Developing Parallel, Discrete Event Simulations in Python: First Results and User Experiences with the SimX Library*

Sunil Thulasidasan[1], Lukas Kroc[2] and Stephan Eidenbenz[1]
[1]*Los Alamos National Laboratory, Los Alamos, USA*
[2]*SpaceX, Hawthorne, USA*
sunil@lanl.gov, kroc@cs.cornell.edu, eidenben@lanl.gov

Keywords: Parallel Discrete-Event Simulation, Distributed Simulation, Python

Abstract: This paper describes SimX, a recently developed library for developing parallel, discrete-event simulations in Python. Written in C++ and Python, SimX enables rapid development and prototyping of a parallel simulation entirely in Python by providing the simulation modeler with core functionality such as processes, event queuing, time advancement, domain partitioning, synchronization and message passing. Designed for both ease-of-use and scalability, applications built using SimX can be executed on multi-core workstations or high performance clusters and can also be easily integrated with other Python tools for scientific computing. In this paper, we briefly discuss the motivation for developing SimX, provide a brief but illustrative example on using SimX to develop an application, a short description of its architecture and some of our initial experiences using SimX in a diverse array of domains. SimX is free software and is publicly available at `http://github.com/sim-x` under the GNU LGPL license.

1 INTRODUCTION

SimX is a generic library for developing parallel, discrete event simulations in Python, designed for both scalability and ease of use. Simulations are the only feasible approach to studying large-scale, complex interacting systems – ant colonies, traffic jams, social and infrastructure networks, to name a few – where the purely local interactions of numerous entities give rise to emergent behavior, that can neither be observed at the local level nor be modeled analytically. For modeling large scale systems, such as cities and social networks, the scalability of a simulation framework then becomes critically important. Simulating the behavior of millions of interacting entities demand significant computational resources, and the need for a framework to support distributed-memory parallelism becomes readily apparent.

On the other hand, developing a scalable simulation model that supports parallelism often becomes a complex undertaking unto itself. The modeler has to take into account issues such as synchronization, distributed message passing and object serialization, to name a few. While libraries for all these tasks readily exist, they often involve steep learning curves and considerable software engineering expertise. A domain expert, such as a social scientist for example, who wishes to study the large scale interactions of intelligent agents in a social network is faced with the task of investing considerable time and effort in complex software engineering issues related to parallel programming, or limiting herself to studying small-scale models.

SimX has been designed keeping both ease-of-use and scalability in mind. The core of SimX is written in C++, providing it with speed for frequently used functionality such as event-queueing, time advancement, domain partitioning and synchronization. APIs needed to develop a simulation application are exported to Python using the Boost Python library (Abrahams and Grosse-Kuntsleve, 2003); these and additional Python wrappers in SimX enable application developers to program entirely in Python. Message passing uses the object serialization facility already present in Python via the fast cPickles (cPickles, 2006) serialization library. Thus any arbitrary Python object that can be serialized (or pickled, as it known in Python parlance) can be sent and received between simulation processes. SimX also supports process oriented simulation, with the facility of suspending an executing simulated process and re-entering at the point of suspension. This is im-

*Los Alamos Unlimited Release LA-UR 12-26739

plemented using light-weight micro-threads (called greenlets (Schmitt, 2012)) that can be used to simulate a large number of concurrent processes with very little overhead. A process oriented paradigm can often greatly simplify the implementation of simulation models, especially those used for computer systems modeling.

The main goal in developing SimX was to enable domain researchers to rapidly develop simulation applications and deploy it on clusters without having to get bogged down by the intricacies of parallel programming, managing MPI communication or worry about issues such as load balancing. In recent years, Python has emerged as a popular and attractive choice for domain scientists due to its ease of use, flexibility and expressive power. This made it a natural choice as a front-end for SimX. A thriving and substantial ecosystem for scientific computing in Python already exists and an application developed in SimX can be easily integrated with other modules available for scientific computing in Python. SimX has been written as a tool for research where developer cycles, are in general, more valuable than program execution time. Indeed, our initial experiences with SimX have shown that the code base of a Python simulation application is often smaller than its C++ counterpart by a factor of four or more. While the flexibility of Python comes with a performance cost, the implementation of the core functionality in C++ provides a reasonable trade-off between ease-of-use and performance. For situations where performance is an over-riding concern, computationally intensive parts of the program can be re-written in pure C++. In fact the entire simulation application itself can be written in C++ as SimX allows for applications to be pure Python, a Python/C++ hybrid or even pure C++.

In the following sections we discuss related work and differences between our approach and others. We then provide a brief architectural overview of SimX, which includes a simple example illustrating some of the features of SimX. We also provide some initial experiences with SimX in terms of the program size of Python simulations compared to equivalent C++ programs, and some initial scaling results.

2 RELATED WORK

The hybrid programming-language approach to simulation is not new, the most notable example of this being the NS-2 network simulator (NS-2,), which uses a C++ backend with a TCL front end, it's parallel and distributed version called PDNS (PDNS,) and more recently NS-3 (NS-3,) which uses C++ and Python for parallel network simulation. SimX, however, is a general purpose library, and not tied to any one domain; it can be used to develop applications in any context where discrete-event simulation is an appropriate modeling paradigm.

There are a number of general purpose libraries available for developing discrete-event simulations. RePast (North et al., 2006) is a collection of open-sourced modules for parallel agent based simulations in C++ and Java. SASSY (Hybinette et al., 2006) is a scalable agent simulation system for PDES that provides a middleware between an agent-based API and a PDES simulation kernel. MASON (Luke et al., 2005) is a discrete-event multi-agent simulation library core in Java, designed to be the foundation for large custom-purpose Java simulation. Another Java-based simulation system is JiST (Barr et al., 2005), that uses a virtual-machine based simulation approach by embedding simulation semantics directly into the Java execution model. PrimeSSF (PRIME,), originally known as DaSSF, is a parallel simulation framework, developed originally for network simulation, that supports both distributed and shared-memory implementations.

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. A more recent effort is the ROme OpTimistic Simulator (ROOT-Sim) (Pellegrini et al., 2011), an open source C/MPI-based simulation package targeted at POSIX systems, which implements a general purpose PDES environment with optimistic synchronization; the simulation models, however, need to be implemented in C.

On the Python side, SimPy (SimPy,) 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, PC-Sim (Pecevski et al., 2009) is a C++ based neural network simulator with a python front-end, that supports both sequential and distributed memory simulations.

Thus while numerous libraries exist for parallel discrete-event simulations for languages in languages such as C, C++ and Java, and while packages also

exist for developing sequential discrete-event applications in Python, to the best of our knowledge SimX is the first publicly available general purpose library for developing parallel discrete-event simulations in Python.

3 A QUICK INTRODUCTION TO USING SIMX

SimX itself is a successor to the SimCore library descrbed in (Thulasidasan et al., 2012), which was primarily a C++ library with a thin Python front-end. In SimX, the architecture is more tightly integrated with Python, with SimX now being a module that can be imported from within Python. SimX is currently distributed in source format, and like many other open-source packages, requires to be compiled and built by the user via an automated build process.

The software prerequisites for building and using SimX are a Python interpreter, a C++ compiler, the Boost libraries (Boost,) and MPI. The source along with build and installation instructions are available at `github.com/sim-x` or can be downloaded and installed via the Python package repository at `https://pypi.python.org/pypi/simx`

The main objects in a SimX simulation application are Entities, Processes and Services. Entities represent physical objects (e.g. an agent) while services (which live on entities) represent the behavior of an agent. Processes represent simulated threads that execute concurrently

Let us consider a simple `HelloWorld` example that consists of a `Person` entity and a `HelloHandler` service attached to a `Person` object. In our simple set up, when a `HelloHandler` residing on a `Person` entity receives a `Hello` message, it sends a `Reply` message to the sending `Person`, to be delivered at some specified time.

Even though this example is highly simplified, it illustrates some of the key ideas for building a SimX application. The Python definition for the `Person` entity is:

```
import simx
class Person(simx.PyEntity):
  def __init__(self,ID,lp,entity_input):
   #do some initialization here
   self.install_service(HelloHandler,Address)
```

Each entity in a SimX application inherits from the `PyEntity` class exported from SimX. At the time of creation an entity is informed of its identity, the id of the simulation process on which it lives and any input parameters if required. The current simulation time is always available to entities (as well as services) via the `get_now()` method. Information regarding which services are to be created on the entity is passed to the entity via the `entity_input` object. The `create_services` method is used to explicitly create services on an entity.While some initializations have been omitted here, the code above captures the essence of the Python class definition. Let us also define the two message objects, `Hello` and `Reply` which are quite simply:

```
# module Message
class HelloMessage:
         def __init__(self, source_id,
                      dest_id):
               self.source_id = source_id
               self.dest_id = dest_id

class ReplyMessage: pass
```

Next consider the `HelloHandler` service that lives on a `Person`. The Python class definition is:

```
import simx
import  Message
class HelloHandler(simx.PyService):

    def __init__(self,name,person,
            service_input):
        # do some initialization here
        self.person = person

    def recv_HelloMessage(self,msg):
          self.send_info(
                    Message.ReplyMessage(),
                    simx.get_min_delay(),
                    msg.source_id,
                    HelloHandlerAddress)

     def recv_ReplyMessage(self,msg):
           print "HelloHandler at",
                  self.get_entity_id
                  "received Reply at"
                  self.get_now()
```

Each service object in a SimX application inherits from the `PyService` class exported from SimX. At creation time, a service is informed of its identity, the entity on which it lives and any input parameters that have been passed in. Since a service and its entity always lives in the same memory space, all the entity functions and data members are available to a service object.

The `send_info()` method referenced above is the communication work-horse of SimX and follows the simple (`what, when, who, where`) paradigm. The parameters to it are the object to be sent, the sending time (offset from current time), the entity to send it to, and the service address on the entity that will receive the message. Any Python object that can be serialized (via the `cPickles` Python module (cPickles, 2006)) can be sent and received inside through SimX.

In addition to sending, services are also capable of receiving messages. If a service needs to receive more than one type of message, as in the above example, the Pythonic way to determine which receive function gets called would be to define a Python dictionary that hashes object types to receive functions, as in:

```
recv_function =
       {'HelloMessage':recv_HelloMessage,
        'ReplyMessage':recv_ReplyMessage}
```

All Python services are expected to define a `recv` function, which gets called each time a message is received at a service. Using the dictionary as defined above, it then becomes quite straightforward to determine which of the two receive handlers defined above gets invoked.

```
def recv(self, msg):
       msg_type = msg.__class__.__name__
       self.recv_function[msg_type]( msg )
```

To actually run the simulation, one simply writes a Python script that creates the desired simulation scenario. Snippets from the `HelloWorld` example are shown below:

```
import simx
from Person import *
from HelloHandler import *
from random import choice

# set some configuration values here
# such as number of processes etc.

simx.init_env()

# create person entities
num_ent = 10
for i in xrange(num_ent):
        simx.create_entity(('p',i), 'Person')

# create a message generation process
class MessageGen (simx.Process):
  def run(self):
   for evt_time in range(1,end_time):
    hello_rcvr = ('p',choice(xrange(num_ent))
    reply_rcvr = ('p',choice(xrange(num_ent))
    hm = HelloMessage(source_id=reply_rcvr)
    simx.schedule_event(evt_time, hello_rcvr,
              HandlerAddr,hm)

    #schedule in chunks of
    #10 time units
    if (evt_time % 10 == 0):
     # go to sleep, and wake up in time
     # to schedule next batch of events
        self.sleep(evt_time - simx.get_now())

mg = MessageGen()
simx.schedule_process(mg)
simx.run()
```

The message generation process in the code snippet above is an example of a simulated *process* in SimX. When the process goes to sleep, other simulation processes execute in the background, advancing time as required. When the process wakes up, it resumes execution at the suspended point. This mechanism is implemented using pseudo-threads in Python called greenlets (Schmitt, 2012) which are essentially extremely light-weight co-routines whose scheduling is explicitly specified by the SimX process scheduling mechanism. Greenlets incur very little context switching overhead; together, with the message passing mechanism of SimX, these are used to implement functionality for process oriented simulation.

Notice that in the above example, entities did not have to be explicitly assigned to processors; by default SimX deals with this by scattering the entities in a round-robin manner, which, in some cases can outperform more sophisiticated algorithms (see (Thulasidasan et al., 2010)). The user, of course, can always explicitly partition the domain based on the characteristics of the problem. Services are always assigned to the same memory space as the entities to which they are attached. The parallelism in a SimX simulation is almost transparent to the user in the sense that the same simulation code can be used for a sequential simulation or a parallel one. The only parameter that needs to be specified is the number of parallel processes to instantiate, which defaults to the number of running MPI processes. For a more involved implementation of the `HelloWorld` example, the reader is referred to the `examples/HelloWorld` directory in the SimX source distribution.

4 SIMX INTERNALS

We present a quick overview of the architecture of SimX in this section; a thorough treatment of SimX internals will be presented in the user manual that the authors are currently working on.

There are four major SimX 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
- **Service:** A service on an entity determines the *behavior* of entities, i.e responses to events.
- **Message:** These are the events in the simulation which include timer events, control events and messages passed between entities in the simulation. Messages are passed and received be-

tween services living on entities. If the entities live in different memory spaces, SimX serializes and packs the data, A reception of a message at a service triggers the handling routine for that particular type of message.

The above three classes are defined in the C++ core. In addition we also have the following class defined in the Python space:

- **Process**: These are Python objects that provide functionality for process oriented simulation. Implemented via Greenlets (Schmitt, 2012), which are extremely light-weight pseudo-threads in Python, processes can suspend execution via sleeping, or waiting on another process (or resource), and then resume execution at the point of suspension.

In Python space, arbitrary objects can be packed and sent to other processes via encapsulation inside a C++ Imessage object. SimX also includes its own multi-threaded simulation engine built on top of the MPI library that provides a conservative barrier-based synchronization and message passing functionality to the application. However, this simulation engine can easily be swapped out with another engine using a simple compile time switch; for example, as an alternative to the native simulation engine, SimX also works with miniSSF (Liu,), a light-weight version of PrimeSSF with conservative synchronization.[2] The modular nature of SimX allows one to plug-in any PDES synchronization and message-passing engine (including those that use optimistic synchronization schemes) with very few changes to the C++ core.

The SimX APIs are exported to Python using the Boost Python library (Abrahams and Grosse-Kuntsleve, 2003) library which provides 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, STL containers, 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++. A graphical schema of the architecture is illustrated in Figure 1

[2]The current version of SimX is distributed with miniSSF.

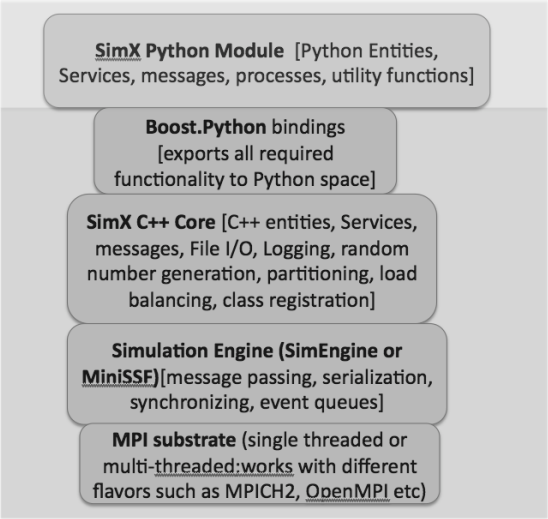

Figure 1: SimX architecture overview

5 SIMX: INITIAL EXPERIENCES AND PERFORMANCE RESULTS

SimX has been written as a tool for research where developer cycles, are in general, more valuable than program execution time. The choice of Python as a front-end for developing simulation applications was motivated by the low learning barrier and flexibility of the language, in addition to its well deserved reputation for rapid prototyping and ease of application development. The flexibility of Python is largely an outcome of its dynamically typed nature, which does result in applications running slower than a pure C++ implementation. In this section we briefly present some initial experiences in porting pure C++ simulation applications to Python using SimX. As a loose metric of programmer productivity, we report on the number of lines of code in the C++ application and compare it to its Python counterpart. We also present initial scaling studies using a simple `HelloWorld` type of application that was discussed in Section 3.

For our initial test-cases, the following applications were ported to Python

- **HelloWorld**. This is a simple sending-receiving application, similar to the application described in Section 3.
- **EduSim** An agent-based simulator for modeling performance of students in school systems. The agents in the simulation are students, teachers and the school system, while the services model the various types of behavior of these agents that are thought to significantly influence the performance

Table 1: Comparison of number of lines of code between equivalent C++ and Python simulation applications

	C++	Python	Ratio
HelloWorld	386	107	3.6
EduSim	3057	901	3.4
AgentCore	1936	561	3.45
TADSim	4057	799	5

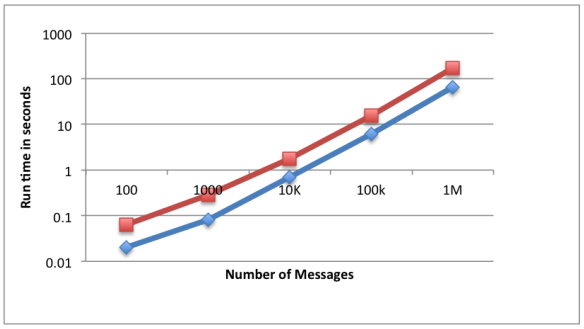

Figure 2: Run time comparison between C++ (Blue) and Python (Red) implementations of HelloWorld

of a school system

- **AgentCore** (Galli et al., 2009), a reactive agent model library that was developed at Los Alamos National Laboratory for developing agent-based simulations.
- **TADSim** A simulation of the execution of a molecular dynamics simulation program (i.e. a simulation of a simulation) that was developed at Los Alamos as part of an effort to better understand and optimize the execution of parallel programs on high performance computing clusters.

Each of the above applications (with the exception of HelloWorld) consisted between 2000 and 4000 lines of code, and were developed over a course of many weeks, often months. While porting, one obviously has the advantage that a redesign of program architecture isn't required during a porting exercise. Nevertheless, it is instructive to see the difference in the amount of lines of code while moving from C++ to Python, shown in the table below.

From the table we see that an equivalent Python program reduces the code base size by upto a factor of 4x compared to the C++ counterpart. The Python porting was often completed in a few hours, and most of this saving comes from the dynamically typed nature of Python whereby type declaration statements are completely absent. Also, in the C++ versions of the simulator, objects have to be explicitly serialized, whereas in Python, it simply involves an invocation to the pickling module.

Figure 2 shows the run-time of a pure C++ implementation of `HelloWorld` in comparison to its Python counterpart, for increasing number of messages, ranging from 100 to upto a million messages in the simulation. This is a simplistic example, completely dominated by message passing, but interestingly enough, the speed-ratio of about two holds constant even as the number of messages are increased. One would expect more computationally intensive scenarios that are able to leverage the highly tuned python libraries for numerical computation to decrease the Python-C++ performance gap.

6 ONGOING AND FUTURE WORK

SimX is, to the best of our knowledge, the first publicly available general purpose library for developing parallel discrete-event simulations in Python. It is currently under active development; new features and bug-fixes are regularly updated on the project code site at `github.com/sim-x`. At Los Alamos, SimX is currently being used to model a variety of complex systems using PDES, such as the performance of computational physics codes running on supercomputers, both from a software and hardware perspective, and modeling of a modern financial reserve system (Williams and Eidenbenz, 2013). We invite simulation researchers and domain scientists to try SimX for their research and provide valuable early feedback to the developers. We also encourage interested simulation developers to participate in further developments to SimX.

REFERENCES

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

Barr, R., Haas, Z. J., and van Renesse, R. (2005). Jist: an efficient approach to simulation using virtual machines: Research articles. *Softw. Pract. Exper.*, 35(6):539–576.

Boost. Boost c++ libraries. http://www.boost.org.

cPickles (2006). http://docs.python.org/2/library/pickle.html.

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*.

Hybinette, M., Kraemer, E., Xiong, Y., Matthews, G., and Ahmed, J. (2006). Sassy: a design for a scalable agent-based simulation system using a distributed discrete event infrastructure. In Perrone, L. F., Lawson, B., Liu, J., and Wieland, F. P., editors, *Winter Simulation Conference*, pages 926–933. WSC.

Liu, J. *Minimalistic Scalable Simulation Framework*. Available at https://www.primessf.net/minissf/.

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., and Balan, G. (2005). Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527.

North, M. J., Collier, N. T., and Vos, J. R. (2006). Experiences creating three implementations of the repast agent modeling toolkit. *ACM Trans. Model. Comput. Simul.*, 16(1):1–25.

NS-2. http://www.isi.edu/nsnam.

NS-3. http://www.nsnam.org.

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

PDNS. Parallel/distributed ns. http://www.cc.gatech.edu/computing/compass/pdns/.

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).

Pellegrini, A., Vitali, R., and Quaglia, F. (2011). The rome optimistic simulator: Core internals and programming model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, pages 96–98, ICST, Brussels, Belgium, Belgium. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

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

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

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*.

Schmitt, R. (2012). http://greenlet.readthedocs.org.

SimPy. http://simpy.sourceforge.net.

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)*.

Thulasidasan, S., Kroc, L., and Eidenbenz, S. (2012). Simcore: A library for rapid development of large scale parallel simulations. In *SIMULTECH*, pages 71–76.

Williams, S. and Eidenbenz, S. (2013). Themis-1: An agent-based model of a modern monetary reserve system. In *Proceedings of the Agent-Directed Simulation Symposium*, ADSS 13, pages 6:1–6:8, San Diego, CA, USA. Society for Computer Simulation International.