pfs: Parallelized, Flow-based Network Simulation

Mukta Gupta[†], Ramakrishnan Durairajan[†], Meenakshi Syamkumar[†], Paul Barford[†+], Joel Sommers[*]
{mgupta,rkrish,ms,pb}@cs.wisc.edu, jsommers@colgate.edu

[†]Department of Computer Sciences
University of Wisconsin - Madison
1210 W. Dayton St.
Madison, WI 53706, USA.

[+]comScore, Inc.
11950 Democracy Drive
Suite 600
Reston, VA 20190, USA.

[*]Department of Computer Science
Colgate University
306 McGregory Hall
Hamilton, NY 13346, USA.

***Abstract*—Simulation is a compelling option for evaluating Internet protocols, configurations and behaviors. While current simulation tools have been used effectively to consider questions in small-scale networks, they are incapable of evaluating large scale phenomena such as routing configurations, DDoS attacks and data center deployments. In this paper, we describe *pfs*, a parallelized version of the *fs* flow-level simulator [1] that offers the opportunity to conduct very large-scale simulations of networks. Our approach to parallelization is based on decomposing simulation configurations both spatially and temporally into independent chunks that can be run simultaneously on massively scalable, parallel processing infrastructures. We demonstrate the capabilities of *pfs* through a series of experiments that highlight both the speedup that can be achieved as well as the costs that are incurred in terms of the accuracy of the simulation results.**

***Keywords*—*fs*, *parallelization*, *very large topology*, *flow-based simulation*.**

I. Introduction

The ability to thoroughly test and evaluate new Internet systems, protocols and configurations is a key component of the design, development and deployment processes. Standard requirements for testing include control and repeatability, realism, efficiency (in terms of time required to specify and conduct tests), visibility (in terms of being able to collect data about the outcome of tests) and the ability to test at an appropriate scale. Unfortunately no single test method can satisfy all of these requirements, thus a combined approach is typically taken.

Standard methods for test and evaluation of networked systems include: *analytic modeling*, which is typically used to examine idealized or asymptotic behaviors and is efficient, but generally lacks realism and detail; *testbed-based evaluation*, which is often used to test the details of prototype implementations and offers control, repeatability, visibility and a good measure of realism but lacks the ability to conduct tests at scale; *in-situ evaluation*, which is typically used to test more complete implementations and offers high realism but lacks visibility, control and repeatability. *Simulation* offers a compelling opportunity to complement other methods by enabling tests to be conducted in an efficient, visible, realistic and repeatable fashion.

While simulation has been widely used in prior research efforts (*e.g.,* the development of many variants of TCP), standard network simulators like ns-2 [2] and its more recent variants have well known, inherent limitations. In particular, ns-2 was developed to capture the low-level packet interactions that lead to congestion on end-to-end paths. While this level of detail is critical for understanding congestion, it requires a simulation engine that must operate at fine timescales. As a consequence, the high computational demands of ns-2 precludes its use in evaluations of large scale network phenomena such as routing configurations, denial of service attacks or network service deployments.

In prior work, the *fs* [1] simulator was developed to enable test and evaluation of large networks. Instead of focusing on packet-level behavior, *fs* is a flow-based simulator that produces results consistent with ns-2 down to single second aggregations. By focusing on higher-level behavior, *fs* achieves significant reduction in processing overhead, enabling a wide range of tests that are beyond the capability of packet-oriented discrete event simulators such as ns-2. However, the current implementation of *fs* is limited to *a single system*, which makes it inappropriate for testing and evaluating very large networks such as large service providers, data centers, ensembles of networks, or large scale events.

In this paper, we describe *pfs*, a parallelized version of *fs* developed to test and evaluate large networks and large network events. The deployment target for *pfs* is any massively scalable, parallel processing computational environments like Hadoop [3], HTCondor [4], or Spark [5], each of which enable large data sets to be efficiently processed in parallel. The challenge in this work is to develop methods for decomposing *fs* simulations in a way that enables parallel processing while preserving accuracy in results.

We developed three different methods for decomposing *fs* simulations for parallel evaluation. *Temporal* decomposition divides the simulation into separate time chunks, which can be run in parallel. *Spatial* decomposition divides the simulation topology into separate chunks, which can be run in parallel. *Spatio-temporal* decomposition, as the name suggests, combines the other two methods and thus may offer the best opportunity for speedup.

The key to parallelization in *pfs* is refactoring the traffic generation process to ensure that the conditions in each chunk are as close as possible to a serial simulation, which we refer to as the baseline, or simply base, case. Refactoring is done in a preprocessing function that establishes traffic and topological pre-conditions for each chunk. This enables each chunk to be run independently in massively-parallel computing infrastructures. By focusing on traffic effects at chunk boundaries, *pfs* is able to maintain accuracy with minimal loss versus the base case. This design choice is supported by the fact that many classes of applications, including traffic engineering, rate control and streaming, are tolerant to minimal loss.

We demonstrate the capability of *pfs* in a series of case studies. We begin with a simple network topology that we use to show the tradeoffs between speedup and accuracy for temporal, spatial and spatio-temporal decomposition. Our results show that simulation times can be reduced by several orders of magnitude at the cost of some reduction in accuracy when spatio-temporal parallelization is used. We then demonstrate speedup and accuracy on a larger network topology. Using spatio-temporal parallelization, the results show that simulation times can be reduced by nearly two orders of magnitude.

The remainder of this paper is organized as follows. In Section II, we provide an overview of the *fs* simulator. In Section III, we describe the details of the *pfs* implementation. The case study evaluations of temporal, spatial and spatio-temporal parallelization in *pfs* are described in Section IV. We describe related studies in Section V. We summarize, conclude and discuss future directions for our work in Section VI.

II. *fs* Overview

fs is a Python-based system developed for the purpose of generating network flow records and interface counters à la SNMP [1]. Although it was not originally designed for simulating network activity, it uses discrete-event simulation techniques for synthesizing the network measurements that it produces. We illustrated in our initial work on *fs* that it not only generates measurements extremely fast compared with identical setups in the ns-2 [2] packet-level simulator, but that the measurements it produces are accurate down to 1-second timescales. More recently, we extended *fs* to support simulation and debugging of software-defined networking applications [6], [7].

fs is designed with four key considerations in mind. First is the goal to generate representative network measurements similar to those that can be collected from operational routers today. In particular, *fs* generates flow export records (*e.g.*, Cisco Netflow records [8]) and SNMP-like counters (*e.g.*, packet and byte counters from router interfaces). In addition to exporting commonly-used measurements, *fs* employs a familiar and easy-to-use method of configuration. In particular, *fs* uses a declarative configuration style using a syntax based on Graphviz DOT files [9].

The second goal is to ensure sufficient realism in the measurements that *fs* generates. *fs* is designed to generate measurements from *benign* flows as well as particular types of anomalous flows. For benign flows, *fs* builds on the Harpoon model for traffic generation [10]. Similar to Harpoon, *fs* creates flows between a given source and destination that have particular distributional properties. Namely, flows are initiated between a source and destination according to one distribution, and flow sizes are drawn from another distribution. It further leverages existing TCP throughput models (*i.e.,* [11] and [12]) to simulate individual TCP flows. More generally, *fs* includes the capability to generate a broad range of simulated traffic conditions through its flexible configuration format.

The third goal of *fs* is to scale to large network configurations—not only to generate measurements quickly, but to use modest memory resources while doing so. Because the kinds of measurements that *fs* can generate do not contain fine-grained information (*e.g.*, packet-level timings), we ignore many packet-level details. This design decision results in major computational and memory savings while generating realistic data over time scales of 1 second and longer, as shown below.

The main reason why *fs* exhibits good scaling properties has to do with the fact that the key network abstraction it operates on is not the packet, but a higher-level notion called a *flowlet*. A flowlet refers to the volume of a flow emitted over a given time period, *e.g.,* 100 milliseconds, which may be 1 or more packets. By raising the level of abstraction and thus the entity around which most simulator events revolve, *fs* achieves much higher speed and efficiency than existing packet-level simulators, like ns-2 [2] and ns-3 [13]. *fs*'s better scaling properties are particularly relevant to this work, since our longer-term goal is to scale to networks with millions of nodes [1]. In this work, we greatly extend the scalability of *fs* by adapting it to run in a parallelized cluster setting.

The fourth goal of *fs* is to enable prototyping and evaluating new SDN-based applications accurately, at large scale, and in a way that enables incorporation of real SDN controllers and applications. The SDN extensions to *fs* seamlessly allow use of any standard Openflow controller, and include switch components that can be controlled and configured through the Openflow protocol. As a result, controller components developed for standard SDN platforms can be used *directly* and without modification in *fs*.

III. Parallelizing *fs*

In this section we provide an overview of the objectives, challenges and approaches to parallelize *fs*, and discuss the design, implementation, and features of *pfs*

A. Design Objective

The main objective of *pfs* is to enhance the scalability of *fs* by *temporal*, *spatial* and *spatio-temporal* parallelization of simulations. Temporal parallelization splits network simulations into multiple smaller simulation chunks based on time. Spatial parallelization splits network simulations into multiple smaller simulation chunks based on topology. Spatio-temporal parallelization is a combination of both temporal and spatial parallelizations. Using *pfs*, network simulations can be run on any parallel processing infrastructure by splitting the simulations into various temporal, spatial or spatio-temporal chunks. Our main objective is to significantly enhance speedup through parallelization with minimal loss in accuracy.

B. Parallelization in pfs

pfs consists of two components: the *parallelization unit* and *cluster manager*. The overall architecture is illustrated in Figure 1, and we describe each part below.

The parallelization unit consists of algorithms (described below) for temporal, spatial and spatio-temporal decomposition and is implemented as a pre-processing step in simulation execution. This pre-processing step in *pfs* takes a standard *fs* configuration file (known as a *scenario* file) as input and divides the traffic generation specifications temporally

[1] We believe that goal is entirely feasible although it will require supporting systems that enable representative network configurations to be generated. Such capabilities are the focus of future work.

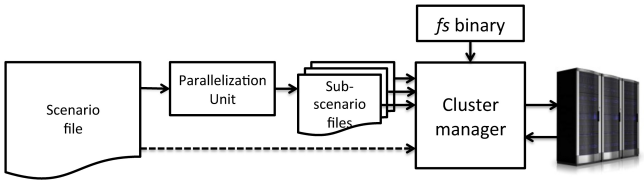

Fig. 1: Architecture of *pfs*.

and/or spatially, thus producing a new set of configuration files (known as *sub-scenario* files). These new configuration files describe independent temporal or spatial *chunks* of the original simulation, and can be run in parallel since they have no dependencies on each other. The original simulation is divided in such a way that the network measurements generated through each chunk can be effectively aggregated to give a complete and accurate picture of the full network. The parallelization unit is implemented as a lightweight extension of the *fs* simulator in approximately 600 lines of Python code.

The cluster manager acts as a simulation coordinator and has the twin goals of *(i)* scheduling scenario and sub-scenario files for execution, and *(ii)* merging the simulation results. First, to enable parallel simulations, the cluster manager packs individual sub-scenario files with *fs* binaries (created using `cx_Freeze` [14]) and creates platform-agnostic executables. These executables can be directly executed in a variety of parallel processing environments like Hadoop, HTCondor, and Spark. In our evaluation, we leverage HTCondor clusters. Next, the aggregation modules in the cluster manager keeps track of sub-scenario simulation statistics and merge them at the end. The cluster manager is implemented in approximately 200 lines of shell script.

C. Temporal Parallelization

Algorithm 1 shows the key steps of Temporal Parallelization. The inputs to the algorithm are the original *scenario* file, the amount of time to simulate, and the desired number of temporal simulation chunks. Pre-processing (steps 1 to 11) for temporal parallelization configures the traffic modulator to generate flows. The traffic generator component of *fs* is run in temporal pre-processing (steps 2 and 3), however, flows are not actually generated. Instead, start times for each flow created by each traffic generator in a given topology are stored (step 4), where the start time is taken as the current simulator time. When all flow start times have been captured, they are divided into different temporal chunks. Flows are segregated into a particular temporal chunk depending upon start time (step 6 and 7). For instance, if a simulation is to be run for T seconds and the number of temporal chunks is n, then flows starting from 0 to T/n are put in chunk one and flows starting after T/n or before 2*T/n are put into second chunk, and so on. After flows are distributed across n temporal chunks, simulations are run in parallel across different chunks (steps 10 to 12).

D. Spatial Parallelization

Algorithm 2 shows the key steps of Spatial Parallelization. Pre-processing (steps 1 to 15) for spatial parallelization involves splitting the original network topology into various sub-topologies, which can be run in parallel. The inputs to the algorithm are the original *scenario* file, simulation time,

Algorithm 1: Algorithm for Temporal Parallelization

```
   input: scenario = original config file
   input: simTime = simulation time
   input: n_chunk = number of temporal chunks
1  preStart = Pre-processing start time;
   // Simulator is run in temporal
      pre-processing mode to generate flows
      tagged with flow start times
2  sim = Simulator('temporal_preprocessing', scenario,
   simTime, NULL);
3  sim.run();
4  flows = getTimes(sim.flows);
5  clusters = initialize temporal clusters;
6  foreach cnt in n_chunk do
7      foreach flow in flows do
8          if flow.startTime lies in the current temporal
           cluster then
9              clusters[cnt].append(flow)
10 preEnd = Pre-processing end time;
11 record 'preEnd – preStart' as pre-processing time;

   // Simulator is run on pre-processed temporal
      chunks in parallel
12 foreach count in n_chunk do
13     sim = Simulator('temporal', scenario,
       simTime/n_chunk, cluster[count]);
14     sim.run();
```

and the number of desired spatial simulation chunks. Similar to temporal parallelization, traffic generators are run in spatial pre-processing mode (steps 3 and 4). However, in this case, the focus is on identifying source-destination pairs for each flow. In particular, each flow is tagged with the hop-by-hop path that each flow would have taken in order to reach their respective destination nodes. If the number of desired spatial simulation chunks is greater than the total number of flows generated by the simulator, then we assign each source-destination pair to a separate sub-scenario and run those sub-scenarios in separate chunks (step 5) *i.e.,* one flow per source-destination pair. Otherwise, source-destination pairs are uniformly distributed among various spatial clusters (steps 7 to 9). Once we have flows segregated into different spatial clusters, sub-scenario files are created for them (steps 10 to 13). We parse the original input configuration file and copy all the nodes and links associated with flows present in the current spatial cluster into the sub scenario file associated with it. Finally, simulations are run in parallel on different spatial chunks with their respective sub-scenario file as the input configuration file (steps 15 to 17).

E. Spatio-Temporal Parallelization

Spatio-Temporal Parallelization is a combination of both temporal and spatial parallelizations described above. First, the simulations are spatially parallelized, that is, they are divided into multiple simulation chunks based on topology using Algorithm 2. Next, these spatially parallelized chunks are further temporally parallelized using Algorithm 1, that is,

Algorithm 2: Algorithm for Spatial Parallelization

```
   input: scenario = original config file
   input: simTime = simulation time
   input: n_chunk = number of spatial chunks
 1 clusters = Initialize spatial clusters;
 2 preStart = Pre-processing start time;
   // Simulator is run in spatial pre-processing
      mode to generate flows per SD pair
 3 sim = Simulator('spatial_preprocessing', scenario,
   simTime);
 4 sim.run();
   // If number of chunks > total flows, assign
      each SD pair into a separate sub-scenario
 5 sp_chunk = min(len(sim.flows), n_chunk);
   // Uniformly distribute SD pairs across
      spatial clusters
 6 cnt = Initialize cnt to 0;
 7 foreach flow in sim.flows do
 8 |   clusters[cnt%sp_chunk].append(sim.flows[flow]);
 9 |   cnt = cnt + 1;
10 foreach i in sp_chunk do
11 |   f = Create sub-scenario file;
12 |   foreach flow in clusters[i] do
13 |   |   Get associated nodes and links and write into f;
14 preEnd = Pre-processing end time;
15 record 'preEnd – preStart' as pre-processing time;

   // Simulator is run on pre-processed spatial
      chunks in parallel
16 foreach count in sp_chunk do
17 |   sim = Simulator('spatial', sub_scenario(count),
   |   simTime);
18 |   sim.run();
```

they are sub-divided into even smaller simulation chunks based on flow start times.

F. Parallelization Challenges

Central to the issue of parallelization is how to maintain close correspondence between the parallelized simulation and the baseline serialized simulation. The specific problem that arises is discontinuity of traffic flows at boundaries of the spatial and temporal chunks. We considered multiple methods for attempting to maintain exact consistency across boundaries. These techniques either significantly increased preprocessing time or required dependencies between chunks. We ultimately decided to sacrifice aspects of consistency across boundaries (and therefore accuracy) in order to maximize speedup. We argue that this design choice is supported by the fact that *pfs* is focused on large scale simulations where behaviors such as individual congestion events are of less importance. Many classes of applications are tolerant to a modest reduction in the accuracy in simulation results including traffic engineering, rate control, data analytics, and image/audio/video streaming [15].

The main challenge in *temporal* parallelization of *fs* is the ability to identify flows that will be active at the chunk boundaries and flow start times at different times during the simulation. For the latter, we have added the capability in *pfs* to append timestamps with each flow generated during the pre-processing phase. For the former, we estimate the flows which will be active at the chunk boundary by examining the flow distribution and number of temporal chunks selected by the user. To account for flows that would have been active in the original *fs* simulation, flows are initiated at the start of each temporal chunk. We also need to estimate various network states at different points throughout the simulation. This state information includes *(1)* size of input queue buffer, *(2)* flowlet arrival times at various nodes and links in the network topology, and *(3)* size of the flowlets. All these are estimated in pre-processing prior to running the simulations on temporal chunks.

The main challenge in *spatial* parallelization is to accurately estimate the load imposed by flows on shared links. A link is considered *shared* if it is traversed by more than one flow. In order to run the simulations on a spatial chunk, we need to identify and account for the load on shared links caused by flows from other sub-scenarios. To do this we need to isolate the hop-by-hop path of flows such that there is no interference of flows from one sub-scenario on flows in other sub-scenario. We also need a method for clustering paths from simulation configurations with potentially millions of source-destination pairs.

IV. Evaluation

In this section, we describe a series of tests that demonstrate the performance of *pfs*. Our focus is on highlighting speedup and accuracy versus baseline for the three parallelization methods described in Section III. We begin by conducting tests on a simple network configuration, followed by tests on a more complex configuration and conclude with tests on large and very large topologies.

A. Simulation Methodology

Since simulation performance is directly tied to the complexity of the configuration, we initially use a simple network configuration to elucidate the basic aspects of speedup and accuracy in *pfs*. Results from the simple configuration should thus be considered conservative. Experiments with a more complex configuration are designed to demonstrate what might be achieved by *pfs* in a more typical configuration. We conclude with tests on large and very large topologies to demonstrate the scalability of *pfs*.

The simple configuration is a dumbbell network topology shown in Figure 2. In this topology, nodes A and E are sources *i.e.,* traffic is sent from these nodes, nodes D and F are destinations *i.e.,* traffic terminates at these nodes, and B-C is the link that is shared by all traffic in the baseline case. We configure the Harpoon traffic generators to start at time 0 and to have a modulation profile such that 10 sources will be active for 60 seconds, then 20 nodes will be active for 60 seconds, then 30 for 120 seconds, 20 for 60 seconds, 10 for 60 seconds, and to then repeat the same pattern. While a Harpoon generator is active, a new flow is started after a time duration chosen from an exponential distribution with λ equal to 100, and a random flowsize chosen from Pareto distribution, with *offset* as 10000 and α equal to 1.2.

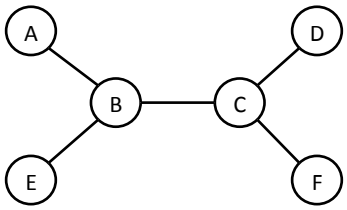

Fig. 2: Simple network topology used to test *pfs*.

For the large and very large configurations, we use two topologies: an extension of the dumbbell topology from the simple configuration (above) and new linear topologies with a sizable number of nodes. For each of these n-node configurations, we configure the Harpoon traffic generator to start at time 0. To have a modulation profile such that n/3 sources and n/3 destinations are always active, a new flow is started after a time duration chosen from exponential distribution with λ equal to 150, and a random flowsize chosen similar to the simple configuration.

In each test, we record the wall clock time for the baseline case *i.e.,* serial simulation in *fs*. We also record wall clock time for preprocessing and simulation in *pfs*. Simulations for simple and complex configurations (for both *fs* and *pfs*) are run on a quad-core Intel-based machine with 2.66 GHz clock speed and 8 GB main memory. For larger topologies (Section IV-F and IV-G), simulations are run on HTCondor clusters comprised of heterogeneous machines where each scenario and sub-scenario files, along with the *fs* binary, are scheduled on the next available CPU in the HTCondor cluster environment by the cluster manager.

To assess the accuracy of the simulations, we measure the Root Mean Square Error (*RMSE*) for the data (bytes) received at destination nodes. RMSE compares the baseline with *pfs*. Specifically, we calculate the total bytes received per second by all the destination nodes over the entire simulation time (T). Then, we calculate the RMSE at each simulation second using the formula:

$$Root\ Mean\ Square\ Error = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Pfs(KBs)_i - fs(KBs)_i)^2}$$

where n is equal to five, *i.e.,* the number of simulation runs. Once, we have *RMSE* of data received (in KBs) per second by all the destination nodes in the topology for the simulation time (T), we calculated the average *RMSE* by dividing *RMSE*, obtained as above, by T. While it could be argued that other metrics should also be considered (*e.g.,* flows-per-second), we argue that bytes received is a good metric since bytes-per-second are often used in research and in operations to assess network state and behavior.

B. Temporal Parallelization Results

To evaluate temporal parallelization using the simple topology, we conduct tests for 600 simulated seconds. We assess speedup and accuracy by dividing the simulation configuration into temporal chunks of size 2, 4, 8, 16, 30 and 60. Figure 3 shows time series graphs for total number of bytes received (per second) for 2 (left) and 60 (right) temporal chunks.

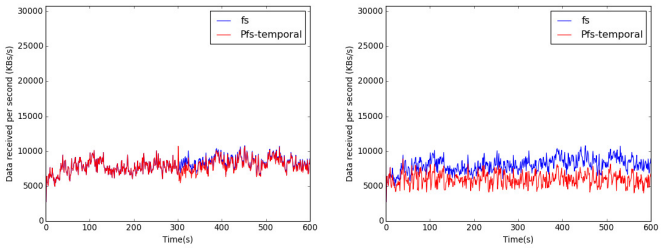

Fig. 3: Data time series graphs for dumbbell topology using temporal parallelization for 2 (left) and 60 (right) temporal chunks.

The figure shows that in the 2 chunk case, *pfs* achieves a high level of correlation with the baseline simulation run without parallelization. There is a decrease in correlation at the chunk boundaries because of our our approach for flow time estimation. The graph for the 60 chunk test shows that while the general characteristics remain similar, there is more pronounced divergence between *pfs* and the baseline.

Figure 4 (left) shows the speedup achieved for different levels of temporal parallelizations. We can see that the preprocessing time remains fairly constant for different temporal chunks and simulation time can be decreased by about 85% when 60 temporal chunks are used. The accuracy achieved by *pfs* is depicted in Figure 4 (right). The figure shows how RMSE values increase as the number of temporal chunks grows. Selecting the best position in the design space is clearly dependent on user requirements and the details of individual tests. However, these results suggest that meaningful speedups and high accuracy can be achieved via temporal parallelization using a small number of chunks.

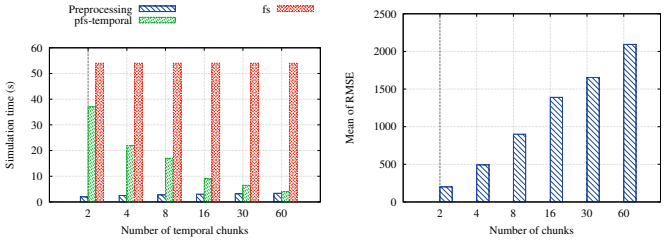

Fig. 4: Speedup (left) and Accuracy (right) achieved by temporal parallelization for different temporal chunks for dumbbell topology.

C. Spatial Parallelization Results

To evaluate spatial parallelization using the simple topology, we conduct tests for 100 simulated seconds. In these tests, traffic load is generated in such a way that there is no congestion on the shared bottleneck link B-C. Figure 5 shows the data time series graphs for total data (in KBps) received by destination nodes D (left) and F (right) after running simulations on *pfs* and *fs*. The graph shows that there is perfect correlation in the traffic time series. When we increase the traffic load, correlation decreased somewhat, but still remained quite high (graphs omitted due to space constraints).

To further assess spatial parallelization, we extended the dumbbell topology in Figure 2 to include eight source-destination nodes. We then ran simulations over a period

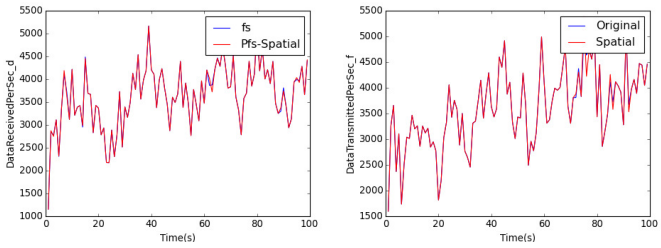

Fig. 5: Time series graphs of data transmitted in the dumbbell topology with two source-destination pairs using spatial parallelization. Data received (in KBps) by destination nodes D (left) and F (right).

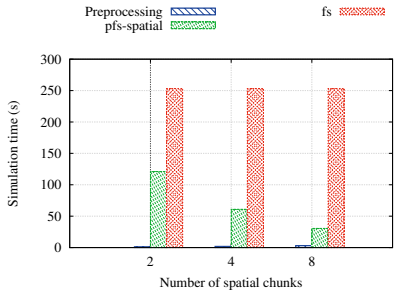

Fig. 6: Speedup achieved by Spatial Parallelization for different spatial chunks for dumbbell topology 8 source-destination pairs

of 600 simulated seconds. Figure 6 shows speedup achieved for different numbers of spatial chunks. The figure shows that preprocessing time is constant and extremely small for different degrees of spatial parallelization. The reason that preprocessing is so small is that analyzing S/D pairs for traffic generation and subsequent spatial decomposition of the topology is simple and efficient. The figure also shows that the simulation time decreases by up to 90% when 8 spatial chunks are used, with essentially negligible loss in accuracy.

D. Spatio-Temporal Parallelization Results

Next, we evaluate the speedup and accuracy achieved by combining the spatial and temporal parallelization. We ran the simulations for 600 seconds for the extended dumbbell eight source destination pairs. Since spatial parallelization achieves maximum speedup when the number spatial chunks is equal to the number of source destination pairs in the topology, as shown above, we use the number of spatial chunks to be equal to 8. We ran the temporal parallelization preprocessor on each of those chunks and varied the number of temporal chunks (2, 4 and 8).

Figure 7 (left) shows the speedup achieved by using 8 spatial chunks and different numbers of temporal chunks. This further shows that significant speedups can be achieved—reducing simulation times by two orders of magnitude. Figure 7 (right) depicts the RMSE values of the data received by the destination nodes in the spatio-temporal tests. We can tune the number of spatial and temporal chunks to achieve desired level of speedup and accuracy. If we increase the number of spatial chunks to be more than the number of source destination pairs in the given input topology, then there is no further benefit of spatial parallelization. We can achieve

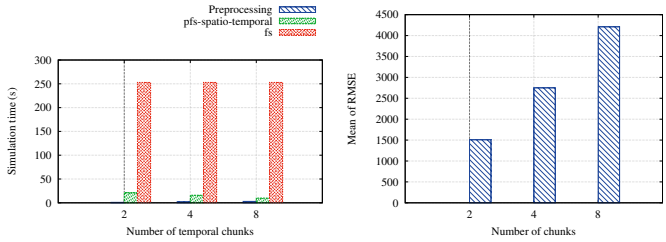

Fig. 7: Speedup (left) and Accuracy (right) achieved by spatio-temporal parallelization for different temporal chunks for dumbbell topology with 8 source-destination pairs.

more speedup by increasing the temporal chunks in that case, but that comes at a cost of loss in accuracy. When there is no congestion, we suggest that if the number of S/D pairs is relatively small, maximum speedup can be achieved without any loss of accuracy by selecting the number of spatial chunks to be equal to the number of flows. After that, the number of temporal chunks can be tuned to achieve further speedup, but at the cost of accuracy.

Practical use of spatio-temporal parallelization depends on the details of the target simulation. Specifically, selecting the number of spatial chunks depends on the number of source-destination pairs. Likewise, selecting the number of temporal chunks depends on simulation length and the characteristics of baseline traffic profiles. In each case, the user must consider the decrease in accuracy that results from increased parallelization. This suggests an iterative process where RMSE is compared across configurations.

E. Parallelization Results for Complex Topology

We conducted tests on a larger network topology to provide further perspective on *pfs*. We use NTT's network topology (obtained from Internet Topology Zoo [16]) as the starting point for these tests. NTT's network consists of 45 nodes and 216 edges. We randomly generate flows across 7 source-destination pairs in the topology. We ran the simulations for 600 seconds using seven spatial chunks and varying the number of temporal chunks (2, 4 and 8).

Figure 8 (left) show the speedup achieved by *pfs* for the complex topology. The figure shows that significant speedups are possible. The accuracy of *pfs* versus the baseline is shown in Figure 8 (right). Similar to results reported above, accuracy is inversely and speedup is directly proportional to the number of temporal chunks. We also observe some degradation in accuracy due to an increased number of shared links with the larger topology, an issue we intend to investigate in future work.

F. Parallelization Results for Large Topologies

Next, we evaluate the speedup achieved by spatial, temporal and spatio-temporal parallelizations for larger configurations. We ran the simulations for 100 seconds for both dumbbell and linear topologies by increasing the number of nodes in logarithmic scale.

Figure 9 shows the speedup achieved by *pfs* for large dumbbell (left) and linear (right) topologies with varying

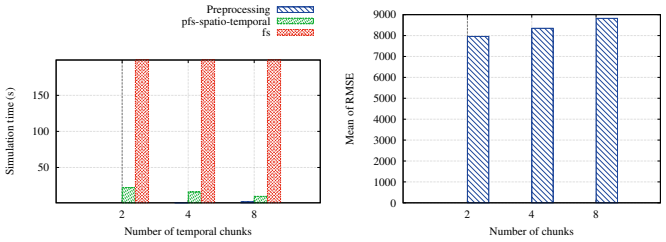

Fig. 8: Speedup (left) and Accuracy (right) achieved by spatio-temporal parallelization for different temporal chunks for simulations of the NTT network topology.

number of nodes. In both topologies, all three decompositions show an order of magnitude speedup. In particular, the spatial and spatio-temporal decompositions are significantly faster than the base with an order of magnitude fewer nodes. We hypothesize that since multiple flows, *e.g.*, in congestion scenarios, potentially end up in the same temporal chunk, temporal parallelization can incur additional simulation time compared to the other two decompositions.

G. Parallelization Results for Very Large Topologies

Finally, we test the speedup achieved by all three parallelizations by scaling the number of nodes further in the large configuration. For these very large topologies, which are atypical in the Internet [17], our simulation goal is to demonstrate the scalability and successful completion of *pfs* for sizable number of nodes. To that end, we ran tests for 100 seconds for both dumbbell and linear 10K topologies, and Table I shows the completion times achieved by *pfs* for the different decompositions. While the original scenario in the very large configuration did not finish, the parallelized sub-scenarios completed in reasonable time demonstrating the scalability of *pfs*.

	Spatial	**Temporal**	**Spatio-temporal**
Dumbbell	2610.62	6862.46	1704.90
Linear	9614.38	20151.03	8679.53

TABLE I: Completion time (in seconds) for 10K dumbbell and linear topologies achieved by *pfs* for spatial, temporal and spatio-temporal parallelizations.

V. Related Work

Developing techniques to improve the execution speed of network simulations has been an area of ongoing research for quite some time. One set of approaches has focused on network simulation at a higher level of abstraction than the packet such as a network *flow*. In this vein, one approach is to use fluid-flow models to simulate aggregate network flow behavior, *e.g.*, [18]. This technique has been recently used in large-scale data center networking studies (*e.g.*, [19], [20]). Two key limitations of fluid-flow models are *(1)* that they assume unrealistically infinite end-to-end flows, and *(2)* that for them to be most scalable, they must operate an open-loop manner. Indeed, Liu *et al.* found that in congested network scenarios, fluid-flow simulation can be *more expensive* than packet-level simulation because of overheads in accommodating network feedback.

A different flow-level simulation approach is exemplified by *fs* [1] in which a higher-level abstraction is used (the *flowlet*), but one that relates directly to real networking features (*e.g.*, discrete packets). Models of TCP throughput behavior are typically used to drive these types of simulators (*e.g.*, [11], [12]), and because they operate at a higher-level of abstraction than the packet, they offer significant speed advantages.

While flow-level simulation is appropriate for some types of experiments, some studies require simulation of low-level packet behavior. In these systems, the main approach toward achieving high performance has been to parallelize the simulation either through *spatial* or *temporal* partitioning of the simulation scenario. The key challenge in either approach is to ensure correct temporal ordering of packet-level events across different partitions either by enforcing particular constraints, or by detecting ordering violations and "recovering" from those violations [21], [22]. These approaches have led to highly scalable packet-level simulators, *e.g.*, [23]–[25]. For example, in [23] scenarios involving 1,536 nodes and very high (simulated) packet rates were reported. Commonly used packet-level network simulators such as ns-2 and ns-3 have received parallelization efforts (typically through spatial decomposition), and substantial speedups are typically reported [24], [26].

Our work differs from all these prior efforts in that we address parallelization for flow-level simulation. While simulating at the flow-level already offers efficiency gains over packet-level simulation, further improvements are clearly possible as we show in this paper. The partitioning approach we take is based on the work of Yao *et al.* [27], which fundamentally differs from prior parallelization approaches (*cf.* [22]). We further apply the approach of Yao *et al.* to the new context of flow-level simulation.

VI. Conclusions and Future Work

A key challenge in assessing new Internet systems, protocols and configurations is understanding how they will behave when broadly deployed. This calls for the ability to test and evaluate at scale in a representative and repeatable fashion. While simulation would appear to be ideally suited for these kinds of tests, standard network simulators such as ns-2 are unable to fulfill this need due to their basic architecture, which is focused on packet dynamics at fine time scales.

In this paper, we describe *pfs*, a set of extensions to the *fs* network simulator, which is based on a higher-level abstraction called a flowlet and enables highly accurate simulations of large networks. *pfs* enables simulations to be parallelized and run in massively-parallel computational environments, which are widely available. The goal of *pfs* is to enable simulations of very large networks such as would be found in large enterprises, service providers and data centers. Spatial and temporal decomposition in *pfs* is performed by careful analysis and reconfiguration of the traffic generation process for a given simulation.

We conducted a series of tests that demonstrate the capability of *pfs*. Using a very simple dumbbell network topology, we show that the preprocessing step in *pfs* has a very modest overhead, and that temporal, spatial or spatio-temporal parallelization all result in significant speedup. These tests also highlight the tradeoff between speedup and accuracy versus

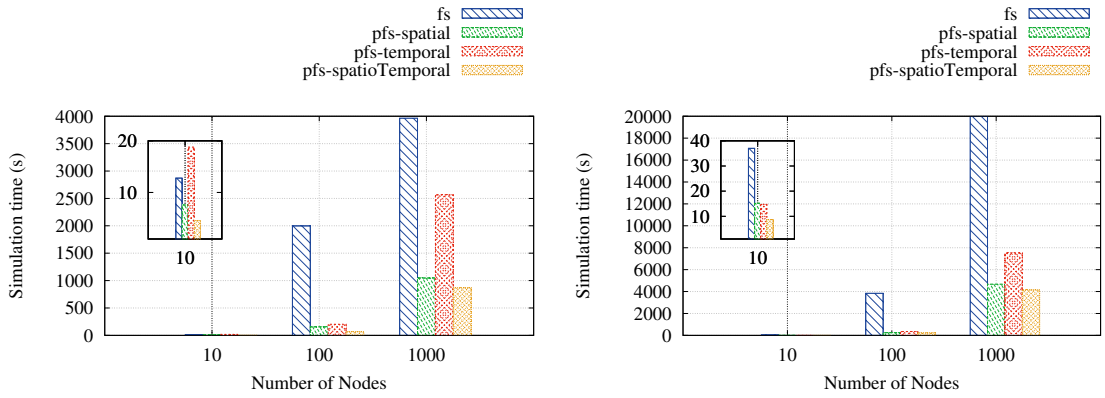

Fig. 9: Speedup for dumbbell (left) and linear (right) topologies achieved by *pfs* for spatial, temporal and spatio-temporal parallelizations.

a baseline serial simulation. We evaluate *pfs* using a larger network topology that is representative of a service provider network, and with large topologies of up to 10K nodes. Results from these tests also show that *pfs* can provide a reduction in simulation time of at least two orders of magnitude.

While we believe that *pfs* offers unique capabilities to simulate very large networks, there are number of issues that we will address in future work. First, we plan to continue to focus on improving simulation accuracy when simulating very large networks. Next, we will focus on the practical problem of how to specify, configure and evaluate simulations of very large networks *e.g.,* by creating canonical configurations that can be used by both researchers and practitioners. Finally, we also plan to benchmark the changes in results by evaluating *pfs* with a wide variety of relatively complex network and typical large-scale data center topologies.

Acknowledgements

This work was supported in part by NSF grant CNS-1054985, an ARL grant W911NF1110227, a DHS grant BAA 11-01 and an AFRL grant FA8750-12-2-0328. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF, ARL, DHS or AFRL.

References

[1] J. Sommers, R. Bowden, B. Eriksson, P. Barford, M. Roughan, and N. Duffield, "Efficient Network-wide Flow Record Generation," in *Proceedings of INFOCOM*, April 2011.

[2] S. McCanne, S. Floyd, K. Fall, K. Varadhan *et al.*, "Network Simulator ns-2," 1997.

[3] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[4] M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," in *Proceedings of ICDCS*, 1988.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of ACM HotCloud*, 2010.

[6] M. Gupta, J. Sommers, and P. Barford, "Fast, Accurate Simulation for SDN Prototyping," in *Proceedings of ACM HotSDN*, 2013.

[7] R. Durairajan, J. Sommers, and P. Barford, "Controller-agnostic SDN Debugging," in *Proceedings of ACM CoNEXT*, 2014.

[8] C. Systems, "Cisco IOS Netflow," http://www.cisco.com/go/netflow, 2010.

[9] A. Bilgin, J. Ellson, E. Gansner, Y. Hu, Y. Koren, and S. North, "Graphviz-Graph Visualization Software," 2010.

[10] J. Sommers and P. Barford, "Self-Configuring Network Traffic Generation," in *Proceedings of ACM IMC*, 2004.

[11] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm," *ACM SIGCOMM CCR*, 1997.

[12] N. Cardwell, S. Savage, and T. Anderson, "Modeling TCP Latency," in *Proceedings of IEEE INFOCOM*, 2000.

[13] "The ns-3 network simulator," http://www.nsnam.org.

[14] "cxFreeze: Python freezing library," http://cx-freeze.sourceforge.net/.

[15] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "Approx-Hadoop: Bringing Approximations to MapReduce Frameworks," in *Proceedings of ASPLOS*, 2015.

[16] "The Internet Topology Zoo," http://www.topology-zoo.org/.

[17] R. Durairajan, S. Ghosh, X. Tang, P. Barford, and B. Eriksson, "Internet Atlas: A Geographic Database of the Internet," in *Proceedings of ACM HotPlanet*, 2013.

[18] V. Misra, W.-B. Gong, and D. Towsley, "Fluid-based Analysis of a Network of AQM Routers Supporting TCP flows with an Application to RED," in *ACM SIGCOMM CCR*, 2000.

[19] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *Proceedings of USENIX NSDI*, 2010.

[20] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "PAST: Scalable Ethernet for Data Centers," in *Proceedings of ACM CoNEXT*, 2012.

[21] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, 1990.

[22] K. S. Perumalla, "Parallel and Distributed Simulation: Traditional Techniques and Recent Advances," in *Proceedings of WinSim*, 2006.

[23] R. M. Fujimoto, K. Perumalla, A. Park, H. Wu, M. H. Ammar, and G. F. Riley, "Large-scale Network Simulation: How Big? How Fast?" in *Proceedings of IEEE/ACM MASCOTS*, 2003.

[24] J. Pelkey and G. Riley, "Distributed Simulation with MPI in ns-3," in *Proceedings of ICST*, 2011.

[25] L. Bajaj, M. Takai, R. Ahuja, K. Tang, R. Bagrodia, and M. Gerla, "Glomosim: A Scalable Network Simulation Environment," *UCLA Computer Science Department Technical Report*, 1999.

[26] S. Lee, J. Leaney, T. O'Neill, and M. Hunter, "Performance Benchmark of a Parallel and Distributed Network Simulator," in *Proceedings of PADS*, 2005.

[27] W.-M. Yao and S. Fahmy, "Partitioning Network Testbed Experiments," in *Proceedings of ICDCS*, 2011.