An Approach for Federating Parallel Simulators

Steve L. Ferenci
Kalyan S. Perumalla
Richard M. Fujimoto
College Of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{ferenci,kalyan,fujimoto}@cc.gatech.edu

Keywords: federating, parallel simulators, proxy entity

ABSTRACT

This paper investigates issues concerning federations of sequential and/or parallel simulators. An approach is proposed for creating federated simulations by defining a global conceptual model of the entire simulation, and then mapping individual entities of the conceptual model to implementations within individual federates. Proxy entities are defined as a means for linking entities that are mapped to different federates.

Using this approach, an implementation of a federation of optimistic simulators is examined. Issues concerning the adaptation of optimistic simulators to a federated system are discussed. The performance of the federated system utilizing runtime infrastructure (RTI) software executing on a shared memory multiprocessor (SMP) is compared with a native (non-federated) SMP-based optimistic parallel simulator. It is demonstrated that a well designed federated simulation system can yield performance comparable to a native, parallel simulation engine, but important implementation issues must be properly addressed.

1. Introduction

There are two principal paradigms for constructing parallel and distributed simulations today. The first, widely utilized by the parallel discrete event simulation (PDES) research community, is to define a parallel simulation engine, associated languages, libraries, and tools to create new high performance simulators. Numerous examples of this approach exist today, e.g., TeD/GTW [1], SPEEDES [2], and Task-Kit [3] to mention a few. Simulation models are specific to the environment for which they were developed, making it difficult, in general, to port models to new environments.

A second paradigm that has emerged in the distributed simulation community is to federate disparate simulators, utilizing runtime infrastructure (RTI) software to interconnect them. This approach is utilized in efforts such as Distributed Interactive Simulation (DIS) [4], Aggregate Level Simulation Protocol (ALSP) [5] and the High Level Architecture (HLA) [10]. This approach places few restrictions concerning the realization of individual simulators. This results in coarse-grained federations, where entire simulations are viewed as black boxes, and designated as federates. The runtime infrastructures used to interconnect the simulations are typically designed for coarse granularity concurrency.

Here, we explore an alternate approach. Unlike the traditional PDES paradigm, explicit support for model interoperability and reuse is defined. Unlike traditional federated approaches such as the HLA, we impose certain restrictions concerning the structure of the simulators that are included in the federation in order to enable entity level interactions between federates. Thus, this approach does not attempt to address the general problem of interoperability and reuse of *arbitrary* legacy simulators. Rather, this paper attempts to explore the question of how simulators might be defined in the future in order to support both model reuse and highly efficient concurrent execution.

A second, related problem addressed in this paper concerns the difficulty of constructing federations of optimistic simulators, and the efficiency of their execution. While interfaces such as the HLA support federations of optimistic simulators, few, if any, federations to date have included multiple optimistic federates. We compare the efficiency of a federation of optimistic simulations with a native (non-federated) implementation executing the same simulation model.

We next describe our approach to realizing federated simulations. The prototype federated simulation is then described that uses an RTI to interconnect optimistic simulations. Implementation issues associated with federating optimistic simulations are discussed, and performance measurements presented.

2. Approach to Federating Simulations

At the highest level, our approach to realizing federated simulations is based on:

- defining a *global conceptual model (GCM)* for the entire (federated) simulation model based on an entity/message-passing paradigm,
- standard entity types and data exchange definitions to achieve semantic interoperability among entities realized in different federates, and
- defining a *mapping* of the GCM to realization of individual model components.

The GCM is central to this approach. Use of conceptual models is not new. Such models are used at least informally as part of the federation development process in the HLA. Our approach differs from that currently used in the HLA in that we formalize this notion so that it can be used to automatically generate and configure federated simulations.

Here, we do not address the second issue concerning standard entity types and data definitions. These must be realized by defining consensus within the modeling and simulation community for each specific domain. Work of this nature is in progress within the Defense community, for example.

2.1. Global Conceptual Model

The GCM is based on an entity/message-passing paradigm. This means the entire federated simulation is viewed as a collection of entities that interact by exchanging time stamped messages.

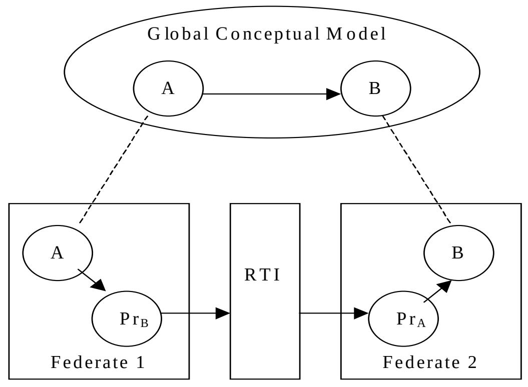
Each entity in the GCM is viewed as a black box. The GCM makes no assumptions concerning the internal realization of an entity, e.g., whether it is based on an event-oriented or a process-oriented world-view. Further, the GCM makes no assumption concerning the actual mechanism for passing information in or out of an entity. This could be done through procedure calls or method invocations, for example. In general, different federates may use entirely different mechanisms to implement and pass information among entities.

We assume each entity defined in the GCM has a corresponding implementation in at least one of the simulators (federates) making up the federation. Thus, we assume each simulator is internally composed of interacting entities. We do not view this as an overly constraining assumption because the entity concept is widely used in modern discrete event simulation. For example, object-oriented and object-based simulation systems typically utilize this approach. Parallel discrete event simulations almost universally are based on logical processes that interact by exchanging messages, so they naturally fall into the entity/message-passing paradigm.

Here, we preclude the use of modifiable, shared state variables between entities. In principle, mechanisms to allow shared state could be easily allowed by the GCM. However, shared state introduces well-known difficulties concerning synchronization. Specifically, references to shared-state inevitably result in zero lookahead interactions, which can have severe performance consequences. Extension of the GCM to allow shared state is an area of future research.

It is clear that not all simulators will be able to conform to the restrictions outlined above. In some cases, it may be necessary to encapsulate the entire simulation as a single entity of the GCM. In other situations, it may not be possible to compose legacy simulations without major redevelopment efforts.

Figure 1. Mapping GCM to Federates. Pr_A is a



proxy of A, and Pr_B is a proxy of B.

2.2. Mapping the GCM to Federates

Each entity in the GCM must have a realization in one of the simulators making up the federation. For this purpose, a table is defined that maps each GCM entity to its corresponding realization in a federate. This mapping is depicted in Figure 1, where entity A is mapped to federate 1, and entity B is mapped to federate 2.

Interactions between entities that reside within the same federate are handled using the mechanisms defined internally within that federate. Interactions between entities residing in different federates are handled using a mechanism called *proxy entities.* A proxy entity is a local representation of an entity that resides within another federate.

An entity initiates an interaction with a remote entity by interacting with the local proxy entity, using whatever interaction mechanisms are defined within the simulator. For example, in Figure 1, suppose entity A wishes to interact with entity B. This will be accomplished by entity A in federate 1 initiating an interaction with Pr_B, the proxy entity for B residing in federate 1. In general, the entity initiating the interaction need not be aware that it is interacting with a proxy rather than the actual entity.

The proxy entity Pr_B is responsible for converting the local interaction to one or more messages that are sent through the RTI to the destination federate, or in this case, Federate 2. These messages are delivered to a second proxy entity (Pr_A) in Federate 2 that represents the entity initiating the interaction. Proxy entity Pr_A must translate the incoming message(s) to interactions with the destination entity B. Pr_A interacts with B using the local interaction mechanisms defined within Federate 2. Again, entity B need not be aware that it is interacting with a local or remote entity.

The principal task performed by the proxy entities is to convert local interactions using the mechanisms defined within the local simulator to interactions that are transmitted through the RTI, and vice versa. A common interface is defined for the RTI, e.g., the Interface Specification defined for the High Level Architecture.

To simplify the previous discussion, we assumed a separate proxy entity was used to represent each remote entity that interacts with a local entity. In some cases it may be more efficient to realize a collection of proxy entities within a single entity, and time multiplex the usage of that realization.

In general, proxy entities may be created or destroyed dynamically during the execution. For example, consider a simulator modeling the operation of a sensor, e.g., radar. As other vehicles, represented in other federates, move within range of the sensor, new proxy entities must be created within the radar simulator to represent them. Further, such proxy entities can be discarded once the simulated vehicles move out of range.

Of particular interest here is the case where each federate is a parallel discrete event simulation. Such simulations are normally defined as collections of logical processes that interact by exchanging messages. Thus, each logical process represents a single entity in the GCM. The mapping of logical processes to federates can be defined as a mapping function:

$$LPtoFed(LPi) = Fedj$$

LPtoFed identifies the federate on which each LP is instantiated.

An example better illustrates the role the GCM plays. Suppose two disparate simulators are to be integrated with one another. The first is an air battle simulation comprised of aircraft entities, the second is a ground battle simulation comprised of tank entities. In the GCM each aircraft entity and each tank entity will be represented as an entity. Each entity in the GCM is permitted to interact with any other entity in the GCM. At this point information about the communication topology between entities can be used to more efficiently federate the simulators. If an entity communicates only with local entities then there is no need to create a proxy entity to represent this entity in the other federates. Next the entities in the GCM are mapped to the federates and proxy entities are created to facilitate communication between the federates.

We will now consider the techniques for implementing the proxy-based federating approach. We illustrate the techniques using an LP-based prototype system, followed by some performance characteristics of the system.

3. Prototype Overview

An initial prototype was developed to enable exploration of this approach to realizing federated simulations, and to identify and evaluate performance issues. As a first step, a realization of a *homogeneous* federation was developed. Specifically, each federate is an instance of the TeD / GTW (Telecommunication Description Language implemented over the Georgia Tech Time Warp parallel simulator) [1, 6]. RTI software from the Federated Distributed Simulation Kit (FDK) was used to interconnect the simulators. In the rest of this section, we describe the relevant aspects of the GTW implementation and the RTI interface, which have a bearing on the effort necessary to realize a federated implementation.

3.1. GTW Overview

Georgia Tech Time Warp (GTW) is an optimistic simulator based on Time Warp [7]. GTW has three main data structures: a message queue (MsgQ) holds incoming messages, a cancellation queue (CanQ) that holds messages that have been canceled (anti-messages), and an event queue (EvQ) that holds processed and unprocessed events. Each processor has each of these structures present and executes a loop that does the following three steps:

(1) All incoming messages are removed from the MsgQ data structure, and the messages are filed one at a time into the EvQ data structure. If a message has a timestamp smaller that the last event processed by the LP, the LP is rolled back. Messages sent by rolled back events are enqueued into the CanQ of the processor holding the event.
(2) All incoming canceled messages are removed from the CanQ data structure, and are processed one at a time. Rollbacks may also occur here, and are handled in essentially the same manner as rollbacks caused by normal messages.
(3) A single unprocessed event is selected from the EvQ, and processed by calling the LP's event handler procedure.

The principal atomic unit of memory in GTW is a buffer. Each buffer contains the storage for a single event, a copy of the automatically checkpointed state, pointers to scheduled messages (direct cancellation) and incremental state-save buffers, and miscellaneous status flags, and other information. Each buffer utilizes a fixed amount of storage. Each processor maintains a list of buffers that are not in use. A buffer may be reused for future events once it has been determined that the time stamp of the event is less than global virtual time (GVT) [7]. GTW uses an efficient GVT algorithm described in [8]. In addition to the GVT algorithm, GTW also employs on-the-fly fossil collection that enables efficient storage reclamation for simulations containing large numbers of simulator objects [8].

3.2. FDK and BRTI Overview

FDK is a modular and reusable set of libraries designed to facilitate the development of Run Time Infrastructures (RTIs) for developing or integrating parallel and distributed simulation systems [9]. Using the libraries provided by the FDK, an RTI was built that implements a subset of the High Level Architecture (HLA) services. This RTI is called the BRTI, and the following is a brief description of the BRTI services that are pertinent to this paper.

- **Publish Object Class/Subscribe Object Class Attribute** - These two services establish a communication pathway between two federates. A federate first publishes an object with Publish Object Class. Other federates can subscribe to the published objects using Subscribe Object Class Attribute.
- **Update Attribute Values/Reflect Attribute Values** - Update Attribute Values sends a message to all federates that have subscribed to an object notifying them of the change in the object's state. On the receiving side, Reflect Attribute Values is the means by which the RTI notifies the federate that an object has been updated.
- **Retract/Request Retraction** - Given an Event Retraction Handle, Retract can be used to cancel a previously sent message. The retraction mechanism is used to implement Time Warp's anti-message mechanism between federates.
- **Flush Queue Request** - This is used by the federate to notify the RTI that the federate wishes all messages currently residing within the RTI to be delivered to the federate as soon as possible. Additionally, Flush Queue Request will also attempt to advance time to the specified time.
- **Time Advance Grant** - This is used by the RTI to notify the federate that its logical time has been advanced to the specified time.
- **Tick** - This is used by the federate to provide the RTI with execution time to perform communication services, time management services, and deliver messages to the federate.

The BRTI includes an underlying efficient asynchronous algorithm for periodically computing lower bound on the timestamp (LBTS) of future incoming events at any federate.

4. Federated GTW

Using the proxy-based approach, we have implemented a system for federating multiple instances of GTW simulations. Each instance of GTW acts as a federate, which communicates using the BRTI with the other GTW federates The implementation process mainly involved three items:

1. Defining a common abstraction of the application objects, called the federation object model, for use by all GTW federates
2. Defining a proxy framework that is used across all GTW applications
3. A set of modifications to GTW for adapting its initialization, message sends, message cancellations and GVT computation modules, to accommodate the proxy-based model.

We describe each of these items next.

4.1. *Federation Object Model*

A federation object model is necessary so that all the federates agree on a certain abstraction of the entities in the GCM. The object model for the GTW federation is defined as follows. Each logical process (LP) has input ports and output ports. Each output port of an LP is mapped at initialization time to an input port of another LP. Whenever an LP sends a message on one of its output ports, the LP that owns the corresponding mapped input port receives the message. The LP actually sends a message by "assigning" the event data as the value of the port object variable. This model based on ports allows the application to expose its communication topology, which is necessary to prevent broadcast semantics for inter-federate communication; at the same time, it does not exclude applications that do need all-to-all communication.

During initialization, the communication links between federates are established with the help of BRTI services **Publish Object Class** and **Subscribe Object Class Attribute**, using port names as unique object classes. Each federate publishes a list of output ports corresponding to the LPs that are owned by this federate. A federate will subscribe to an output port if it owns an LP whose input port is mapped to that output port. Port mapping can be specified in a file that is read in at run time. If no file is specified then all-to-all communication is assumed by default.

At runtime, event exchanges are realized using the ports as follows. Since each port is an RTI object, its value can be updated by the federate that can publish new values to the object. Events (event data) are assigned as *values* for the port objects. This is done using the **Update Attribute Values** service of the BRTI. On the other side of the output port, updates to the output port are received via the **Reflect Attribute Values** callback service of the BRTI.

Based on the port descriptions of the LPs, the object class creation, publication and subscription services are automatically invoked by GTW, in order to initialize the communication services.

4.2. *Proxy Framework*

By default, a replication-based proxy framework is supported. In this framework, every federate instantiates every LP that is present in the simulation's GCM. At any federate, only those LPs that are mapped to that federate are executed as regular LPs. The rest of the LPs are executed as proxies. Exactly one federate *owns* any given non-proxy LP. When any proxy LP receives an event, it forwards exactly one copy of that event to the federate that owns that LP. In addition to the forwarding semantics, every GTW proxy LP implements an initialization function, which is the same as its corresponding non-proxy LP initialization function. This is used in constructing global read-only data structures during the initialization stage at each GTW federate, as described next.

4.3. *Initialization and Read-only State*

The GTW federations are initialized as usual, similar to the non-federated GTW simulation, with one important distinction as follows. When LPs are created and initialized in GTW, they are permitted not only to schedule their initial events, but also to cooperate in creating and initializing global data structures intended for read-only use during the actual simulation. It is clear that, if the initialization procedures of *all* the LPs are invoked in *all* the federates, then identical copies of the global state are correctly created automatically in *all* the federates. This approach is what the proxy framework as described previously supports.

This approach has the advantage that no source-code changes are required for the applications. Since we are interested in minimizing the changes to the application, we implemented this approach. The initialization must be carefully controlled, however, in order to preserve the semantics of proxy LPs. This is done at each federate by ignoring any message-sends performed by a proxy LP during its initialization, permitting the proxy LP to cooperate in the global data creation, but disallowing it to be scheduled during simulation at this federate. Turning off message-sends was quite easy to implement in GTW -- for any message-send by a proxy LP, a dummy "abort" message buffer is supplied by GTW to the LP, which is later discarded, instead of being scheduled, by the kernel.

4.4. *Sending and Receiving Messages and Message Cancellations*

The original native GTW includes a mechanism for sending messages and cancellations among LPs. This mechanism needed to be augmented such that events and cancellations destined to a proxy LP at a federate get automatically forwarded to the federate where the destination LP is actually simulated. For events exchanged between regular (non-proxy) LPs, the usual fast communication path of GTW is preserved. At each GTW federate, the processor whose ID is zero acts as the gateway to route events to and from other GTW federates.

With a view to minimizing the source-code changes, while not compromising on efficiency, we preserved the method by which any GTW LP sends an event to another LP, irrespective of whether the destination LP is a proxy or not. This essentially appends the event to the MsgQ of the owner processor of the event's destination LP. The distinction between local and proxy LPs is, however, made at the time the event is actually extracted from the MsgQ by the destination processor. If the event is for a local LP then the event handler is called as usual. Otherwise (if the LP is a proxy), then this message is forwarded to the federate that owns the LP by invoking **Update Attribute Values** on the object of the corresponding port.

On the receiver side, the BRTI accepts the message and stores it internally until the federate notifies the BRTI to deliver the messages. In the main scheduling loop of GTW, **Flush Queue Request** is invoked to notify BRTI to delivery any messages received so far. (The messages are actually delivered when the next time Tick is called). Messages are delivered to the GTW federate by the BRTI via **Reflect Attribute Values** callback. Once the message has been delivered it is appended to the MsgQ of the processor that owns the destination LP.

Message cancellations (retractions) are treated in a manner analogous to normal events. If the cancellation is meant for a local LP, then the existing GTW mechanism for processing cancellations is performed. If the retraction was in fact destined to a proxy LP, then the BRTI services will have to be used to cancel the message. Every time a normal message is sent using **Update Attribute Values,** an event retraction handle is returned. This handle is stored in the event buffer so that the handle can be used if it has to be canceled. When a proxy LP receives a cancellation it invokes the BRTI **Retract** service with the event retraction handle. When a federate receives a retraction, it makes a call back specified by the GTW federate. The GTW federate performs a handle-to-pointer hash to identify the retracted event, and places it on the CanQ of the processor who owns the destination LP. Cancellations then proceed as usual in GTW.

4.5. Synchronization

Both GTW and BRTI have their own concept of a global time. In GTW it is GVT (Global Virtual Time) and in BRTI it is LBTS (Lowest Bound on Time Stamps). Coordinating the algorithms is crucial to obtaining correct and efficient performance. In the main scheduling loop of GTW, Flush Queue Request is called to notify the BRTI to deliver all messages it has received. A target time is passed as argument to the BRTI, which indicates when the next event is scheduled to the best of GTW's knowledge at that time. When this federate participates in an LBTS computation it will use this to determine its contribution to the computation. When the LBTS computation completes, BRTI issues a Time Advance Grant which notifies the GTW federate that time has been advanced. This time is what is used as the global GVT. In the federated GTW, LBTS is equivalent to GVT of the non-federated GTW.

5. Insights, Lessons, and Challenges

The process of federating GTW was straight forward. The GTW kernel was augmented to use RTI services where necessary, e.g., sending messages to remote LPs or performing necessary time management services. Only the GTW kernel was modified, thus avoiding making significant modifications to the applications. In the case of phold and PNNI no changes were required in these applications. During the course of the implementation it became apparent that care must be taken to ensure good performance as well as correctness. Two major challenges included coordinating the GVT computation and LBTS computation, and buffer management.

5.1. GVT and LBTS

The GVT algorithm [8] incorporated into GTW is specially optimized for a shared-memory implementation, which relies on the actual order of operations on the MsgQ, CanQ and EvQ for handling transient messages efficiently. When such an algorithm is integrated with an RTI, which presents incoming events at unpredictable moments, race conditions can arise with respect to accounting for transient messages in both the local GVT and LBTS computation, potentially leading to incorrect GVT values being computed.

Ensuring that the LBTS computation and the incoming message delivery do not overlap with the local GVT computation easily solves this problem. In our implementation, an LBTS computation gets initiated when a Flush Queue Request is made. By calling Flush Queue Request only when no GVT computation is active locally, we can prevent the race condition. This ensures that all messages that have to be considered in the local GVT computation have already been delivered by BRTI, and the most accurate local GVT value is used in the LBTS computation. Thus, we were able to preserve the efficient asynchronous GVT algorithm of GTW without compromising its correct integration with the time management services of the BRTI.

5.2. *Buffer Management*

The second major challenge actually became relatively simple to solve using the proxy-entities. When sending or receiving messages using the BRTI we either had to decide what to do with a buffer once used, in the case of sending, or where to obtain a buffer when receiving a message. Message sends occur when an event is removed from the MsgQ and its destination is a non-local LP. At this point, the BRTI is used to send the message, but what can be done with the buffer after the send? The solution is quite simple: mark the buffer as 'processed', as if an event handler was called for this event, and place it on the LP's processed list. This way, the on-the-fly fossil collection can recover the buffer and place it on the free buffer list. The proxy LPs essentially served as convenient buffer repositories for remote messages, to facilitate in generating anti-messages if necessary. This made it relatively easy to integrate the proxy behavior with on-the-fly fossil collection, GVT computation and cancellations, thus minimizing the amount of code changes required.

When receiving messages, BRTI first asks GTW where it can place messages arriving off the wire. This ensures that there will not be a need for an extra memory copy from a BRTI buffer to a GTW buffer. GTW will give BRTI a buffer from processor 0's free buffer list to store the message. At this point the message will be stored in BRTI's internal message queue and delivered when GTW requests that the message be delivered. When the messages are delivered, processor 0 will forward the message to the destination LP's owner processor as if processor 0 scheduled it. As far at the destination processor is concerned processor 0 sent the message. Using the proxy-entities makes buffer management much simpler since the existing buffer manager can be relied on.

5.3. *Deadlocks and Flow Control*

Deadlock is another disconcerting possibility that arises when the simulators are federated together. Even though the native simulators are designed to be deadlock free in isolation, the deadlock problem arises all over again when they are federated together. In fact, we have observed this problem empirically early on in our implementation. Even though GTW is deadlock free in isolation, a naïve implementation of a GTW federation can deadlock due to a circular hold-and-wait condition on memory buffers used for sending events between federates.

The flow control problem also shares this feature with respect to federating, and deserves careful attention in a federation.

6. Performance Study

We used two separate GTW applications in our performance study. The first is the PHOLD application, which is a synthetic benchmark commonly used for parallel simulators. The second is a practical application, called the PNNI (Private Network to Network Interface) model suite, written in the TeD language, and compiled as a GTW application. The two applications exhibit contrasting characteristics compared to each other. PHOLD is a relatively fine-grained application, with small state and event sizes. PNNI, on the other hand, is a relatively coarse-grained application, with very large state and event sizes.

With both applications, we compared two scenarios:

1. A non-federated multi-processor parallel simulation, using a single instance of the original GTW kernel. The LPs of the application are statically mapped to different processors.
2. A federated simulation in which multiple instances of single-processor GTW kernels communicate using the BRTI. The LPs are partitioned across the federates, but proxies are instantiated for non-local LPs in every federate.

The LP-to-federate mapping used in the federated simulation is the same as the LP-to-processor mapping of the corresponding native GTW simulation.

The test configuration for PHOLD included 40 LPs with a message population of 40. The test configuration for PNNI included a 200-node real-life network with a user node attached to each network node, giving a total of 400 LPs. All the simulations were run on an SGI Origin multiprocessor (R10K processors) and 4GB RAM. All communication is through shared memory.

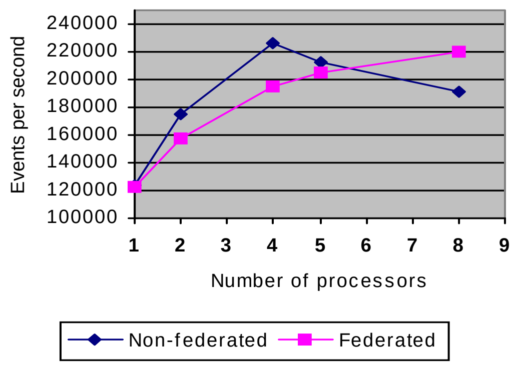


Figure 2. Performance of PHOLD simulation.

The performance of PHOLD is shown in the Figure 2. The rollback statistics of the federated and non-federated simulations were comparable on smaller number of processors. But when larger number of processors was used, the federated simulation incurred fewer rollbacks, accounting for its higher performance than the non-federated simulation.

The performance of PNNI is depicted in the Figure 3. In all cases, the rollback behavior of federated and non-federated simulations was comparable.

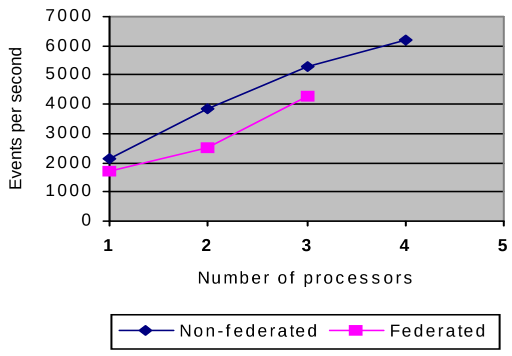


Figure 3. Performance of PNNI.

In both types of applications, we observe that the performance of the federated simulation is comparable to that of the optimized monolithic parallel simulator. This indicates that for an application, not only the reuse capabilities but also performance needs can be met using a standards-based federated implementation.

7. Conclusions and Future Work

In this paper we presented an approach to federating parallel simulations. We introduced the Global Conceptual Model (GCM) as a key concept to federating parallel simulations, and presented the implementation techniques for a prototype federation of GTW optimistic parallel simulators. The implementation of a proxy-based GTW federation involved a relatively small amount (15%) of source-code changes to the GTW kernel, along with the use of the RTI library. Our implementation ensured that GTW applications written to run on non-federated GTW would run on the GTW federation with almost no source-code changes. By keeping overheads low, such as by minimizing memory copy operations via careful buffer management, and by adopting asynchronous GVT/LBTS algorithms, we demonstrated that the federation of parallel simulators could perform nearly as well as native parallel simulators. We intend to extrapolate further from this observation: if any parallel simulator shall be developed from scratch in the future, it is actually feasible to develop it as a federation of sequential simulators (rather than as yet another monolithic parallel simulator) without the fear of a significant performance penalty! The system presented here is one of the first works to evaluate the performance of a standards-based federation of optimistic parallel simulators.

Although there has been much work on GVT algorithms, the problem of hierarchical composition of different GVT/LBTS algorithms appears to be an interesting research area to be explored further. Specifically, this is a problem that becomes more relevant in a federation of heterogeneous *parallel* simulators, rather than of sequential simulators.

8. Acknowledgements

This work was supported under contract #DASG-60-99-C-0052 from the U.S. Army Space and Missile Defense Command

9. References

1. Bhatt, S., *et al.*, *Parallel Simulation Techniques for Large-Scale Networks.* IEEE Communications, 1998. **36**(8): p. 42-47.
2. Steinman, J.S., *SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete Event Simulation.* International Journal on Computer Simulation, 1992: p. 251-286.
3. Unger, B., *et al.*, *Scheduling Critical Channels in Conservative Parallel Discrete Event Simulation*, in *Proceedings of the Workshop on Parallel and Distributed Simulation*. 1999.
4. IEEE Std 1278.1-1995, *IEEE Standard for Distributed Interactive Simulation -- Application Protocols*. 1995, New York, NY: Institute of Electrical and Electronics Engineers, Inc.
5. Wilson, A.L. and R.M. Weatherly, *The Aggregate Level Simulation Protocol: An Evolving System*, in *Proceedings of the 1994 Winter Simulation Conference*. 1994. p. 781-787.
6. Das, S., *et al.*, *GTW: A Time Warp System for Shared Memory Multiprocessors*, in *Proceedings of the 1994 Winter Simulation Conference*. 1994. p. 1332-1339.
7. Jefferson, D., *Virtual Time.* ACM Transactions on Programming Languages and Systems, 1985. **7**(3): p. 404-425.
8. Fujimoto, R.M. and M. Hybinette, *Computing Global Virtual Time in Shared Memory Multiprocessors.* ACM Transactions on Modeling and Computer Simulation, 1997. **7**(4): p. 425-446.
9. Fujimoto, R.M. and P. Hoare, *HLA RTI Performance in High Speed LAN Environments*, in *Proceedings of the Fall Simulation Interoperability Workshop*. 1998: Orlando, FL.
10. Defense Modeling and Simulation Office, http://hla.dmso.mil.