Scalable RTI-Based Parallel Simulation of Networks

Kalyan S. Perumalla
Alfred Park
Richard M. Fujimoto
College of Computing, Georgia Tech
Atlanta, Georgia, USA 30332-0280
{kalyan,park,fujimoto}@cc.gatech.edu

George F. Riley
Department of Electrical and Computer Engineering, Georgia Tech
Atlanta, Georgia, USA 30332
riley@ece.gatech.edu

Abstract

Federated simulation interfaces such as the High Level Architecture (HLA) were designed for interoperability, and as such are not traditionally associated with high-performance computing. In this paper, we present results of a case study examining the use of federated simulations using runtime infrastructure (RTI) software to realize large-scale parallel network simulators. We examine the performance of two different federated network simulators, and describe RTI performance optimizations that were used to achieve efficient execution. We show that RTI-based parallel simulations can scale extremely well and achieve very high speedup. Our experiments yielded more than 80-fold scaled speedup in simulating large TCP/IP networks, demonstrating performance of up to 6 million simulated packet transmissions per second on a Linux cluster. Networks containing up to two million network nodes (routers and end systems) were simulated.

1. Introduction

The parallel discrete event simulation community has traditionally realized high-performance simulators using a monolithic approach where the parallel simulator is constructed "from scratch" and all simulation software is designed specifically for a particular simulation environment. Examples of parallel network simulators using this approach include GloMoSim [1], TeD [2, 3], SSFNet [4], DaSSFNet [5], TeleSim [6], and the ATM simulator described in [7]. One advantage of this approach is that the software can be tailored to execute efficiently in a specific environment. A disadvantage is the models must be developed "from scratch," leading to much duplication of effort and lengthy delays in realizing the parallel simulator.

Another approach to parallel/distributed simulation is to interconnect existing simulators. These federated simulations may include multiple copies of the *same* simulator (modeling different portions of the system), or entirely *different* simulators. The individual simulators that are to be linked may be sequential or parallel. SIMNET was perhaps the first system utilizing this approach to realize distributed training simulations [8]. An approach linking multiple copies of the commercial CSIM simulator is described in [9]. Industry standards for linking simulations have been developed, notably the Distributed Interactive Simulation (DIS) [10, 11] and the High Level Architecture (HLA) [12] standards. The federated approach offers the benefits of model and software reuse, and the potential of rapid parallelization of sequential simulators. It also offers the ability to exploit models and software from different simulators in one system [13].

This paper is concerned with the use of federated distributed simulation techniques using industry-wide standards to realize scalable parallel network simulations, thereby offering the potential to enjoy the benefits of both software reuse and high performance. The effectiveness of the federated approach for large-scale network simulations has not previously been proven. Of particular concern are the performance overheads incurred due to the runtime infrastructure (RTI) software that links the simulators, especially for network simulators that have fine-grained event computations on the order of 10 microseconds or less for each event.

2. Related Work

Efforts to realize federated simulations for high performance computing include the SF-Express project that realized a DIS-based system using ModSAF [14], and a parallel version of CSIM that was developed for simulating queuing networks [9]. Our work differs from SF-Express in our focus on network simulations that require time synchronization of small granularity events. Our work differs from the CSIM system in our focus on industry-wide standards and network simulation. The latter introduces significant complexities concerning interoperability that do not arise in queuing network simulations, e.g., see [15]. Other related work includes the backplane software described in [13, 16], and parallel simulators based on Opnet [15], and an aviation simulation called TAAM [18]. These efforts focus on

Proceedings of the Seventeenth Workshop on Parallel and Distributed Simulation (PADS'03)
1087-4097/03 $17.00 © 2003 IEEE

interoperability issues, however, and do not address the question of large-scale network simulation.

3. Background

We briefly review implementation features of the monolithic and federated approaches to parallel simulation, and highlight some of the performance-critical aspects of the federated approach.

We use HLA terminology throughout this paper. Specifically, a parallel/distributed simulation is referred to as a federation, which is composed of a number of simulators (termed federates) that interact through services provided by runtime infrastructure (RTI) software. The RTI is middleware software that lies between the operating system and the federates. It implements a set of services defined in the HLA Interface Specification (IFSpec). For our purposes, the most important are the Object Management services that provide communication primitives, and the Time Management services that support synchronization. See [12, 17] for an introduction to the HLA and IFSpec services.

3.1. Monolithic vs. Federated

In traditional parallel simulators, the simulator kernel directly handles inter-processor communication. Typically, the kernel is also closely coupled with time synchronization algorithms tailored specifically for that parallel simulator and computation platform. Such direct communication and close coupling can potentially result in an optimized, high-performance implementation.

However, in federated simulation, the RTI acts as an intermediate layer that decouples the federation from specific implementations of message passing and time synchronization protocols. Event exchange is generally achieved via indirect communication using multicast group semantics. This permits both destination naming independence as well as messaging optimizations in multi-destination communication. Time synchronization is also decoupled from the simulators, and implemented within the RTI, thus making the federation less dependent on any specific time synchronization protocol.

Further, the RTIs supporting standards such as the HLA must accommodate a variety of simulators and support different synchronization protocols including time stepped, and conservative and optimistic event driven execution, possibly all within the same federation execution. While any given federation typically uses only a subset of these options, the RTI must support all of them, leading to increased implementation complexity. Due to the general nature of a standard such as the HLA, the RTI cannot exploit application specific characteristics, e.g., a static topology among simulation processes. Other potential optimizations, such as lookahead between specific pairs of processes, are difficult to define in a general way, and hence are not supported in the HLA. Thus, certain performance optimizations that might be possible in a monolithic simulation environment are difficult or impossible to realize in a general standard.

Our performance study is based on two different network simulators: *pdns* and *GTNetS*. Each of these two simulators is parallelized by self-federating the simulator with itself. In other words, the parallel version of *pdns* is essentially a federation of multiple copies of sequential *ns* (and similarly for *GTNetS*).

For our current purpose, namely, evaluating RTI-based parallelization approach, we focus on homogeneous federations (self-federation), and do not consider a mixture of dissimilar simulators.

3.2. Federated Implementation

```
TimeAdvanceGrant(T) {
      granted = true; grantedtime = T;
}
SendEventToGroup (Event e, Group g) {
      e->timestamp = Now + LA;
      Update(e, g);
}
Reflect(Event e, Group g) {
      Now = e->timestamp;
      ProcessEvent(e);
}
MainLoop() {
      SetLookAhead(LA);
      While (not end of simulation) {
            e = local min time-stamped event
            If (e->timestamp <= grantedtime) {
                  Dequeue(e);
                  Now = e->timestamp;
                  ProcessEvent(e);
            } else {
                  t = e->timestamp;
                  NextEventRequest(t);
                  granted = false;
                  While(not granted) RTITick();
            }
      }
}
```

Figure 1: Outline of NER-based approach. RTITick() results in zero or more calls to Reflect() to transfer events from RTI to federate, followed by a call to TimeAdvanceGrant() to the next safe time. Update() is called by the federate during event processing to send events to other processors via group communication.

An HLA RTI must support multiple primitives for advancing simulation time to accommodate different time flow mechanisms. Despite the multitude of primitives, they can be implemented over a single, shared core functionality, based on the computation of a value called Lower Bound on Time Stamp (LBTS).

Here we focus on the **NextEventRequest** primitive (abbreviated as NER), which is used by *pdns* and *GTNetS*. NER is the primitive designed for use by conservative event driven federates, so is the most natural choice for these implementations. The pseudo code for the main simulation loop using NER is shown in Figure 1. Equivalent implementations using other RTI primitives, namely, **TimeAdvanceRequest** (TAR) and **FlushQueueRequest** (FQR), are possible. Nevertheless, among the different alternatives, NER is the most critical with respect to runtime overhead, since it is normally invoked prior to *every* local event.

The optimizations we will describe later in the paper are based on the core LBTS functionality, and hence are generic in nature. Thus, it is important to note that our approach is not necessarily limited to the **NextEventRequest** primitive, although further work is needed to more thoroughly evaluate the performance on the rest of the primitives.

3.3. RTI Overhead

After initialization is completed and the main simulation loop is entered, the federate periodically enters the RTI for (a) sending events (b) receiving events, and (c) synchronizing virtual time. In line 6 of Figure 1 the **Update** RTI primitive is used by the federate to send events. Events are received by the federate using the **Re flect** callback on line 8. On line 22, the federate uses **NextEventRequest** to request delivery of events and to advance simulation time. This service will cause the RTI to deliver the smallest time stamped event from another federate (if any) that has a timestamp earlier than the minimum timestamp of any event in the calling federate's local event list. The RTI is given CPU cycles on line 24 using the **RTITick** call (henceforth abbreviated as Tick). The RTI performs time synchronization and network processing during this call, and provides incoming event(s) to the federate via the **Reflect** callback, and grants time advances via the **TimeAdvanceGrant** callback (henceforth abbreviated as TAG).

Clearly, when inter-federate communication is sparse, the bulk of the RTI overhead occurs within lines 22 to 24.

4. Performance Optimization

When we began to examine the performance of the federated *pdns* and *GTNetS* simulators, we observed that a simple 2-processor *pdns* run exhibited disappointing performance. This experiment scaled the size of the simulated network in proportion to the number of processors. The metrics observed in the initial 2-CPU *pdns* run are shown in Table 1.

While we expected to see negligible RTI overhead, and hence near-linear speedup, we instead observed an average RTI overhead of over 3.6 microseconds per event.

CPUs	Nodes / CPU	Events/CPU (millions)	Run Time (s)	Mics/ Event
1	3766	73.7	485	6.58
2	3766	73.7	751	10.19

Table 1: Initial performance of *pdns* on baseline RTI.

Although an RTI overhead of 3.6 microseconds per event is negligible for traditional RTI-based applications, it is significant for the fine-grained *pdns* federation. The RTI overhead was almost 55% of sequential event execution time, even with little inter-processor communication, warranting a closer look at the overhead.

Upon investigation, we discovered different sources of overhead that, although contributing minor amounts, eventually added up to the cumulative overhead that was observed. We now describe each of these sources, and outline the solutions we adopted to eliminate them and substantially improve the parallel performance. For example, in the 2-CPU run of Table 1, as we will see, we were able to optimize the RTI to reduce the overhead down from 3.6us (55%) to just 1.2us (18%) per event.

4.1. NLBTS

One of the first problems we noticed was that the RTI spent a significant portion of the time trying to advance simulation time in small increments. The RTI computes a quantity called Lower Bound on Time Stamp (LBTS) of future events that may be received by a processor in order to implement HLA's timestamp-ordered message delivery service. The RTI software used in this study computes LBTS values asynchronously, in the background with other federate computations. When a federate needs to advance its simulation time, the RTI initiates an LBTS computation if the last computed LBTS value is not large enough to issue the TAG. However, in the presence of a load imbalance where a lightly loaded processor has advanced ahead of the others in simulation time, this processor can inundate the more heavily loaded processor with LBTS computations. This phenomenon can happen despite the presence of a large lookahead, and despite the fact that eventually all processors are loaded similarly. The fact that one processor is ahead in simulation time of the others is sufficient to initiate this problem. We have noticed this phenomenon not only in *pdns* and *GTNetS*, but also in other network simulator federations.

A simple solution is to have each processor participate in an LBTS computation only when it itself needs a time advance, thereby forcing the processor that is ahead to wait. Even if a lightly loaded processor initiates an LBTS computation, the other processors refrain from eagerly responding to that computation, but instead participate only when they themselves run out of local computation. This simple rule not only tends to reduce the load imbalance, but also evens out any transient imbalances in an otherwise well-balanced federation.

The correctness of this optimization is ensured by the

fact that every federate eventually reaches its most recently granted time, and hence will initiate another LBTS computation in order to make progress. Due to this effect, it is impossible for any federate to wait indefinitely, and overall progress is ensured in the federation.

Of course, a drawback to this solution is that processors could potentially waste some amount of time waiting for a time advance. However, the waiting time is not significant in federates that advance their time at roughly identical pace.

4.2. Short Circuiting Tick Calls

Another significant portion of the overhead is incurred by the federates due to the NER and Tick call combination required before processing each local event. Note that the RTI keeps an ordered list of timestamp-ordered (TSO) events, so that they could be delivered in timestamp order according to NER semantics. The minimum timestamped event in the RTI's TSO event list is referred to as TSOMin. The RTI also keeps the most recently computed LBTS value in a variable named LBTS. Figure 2 shows the actions performed for each NER and Tick call.

```
1.  NextEventRequest(T) {
2.      Check for error conditions;
3.      Set up NER pending request state;
4.      Compute & update new local RTI time;
5.      Initiate a new LBTS if necessary;
6.  }
7.  RTITick() {
8.      Check incoming network messages;
9.      Process LBTS messages;
10.     While (there is a deliverable TSO event e) {
11.         Reflect(e);
12.     }
13.     If (can issue a time advance grant) {
14.         T = grantable time;
15.         TimeAdvanceGrant(T);
16.     }
17. }
```

Figure 2: NER and Tick implementation within RTI.

Although the individual operations within NER and Tick are relatively simple and efficient, it is clear that they can accumulate if invoked very frequently. For example, the check for incoming messages takes less than 0.5 microseconds with our shared memory communication implementation. Similarly, the time to process LBTS messages is also insignificant when considered in isolation. However, together they add up to more than 3 microseconds on a 2-CPU execution. Furthermore, the overhead increases significantly when TCP communication is used.

In order to address this problem, we analyzed how often each of the operations was indeed required, and how often the operations were superfluous. It was found that in the vast majority of Tick calls, no incoming messages needed to be processed.

Thus, an effective optimization is to short-circuit Tick calls to optimize the common case. We call this approach the "fast-path" implementation of NER and Tick calls. It optimizes for the case where there are no "deliverable" events in the RTI's TSO event queue, and when the previously computed LBTS is beyond the time advance requested in the NER call. In other words, in the most frequent case, NER and Tick are effectively no-ops, doing nothing more than checking for the no-op condition and issuing a grant to the requested NER time.

Figure 3 shows the "fast-path" optimization code pre-pended to their function bodies.

```
1.  NextEventRequest(T) {
2.      If (T < TSOMin and T < LBTS) {
3.          Mark pending request as NER(T);
4.          return;
5.      }
6.      Execute as usual;
7.  }
8.  RTITick() {
9.      If (NER(T) is pending and
10.         T < TSOMin and T < LBTS) {
11.         TimeAdvanceGrant(T);
12.         return;
13.     }
14.     Execute as usual ;
15. }
```

Figure 3: Fast path optimization for NER and Tick implementation within the RTI.

The lines 2-5 and 9-13 of Figure 3 correspond to the fast path optimizations. Notice that, as a result of the fast path code, both the functions return immediately upon detecting the fast path condition, which is that no RTI TSO events can be delivered (T < TSOMin), and no new LBTS computation is required (T < LBTS). The net effect of this optimization is that the NER, Tick and TAG calls together degenerate to a fast sequence of three short function calls.

5. Performance Study

We now present a performance analysis of federated execution, to demonstrate the relevance of RTI-based federated approaches to high-performance parallel/distributed simulation. We do this using the two network simulators mentioned previously, namely, *pdns* and *GTNetS*, each of which has been parallelized using our HLA RTI implementation. The HLA software in question implements a subset of the HLA Interface Specification (version 1.3). It utilizes one notable simplification of the IFSpec: attribute-handle-value pair sets are not implemented, in favor of a simpler mechanism to pass attribute values to the RTI.

5.1. *Network Configuration*

The network topology, traffic, and parameters were based on the benchmark specification developed by the research group at Dartmouth College [4]. The benchmarks were developed as a set of baseline models for the network modeling and simulation community. The benchmark configurations were developed with the intention of facilitating the demonstration of network simulator scalability. To aid in scalability studies, replication and expansion can be used on the original smaller network topology to easily create larger sized networks.

Topology

Each portion of the network is referred to as a Campus Network (CN). Figure 4 shows the schematic for a typical CN. Each CN consists of 4 servers, 30 routers, and 504 clients for a total of 538 nodes. The CN is comprised of 4 separate networks. Net 0 consists of 3 routers, where node 0:0 is the gateway router for the CN. Net 1 is composed of 2 routers and 4 servers. Net 2 consists of 7 routers, 7 LAN routers, and 294 clients. Net 3 contains 4 routers, 5 LAN routers, and 210 clients.

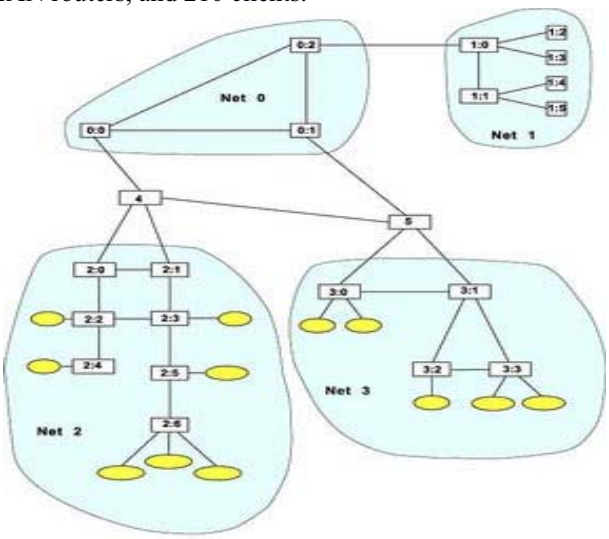

Figure 4: Basic campus network (CN) model.

Net 0 is connected to Net 2 and Net 3 via standalone routers. Net 1 is connected directly to Net 0 through a single link. All non-client links have a bandwidth of 2Gb/s and have a propagation delay of 5ms with the exception of the Net 0 to Net 1 links, which have a propagation delay of 1ms. Clients are connected in a point-to-point fashion with their respective LAN router and have links with 100Mb/s bandwidth and 1ms delay.

Multiple CNs may be instantiated and connected together to form a ring topology. This aspect of the network allows the baseline model to be easily scaled to arbitrarily large sizes. Multiple CNs are interconnected through a high latency 200ms 2Gb/s link via their Net 0 gateway router.

Traffic

In our performance study, we focus on pure TCP traffic requested by clients from server nodes. All TCP traffic is "external" to the requesting CN clients, *i.e.*, all the clients generate TCP traffic to/from servers in an adjacent CN in the ring (CN *i* communicates with CN *i+1*, etc.). Also, we use the short transfer case of the baseline model, in which clients request 500,000 bytes from a random Net 1 server. The TCP sessions start at time selected from a uniform distribution over the interval from 100 and 110 seconds of simulation time.

5.2. *Scaling Methodology*

The experiments described here scale the size of the simulated network in proportion to the number of processors used. This is a widely accepted approach for scalability studies in the high performance computing community. It also circumvents the problem of having a sequential machine with enough memory to execute the entire model, which would not be possible for the large simulations that are considered here.

A principal performance metric used here is the number of simulated "packet hops" that can be processed by the simulator in one second of wallclock time. A "packet hop" represents the transmission of a packet from one node (a router or end node system) to another over a link in the network. Network simulators will typically require more than one event to simulate a packet hop. For example, *pdns* and *GTNetS* both require exactly two simulator events to model a packet hop.

5.3. *Simulation Platform*

All our experiments are executed on a large Linux cluster consisting of 16 machines. Each machine is a Symmetric Multi-Processor (SMP) machine with eight 550MHz Pentium III XEON processors. The eight CPUs of each machine share 4 GB of RAM. Each processor contains 32KB (16KB Data, 16KB Instruction) of non-blocking L1 cache and 2MB of full-speed, non-blocking, unified L2 cache. An internal core interconnect utilizes a 5-way crossbar switch connecting two 4-way processor buses, two interleaved memory buses, and one I/O bus. The operating system is Red Hat Linux 7.3 running a customized 2.4.18-10smp kernel.

The 16 SMP machines are connected to each other via a Dual Gigabit Ethernet switch with EtherChannel aggregation. Our RTI software uses shared memory for communications within an SMP, and TCP/IP for communication across SMPs.

Note that, in the following sections, the performance metrics are consistent across multiple runs, and hence error bars are not shown.

5.4. *RTI Primitive Timings*

Execution times for key RTI primitives are shown in Table 2. The first two lines report the time required for each invocation of NER and Tick, as discussed earlier. **UpdateAttributeValues** is an HLA service to send a message. **ReflectAttributeValues** is a callback from the RTI that is invoked to deliver a message to the federate. The reported times indicate the execution time required in the RTI to deliver the message, and the amount of time in the federate (for *pdns*) to process incoming event.

Primitive	Portion	Average Time (microsecs)
NextEventRequest	RTI	1.99
RTITick	RTI	3.03
UpdateAttributeValues	RTI	36.61
ReflectAttributeValues	RTI	21.28
ReflectAttributeValues	*pdns*	37.13

Table 2: Micro timing measures with *pdns* on 16 CPUs, 7 CN/CPU, for RTI primitives after optimizations.

5.5. *Performance after Optimizations*

The individual and cumulative performance improvements provided by the NLBTS and fast path optimizations are shown in Figure 5 and Figure 6 for *pdns*, and in Figure 7 for *GTNetS*. The 1-processor data point in the figures corresponds to executing the parallel version on a single CPU.

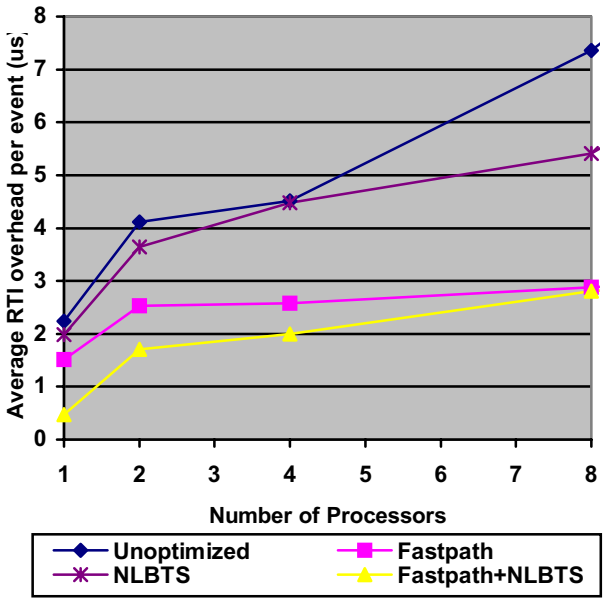

Figure 5: Decrease in overhead per *pdns* event with each optimization on a single 8-CPU machine with shared memory inter-processor communication.

The benefits of NLBTS optimization are more pronounced when all communication is performed via shared memory, as seen in Figure 5. In this case, due to the high speed of shared memory messaging, each LBTS computation completes rapidly, and hence provides more opportunity for the lightly loaded processor to initiate many more LBTS computations. However, this effect is less severe when TCP communication is introduced when scaling to a large number of processors, as see in Figure 6. The number of LBTS computations is automatically reduced due to longer messaging delay, and hence the reduction in overhead is negligible beyond 16 processors.

On the other hand, the fast path optimization fetches significant savings in overhead in all processor configurations. The savings are in fact greater on a larger number of processors, partly because it avoids the high cost frequent network polling.

Similar performance improvement trends are seen with *GTNetS* as well, as shown in Figure 7.

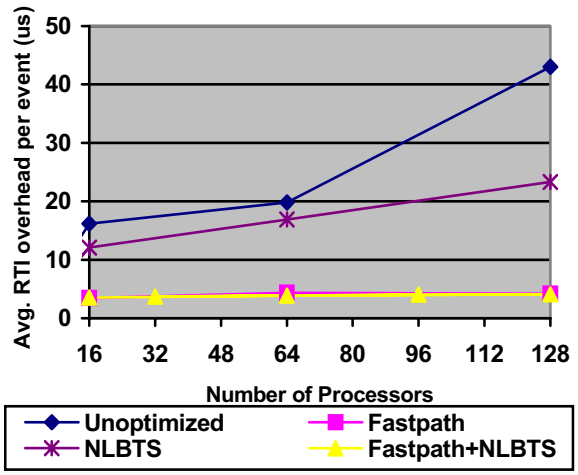

Figure 6: Decrease in overhead per *pdns* event with each optimization on multiple 8-CPU machines.

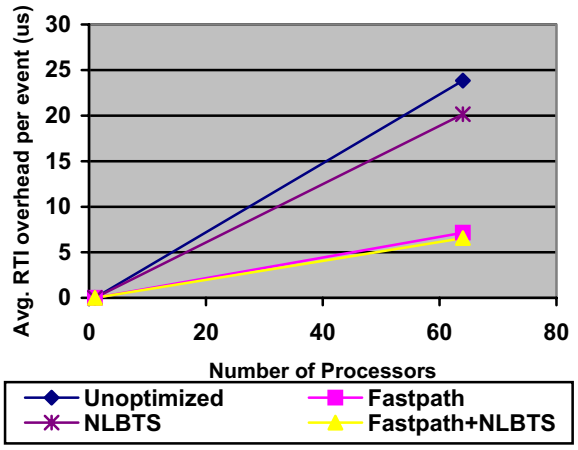

Figure 7: Decrease in overhead per *GTNetS* event with each optimization on multiple 8-CPU machines.

It is clear from the data that the optimizations are necessary in order to lower the overall amortized runtime overhead of each event. As a net result, the RTI overhead levels off at around 3-4 microseconds per event, even when the number of processors is increased up to the

maximum available number of processors.

5.6. Scalability Study

We now consider the scalability of the federations using the optimized version of the RTI. As described in the scaling methodology earlier, the network is scaled with the number of processors for all our scalability experiments. Scalability is tested along two fronts: (a) simulation runtime/speed (b) maximum network size that can be simulated. Initialization time is excluded in simulation runtime.

As can be expected with any set of different simulators, *pdns* and *GTNetS* exhibit slightly different speed and memory characteristics. *pdns* events execute faster since they model slightly lesser amount of detail than *GTNetS* events, while *GTNetS* is more memory-efficient than *pdns*.

Parallel Speedup

The parallel speedup afforded by the simulators is shown in Figure 8. Both simulators scale very well with increasing number of processors. *pdns* exceeds a speedup of 80 on 128 processors (16 8-CPU machines), while *GTNetS* reaches 80-fold speedup on 120 processors (15 8-CPU machines).

Packet Hop Rate

The simulation speed of *pdns* is shown in Figure 9 for simulating 7-CN per CPU. *pdns* achieves a speed exceeding 6 million packet hops per second when executing on 128 processors. *GTNetS* clocks approximately 3 million packet hops per second on 120 processors for the same network model.

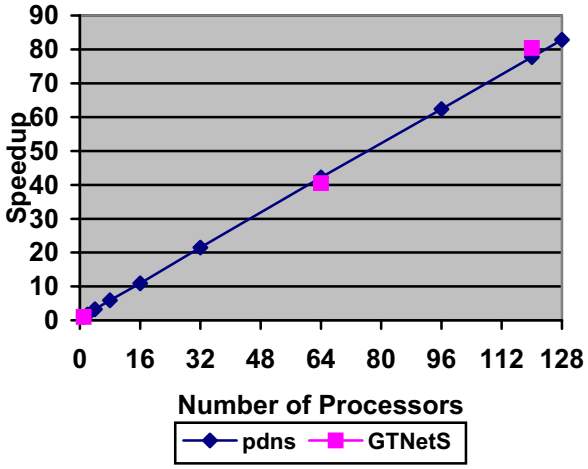

Figure 8: Scalability of runtime.

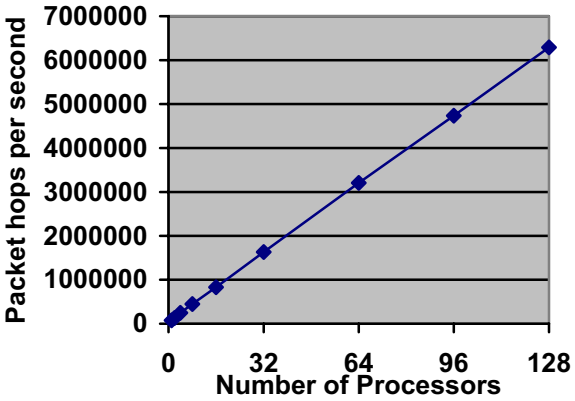

Figure 9: Scalability of *pdns* showing over 6 million packet hops per second on 128 processors.

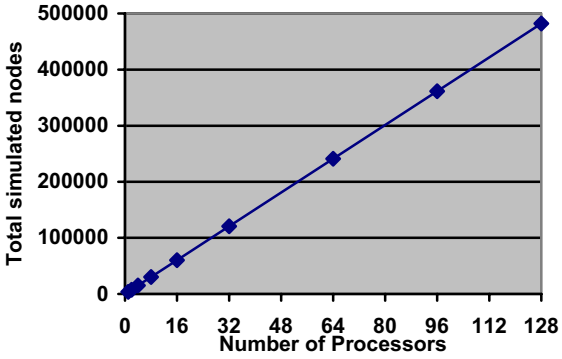

Figure 10: Scalability of *pdns* showing almost half a million simulated nodes on 128 processors.

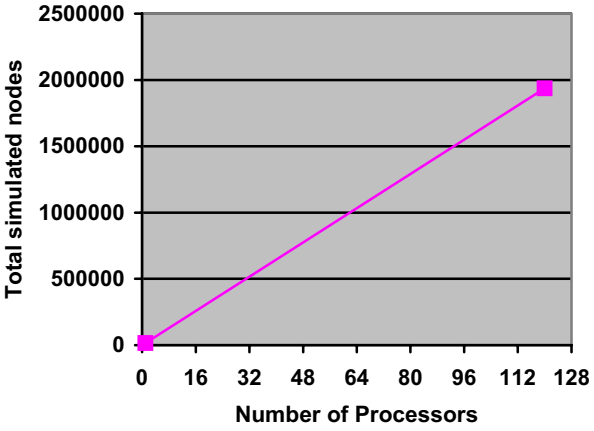

Figure 11: Scalability of *GTNetS* showing nearly 2 million simulated nodes on 120 processors.

Network Size

The increasing network sizes shown in Figure 10 and Figure 11 are interesting when considered in conjunction with packet hop rates shown in Figure 9. Not only the packet hop rate but also the network size increases linearly with number of processors. This demonstrates scalability

along each dimension without affecting the scalability along the other dimension.

pdns reaches the memory limit (4GB) on each 8-CPU box when simulating one 7-CN (3766 nodes) per CPU. *GTNetS* simulates over 20,000 nodes per CPU.

6. Future Work

The fast path optimization method could potentially be applied to other RTI primitives. For example, the FQR primitive could be optimized for optimistic simulations such as TeD and Telesim, and the TAR primitive could be tuned for efficient time-stepped simulations such as vehicular traffic simulations.

More generally, the fast path and NLBTS optimizations are examples of the types of improvements that can be performed on an RTI implementation. We believe it is possible to generalize such optimizations, and make them automatically detected and tuned by the RTI at runtime, depending on the dynamics of the executing federation. We are investigating adaptive mechanisms for automatically tuning different optimizations based on performance monitoring at runtime.

7. Conclusions

We have demonstrated that HLA-like federated simulation interfaces, although originally defined for interoperability and ease of integration, can also be efficiently implemented for high performance. The parallel execution performance can rival that of monolithic approaches, delivering extremely good speedups even in the challenging case of fine-grained event processing. Using our optimized RTI implementation, we are able to achieve some of the largest packet-level network simulations to date.

An interesting corollary to our work is that the use of un-optimized RTI implementations can convey the incorrect notion that RTI-based federated execution is inherently slow. Our initial performance runs using an un-optimized RTI implementation substantiate such a false notion. Our subsequent optimizations and the resulting excellent speedup demonstrate that federated simulation interfaces can indeed be implemented efficiently.

In favor of the RTI-based approach, it is also noteworthy that the same optimized RTI implementation was easily reusable for parallelizing multiple different simulators. We were able to realize efficient parallel implementations of both *pdns* and *GTNetS* simply by linking the exact same library of our RTI software into both simulators. While reuse of optimizations is not nearly as straightforward across different monolithic parallel simulators, the RTI reuse was natural in our federated approach.

References

1. Zeng, X., R. Bagrodia, and M. Gerla, GloMoSim: A Library for Parallel Simulation of Large-Scale Wireless Networks, in Proceedings of the 1998 Workshop on Parallel and Distributed Simulation. 1998. p. 154-161.
2. Perumalla, K., R. Fujimoto, and A. Ogielski, TeD - A Language for Modeling Telecommunications Networks. Performance Evaluation Review, 1998. **25**(4).
3. Poplawski, A.L. and D.M. Nicol, Nops: A Conservative Parallel Simulation Engine for TeD, in 12th Workshop on Parallel and Distributed Simulation. 1998. p. 180-187.
4. Cowie, J.H., D.M. Nicol, and A.T. Ogielski, Modeling the Global Internet. Computing in Science and Engg., 1999.
5. Liu, J. and D.M. Nicol, DaSSF 3.0 User's Manual. 2001.
6. Unger, B., The Telecom Framework: a Simulation Environment for Telecommunications, in Proceedings of the 1993 Winter Simulation Conference. 1993.
7. Pham, C.D., H. Brunst, and S. Fdida, Conservative Simulation of Load-Balanced Routing in a Large ATM Network Model, in Proceedings of the 12th Workshop on Parallel and Distributed Simulation. 1998. p. 142-149.
8. Miller, D.C. and J.A. Thorpe, SIMNET: The Advent of Simulator Networking. Proceedings of the IEEE, 1995. **83**(8): p. 1114-1123.
9. Nicol, D. and P. Heidelberger, Parallel Execution for Serial Simulators. ACM Transactions on Modeling and Computer Simulation, 1996. **6**(3): p. 210-242.
10. IEEE Std 1278.1-1995, IEEE Standard for Distributed Interactive Simulation -- Application Protocols. 1995, New York, NY: Institute of Electrical and Electronics Engineers.
11. IEEE Std 1278.2-1995, IEEE Standard for Distributed Interactive Simulation -- Communication Services and Profiles. 1995, New York, NY: Institute of Electrical and Electronics Engineers Inc.
12. Kuhl, F., R. Weatherly, and J. Dahmann, Creating Computer Simulation Systems: An Introduction to the High Level Architecture for Simulation. 1999: Prentice Hall.
13. Perumalla, K., et al., Experiences Applying Parallel and Interoperable Network Simulation Techniques in On-Line Simulations of Military Networks, in Proceedings of the 16th Workshop on Parallel and Distributed Simulation. 2002. p. 97-104.
14. Brunett, S., et al., Implementing Distributed Synthetic Forces Simulations in Metacomputing Environments. 1998, California Institute of Technology, Center for Advanced Computing Research (CACR-158).
15. Wu, H., R. Fujimoto, and G. Riley, Experiences Parallelizing a Commercial Network Simulator, in Proceedings of the Winter Simulation conference. 2001.
16. Riley, G., et al. Distributed Network Simulations using the Dynamic Simulation Backplane. in International Conference on Distributed Computer Systems. 2001.
17. Fujimoto, R.M., Time Management in the High Level Architecture. Simulation, 1998. **71**(6): p. 388-400.
18. Bodoh, D., and F. Weiland. Self Federating an Aviation Simulation using HLA: Is it Feasible? in Proceedings of the Workshop on Distributed Simulation and Real-Time Applications, 2001.