SimCore: A Library for Rapid Development of Large Scale Parallel Simulations

Sunil Thulasidasan[1], Lukas Kroc[2] and Stephan Eidenbenz[1]
[1]*Computational & Statistical Sciences Division, Los Alamos National Laboratory, Los Alamos, NM, U.S.A.*
[2]*Claremont College, Claremont, CA, U.S.A.*

Keywords: Discrete-event Simulation, Parallel Simulation Library, Python Simulation.

Abstract: We present the SimCore parallel simulation library, an object-oriented framework for developing parallel distributed discrete-event simulation applications, implemented in C++ with a Python front-end. SimCore is designed to scale to thousands of processors but is simple enough to use for application programmers, an outcome of its fast C++ core and message passing routines integrated with the full expressive power of Python. We discuss the design philosophy of SimCore including the software architecture and the C++/Python interface implementation that allows applications to be written in pure Python or a hybrid of Python and C++ or even pure C++. We also provide real world examples of the scalability and briefly describe a few diverse applications that have been deployed using SimCore.

1 INTRODUCTION

SimCore is a generic library for designing simulation applications for large-scale, parallel discrete event simulations. Scalable simulations are an important tool in the modeling of complex systems where simulating the behavior of millions of entities and their interactions demand significant computational resources. SimCore has been designed with the primary goals of flexibility and scalability, and has allowed us to construct a suite of various domain specific simulators that can be run independently, or integrated with each other. For instance, we have been able to integrate a parallel transportation simulator with an agent-based activity generator and simulator (both SimCore applications) that allows us to generate realistic activity schedules for millions of intelligent agents, route the tens of millions of vehicular trips generated from these activities, and observe how traffic conditions and the variations they cause in expected travel times impact activity scheduling and vice versa.

SimCore, is in general, designed to be usable in any scenario where discrete-event simulation techniques are a suitable modeling paradigm. In this paper, we focus on the the motivation and design principles in developing SimCore and give examples of its usage and performance in diverse application areas.

2 MOTIVATION

SimCore has been designed to be a general purpose simulation library for developing parallel and distributed simulation applications that use a discrete-event simulation paradigm with conservative synchronization (Fujimoto, 1990). The goal of SimCore is to enable domain researchers to rapidly develop simulation applications and deploy it on large clusters without having to get bogged down by the intricacies of parallel programming, managing MPI communication or worry about issues such as load balancing. The Python front-end of SimCore provides an intuitive and easy to use interface to the simulation library that makes use of the full expressive power of Python; this also enables easy integration of SimCore applications with other numerous scientific packages available for Python including visualization and numerical computation.

The need for computational performance, however, dictates that a large-scale simulation code implemented in pure Python will inevitably run into performance issues. At this point, the traditional route is to port the initial prototypes to a performance oriented language such as C, C++ or Fortran. While SimCore has been written as a tool for research – where developer cycles are significantly more valuable than program execution time – the decision was made to implement the most general and frequently used parts of

Thulasidasan S., Kroc L. and Eidenbenz S..
SimCore: A Library for Rapid Development of Large Scale Parallel Simulations.
DOI: 10.5220/0004063700710076
In *Proceedings of the 2nd International Conference on Simulation and Modeling Methodologies, Technologies and Applications* (SIMULTECH-2012), pages 71-76
ISBN: 978-989-8565-20-4
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

the library – message passing, synchronization, logging and file I/O – in pure C++. This hybrid approach provides a reasonable balance between performance and ease of use. Further, any computationally intensive bottle-neck can be re-written in pure C++, but it has been our experience that performance optimization is best done after the many iterations that are usually needed for the model and design to stabilize.

Multiple simulation applications have been previously developed using SimCore (see Section 6); many of these applications were implemented in pure C++. Indeed the decision to develop a Python front-end for ease of application development was based on our observation that domain researchers often spend significant amounts of time learning how to write advanced C++ code. While a full detailed description of the SimCore library is outside the scope of the paper, in what follows, we discuss the salient architectural features of the library and the Python integration aspects. We also briefly describe some of the applications that have been developed using SimCore and provide some performance results of large scale simulations.

3 RELATED WORK

There are a number of non-hybrid simulation libraries available for parallel discrete event simulation (PDES). PrimeSSF (PRIME, 2012), originally known as DaSSF, is a parallel simulation framework, developed originally for network simulation, that supports both distributed and shared-memory implementations. It uses the Entity-Message simulation paradigm; in a distributed memory setting, messages are passed via MPI, and causality is maintained via barrier synchronization. SimCore initially used PrimeSSF for message passing and synchronization, and can still be compiled with PrimeSSF as one of the options. OMNET++ (OMNET++, 1996) is a component-based C++ simulation library and framework, primarily focussed on the domain of network simulation. OMNET++ also supports parallel simulations. ClusterSim (Ramos and Martins, 2004) is a Java-based parallel discrete-event simulation tool for cluster computing. ClusterSim supports visual modeling and simulation of clusters and their workloads for performance analysis. μsik (Perumalla, 2005) is a micro-kernel for PDES that supports both conservative and optimistic parallel simulations.

On the Python side, SimPy (SimPy, 2012) is an object-oriented, process-based discrete-event simulation language written in pure Python and provides the modeler with classes for both active and passive components in a simulation. Parallel support was later added to SimPy, but the parallelism remains non-transparent to the user. In the hybrid-approach category, PCSim (Pecevski et al., 2009) is a C++ based neural network simulator with a python front-end, that supports both sequential and distributed memory simulations. NS-3 (NS-3, 2012) is another example of a hybrid C++/Python approach for network simulation. We note that most of the simulation libraries mentioned here were developed for a specific domain. SimCore, on the other hand, aims to be as generic as possible; it can be used in any environment that can be modeled as a collection of interacting entities, using discrete event simulation. Thus, our main contribution is the development of a hybrid general purpose library for parallel discrete event simulation that offers a good trade-off between ease of use and scalability

4 ARCHITECTURE OVERVIEW

SimCore provides the application (an end simulation, such as a network Simulator) with APIs for easily developing a simulation application and hides the complexity of message passing, event synchronization and domain partitioning from the application developer. For message passing and synchronization, SimCore also includes its own simulation engine built on top of the MPI library. However, a compile-time switch will allow SimCore to use PrimeSSF as the underlying event engine. An overview of the software architecture and library stack is illustrated in Figure 1. The simulation library classes are all implemented in C++, using Boost (Boost, 2012) for templated constructs and pointer management, and MPI for communication. Applications can be written entirely in C++ on top of SimCore, or written as Python modules. For the latter facility, a subset of SimCore classes are exported to Python using the Boost.Python interface library to produce a module PySimCore that can then be imported inside Python.

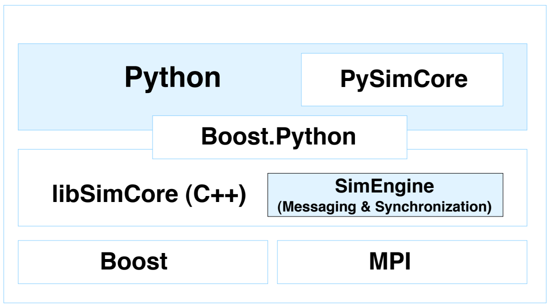

Figure 1: Software architecture of SimCore.

4.1 SimCore Classes

There are three major SimCore constructs that form the basis of a simulation application. These are:

- **Entity:** This is the primary active element of the simulation (an agent, a network device etc.); an entity has associated properties and behaviors that are implemented using services (below)
- **Service:** A service on an entity determines the *behavior* of entities, i.e responses to events.
- **Info:** These are the events in the simulation which include timer events, control events and messages passed between entities in the simulation. Info's are passed and received between services living on entities. For distirbuted memory applications, SimCore serializes and packs the data, before invoking the MPI communication routines. A reception of an info at a service triggers the handling routine for that particular type of message.

Let us consider a Network Simulator for an example: the entities would model devices (e.g. computers), interfaces (e.g. network cards) and media (e.g. network cable or a frequency as a wireless medium). Entities, therefore, correspond to *hardware*. Services would model various protocols (e.g. FTP, TCP, IP on a network device and a "signal transmission functionality" for a communication medium). Thus services correspond to *software* in this context. Packets travelling between various devices are implemented using Info's and appropriate packet-handlers. The interaction of the various services living on an entity will then determine the overall behavior of an entity. The most important properties of each SimCore construct are given below:

Entity

- Models objects in a simulation
- Uniquely identified throughout the simulation system (by an EntityID).
- Has general features (e.g location, capacity, status, etc.)
- Has its own services. Each service lives at a Service address
- Forwards events to an appropriate event handler.
- Entities might reside in different memory spaces depending on the partitioning logic, but this is transparent to the user.

Service

- Models behavior of entities
- Identified using an Entity ID and a service address
- Processes incoming messages.
- All Services living on the same entity are forced to reside in the same memory space i.e. on the same computing node.

Info

- Data and messages to be exchanged between Entities
- Destination address is a tuple of ¡EntityID, Service address¿
- Can be packed with user defined content.

Having a clear distinction between objects (entities) and their behavior (services) allows us to easily extend or modify the behavior of an entity; adding new functionality to an entity amounts to adding a new service to it, without having to change existing code. Interaction between different entities is (mostly) restricted to message passing though communication between services on the same entity can be made via regular function calls. The parallelism and partitioning in the simulation is completely transparent to the end user (simulation application developer). Thus the same logic can be executed on a single processor or a handful of compute nodes or a large-scale cluster consisting of thousands of processors without making any changes to the code.

4.2 Managing SimCore Objects

Each of the main SimCore object types – Entities, Services and Info's – has an associated object manager that handles the task of object creation and maintenance. No SimCore object is created directly by the application; rather, they are created via calls to methods of the managers. SimCore uses the smart pointer (implemented via reference counting) functionality provided by the Boost library to manage object creation and destruction. This only adds a small amount of overhead to the code, while greatly reducing, if not completely eliminating, the occurrence of pointer related bugs.

- The **Entity Manager** is responsible for creating entities, and letting user code access them later, if needed. In the current model, Entities are created inside logical processes (LP) and are uniquely identified in the simulation. Before creating an

Entity, it is necessary to decide which LP it will live at, based on the partitioning strategy used. An LP represents a single running thread of the simulation with a common clock value; in SimCore, each LP is mapped to a distinct physical process.

- The **ServiceManager** is responsible for creating the services on the entities, as well as reading in the service related data before creating the services.
- The **InfoManager** is responsible for mapping Info types to Info objects and calling the appropriate Info handler object when an info is received.

4.3 Partitioning and Load Balancing

Entities are mapped to LPs by a placement function. The same function is later used to find which LP each entity lives on, while sending messages. The user may use any function and register it with a call to the Entity Manager. The default placement function is a modulo function based on the entity identifier, though different placement schemes that map entities to LPs can also be used. Entity placement and partitioning is one of the crucial factors in the performance of a parallel, discrete event simulation. We provide a detailed discussion of the performance of different partitioning strategies from a load balancing perspective in (Thulasidasan et al., 2010) and (Thulasidasan et al., 2009).

5 PYTHON INTEGRATION

To enable construction of simulation applications in Python, we export the APIs from the three main SimCore classes (Entity, Service, Info) to the Python space. For the Python interface generation we use the Boost.Python (Abrahams and Grosse-Kuntsleve, 2003) library which provides fairly seamless operability between C++ and Python. Boost.Python provides comprehensive mapping between C++ and Python constructs, and supports advance templated meta-programming techniques. There is support for exception handling, iterators, operator over-loading, standard template library (STL) containers and Python collections, smart pointers and virtual functions that can be over-ridden in Python. This feature makes the interface bidirectional i.e user-extensions in Python can be also invoked from C++.

Since each class and function has to be manually wrapped in Boost.Python code, to make this process less cumbersome we use the Py++ (Py++, 2012) automatic code generation utility that wraps input C++ code inside Boost.Python constructs. Py++ is based on the GCC-XML (GCC-XML, 2012), a GCC based parser that generates XML representation of C++ code. Using the XML output and user generated rules regarding API export, Py++ generates the Boost.Python interface. This is then compiled and linked against the Boost.Python and SimCore libraries to produce a module that can be imported inside Python. A schematic of the build-chain is shown in Figure 2.

6 SimCore APPLICATIONS

In this section we briefly present some of the simulation applications that have been developed at LANL using the SimCore library. These applications were built to model the large scale behavior and complex interdependencies between various socio-technical systems, and have been used in numerous real world simulation studies to analyze the dynamics of various national infrastructure The interested reader is referred to the relevant publications that describe these applications in more detail.

- **FastTrans** Described in (Thulasidasan and Eidenbenz, 2009; Thulasidasan et al., 2009), FastTrans is a scalable, parallel microsimulator for transportation networks that can simulate and route tens of millions of vehicles on real-world road networks in a fraction of real time. Vehicular trips are generated using agent-based simulations that provide realistic, daily activity schedules for a synthetic population of millions of intelligent agents.

  In FastTrans, simulation entities are the fixed elements of the road network – road links and traffic intersections. All the properties of the network – capacity, flow rate, etc – are members of the relevant entity class. The scheduling logic at a traffic intersection is implemented as a *service* on the traffic-node entity. The modular design allows the scheduling policy to be easily changed by simply replacing one scheduling service with another.

  The mobile elements of the simulation (vehicles) are represented using messages. Vehicle objects (messages) are created and destroyed during the start and end of a trip, respectively. Since FastTrans uses a distributed-memory model, different entities of the road network are created in different memory spaces during simulation start-up. Each simulation process (also known as Logical Process, or LP) in the Entity creation, partitioning and LP set up are all, as described before, handled seamlessly by the SimCore layer.
- **MIITS** (Multi-Scale Integrated Information

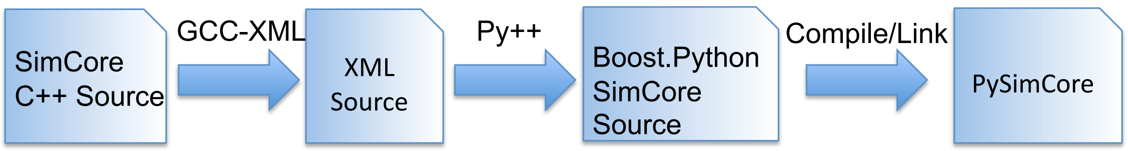

Figure 2: Python integration chain.

and Telecommunications System), introduced in (Waupotitsch et al., 2006) tool is a scalable, end-to-end simulation environment for representing and analyzing communication networks including cellular networks, public switched telephone networks (PSTNs), the Internet, and ad hoc mesh networks. offering network representation at several resolutions, ranging from packet-level simulation to flow-based approaches.

- **ActivitySim,** introduced in (Galli et al., 2009) is another SimCore module, that implements agent-based models combining traditional agent technology from the artificial intelligence community and numerical methods for activity schedule calculations. The simulation entities in ActivitySim are individuals and locations; generic agent classes are used to implement entities with intelligent behavior. The behavior of a person in ActivitySim is modeled using services that implement cognitive functionality which allows people to change and adapt their activity schedules. A more detailed description of the ActivitySim architecture is provided in (Galli et al., 2009).

7 SimCore APPLICATION PERFORMANCE

In this section, we briefly present some scaling and execution performance results from two SimCore applications, FastTrans and ActivitySim, running seperately and also coupled together. For the experiments, we simulate two real world scenarios – an entire day's worth of road traffic for New York City (for the FastTrans runs), and activities and road-trips for the Twin Cities region. The Twin Cities road network consists of approximately 300,000 road links and 150,000 intersections; the New York graph consists of half a million intersections and about 1.1 million road links. The experiments were conducted on an HPC cluster for different processor configurations, ranging from 32 to 1024 processors.

Figure 3 shows the scaling performance in terms of execution time for FastTrans for the two scenarios, and exhibits strong scaling. Figure 4 show the scaling behavior performance of the integrated simulation in the Twin Cities scenario, compared to the performance of the modules when run separately. The performance of the integrated simulation follows the performance of the dominant module. All simulations for both cities run significantly faster than real-time even on 32 processors. The high speed-ups over real-time allow us to simulate multiple scenarios and provide timely feedback and analysis in real-world situations. More detailed performance results are described in (Thulasidasan et al., 2009).

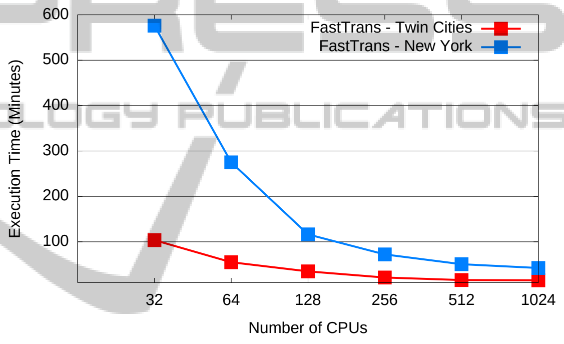

Figure 3: Execution time of FastTrans as a function of #CPUs.

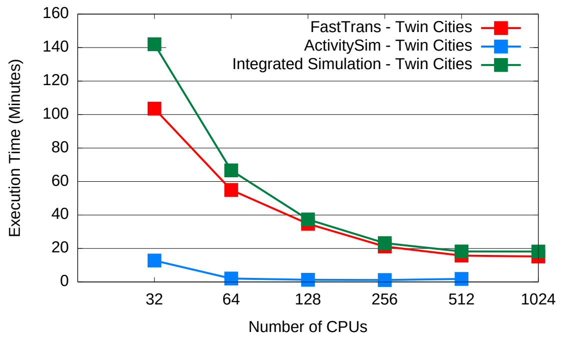

Figure 4: Comparison of execution times of FastTrans, ActivitySim, and the integrated simulation as a function of #CPUs.

8 CONCLUSIONS AND FUTURE WORK

We presented SimCore, a generic library for developing scalable, parallel discrete-event simulations using a hybrid language approach that offers a balance between development time and performance. The common framework and modular software architecture employed in building SimCore applications systems allows us to easily combine different modules for interacting systems. Experiments on HPC clusters have shown near-linear scaling, and we are currently performing studies to calibrate the performance of the Python layer for large scale simulations. Efforts are also currently under way to make the SimCore source code freely available for simulation application developers.

REFERENCES

Abrahams, D. and Grosse-Kuntsleve, R. (2003). Building hybrid systems with boost.python. http://www.boostpro.com/writing/bpl.html.

Boost (2012). Boost c++ libraries. http://www.boost.org.

Fujimoto, R. M. (1990). Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53.

Galli, E., Eidenbenz, S., Mniszewski, S., Teuscher, C., and Cuellar, L. (2009). Activitysim: Large-scale agent-based activity generation for infrastructure simulation. In *Proceedings of the 2009 Spring Simulation Conference*.

GCC-XML (2012). Gcc xml parser. http://www.gccxml.org.

NS-3 (2012). http://www.nsnam.org.

OMNET++ (1996). http://www.omnetpp.org.

Pecevski, D., Natschlager, T., and Schuch, K. (2009). Pcsim: a parallel simulation environment for neural circuits fully integrated with python. *Frontiers in Neuroinformatics*, 3(0).

Perumalla, K. (2005). μsik - a micro-kernel for parallel/distributed simulation systems. In *Workshop on Principles of Advanced and Distributed Simulation*.

PRIME (2012). *Parallel Real-time Immersive network Modeling Environment*. Available at http://prime.mines.edu/.

Py++ (2012). Py++ automatic code generator. http://sourceforge.net/projects/pygccxml.

Ramos and Martins (2004). Clustersim: a Java-based parallel discrete-event simulation tool for cluster computing. In *Proceedings of IEEE International Conference on Cluster Computing*.

SimPy (2012). http://simpy.sourceforge.net.

Thulasidasan, S. and Eidenbenz, S. (2009). Accelerating traffic microsimulations: A parallel discrete-event queue-based approach for speed and scale. In *Proceedings of the Winter Simulation Conference*.

Thulasidasan, S., Kasiviswanathan, S., Eidenbenz, S., Galli, E., Mniszewski, S. M., and Romero, P. (2009). Designing systems for large-scale, discrete-event simulations: Experiences with the fasttrans parallel microsimulator. In *HiPC'09*, pages 428–437.

Thulasidasan, S., Kasiviswanathan, S., Eidenbenz, S., and Romero, P. (2010). Explicit spatial scattering for load balancing in conservatively synchronized parallel discrete event simulations. In *Proceedings of ACM/IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*.

Waupotitsch, R., Eidenbenz, S., Smith, J., and Kroc, L. (2006). Multi-scale integrated information and telecommunications system: First results from a large-scale end-to-end network simulator. In *Winter Simulation Conference*, volume 0, pages 2132–2139. IEEE Computer Society.