Parallel Simulation Techniques for Large Scale Networks

Sandeep Bhatt, Richard Fujimoto, Andy Ogielski, Kalyan Perumalla

Abstract

Simulation has always been an indispensable tool in the design and analysis of telecommunication networks. Due to performance limitations of the majority of simulators, usually network simulations have been done for rather small network models and for short time scales. In contrast, many difficult design problems facing today's network engineers concern the behavior of very large, hierarchical multi-hop networks carrying millions of multiprotocol flows, over long time scales. Examples include scalability and stability of routing protocols, packet losses in core routers, or long-lasting transient behaviors due to observed self-similarity of traffic patterns. Simulation of such systems would greatly benefit from application of parallel computing technologies, especially now that multiprocessor workstations and servers have become commonly available. However, parallel simulation has not yet been widely embraced by telecommunications community due to a number of difficulties. Based on our accumulated experiences in parallel network simulation projects, we believe that parallel simulation technology has matured to the point that it is ready to be used in industrial practice of network simulation. This article highlights recent work in parallel simulations of networks and its promises.

1 Introduction

Network engineers and researchers routinely use simulations in their daily network design and analysis tasks. Simulation provides a practical methodology for understanding system behaviors that are either too complex for mathematical analysis, or too expensive to investigate by measurements or prototyping, or both.

With the emergence of global multi-hop packet networks and Gigabit/s links the network simulation community is faced with significant challenges. First, actual packet traffic is dominated by long-range correlations (first characterized by [1]), thus realistic models have to be simulated for very long time scales to avoid misinterpreting long-lasting transient behaviors. Second, network configurations of really large size have to be simulated to study issues such as scalable routing, survivability, or packet loss correlations in multi-hop, multi-domain networks. Such features just cannot be captured in small network models. The immediate need for such extensive modeling capabilities for planning, growth management, and network management has been, for instance, repeatedly stressed in a recent multi-agency white paper on the Next Generation Internet Implementation Plan [2]. In that document, the milestones include requirements for network planning which must be verified by simulation, for a 100,000 node, five protocol layer network by the end of year 2000, and for a 10 million node, seven protocol layer network in year 2001.

It is widely acknowledged that the capabilities of conventional sequential simulation techniques are inadequate to address such simulation requirements, and that *parallel* simulation techniques must be brought to bear on these challenges. However, parallel simulation techniques are not yet commonplace in network simulation. Lack of established, easy to use modeling methodologies suitable for parallel execution, absence of mature software environments and comprehensive feasibility demonstrations have so far prevented the widespread use of parallel simulations in network research and industrial practice.

Parallel simulation is the process of using multiple processors simultaneously for executing a single simulation, with the goal of reducing the total execution time. As we will demonstrate later, for large and complex networks the parallel simulation can indeed reduce the simulation time to tolerable levels, such as from several days down to a few hours. Importantly, parallel simulations can be performed on commodity machines such as multiprocessor personal workstations and servers, and on workstation clusters, all of which are now widely available in research and industry.

Parallel simulation techniques are inherently difficult, but have been intensely researched for over a decade, and now are opening up the possibilities for their exploitation in production use. To further their application to network engineering, a comprehensive feasibility demonstration is now necessary, coupled with the introduction of mature software environments for routine parallel network simulations. We believe that presently both requirements are getting satisfied, thus clearing the way for industrial strength products.

The feasibility of applying parallel simulation techniques to diverse, large and complex network models has been recently illustrated in [7] and elsewhere with the TeD simulation framework that is described here. Also, several other good, research-oriented parallel network simulators are available, such as UCLA's Maisie/Parsec [3], or the TeleSim project in Canada and New Zealand [4]. For promise of success, usability requirements are paramount in such systems. To promote widespread industrial use, the systems must support a framework for model specification, development, testing and maintenance, which must be natural for network engineers, portable across multiple platforms, and successfully exploit the potential of high performance via parallel execution.

We and our colleagues are currently involved in projects to develop and stress-test a robust network modeling and simulation framework aimed at large-scale modeling of networks. These projects also serve as an exercise in exposing the problems that may be encountered in production-quality parallel network simulation efforts, and addressing their resolution. The end-goal of this work is the realization of a mature, high-usability software suite for parallel simulation of networks. Although much work still remains, we believe that our current experiences can serve to support the feasibility and usability claims.

In our approach to the development of the software framework for network simulations, we started with the premise that the value to a user lies in the model descriptions, and the results of the simulations, and not in the simulator engine. Our main objective, therefore, has been to shield the modelers from the underlying complex simulation mechanisms and computing platforms, and to investigate the canonical model design patterns that can routinely speed up the simulations. This article describes the salient features of our framework, and our experiences in applying parallel simulation technology in the framework. We illustrate its use with an example of parallel simulations of ATM/PNNI internetworks. The discussion is limited, for reasons of space, to fairly high-level; interested readers are referred to [7] for details.

2 The Layered Approach

Modern simulation software is composed of several layers (see Figure 1). Each layer provides an API (Application Programming Interface) to the layer above it; provided that the semantics of the API remain unchanged, layers can be maintained independently. The parallel simulation kernel provides a framework (in the software engineering sense) that ensures correct parallel execution for any model built in a layer above it. The modeling framework layer supplies a parsimonious API for model development, without compromising parallel performance. The extensible model layer provides canonical designs for network elements and protocols that can be recursively composed to build models of very large and complex systems.

Our discussion is focused on one operational parallel simulation system, TeD/GTW. TeD, described in more detail below, is a small language expressing a natural modeling framework, that is transparently mapped onto a high performance parallel simulation kernel, the Georgia Tech Time Warp (GTW)[5]. TeD itself is independent of the underlying parallel simulator, and can be used with other parallel simulators, such as **Nops** that has recently been developed at Dartmouth [11]. The TeD/GTW software executes on multiprocessor Sun and SGI machines.

Deciding how to abstract the real system behavior of interest into a simulation model is an art in itself. Assuming the abstract system behavior is decided separately, we focus on naturally expressing the behavior, and efficiently executing it. It is worth stressing that design of a good modeling interface is an iterative feedback process, involving software designers and model developers. Only many large trials can ensure that the modeling framework evolves towards a

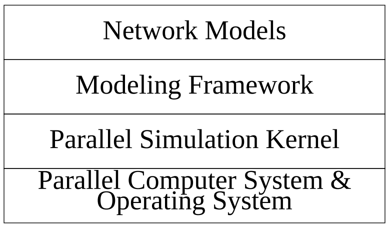

Figure 1: Software layers in a parallel network simulation system.

sufficiently expressive and easy to use system. Besides the modeling examples described here, our colleagues are using the TeD/GTW system to model and investigate TCP/IP networks [9], wireless networks [8], reliable multicast protocols [10], and communication scenarios for defense applications. Experimentally verified traffic models are developed at Boston University.

3 Parallel Discrete Event Simulations for Network Modeling

Network simulations can benefit from parallel processing because network operations are inherently distributed and concurrent. Unlike scientific applications, the traditional focus for high–performance parallel computing, sophisticated algorithms to discover parallelism are not required; rather, concurrency is specified explicitly in the model description. For example, Figure 2(a) shows a portion of a network containing four elements: two traffic sources, a multiplexer, and a switch. Software description of the behavior of each element could be compiled to simulation code that executes on a separate CPU. Although specification of the model to explicitly represent concurrency is not always so straightforward, once a suitable representation has been developed, the design ideas can be successfully reused over and over again. Another important distinction from parallel scientific computing is that managing concurrency is conceptually quite different, as will be discussed next.

A parallel discrete event simulation program can be viewed as a collection of interacting sequential simulators, called logical processes (LPs). The computation performed by each LP is a sequence of event computations, where each event represents some interesting action in the model, e.g., the arrival of a new packet on an input link. Each event contains a time-stamp indicating when that event occurs. LPs interact by scheduling events for each other. For example, Figure 2(b) shows LP A (a traffic source) scheduling an event for LP B (the multiplexer) to indicate a new packet has arrived at the multiplexer at simulation time 100.

Each LP must process its events in increasing time-stamp order to correctly reproduce temporal relationships in the simulated system. Ensuring time-stamp ordered event processing is non-trivial. Consider the situation depicted in Figure 2(b). LP B is ready to process the arrival event with time-stamp 100, but how does it know LP C, now at simulated time 80, will not send it a new event with time-stamp less than 100? Ensuring time-stamp ordered processing of events is a fundamental problem in parallel discrete event simulation.

Two methods are commonly used for process synchronization: *conservative* and *optimistic* algorithms. In the conservative approach an LP can only process an event when it has a guarantee no other event containing a smaller time-stamp may arrive at some later time. For example, if the behavior of LP C in Figure 2(b) were constrained so that it could not generate any new events with time stamp less than 100, e.g., because the minimum transmission delay to send a packet over a link is at least 20 units of simulation time, LP B could safely process the arrival event at time 100. Constraints regarding the time-stamp of new events scheduled by an LP are referred to as *lookahead constraints*. Lookahead is clearly model dependent, and constructing the model to adhere to lookahead constraints places certain burdens on the modeler.

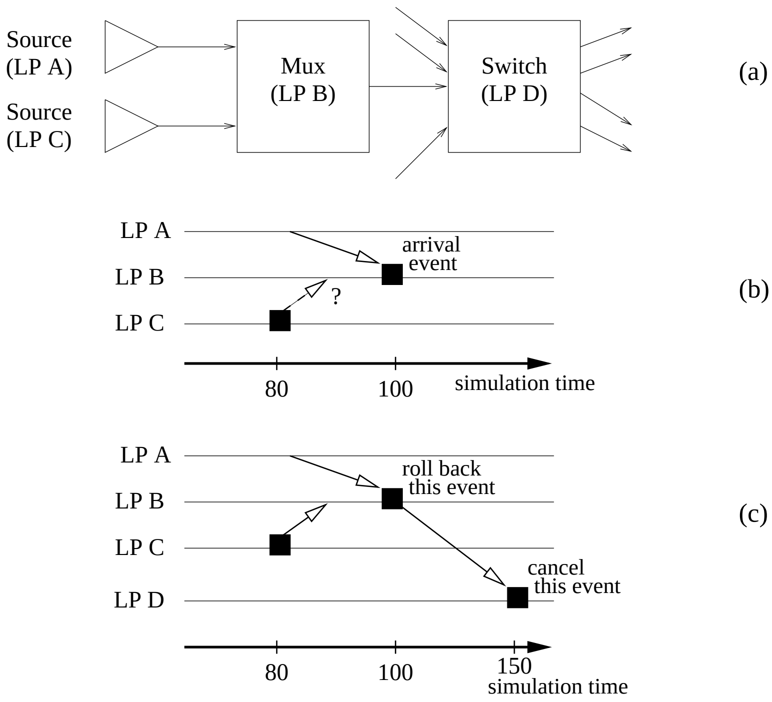

Figure 2: Synchronization in a parallel simulator. (a) simulated network. (b) conservative synchronization. (c) optimistic synchronization.

A possible drawback of this approach is that LPs may be forced to wait unnecessarily until they acquire a guarantee of time-stamp ordered event processing. LP B must block if LP C *could* send an event with time-stamp less than 100; if LP C does not actually send such an event, this blocking was unnecessary.

In contrast, optimistic mechanisms do not prevent out-of-order event processing, but rather LPs process events as they arrive without concern of potential causality violation. Should an out-of-order event arrive, the LP uses a clever rollback mechanism to recover from potential error. In Figure 2(c) LP B has processed the time stamp 100 event, resulting in a new arrival event being scheduled at LP D with time-stamp 150. If LP B now receives an event with time-stamp 90, the processing of the time-stamp 100 event must be rolled back. This means state variables modified in processing this event must be restored to their original value, and the message sent to LP D must be canceled. Checkpointing and incremental state saving techniques are usually used to undo modifications to state variables. Message cancellation is accomplished by sending a special *anti-message* to cancel the incorrect message. If the message being canceled has already been processed by the receiver (LP D in Figure 2(c)), the receiver must also roll back, possibly resulting in the generation of additional anti-messages and rollbacks. This approach avoids the blocking associated with conservative execution, but at the cost of additional computational overheads to realize the roll back mechanism.

In general, these are complex techniques, and it has taken years of research to investigate them and develop practical realizations. This is the primary reason for hiding the internals of parallel simulation software from network modelers. We refer the interested reader to [6] for an extensive discussion of parallel simulation algorithms and techniques.

4 The Modeling Framework

The communicating logical processes managed by a parallel simulation kernel are fairly low level constructs. They are certainly not at the level of aggregated concepts such as a router, switch, protocol stack or communication channel that are employed by network designers and modelers.

A successful parallel simulator must provide a modeling interface layer which is more convenient for representing network elements. To illustrate the constructs provided in this layer, we describe a specific object-oriented framework called TeD. Another approach to defining the modeling layer is described in [3].

TeD has been designed to facilitate network modeling in a manner that exploits the models' latent concurrency for efficient parallel simulations. TeD allows to design new network element or protocol models as well as to reuse the "black box" designs created by others, all within the *same* carefully designed small modeling framework with rich semantics. This permits modeling of all network elements and protocols, and does not limit the possibility of modeling future systems that were not imagined by the designers of TeD.

TeD supports a modeling view that has syntactic resemblance to successful hardware design languages such as VHDL, but with a significantly different semantics. Technically speaking, TeD is a small language expressing the design pattern formalized in the modeling framework. TeD uses an approach of carefully separating the core modeling constructs from the executable constructs. The modeling constructs are defined in terms of a meta language. These constructs provide scope for enclosing executable code fragments written in any general-purpose programming language. The current TeD compiler accepts executable content expressed in C^{++}, with a macro interface to mediate between the executable language and the runtime simulator services.

The advantages of this separation are that the meta language is useful for naturally capturing the structure of models and enforcing requirements necessary for efficient execution on a parallel computer (e.g., the avoidance of global variables), while existing software development tools such as compilers and debuggers can be brought to bear on the general purpose language used to specify the executable content.

4.1 TeD Abstractions

In the following, the principal TeD language constructs and semantics are described. More complete, detailed documentation and examples can be found in the language manuals.

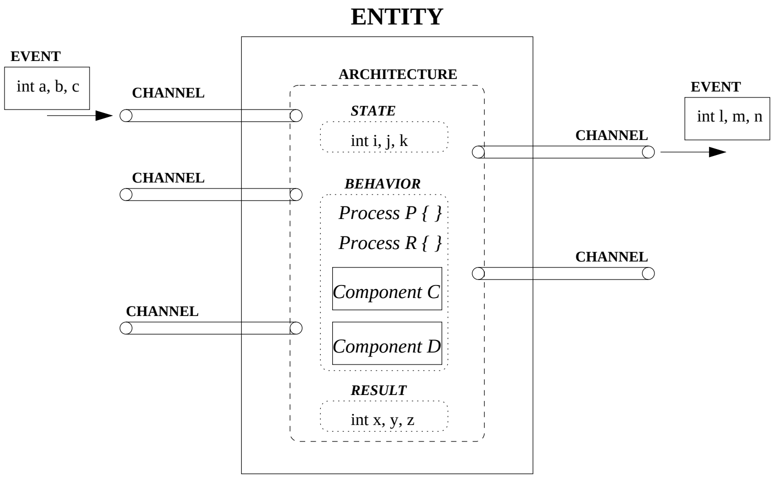

Figure 3: Basic framework of an Entity

Figure 3 illustrates the basic framework underlying the constructs in TeD, and their relation to each other. Figure 4 illustrates the basic elements of TeD in the context of a simple ATM multiplexer model.

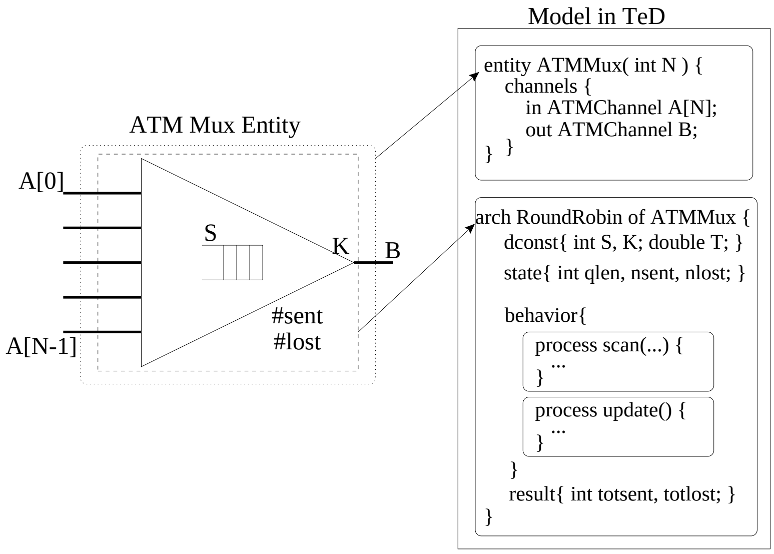

Figure 4: Illustration of a physical entity and its model in TeD

Entity

The physical and conceptual objects in the telecommunication domain are modeled in terms of *entities*. Entities are connected to each other by typed *channels*. Channels transmit the *events* that describe dynamic interactions between entities. A channel is a port of input or output for an entity. An output channel of an entity can be mapped to an input channel of another entity. A channel type is defined as a set of event types.

The *entity* declaration is used to define a black-box view of an entity. The entity could be a model of a real-life object, such as a multiplexer, or of an abstract object, such as a protocol. The behavior of an entity is defined solely in terms of its dynamic reactions to events on its input-mode channels, and the production of events on its output-mode channels. Special semantics are supported for entity inheritance and enhancement.

Architecture

The dynamic behavior of each entity is described by its *architecture*. The architecture of an entity is expressed using concurrent process semantics. It describes the actions of the entity upon event arrivals on its input channels, and the production of events on its output channels. One or more *processes* can be defined to act upon events arriving on the input channels of the entity. The processes may generate events on output channels as part of their computation.

The architecture of an entity can also be expressed as a composition of interacting *components*. Components are themselves entities that are "logically enclosed" inside the surrounding entity. The entity format of Figure 3 is thus recursive in the components, allowing the construction of hierarchical entities. The channels of the components can be arbitrarily mapped among each other or to the channels of the enclosing entity. Thus, the behavior of an entity can be described in terms of its structure (components) or dynamics (processes), or a combination of both.

The architecture of an entity is composed of a small number of code blocks. Their simplified description is as follows:

- **Deferred Constants**: Variables whose values are held constant during runtime, but can be initialized to different values for different architecture instances.
- **State**: A set of variables that together form a part of the abstract state of the modeled entity.
- **Processes**: Threads of computation that act on the events arriving on the interface and internal channels, or that act upon time advances.
- **Components**: A set of entities that *logically* form subentities of an entity's behavior.

Structural and behavioral descriptions of an entity can be inherited, thus allowing object-oriented hierarchical design and development. A derived architecture inherits all the items of the parent architecture (state, processes, etc.), which can also be redefined (overridden).

Process

Processes are dynamic threads of computation acting on behalf of the entities that *own* them. Only the *owner* entity's channels (internal and external) and state variables are accessible to the entity's processes. The functionality of processes consists of combinations of two types of actions: *computation*, and *synchronization*. Computation is a sequence of operations performed on the state variables. Synchronization is a sequence of actions involving events on the channels or the lapse of time (wait).

Computation actions are specified in the executable language. Synchronization actions are specified using different variants of the **wait** statement in the process body. According to the TED semantics, the processes never stop executing, but repeatedly suspend and resume their activity depending on arrival of events or the timeout of a **wait** statement. TED processes behave like conventional thread computations, but with the capabilities to make nested procedure calls and invoke the **wait** statements at arbitrary points in the process and procedure body.

4.2 Simulation

TED models are designed without reference to any specific parallel simulation engine. TED models are thus translated by a special TED compiler that generates the C++ code which can directly use the services of the parallel simulation kernel. In our projects both optimistic and conservative synchronization kernels have been implemented at TED targets (GTW and Nops). The generated C++ code is then compiled with a standard C++ compiler and linked with the parallel simulator.

Once the models TED objects have been compiled, a separate *configuration* specification is used to instantiate a desired network model configuration, thus avoiding recompilation when a modeler wishes to change model parameters, or even a modeled network topology. The network thus instantiated can now be simulated on a multiprocessor machine.

We have successfully used TED to model a variety of networks and protocols, including ATM, Internet, and wireless. For reasons of space, we will use only our work on the ATM PNNI protocol simulations to illustrate the modeling process and the achieved parallel performance.

5 An Example: Parallel Simulations of PNNI Internetworks

We have built a TED simulator for the ATM Private Network-Network Interface (PNNI) suite of protocols. The PNNI protocols (ATM Forum 1996) is an international draft standard proposed by the ATM Forum for ATM internetworking. It supports protocols for topology discovery and (re)configuration, and for dynamically routing virtual circuit connections. The PNNI suite includes the most sophisticated signaling protocols devised to date, with the goal of supporting QoS–based routing for global–scale networks.

We are using the virtual PNNI testbed to study tradeoffs between scalability and network performance. For the network sizes and time scales of interest, high-performance parallel and distributed simulation engines are essential. It is equally important that the underlying models be reusable and independent of simulation-engine code. This will allow us to leverage advances in parallel discrete-event simulation technology without having to rewrite the models.

5.1 PNNI Overview

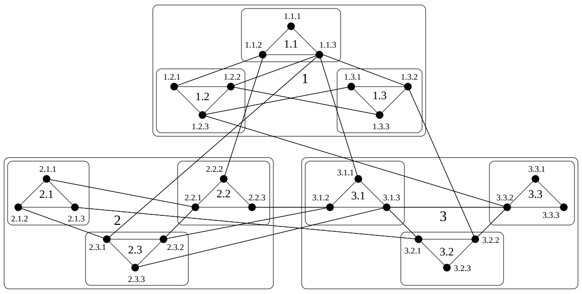

Figure 5: An Example PNNI Network

The PNNI protocols construct a logical hierarchy atop a physical ATM network. The hierarchy provides aggregate views of the current network topology and state to ATM switches at the bottom of the hierarchy. Figure 5 shows a PNNI hierarchy with 4 levels. Nodes at all levels are organized into peer groups. Each peer group selects a leader and these are grouped together into higher level peer groups. Peer group leaders (PGLs) help establish and maintain topology views for network switches; they are not used for call routing.

The hierarchy helps control the propagation of changes in the network state; both processing time and storage space are reduced at the possible expense of the fidelity and timeliness of the views of the network at a switch. Each switch obtains complete knowledge of the switches and links within its peer group by a flooding mechanism. At higher levels, PGLs within the same peer group exchange aggregated information and transmit this information to their descendants. Each switch thus has detailed information about nearby switches but approximate information about more distant nodes. This exchange of state information is carried out using short `Hello` messages and longer PNNI Topology State Exchange (`PTSE`) messages. The state information is updated periodically, and is used for distributed signaling, routing and call admission, call setup and call teardown.

The call admissions and routing algorithm at each switch must make tentative route assignments based on two kinds of approximate information: the aggregated view of network topology, and the snapshot of network state at a previous point in time. Thus tentative routes must be determined on the basis of incomplete information that is also potentially stale. We are using the testbed to investigate the design of aggregation mechanisms and routing algorithms which achieve desirable balance between scalability and performance.

5.2 TED Models

The basic TeD entity is the ATM switch. The entity interface consists of a parameterized array of channels, one per ATM link, and a `user` channel for the local point of attachment of a user node. Large networks are assembled by connecting channels appropriately.

Each switch has the basic PNNI routing and signaling functions, including routing state information exchange, topology aggregation, routing hierarchy formation, connection setup message processing, and crankback. The behavioral specification of a PNNI switch in TeD partitions nicely into the linear inheritance hierarchy shown in Figure 5.2.

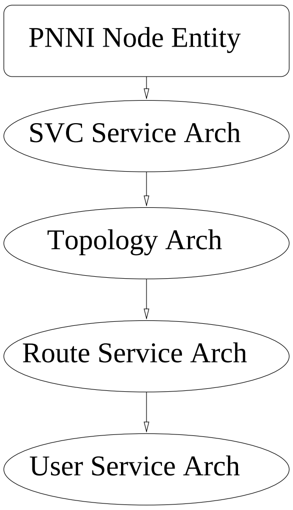

Figure 6: Entity and architecture inheritance hierarchy in PNNI model

The PNNI models constitute a complex set of standards-based specifications of real network interfaces. Our modeling experiences fed back into the development of TeD and also led to enhancements to GTW. We found that, unlike most academic projects, the amount of state information that can change in unpredictable ways can be very large; indeed, this led us to incorporate a transparent incremental state saving facility within GTW. Similarly, while most research simulators fix all event sizes to be equal, event sizes in PNNI vary considerably, causing inefficiencies in memory management. We have found simple compression techniques to help mitigate our specific problems, and we are exploring more general techniques.

One of the tricky modeling challenges was posed by mutually recursive dependencies in the PNNI protocols. For example, setting up and maintaining the hierarchy requires the routing service to set up connections between PGLs; in turn, the routing service uses the hierarchy for call set up. TeD internal channels were crucial to resolve this problem.

5.3 Simulation Studies

We are using the PNNI testbed in network performance studies. In particular, we examine the effects of topology and traffic aggregation schemes, grouping methods, and call admissions and routing protocols on overall network performance. The performance of typical simulation runs on networks with 1000 switches and over 2200 links is shown in Figure 7. The experiments run for 1650 time units from cold start; the initial network setup time is 150 units, the total number of successful SVC requests is slightly over 1,450,000. The work load corresponds to nearly 94,000,000 discrete events that are simulated. The simulations were performed on an SGI Origin shared-memory multi-processor containing `R10000` processors.

In Figure 7 we observe that a large PNNI network model exhibits almost linear speedup, reducing simulation times from over a day on 1 processor down to about 4 hours on 8 processors! Similar simulation speedups are observed on several other network models, such as multicast and wireless networks[7]. This serves as clear indication that large speedup can in fact be achieved in large networks with complex communication patterns. More importantly, such speedup can be obtained without burdening the modeler with issues of parallelism. These results also demonstrate that exotic computers are not necessary to execute the parallel simulations with very good speedup — commodity multiprocessors that are widely available are entirely sufficient.

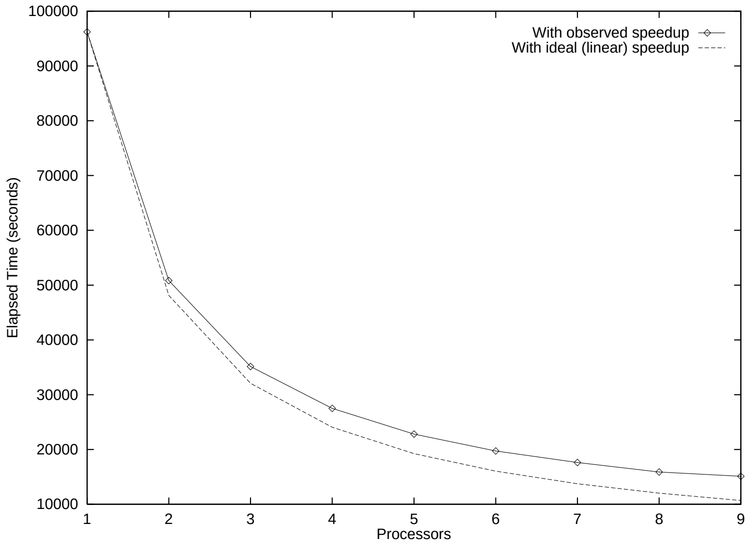

Figure 7: Performance of parallel simulations of a 1000-node PNNI network.

6 Conclusions

The central point of this article is that recent advances in software design for parallel discrete event simulations have brought us to the point where it is possible to routinely use them for modeling and analysis of large and complex network models. From the perspective of network modelers, the simulators present a simple and natural high level modeling framework, such as TED, which hides the details of parallel simulator implementation and allows to build any network model from simpler, reusable components.

Network models designed in TED routinely achieve high parallel performance on multiprocessor machines, speeding up the simulations by a factor proportional to N for N-processor machines. This has been demonstrated for elaborate, large models of standards-based, realistic models of ATM, Internet and mobile wireless networks.

This and related work clearly indicate that once the techniques for parallelizing the simulations of networks have been implemented in the design of primary simulation objects (which is not always easy), the modelers can routinely reuse and refine them to create models exhibiting quite significant speedups. Even without large multiprocessor servers, when only a standard 4–processor personal workstation is available, speeding the simulations by a factor of 3 means a lot in everyday engineering design!

This work is partially supported by DARPA Contract N66001-96-C-8530 and by NSF Grant NCR-9527163.

References

[1] "Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level," W. Willinger, et al, Proceedings of ACM SIGCOMM, 1995.

[2] "Next Generation Internet Implementation Plan, February 1998. `http://www.ngi.gov`

[3] "Maisie/PARSEC: Parallel Simulation Environment for Complex Systems," `http://may.cs.ucla.edu/projects/parsec/`.

[4] "TeleSim Project Home Page," `http://bungee.cpsc.ucalgary.ca/TeleSim/`.

[5] S. Das, R. Fujimoto, K. Panesar, D. Allison, M. Hybinette, “GTW: A Time Warp System for Shared Memory Multiprocessors,” Proc. of Winter Simulation Conference, 1994.

[6] *Parallel and Distributed Simulation Systems.* R. M. Fujimoto, Wiley-Interscience, New York, NY, to appear.

[7] *Special Issue on the Telecommunications Description Language*, D. M. Nicol, editor, ACM SIGMETRICS Performance Evaluation Review, Vol. 25, No. 4, March 1998.

[8] “WiPPET, A Virtual Testbed for Parallel Simulations of Wireless Networks,” J. Panchal, O. Kelly, J. Lai, N. Mandayam, A. T. Ogielski and R. Yates. Proceedings of PADS98, May26-29, Banff, Alberta, Canada.

[9] “Parallel Simulation of TCP/IP using TeD,” B. J. Premore, D. M. Nicol, Proceedings of the 1997 Winter Simulation Conference, December 1997.

[10] “Optimistic Parallel Simulation of Reliable Multicast Protocols,” D. Rubenstein, J. Kurose, D. Towsley, ACM SIGMETRICS Performance Evaluation Review, Vol. 25, No. 4, March 1998.

[11] “Nops: A Conservative Parallel Simulation Engine for TeD,” A. L. Poplawski, D. M. Nicol, Proceedings of the 12th PADS Workshop, 1998, to appear.

7 Biographies

SANDEEP BHATT is a Senior Research Scientist at Bellcore and member of research faculty at the Rutgers University. His research interests include models and algorithms for parallel and distributed computing, graph embeddings, and distributed simulations of many-body systems and communication systems.

RICHARD M. FUJIMOTO is a professor in 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 over 70 technical papers in this field. He has given several tutorials on parallel and distributed simulation at leading conferences, and has co-authored a book on parallel processing. He served as chair of the technical working group responsible for 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.

ANDY T. OGIELSKI is a Research Professor with joint appointment at Discrete Mathematics and Computer Science Center (DIMACS) and Wireless Information Network Laboratory (WINLAB) at Rutgers University since 1997. He held academic positions in Europe and in USA from 1976 to 1981, was a Member of Technical Staff in Physics Research and in Mathematics Research at Bell Laboratories from 1982 to 1989, then joined Bellcore where he was a Director of Parallel Computing and Algorithms Research, and later of Internet Communications Research until 1997. He received the M.Sc. in Physics in 1972 and the Ph.D. in Theoretical Physics in 1976, both from The University of Wroclaw, Poland. His current research focuses on network dynamics, protocols, and wireless communications, in particular on the modeling techniques and parallel simulations of large telecommunication systems and on parallel simulation software design.

KALYAN PERUMALLA is a Research Scientist at the College of Computing at the Georgia Institute of Technology, where he is also working towards the Ph.D. degree in Computer Science. He received the M.S. degree in Computer Science from the University of Central Florida in 1993, and B.E. degree in Mechanical Engineering from the Osmania University, India, in 1991. His current research interests include parallel simulation, network modeling, and parallel combinatorial optimization.