Proceedings of the 2001 Winter Simulation Conference
B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, eds.

EXPERIENCES PARALLELIZING A COMMERCIAL NETWORK SIMULATOR

Hao Wu
Richard M. Fujimoto
George Riley

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

ABSTRACT

Most current approaches of parallel simulation focus on building new parallel simulation engines that require the development of new models and software. An alternate, emerging approach is to extend sequential simulators to execute on parallel computers. We describe a methodology for realizing parallel simulations in this manner. This work is specifically concerned with parallelization of commercial simulators where source code for some or all of the sequential simulator is not available. We describe our experiences in applying this methodology to realize a parallel version of the OPNET simulator for modeling computer networks. We show significant speedup can be readily obtained for some OPNET models if proper partitioning strategies are applied and the simulation attributes are tuned appropriately. However, we observe that substantial modifications to other OPNET models are needed to achieve efficient parallel execution because of their extensive use of global variables and "zero lookahead events".

1 INTRODUCTION

Simulation is often the method of choice in addressing many network research problems. This is because it is often too costly and time-consuming to create physical networks requiring a large number of network nodes. For example, to study the impact of varying RED algorithm parameters in the Internet (Christiansen et al. 2000), it would be impractical to deploy the algorithm throughout the Internet. Analytic modeling may be used in some cases, but the complexity and scale of modern networks often necessitate the use of simplifying and unrealistic assumptions, e.g. Poisson traffic models. Furthermore, it is hard to capture the dynamic nature of a network with a mathematical model. Simulation is often a critical step before deploying real networks.

The increasing scale, speed and complexity of modern networks have dramatically increased the time and resource requirements of network simulations. In one of our experiments, the simulation of one minute of the operation of a 200-node network model required the processing of more than 4,600,000 events and 16 minutes of CPU time. High-speed gigabit networks carry much more traffic, resulting in a proportional increase in the number of events the simulator must process. The increasing complexity of protocol stacks in communication end systems further aggravates this problem.

Parallel simulation, i.e. distributing the execution of the simulation over multiple computers, provides one approach to addressing this problem. There are already several research efforts in building parallel network simulators. Bagrodia et al. developed GloMoSim (Global Mobile System Simulation) for parallel simulation of wireless networks (Zeng et al. 1998). Cowie et al. addressed a scalable, parallel discrete event simulator capable of modeling the Internet (Cowie et al. 1999). That work was based on a previous effort by Perumalla et al. who created TeD (Perumalla et al. 1998), which is capable of creating multithreaded network simulations. Nicol et al. described IDES, a java based distributed simulation engine (Nicol et al. 1998). Ferenci et al. federated parallel optimistic simulators for network simulation (Ferenci et al. 2000). Unger et al. developed a conservative parallel network simulator in the TeleSim project (Unger 1993).

Despite these efforts, most network simulations today are performed using sequential simulators such as OPNET or *ns*. One important reason is that transitioning to a new simulator, especially a parallel simulator is often difficult, requiring one to abandon familiar software environments, models, and tools. Often, one must abandon previous investments in sequential simulators and master a new, unfamiliar simulation language.

One approach to solve this problem is to parallelize existing sequential simulators by decomposing the system being modeled into subsystems, instantiating a separate simulator for each subsystem on a different processor, and adding extensions to exchange data and synchronize the

separate instantiations. For example, Riley et al. designed and implemented Parallel/Distributed ns *pdns*, which allows a single ns simulation to be distributed on a cluster of workstations (Riley et al. 1999). A similar effort is reported in (Jones et al. 2000). Nicol et al. described an approach in (Nicol et al. 1996) allowing the modeler to develop sub-models with an existing sequential simulation modeling tool, using the full expressive power of the tool. A set of modeling language extensions permit automatically synchronized communications between sub-models.

These approaches require access to the underlying simulation executive to implement extensions that are necessary to ensure proper synchronization of the parallel simulator. Here, we explore parallelization of commercial simulators where arbitrary changes to the underlying simulation engine are not possible. Specifically, we assume the source code of the simulation engine is not available. Our approach does require the simulation engine to support a small set of capabilities. Our work is closer in spirit to that of Strabburger concerning application of HLA to commercial applications (Strabburger 2000).

An important goal of this approach is to minimize changes to the original sequential simulator. This will facilitate model and software reuse, which is a central objective of the approach. But this constrains the inter-federate synchronization to be conservative.

Here, we use terminology used by the High Level Architecture (HLA) (Defense Modeling and Simulation Office 2000). HLA is a framework intended to facilitate the interoperability and reuse of simulation models. In HLA, a parallel/distributed simulation is called a federation, and each individual simulator is referred to as a federate. The software providing communication and synchronization services to federates is referred to as the runtime infrastructure (RTI).

The remainder of this paper is organized as follows. In section 2, we introduce our methodology, associated problems, and solution approaches. Section 3 describes our experiences in applying this approach to the OPNET simulator and presents performance measurements of a prototype implementation. In section 4, we discuss our conclusions.

2 APPROACH TO PARALLELIZATION

2.1 Simulator Assumptions

We make the following assumptions:

- Here, we are only concerned with discrete event simulations utilizing an event-driven style of execution. We assume links and nodes of the model are represented with link objects and node objects, respectively. Data flow is represented by transmission of time-stamped events. Many network simulators, e.g., ns (McCanne et al. 1997) and OPNET (Chang 1999) are structured this way.
- We assume new user-defined object models can be defined and added to the simulator. This provides a means to add communication and synchronization software to the sequential simulator.
- We assume the pending event list data structure is accessible to user-defined model code. We do not require source code for the sequential simulator executive, e.g., the central event processing loop.
- An important optimization utilized in our methodology involves a construct called *ghost object.* Utilization of *ghost objects* requires access to the simulation model code, though access to the simulator executive is still not required.

2.2 Parallel Network Simulation Architecture

Figure 1 shows our overall architecture for the parallel network simulation. Each federate is a sequential simulator modeling a subnetwork. Communication and synchronization services are provided by the RTI. Every federate manages its local components (a subset of the entire network model). A proxy model is added as an extension to the simulation model on each federate, defining an interface between the sequential simulator and RTI.

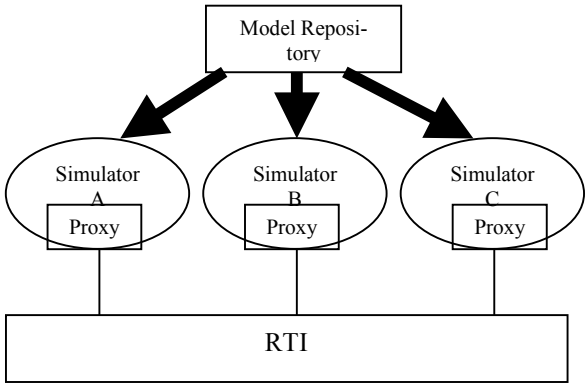

Figure 1: Parallel Network Simulation Architecture

To illustrate our approach, we will use the simple network model depicted in Figure 2. This model consists of 4 end hosts (H0-H3) and 4 routers (R0-R3). The four hosts are both "UDP source" and "UDP sink" nodes. There is traffic from each host to every other host. Let us assume we partition this model into 2 sub-models and simulate each on separate federates called A and B respectively.

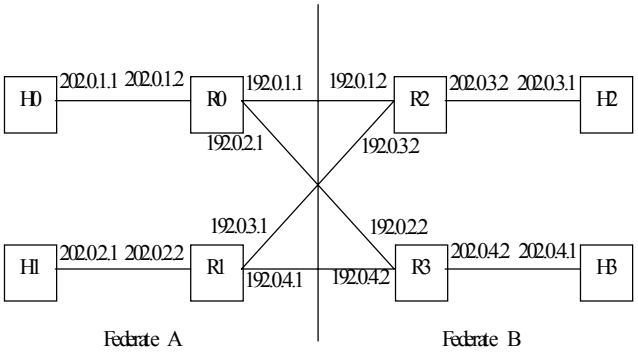

Figure 2: Sample Network Model

In the remaining part of this section, we illustrate the general methodology for parallel simulation, followed by issues concerning the construction of such a parallel simulator:

- Data flow across federates
- Time & Event management
- Optimizations.

2.3 Methodology

Our general methodology follows from issues addressed by Riley et al. in (Riley et al. 1999):

1. Partition the network model into sub-models, one per processor. Each sub-model represents a subnet. The union of all the subnets forms the original network model. The partitioning strategy has an important effect on the overall performance. As we will see later, an optimal partitioning must trade off among several factors: load balance, connectivity, lookahead and event locality. The partitioning strategy is beyond the scope of this paper. Interested readers can refer to (for instance) (Zeng et al. 1998).
2. Map a sub-model to a sequential simulator on each processor. Create the sub-model in its mapped simulator. Those components not existing in local sub-model can be initiated as *ghost objects* if needed, which will be addressed in section 2.6.
3. Anywhere a sub-model interacts with a model element that is instantiated on another processor an interaction with a proxy model is defined. The proxy model is responsible for implementing interactions with entities instantiated on other processors.
4. Apply optimizations to improve performance. For example, one can modify the model to reduce memory requirements.

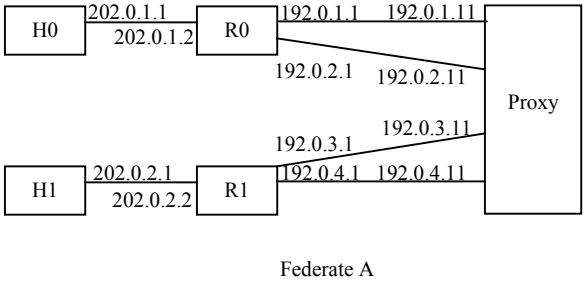

Figure 3: Sub-model in Federate A

Figure 3 shows the architecture of the sub-model in Federate A after applying our approach.

2.4 Data Flows Across Federates

A network model includes node objects that are interconnected using link objects. A link object is associated with endpoint node objects that are connected to it. Once we partition the network model into sub-models, the links that cross a partition boundary are broken and lose their endpoints in other sub-models. In order to conserve memory, node objects not existing in local sub-models may not be created locally (see step 2 above), making it impossible to reference these non-existing node objects as end points. We also need to ensure that data traffic can flow across the partition boundary.

To address these problems, we define a proxy model in every sub-model. The proxy model is constructed using the model construction methods provided by the sequential simulator so the proxy appears within the simulator to be no different than any other model element. However, the proxy model is linked with RTI libraries to make use of the services provided by the RTI. The functions implemented by the proxy model include:

1. Utilize RTI services to realize simulation time advances and event synchronization. This will be discussed in section 2.5.
2. Provide the endpoints for all the broken links, and transfer events between federates using RTI services.

Further, in our implementation message format transformations are performed by the proxy. Packets destined outside the local sub-model are transformed from their native format representation internal to the sequential simulator, to a common message format used by the proxy. Correspondingly, messages from other federates are transformed into the native format by the proxy as well. While not strictly necessary for federating a simulation package with itself, this capability is important when federating different network simulation packages.

In our design a modular approach is used where the proxy model is decomposed into two parts. One is a protocol independent component called the *gen_proxy* and the second is a set of protocol dependent components called the *pro_proxy*. The protocol independent functions that are implemented in *gen_proxy* include event and time management. Every network protocol that is modeled has one *pro_proxy* in the proxy model to process protocol packets. For example, IP is associated with the *ip_proxy* component. All the broken links carrying the same kind of protocol packets will be connected to their unique *pro_proxy*.

Every broken link is mapped to a data channel in the proxy model. Riley et al. introduce the concept of *rlink* in (Riley et al. 1999) for this purpose. Here, data channels provide a similar function as *rlink*. A channel is unidirectional or bi-directional depending on the link that it is modeling.

In our implementation channels are implemented using the HLA declaration management services. A publishing class represents an outgoing data flow while a subscribing class represents an incoming data flow. A bi-directional channel is implemented with a publishing and a subscribing class on each side while a unidirectional channel is implemented with a publishing class on the sending side and a subscribing class on the receiving side. We use the addresses of the endpoints as the names of the publishing classes, since they are unique within the entire network model and are easily associated with links. In the sample network of Figure 2, for the link between R0 and R3, the publishing class names are 192.0.1.1 on Federate A and 192.0.1.2 on Federate B. Federate A subscribes to class 192.0.1.2 while federate B subscribes to class 192.0.1.1. A bi-directional channel is constructed across federates through RTI.

The next step is to instruct all the node objects adjacent to the proxy model to route data traffic destined for other sub-models to the proxy model. In the case of IP traffic, this can be realized by modifying the routing table. For example, in Figure 3, R0 is instructed to route all the packets destined to address 202.0.3.* to next hop 192.0.1.11, instead of the nonexistent 192.0.1.2. Then the proxy model can be responsible for forwarding packets across federates through data channels.

2.5 Simulation Time and Event Management

The proxy model interfaces the simulator (federate) to the RTI's time management services. The proxy must synchronize local simulation time of each simulator with that of others.

In sequential discrete event simulation, unprocessed events are stored in an event queue and processed in time stamp order. In parallel simulation, a federate cannot autonomously advance its local simulation time because this might result in receiving an event in its past, i.e., the time stamp of an incoming message may be smaller than the local time of the federate. Every federate has to process events, both those generated locally and those generated by other federates, in time stamp order. This is the well-known *synchronization* problem. When conservative synchronization is used, each federate must wait until it can be sure that no events will arrive in its past. A global consensus protocol can be used to compute the lower bound on time stamp (LBTS) of messages the federate may later receive. Events with time stamp less than LBTS are called safe events because they can be safely processed without concern that a smaller time-stamped event might later arrive. No federate can safely advance its local simulation time beyond its current LBTS value. This will guarantee no event will later arrive in the federate's past. With this approach, each federate repeatedly cycles through "phases" of (1) processing safe simulation events, and (2) waiting for its LBTS value to advance so that more safe events can be identified. LBTS computations and the protocol for advancing simulation time are implemented by the time management services of the RTI.

Here, because we are dealing with an existing sequential simulation executive that cannot be modified, each federate can only process events stored in its local event queue. A mechanism is required to allow a sequential simulator to process externally generated events. For this purpose we define a checking algorithm that executes within the proxy model. The proxy model schedules checking events that are inserted into the simulator's local event queue so that the sequential simulator will process these checking events in the same manner as any other local event. Each checking event results in the execution of the proxy model. The proxy model will, in turn, receive and process external events and identify safe events in the local event queue. As stated earlier, one assumption made for this algorithm is that we have the access to the event queue. This is a reasonable assumption since the simulation application should be able to schedule, cancel and search for events. The following steps describe the checking algorithm:

1. Step through the local event queue from the current event forward to identify all the events with time stamp less than the current LBTS value. If any such events are found, go on to step 2, otherwise go to step 3.
2. The events with time stamp less than LBTS found in step 1 can be safely processed. Then schedule the next checking event at the time stamp of the last safe event identified with the lowest priority. This means checking events will be processed last among all the events with the same time stamp. The algorithm terminates at this point.
3. Invoke the HLA *NextEventRequest* service to request advancing simulation time to the time stamp

t of the next event in the local event queue. This is necessary to identify any pending external events that exist preceding the next internal event. If no pending external events exist before *t*, go on to step 4, otherwise go to step 5.

4. The simulation time will be advanced to *t*. Then schedule the next checking event at time stamp *t* with lowest priority. The algorithm terminates at this point.
5. If there are any unprocessed external events with time stamp less than *t*, (let the earliest such event e_1 have time stamp t_1), the RTI will advance the federate's local simulation time to t_1 and issue a *Time Advance Grant*. Event e_1 will be delivered to the federate and can be processed. The time stamp of the next internal event may or may not be *t* because the processing of e_1 may or may not generate new local events with time stamp less than *t*. Go back to step 3 and repeat step 3-5 again.

In step 3 above, the RTI will typically initiate or participate a new LBTS computation since the requested time stamp advance is beyond the current LBTS value. The events in the local event queue between two checking events can be processed safely and all events will be processed in time stamp order.

With this method, the sequential simulator can ensure all the local events are processed in time stamp order. The RTI guarantees all external events are delivered to the federate in time stamp order. Using our algorithm, a local event is identified to be safe for processing only after all external events preceding this local event have been delivered and processed during the processing of a checking event. Thus, all local and external events are processed in time stamp order.

2.6 Performance Related Issues

In this section, we discuss two issues that have a significant impact on performance: lookahead and shared information among sub-models.

Lookahead is a concept used to improve performance in parallel/distributed simulations. If a federate's lookahead is *L*, then it guarantees it will only generate messages at least *L* units of simulation time into the future. A larger lookahead allows more parallelism. In parallel network simulation, lookahead is defined as the minimum of the packet delivery delays in all the links of a sub-model that cross boundaries of the partition. The packet delivery delay is the sum of transmission delay and propagation delay. Usually, propagation delay is an attribute of the link object while transmission delay is data packet size divided by link data rate. In order to improve the lookahead capability, we delegate the delay computation of all broken link objects to the proxy model. In place of the normal delay computation, these link objects attach only the relevant information, such as propagation delay and link data rate, to the packet. When the proxy model (actually in *gen_proxy*) receives the packet, it can compute the actual delay and thus predict the simulation time at which the packet arrival occurs on the receiving side of the link. The federate can declare the minimum of all the delay values as its lookahead. This can be computed prior to the execution of the federation.

There are some cases where one federate may need model information beyond its own sub-model. For example, some simulators (e.g., ns) may need to compute routing information during the execution of the federation. This requires the entire network topology to be defined and shortest path routes to be computed on each federate. Another case is a mobile device needing to know whether it is within the power range of a base-station, which may not be in the local sub-model. If so, it needs to subscribe to the data sent by the base-station. These cases require the duplication of the network model in different processors, and may require a large amount of memory. We introduce the concept of a *ghost object*, which is a reduced state version of the object consuming little space. A *ghost object* includes a subset of real object attributes. These attributes can be updated dynamically by the federate responsible for simulating the object through the HLA declaration management services or data distribution management services. The drawback of this approach is that such updates usually result in zero lookahead interactions, which may severely degrade performance. Detailed analysis of the nature of each such attribute is needed to determine whether we can exploit some lookahead capabilities in this case. For example, the attributes of a link can be updated with some delay because a remote node cannot know the link state changes immediately in real network operations.

3 CASE STUDY: BUILDING A PARALLEL OPNET SIMULATION

We have implemented our approach by building a parallel OPNET simulation. We compare the performance of the parallel OPNET simulation with the sequential simulation.

3.1 OPNET Overview

OPNET is a commercial network simulator marketed by OPNET, Inc. First developed at MIT, OPNET was introduced as a commercial network simulator in 1987. OPNET comes with four major components:

1. An event-driven simulation engine. The simulation executive manages an event queue and processes events in time stamp order.
2. A set of application interfaces which are implemented as C libraries. Users can create custom simulation models by utilizing these interfaces.

3. Graphical tools and commands. Graphical tools provide a drag-and-drop style of programming.
4. A large library of network protocol models covering many standards and equipment models from major equipment suppliers.

3.2 Implementation

The Federated Simulations Development Kit (FDK) software package developed at Georgia Tech was used for this work (Fujimoto et al. 2000). Specifically, we used FDK's Basic RTI (BRTI), which provides event and time management services.

In our parallel OPNET simulation, the proxy model is implemented as a *gen_proxy* node model and a set of protocol dependent *pro_proxy* node models. When the OPNET package is updated, the *pro_proxy* may need to be updated (depending on the changes made to OPNET), but changes to *gen_proxy* should be minimal. The behavior of a link object in OPNET is defined by a series of consecutively executed procedures called pipeline stage procedures, e.g., computing propagation delay. We modified the pipeline stage procedures of broken links to delegate the delay computation to the proxy model. Instead of computing delay, the pipeline stage procedures we implemented store relevant information, such as link delay and data rate, in Transceiver Pipeline Data Attributes (TDAs) of the packet and return the delay as zero. The proxy model is responsible for computing the real delay value and predicting the arrival events accordingly.

One major problem we encountered in building the parallel OPNET simulation has to do with OPNET's use of *global state* information. Specifically, sequential simulation models in OPNET assume the existence of certain global data structures. For example, every IP process model registers its address information in global tables shared by all the IP models during initialization. When the IP model of one node is going to send a packet to another node, it will first check whether the address of the destination node is valid by a lookup in these global tables. Our current approach to solve this problem is to utilize the *ghost objects* (as discussed earlier) to initialize these global data structures. This ensures each federate has consistent global state information. This global state information must be static, i.e., it cannot be modified during the execution. The drawback of this approach (in addition to not supporting modifiable state) is detailed analysis of the model code is required to determine what initialization functions must be kept in every *ghost object*.

Another problem has to do with OPNET's communication mechanisms. Packet-based communications are not the only means of communication in OPNET. Several types of interrupts, e.g., self, remote or multicast interrupts, allow one module to schedule events to any module of the entire model, which makes the prediction of events very hard. For example, some OPNET application models utilize remote interrupts to simplify service start and end simulations. Naively converting such interrupts into interactions across the RTI usually results in zero lookahead. Thus detailed analysis of the model code is required to be able to predict non-packet-based communications so as to improve the lookahead capability.

Global state, zero lookahead interactions, and pointer data structures lead to dependencies between elements of the simulation that are hard to identify, making parallelization difficult and/or time consuming. Such problems have been observed by others, e.g., see (Bagrodia 1996). Because of these problems, at present, only a few simple protocol models (e.g., UDP and IP) in OPNET have been parallelized in our current implementation.

3.3 Performance

We constructed two sets of experiments, each simulating three minutes of network operation to evaluate the performance of the prototype system. In the first set, we simulated a network model called a *"regular traffic model"*. It consists of eight subnets, each of which contains eight hosts and one router. Each host is both a data source and sink. It sends and receives data to and from all other subnets. The data sinks are chosen uniformly. The inter arrival time of traffic generated by each host is exponentially distributed. In the second set of experiments we simulated a network model called the *"added traffic model"*. On the base of the *"regular traffic model"*, we added sixteen internal hosts to every subnet, which only transmit data within the local subnet. The parallel simulation was executed on a set of eight Sun Ultra Sparc 1 Model 170 systems, each with 64MB main memory and a 167Mhz Sparc CPU. The Ultra Sparcs are connected by a 100Mb Ethernet. Each subnet is mapped into a Sun system. As the baseline, the entire network model is also simulated in a single system.

Several variations of simulation parameters were used in the experiments. First, the mean inter arrival time of generated packets, which determines the density of events (number of events within a fixed sized window of simulation time), was varied from 1s to 0.05s. Second, the propagation delay ranged from 0 to 1s. This is an important parameter because it determines the lookahead between federates as previously discussed. We varied the propagation delay to allow for different lookahead values.

The results for the *"regular traffic model"* are shown in Figure 4. Figure 5 shows the results for the *"added traffic model"*. The X-axis is the inter arrival time of generated packets. The Y-axis is the speedup factor. Speedup is defined as the execution time taken by baseline sequential simulation divided by the execution time of the parallel simulation, which is executed on 8 simulators in our experiments. The different lines correspond to different propagation delays.

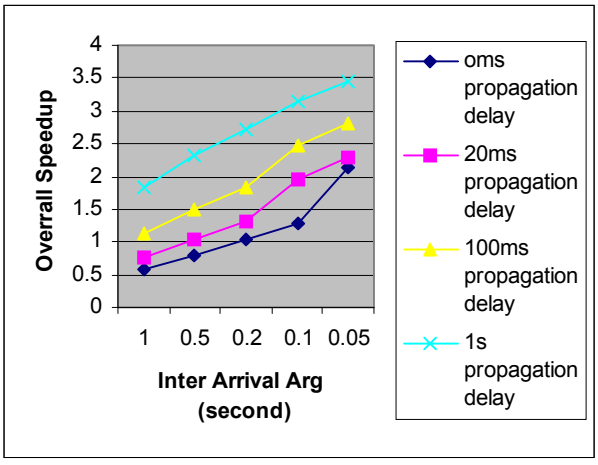

Figure 4: Overall Speedup Factors, Regular Traffic

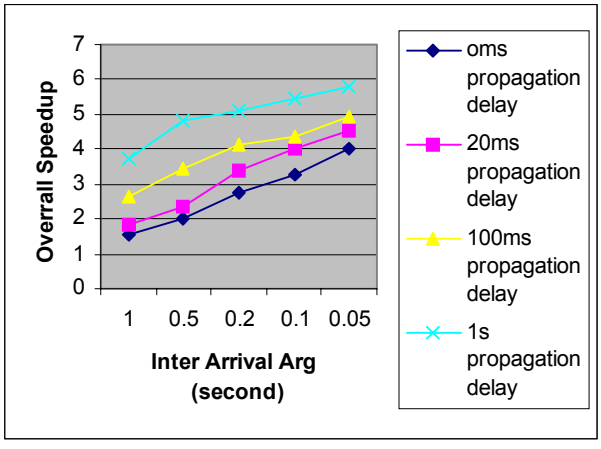

Figure 5: Overall Speedup Factors, Added Traffic

Computation and communication overheads in the parallel execution arise from several sources: LBTS computations, message transmissions over the network, and extra computations to process checking events. The performance results reflect these overheads. First, increased lookahead values do speed up the parallel simulation. This is expected, and can be attributed to a reduced number of LBTS computations. When the propagation delay reaches 1s, fewer than 300 LBTS computations are performed. Second, while fixing the propagation delay we get better performance by reducing the inter arrival time. Reducing inter arrival time means the event density increases. This in turn increases the amount of simulation computations that can be performed between LBTS computations, resulting in improved speedup. On the other hand, increasing the event density also increases the number of message transmissions between federates. Since performance improves with increased message density, we conclude that LBTS computations are much more expensive. Third, by comparing Figure 4 and Figure 5, we note that we observe better performance when the proportion of local events is increased. This is not surprising, since the parallel simulation overheads are amortized among local events.

The performance results are summarized below:

1. Performance improves as lookahead increases. The lookahead should be as large as possible. In the context of network models, lookahead represents the smallest amount of simulation time required for a packet to be transmitted from one federate to another. To increase the lookahead, we need to either partition the network model at links with low bandwidth, or increase the distance between the subnets mapped to federates.
2. Increasing event density (i.e., message traffic) helps to amortize the cost of parallel simulation overheads, and results in better performance.
3. Improving traffic locality leads to reduced costs related to transmission of messages between federates, results in better performance.

4 CONCLUSIONS

We have presented an approach to build a parallel network simulation using existing commercial event-driven simulators where only limited access to the sequential simulation code is provided. This method is relatively straightforward to implement if the original simulator does not make extensive use of global state variables and packet-based communications are the major means of communication. Most of the modifications to the sequential simulator are implemented in a proxy model that interfaces the simulator to the RTI. This approach allows federation of heterogeneous simulators, although this aspect has not yet been implemented in the prototype. Our prototype using the OPNET simulation package yielded reasonable speedup for parallel simulations with good lookahead. But issues such as global state and zero lookahead interactions make parallelization much more difficult for more complex protocol models such as TCP.

Recently, an HLA module has been introduced in OPNET 7. This provides a similar architecture to that used for our parallel simulation, but utilizes native support within the OPNET simulation engine. The current release doesn't provide distributed network models but requires users to develop their own distributed version of models. By contrast, our approach focuses on reuse of currently available models.

REFERENCES

Bagrodia, R. L. 1996. Perils and Pitfalls of Parallel Discrete-Event Simulation. In *Proceedings of the 1996 Winter Simulation Conference.*

Chang, X. 1999. Network simulations with OPNET. *Proceedings of the 1999 Winter Simulation Conference.*

Christiansen, M., K. Jeffay, D. Ott and F. D. Smith. 2000. Tuning RED for Web Traffic. *ACM SIGCOMM*, Stockholm, Sweden.

Cowie, J. H., D. M. Nicol, A. T. Ogielski. 1999. Modeling the Global Internet. *Computing in Science and Engineering.*

Defense Modeling and Simulation Office. 2000. *High Level Architecture*. `http://hla.dmso.mil`.

Ferenci, S., K. Perumalla, R. M. Fujimoto. 2000. An Approach to Federating Parallel Simulators. 14th Workshop on Parallel and Distributed Simulation.

Fujimoto, R. M., T. McLean, K. Perumalla, I. Tacic. 2000. Design of High-performance RTI software. In proceedings of Distributed Simulations and Real-time Applications.

Jones, K. G. and S. R. Das. 2000. Parallel execution of a sequential network simulator. In *Proceedings of 2000 Winter Simulation Conference*.

McCanne, S. and S. Floyd. 1997. *The {LBNL} Network Simulator*.

Nicol, D. and P. Heidelberger. 1996. Parallel Execution for Serial Simulators. *ACM Transactions on Modeling and Computer Simulation* **45**(6): 210-242.

Nicol, D., M. Johnson, A. S. Yoshimura, M. E. Goldsby. 1998. IDES: A Java-based Distributed Simulation Engine. *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems.*

Perumalla, K., A. Ogielski, R. Fujimoto. 1998. TeD - A Language for Modeling Telecommunications Networks. *Performance Evaluation Review* **25**(4).

Riley, G., R. M. Fujimoto, M. H. Ammar. 1999. A Generic Framework for Parallelization of Network Simulations. *Seventh International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*.

Strabburger, S. 2000. *Distributed Simulation Based on the High Level Architecture in Civilian Application Domains*. Ph.D. thesis.

Unger, B. 1993. The Telecom Framework: a Simulation Environment for Telecommunications. In *Proceedings of 1993 Winter Simulation Conference*.

Zeng, X., R. Bagrodia, M. Gerla. 1998. GloMoSim: A Library for Parallel Simulation of Large-Scale Wireless Networks. Workshop on Parallel and Distributed Simulation.

AUTHOR BIOGRAPHIES

HAO WU is Ph.D. student with the College of Computing at the Georgia Institute of Technology. He received his M.S. from Beijing University of Posts & Telecom in 1997 (Computer Engineering). He received his B.S. from Beijing University of Posts & Telecom in 1994 (Electrical Engineering). His current research interests include large scale distributed and parallel simulations and emulations, network simulations, distributed computing and real-time systems. His email address is <wh@cc.gatech.edu>.

RICHARD M. FUJIMOTO is a professor with the College of Computing at the Georgia Institute of Technology. He received the Ph.D. and M.S. degrees from the University of California (Berkeley) in 1980 and 1983 (Computer Science and Electrical Engineering) and B.S. degrees from the University of Illinois (Urbana) in 1977 and 1978 (Computer Science and Computer Engineering). He has been an active researcher in the parallel and distributed simulation community since 1985 and has published numerous papers on this subject. He has given several tutorials on parallel and distributed simulation at leading conferences. He has coauthored a book on parallel processing and recently completed a second on parallel and distributed simulation. He served as the technical lead in defining the time management services for the DoD High Level Architecture (HLA). Fujimoto is an area editor for ACM Transactions on Modeling and Computer Simulation. He also served as chair of the steering committee for the Workshop on Parallel and Distributed Simulation, (PADS) from 1990 to 1998 as well as the conference committee for the Simulation Interoperability workshop (1996-97).

GEORGE RILEY received his Ph.D. from the Georgia Tech College of Computing in August 2001, and is presently an Assistant Professor at Georgia Tech College of Engineering, School of Electrical and Computer Engineering. His research interests are in large scale distributed simulations, computer networks, and distributed computing.