for generating N/P of the individual sources to be aggregated (line 3 of Figure 4). When an LP is assigned to generate a source, it generates, in each loop, P successive slices of this source's activity (loop at line 4 of Figure 4). Only one of these slices per source remains local to LP_i by assigning it to a local array `slices[]` at line 9 of Figure 4. The remaining $P-1$ slices are sent to the LPs which will process the respective slices. Therefore, the source generation process in a message–passing environment is penalized by the cost of sending the slice of the source activity to the LPs to which they correspond.

Symmetrically, each LP_i, waits to receive (lines 14 to 17 of Figure 4) the slices of the source activity corresponding to the i–th slice of all the sources that were not generated locally by LP_i. Again, in a message–passing environment, there is a penalty in waiting for receiving the slices of source activity of the sources generated in other LPs. Note that the *receive_slice*() in line 15 relies on the FIFO property for the communication between any pair of LPs and that there is no need to identify the individual source slices, since they are interchangeable with respect to the merging and aggregation operations.

Once the slices of all N sources have been received, the merging and generation of the departure process can be performed (line 19 of Figure 4) producing the departure RLE tuple trace, m, assuming that the initial state of the multiplexer queue is zero for all LP_i where $i > 0$ (line 18). LP_0 uses as initial queue state the value of q from the previous loop or $q = 0$ if it is the first loop. The final state of the multiplexer queue, q', is therefore generated by LP_i and can be sent to the LP processing the next time slice, i.e., to LP_{i+1} (line 20) where it is received as q''.

With the exception of LP_0, which does not need to perform a fixup, the LPs perform the fixup (line 23) operation taking into account the new final state received from the previous $(i-1)$ LP (line 21). The fact that all *send_state*s are performed prior to the

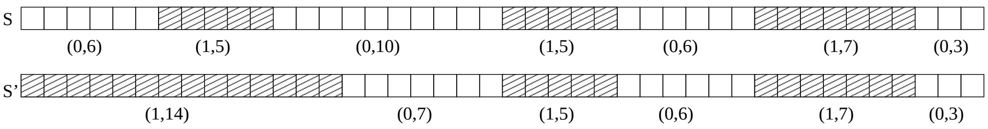

Figure 3: Fixup Operation Example for $Q = 9$.

```
LP_i:
1.   loop forever
2.       t = 0;
3.       for j = 0, ..., N/P - 1
4.           for k = 0, ..., P - 1
5.               s = generate_slice(state[j]);
6.               if k ≠ i then
7.                   send_slice(s, LP_k);
8.               else
9.                   slices[t] = s;
10.                  t = t + 1;
11.              endif
12.          endfor
13.      endfor
14.      while t < N do
15.              slices[t] = receive_slice();
16.              t = t + 1;
17.      od
18.      if i ≠ 0 then q = 0;
19.      (q', m) = merge_mux(slices, N, q);
20.      send_state(q', LP_{(i+1) mod P});
21.      q'' = receive_state(LP_{(i-1) mod P});
22.      if i ≠ 0 then
23.          m' = fixup(m, q, q'');
24.      else
25.          q = q'';
26.      endif
27.      output_trace(m');
28.  endloop
```

Figure 4: The Parallel Algorithm.

receive_states, ensures that there will be no deadlock. Note that LP_0 does not need to immediately receive q'' which will only be used in the *merge_mux* of the next loop, but it is shown performed in the same way as all other LPs for the sake of symmetry. Once the departure RLE tuple sequence has been fixed up from the original m to the correct m', it is ready for output for use by any simulation model (line 27).

4 EXPERIMENTS AND PERFORMANCE EVALUATION

A number of configurations were examined with respect to the number of sources, the length of the time slices, the utilization parameters and the average burst length. The presented experiments are for a configuration of $N = 500$ individual ON/OFF sources which was the largest configuration with respect to the number of sources and, hence, with respect to computation. The length of the time slices was set to 9139150 and 4569575 cell slots representing, respectively, approximately 25 and 12.5 seconds of operation of a 155 Mbps ATM link. The link utilization, U, was set to 20 %, resulting in a per–source utilization for each of the 500 sources set to 0.04 %. The average burst length, B, is set to 10 cells. The limit of the available memory of a workstation places a limit on the length of the time slice (because the more the RLE tuples the more the length of time that can be represented by these tuples), the selection of the size of memory for RLE tuples was dictated by the available physical memory of the workstations. In the experiments, a conservative size of memory for RLE tuples was allocated which was, at all times, no more than 8 Mbytes per workstation for the slice length of 9139150 and 4 Mbytes for the slice length of 4569575 (assuming 8 bytes per tuple: 4 bytes representing the state and 4 bytes representing the duration)

Figures 5 and 6 present the percentage of time spent in the execution at each step of the algorithm of Figure 4 for a slice length of 9139150 and 4569575 respectively. The *Generate & Send Sources* is the time spent between lines 3 and 13. The *Receive* is the time spent between lines 14 and 17. The *Generate Departures* is the time spent in line 19 and, finally the *Wait for Fixup* is the time spent between lines 22 and 26. The *output_trace* function was set to a NO–OP in order to allow measurements independently of any specific file I/O or IPC primitives used to incorporate the code in another simulation. All the remaining time spent in the execution of the algorithm (including the *fixup* function) was less than 0.05 % of the total measured time and it is not reported. Thus, it was verified that the fixup operation for this model is not a major computational burden. The experiments were run on a set of Sun workstations under Solaris 2.5 and PVM 3.3.10 and without any other significant user activity.

The shape of the curves does not differ by much between Figure 5 and Figure 6 despite the difference in the slice length. Similar behavior was observed for other parameter settings as well. The major fraction of time is spent in the generation and sending of the source slices where a slice is sent (by *send_slice*) as soon as it is produced if it is to be processed by a remote LP. Notably, the send operation is non–blocking and not in–place (in the PVM sense). Hence, each send operation involves the copying of the data to be sent in a separate buffer in order to allow the reuse of the same allocated area for the generation of the source RLE tuples in the next loop.

The more the workstations, i.e., the higher the P, the fewer the sources produced by each workstation. Consequently, the *Generate & Send Sources* part of the execution time decreases, as a percentage, for increasing P but only to the point where the overhead due to the send operation becomes the dominant overhead. Thus, the gain out of splitting the generation of sources over a gradually larger set of processors is diminished by the fact that most of the produced source slices have to be sent to other workstations. The net result is that eventually the *Generate & Send Sources* percentage remains almost constant as P increases.

On the other hand, the Receive percentage (from being zero when $P = 1$) increases as expected as the number of sources produced at other LPs increases. The time spent in *Receive* reaches an almost constant percentage for increasing P. This is due to the fact that as the LP gets more delayed in the *Generate & Send Sources* part, a longer time is given to the LPs (in fact to PVM) to receive the slice data from other LPs and hence, then the *receive_slice* (in line 14) returns almost instantly since the received data is already local to the workstation and hence, *receive_slice* does not block as often. At the same time, the volume of received data increases with larger P, and hence the two trends balance out at an almost constant percentage, in a much similar fashion as it occurs in the *Generate & Send Sources* step.

A good example of how the computation is penalized due to the scaling to a larger P, is given by the *Generate Departures* step. The average time spent here is constant (subject to the slice length) independent of P. Hence, its decrease demonstrates the gradual increase of overheads related to the scaling of the simulation to a larger P. However, the combined effect of the *Generate & Send Sources* and the *Receive* steps, results in an eventually constant percentage of time covered by the *Generate Departures* step. The only significant remaining portion of time is the *Wait for Fixup* step. This step is largely unrelated to the message–passing overheads (the data conveyed is very small, only a queue size), but rather to the nature of the time–parallel algorithm.

Note that the time spent waiting for the queue size in the *Wait for Fixup* is less than 10 % of the total time in all configurations. Hence, the overhead due to the time–parallel nature of the simulation is small. In fact, the average time waiting for fixup decreases slightly as the number of workstations increases although its variance increases depending on the different processing speeds of the individual workstations. A fast workstation will complete the *Generate and Send Sources* stage faster but will have to wait in the *Receive* step for longer. Similarly, it will complete the *Generate Departures* faster, but will have to wait longer in the *Wait for Fixup* step. Summarizing, the algorithm deals with load imbalances at the cost of performance due to the synchronization points between processors at the blocking receive operations. To cancel out effects where a certain run for P workstations was performed with the faster workstations and a run at $P + 1$ with slower ones, the set of workstations used at $P + 1$ is the exact same set as the ones used in the run for P plus one additional (new) workstation.

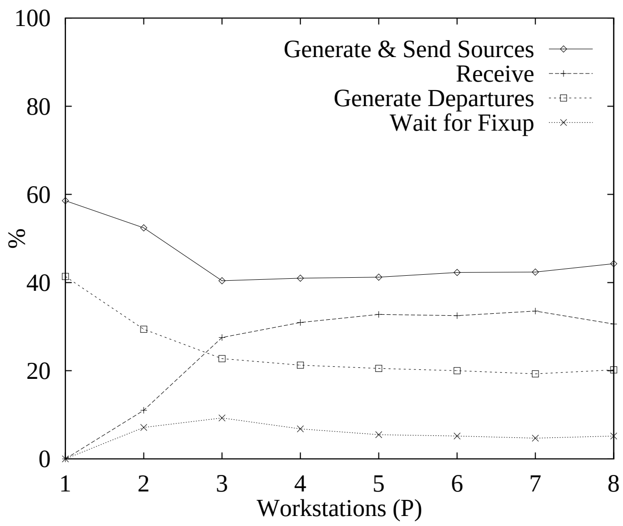

Figure 5: Execution Time Profile (slice=25sec).

Figure 7 compares the percentage of time spent in the *Generate and Send Sources* for the two different slice lengths. The difference is evident in the transient around $P = 2$, where the shorter slice presents a more sudden drop in its percentage. Note that as long as $P = 1$ the two different slice lengths should cause no difference in performance, as indeed is the

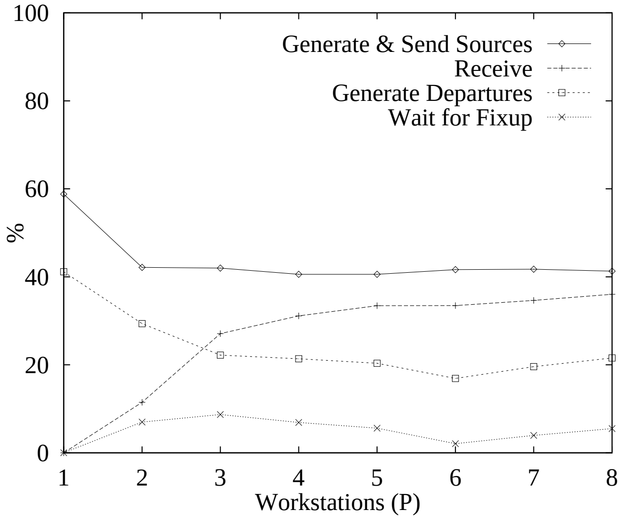

Figure 6: Execution Time Profile (slice=12.5sec).

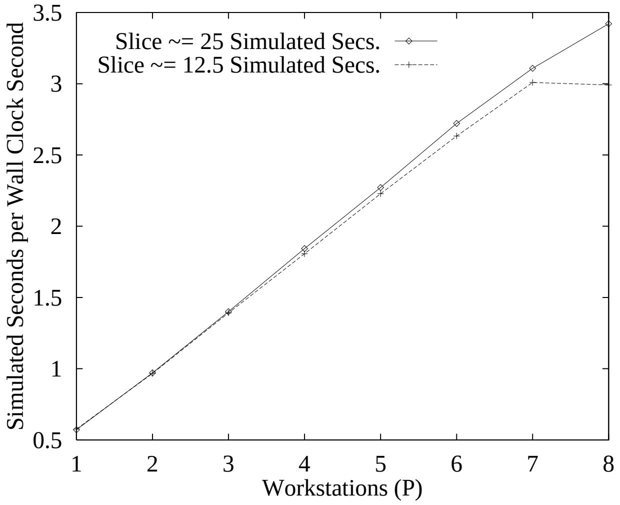

Figure 8: Simulation Speed in Simulated Seconds.

case. However, when $P > 1$, the shorter the slice, the more frequent the communication. Hence, the contention brought to the communication medium and the overheads due to the frequency of the send operations are more intense for shorter slices. Note that the volume of data sent are the same in both slice sizes, but the smaller size sends them by calling more frequently (in the examples: twice as frequently) the send operation. Eventually, the difference due to the different slice lengths becomes less significant as increasingly more data are sent over the network, i.e., for higher P.

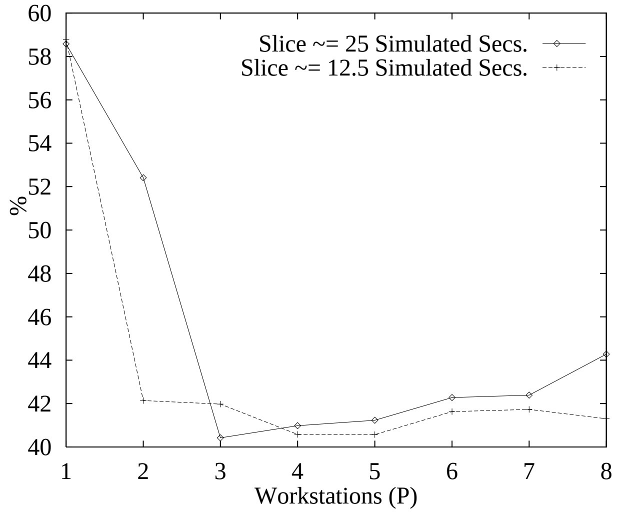

Figure 7: Time Spent in *Generate & Send Sources*.

Figure 8 captures the benefit of time–parallel simulation. That is, the almost linear speedup to the number of used processors, even when the overheads due to communication contention are considered. The waiting time for a fixup value as well as the fixup operation are not the dominant overheads. The speedup for different slice lengths may be very similar but the shorter slices can rapidly reach the point of diminishing returns as the shape of the line for a slice length of 12.5 seconds illustrates (between 7 and 8 workstations). The linear behavior is lost sooner or later for increasing P, but we have systematically produced good results following the linear speedup for a small aggregation of workstations (typically a dozen or less of them).

Figure 8 presents the simulation speed in seconds of simulated operation of a 155 Mbps ATM link. It is easy to see that the technique enables the generation of self–similar traffic faster than real–time. That is, for any $P > 2$, one second of link activity of a 155 Mbps ATM link can be generated in less than a second of wall clock simulation time. Furthermore, the fact that the simulation speed reaches almost four times the speed of the 155 Mbps link (in fact, 3.5) indicates that the technique can be safely used with minor modifications for the generation of real–time ATM link workloads of even 620 Mbps links. Considering that the reported performance is achieved using commodity workstations and low speed (10 Mbps) networking infrastructure, it is safe to assume that it can give even better results on a multiprocessor system for an embedded ATM link testing device.

5 CONCLUSIONS

We have presented a time–parallel technique for the fast parallel generation of self–similar traffic. A message–passing implementation of the algorithm over a cluster of scientific workstations communicating over a 10 Mbps Ethernet shows that it is possible to generate self–similar traffic for 155 Mbps or faster ATM links in real–time by utilizing less than a dozen workstations. The property of real-time generation is of particular importance for use by equipment (ATM switches in particular) manufacturing companies. To our knowledge, currently, no commercial ATM testing product can supply continuous real–time self–similar traffic. The work detailed in this paper targets at solving this problem with the aid of small scale parallel processing using commodity workstations attached on a general purpose Ethernet.

The paper also serves as a case study for implementing time–parallel simulation techniques on a loosely coupled network of scientific workstations. It illustrates that even with a limited communication bandwidth, it is possible to achieve a speedup almost linear to the number of utilized workstations when the fixup stage of the algorithm is computationally inexpensive. Indeed, in the tests we have conducted the linear speedup is only impaired by the communication overhead inherent in any message–passing scheme and *not* by the fixup phase of the algorithm. The results are particularly impressive when compared to previous techniques that were distinctly inefficient due to the fact that their computational complexity was not linear to the number of samples generated.

Future work includes the extension of the technique to fit both LRD and SRD artifacts in the produced traffic. Another objective is the simulation of ATM multiplexers fed by several self–similar streams in order to study the effects of multiplexing self–similar processes with different Hurst parameters.

REFERENCES

Garrett, M. W. and W. Willinger. 1994. Analysis, Modeling and Generation of Self–Similar VBR Video Traffic. In *Proceedings of ACM SIGCOMM 94*. 269–280.

Hoskings, J. R. M. 1984. Modeling Persistence in Hydrological Time Series Using Fractional Differencing. *Water Resources Research* 20:1898–1908.

Huang, C., M. Devetsikiotis, I. Lambadaris and A. R. Kaye. 1995. Modeling and Simulation of Self–Similar Variable Bit Rate Compressed Video. In *Proceedings of ACM SIGCOMM 95*. 114–125.

Leland, W. E., M. S. Taqqu, W. Willinger and D. V. Wilson. 1994. On the Self–Similar Nature of Ethernet Traffic (Extended Version). *IEEE/ACM Transactions on Networking* 2:1–15.

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

Nikolaidis, I., R. M. Fujimoto and C. A. Cooper. 1994 Time–Parallel Simulation of Cascaded Statistical Multiplexers. In *Proceedings of ACM SIGMETRICS 94*. 231–240.

Paxson V. and S. Floyd. 1994. Wide–Area Traffic: The Failure of Poisson Modeling. In *Proceedings of ACM SIGCOMM 94*. 2576–268.

Paxson, V. 1995. Fast Approximation of Self–Similar Network Traffic. Technical Report LBL–36750, Lawrence Berkeley Laboratories.

Willinger, W., M. S. Taqqu, R. Sherman and D. V. Wilson. 1995. Self–Similarity Through High–Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level. In *Proceedings of ACM SIGCOMM 95*, 100-113.

AUTHOR BIOGRAPHIES

IOANIS NIKOLAIDIS is an Assistant Professor with the Computing Science Department at the University of Alberta, in Edmonton, Canada. His research interests are parallel simulation and network performance.

C. ANTHONY COOPER is a member of technical staff in Lucent. Previously he has worked for Bell Communications Research (Bellcore). His interests are in the performance and standards for fast packet switching and ATM networks.

KALYAN S. PERUMALLA is a Research Scientist and a Ph.D. student at Georgia Tech. His interests are in parallel and distributed simulation and in network modeling. He has developed the TeD network modeling language.

RICHARD M. FUJIMOTO is a Professor with the College of Computing at the Georgia Institute of Technology. He has been working in the field of parallel and distributed simulation over the last 14 years.