Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

THE SIMIAN CONCEPT: PARALLEL DISCRETE EVENT SIMULATION WITH INTERPRETED LANGUAGES AND JUST-IN-TIME COMPILATION

Nandakishore Santhi, Stephan Eidenbenz

Information Sciences (CCS-3)
Los Alamos National Laboratory
Los Alamos, NM, USA

Jason Liu

School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA

ABSTRACT

We introduce Simian, a family of open-source Parallel Discrete Event Simulation (PDES) engines written using Lua and Python. Simian reaps the benefits of interpreted languages—ease of use, fast development time, enhanced readability and a high degree of portability on different platforms—and, through the optional use of Just-In-Time (JIT) compilation, achieves high performance comparable with the state-of-the-art PDES engines implemented using compiled languages such as C or C++. This paper describes the main design concepts of Simian, and presents a benchmark performance study, comparing four Simian implementations (written in Python and Lua, with and without using JIT) against a traditionally compiled simulator, MiniSSF, written in C++. Our experiments show that Simian in Lua with JIT outperforms MiniSSF, sometimes by a factor of three under high computational workloads.

1 INTRODUCTION

Parallel discrete-event simulation (PDES) allows large-scale simulation to run on parallel computers with the goal of accelerating simulation execution (Fujimoto 1990). The performance of a parallel simulator is commonly measured using the event rate, which is the number of executed simulation events per second accrued collectively on the parallel computers. Petascale computing has become a commodity, e.g., the top fifty supercomputers in the November 2014's Top500 list are all ranked as petaflops machines (http://top500.org/). As exascale is a likely reality by the end of the decade, it is reasonable to assume we will continue to see considerable growth in the performance of parallel simulators.

Traditional programming languages, such as FORTRAN, C, and C++, are compiled languages: the programs need to be first compiled into machine instructions before they can be run on the target machine. Compilation is typically divided into two steps: the compilation step translates each program source file into an object file which contains the corresponding machine instructions, whereas the linking step builds a single executable file by combining the object files and resolving the cross references of variables and functions among separate object files and libraries. In general, programs written in compiled languages are fast to run since they have already been translated into the native machine code. Moreover, the compilation step can afford deep program analyses and time-consuming optimizations for most efficient code.

Interpreted languages do not require a compilation step ahead of time. Instead, a separate program "interprets" the instructions in the source code as a set of predefined actions to be executed directly on the target machine. In case of Java and Python, for example, the source code is first translated into an intermediate representation (byte code) and subsequently executed on the Java and Python virtual machine that functions as the interpreter. Interpreted languages are more portable as the language designers can customize the interpreters (or the virtual machines) on specific platforms. This would free the users from compatibility issues, which would otherwise have to be dealt with during the compilation phase of a compiled language. Interpreted languages also provide high-level language features, such as garbage collection, dynamic typing and scoping, with which the users would find it much easier to program. However, the ease of development usually comes with a penalty in the execution speed. Interpreted languages need to be translated during run time with program analyses and optimizations. Implementing those advanced language features would take even more time. As a result, high-performance parallel simulators developed so far all use compiled languages, predominantly in C or C++.

Recent advances in "just-in-time" (JIT) compilation (also known as dynamic translation) have blurred the boundary between compiled and interpreted languages. JIT refers to the technique of performing translation on the fly during program execution (Aycock 2003). Consequently, JIT combines the benefits of ease-of-use and portability of interpreted languages, and the performance advantages of (statically) compiled languages. More importantly, since a JIT compiler can access dynamic runtime information (whereas a standard compiler cannot), it can achieve better optimizations, such as inlining frequently invoked functions and unrolling nested loops. In this case, a "jitted" program language can even yield better performance.

In this paper, we investigate the use of interpreted languages for parallel discrete-event simulation. We introduce Simian, a family of open-source, conservatively synchronized, process oriented, parallel simulators written in two interpreted languages, Python and Lua. The language choices stem from Simian's two design goals. The first goal is to lower the barrier for domain modelers to use parallel discrete-event simulation. One can reduce development time by adopting popular interpreted languages that are easy to use and highly portable among various computing platforms. Python is a popular scripting language, due in part to the "batteries included" language design philosophy and a clean syntax, which has resulted in Python being taught in many disciplines as the first programming language in computer science introductory courses. The second goal is to achieve good simulation performance on high-performance computing platforms. Simian should be able to achieve an event rate comparable to the performance of highly-optimized parallel simulators implemented in C or C++. Although perhaps more of a niche language, often used in the gaming industry and in embedded systems, as we demonstrate later in this paper, Lua can bring significant performance benefits resulting from advanced JIT compilation techniques.

We describe four Simian implementations—namely, SimianPie (a Python version), Pypy Simian (a Python version that allows JIT compilation), Lua Simian (a Lua version), and LuaJIT Simian (a Lua version that allows JIT compilation). We examined the performance of the Simian implementations by comparing it against a state-of-the-art PDES engine, called MiniSSF, written in C++ (Rong, Hao, and Liu 2014). We used the La-PDES benchmark suite (Park, Eidenbenz, Santhi, Settlemyer, and Chapuis 2015), a stylized PDES-application that can reveal various performance aspects of parallel discrete-event simulators, including the efficiency of the event queue management, the event computational workload, the spatial and temporal distribution of events, synchronization mechanisms, memory requirements and access patterns, the buffer management of the communication routines, and so on. La-PDES performs these tests as different input parameters of a single PDES model that controls the random selection and distribution of event arrivals, the computational workloads associated with the events, the source and destination entities and communication delays.

Our main findings can be summarized as follows: (1) JIT compilation achieves approximately an eight-fold performance improvement over regular interpretation for both Python and Lua; (2) higher computational load in the model increases the advantage of JIT compilation; (3) Lua outperforms Python by a factor of eight approximately in both "jitted" and interpreted modes; and perhaps most intriguingly, (4) LuaJIT outperforms MiniSSF by up to a factor of three under high computational loads and retains its leading position even under low computational loads by a few percentage points; it is only taken over by MiniSSF for scenarios with low computational load *and* higher event counts per simulation epoch, due to MiniSSF's advanced synchronization scheme and superior interaction with MPI through multi-threading.

The rest of the paper is organized as follows. We present background and related work in Section 2. In Section 3, we describe the Simian design in detail, including the important design features, the major components, as well as its programming interface. We investigate the performance of the Simian implementations through a comparative performance study in Section 4. Finally, we present our conclusions in Section 5.

2 BACKGROUND AND RELATED WORK

Performance is the primary goal of parallel discrete-event simulation. Extended performance studies (for example, Fujimoto et al. 2003, Perumalla 2007, Bauer Jr. et al. 2009, Barnes et al. 2013) have demonstrated that PDES can scale out to run extremely large scale models on today's supercomputers. Almost all advanced parallel discrete-event simulators are written in compiled languages, predominantly in C or C++. Examples include GTW (Das et al. 1994), DaSSF (Liu et al. 1999), ROSS (Carothers et al. 2000), μsic (Perumalla 2005), and MiniSSF (Rong et al. 2014). While there are many sequential discrete-event simulators written in interpreted languages—for example, JiST (Barr, Haas, and van Renesse 2005), SimPy (SimPy 2015), and Simulua (Carvalho 2008), to name only a few—not as many were specifically designed with parallel simulation capabilities.

Scalable Simulation Framework (SSF) is a public-domain standard for object-oriented parallel discrete-event simulation with both C++ and Java bindings (Cowie, Nicol, and Ogielski 1999). SSF has been used extensively to simulate large-scale computer networks. RePast is a collection of open-source modules for agent-based simulation (North, Collier, and Vos 2006). RePast has several implementations, in Java, .NET (C#, C++, Visual Basic), and Python. It includes a scheduler that supports both sequential and parallel discrete-event simulation. D-MASON (Cordasco, De Chiara, Mancuso, Mazzeo, Scarano, and Spagnuolo 2011) is a parallel version of MASON, which is a discrete-event simulation toolkit written in Java for agent-based models. D-MASON adopts the master-worker paradigm with a simple conservative synchronization strategy to allow neighboring worker processes to exchange information. SimX is a Python library designed specifically for parallel discrete-event simulation (Thulasidasan, Kroc, and Eidenbenz 2014). SimX allows users to rapidly develop parallel simulation models (particularly, scientific applications) in Python that can be easily deployed to run on parallel machines. The core of SimX, including event management and parallel synchronization, is written in C++ to provide the high performance necessary for running large-scale models. In the same vein, PCSIM (Pecevski, Natschlger, and Schuch 2009) is distributed simulator for large-scale neural networks. While the core of PCSIM is written in C++, its primary interface is implemented in Python for ease of use. PrimoGENI (Van Vorst and Liu 2012) is another example. It is a real-time parallel simulator (for modeling large-scale computer networks) with a C++ core and a user interface written in Java and Python.

It is reasonable to believe that compiled languages can render significant performance advantage over interpreted languages. Such has been demonstrated (quite easily) in the past. However, with today's advanced compilation techniques, especially Just-In-Time (JIT) compilation, a re-examination of the performance of some interpreted languages for parallel simulation is warranted. JIT compilation is best thought of as a form of runtime inlining and unrolling of code (Aycock 2003). While static Ahead-Of-Time (AOT) compilers can also unroll and inline code (Cooper and Torczon 2003), they quickly hit a point of diminishing returns for long running loops and large functions with many sub-functions. This is because unlimited inlining when applied to nested programs, exponentially increases final executable size. In the case of a JIT compiler, only a section currently running is unrolled and inlined in memory, thereby preventing the blowup. At the same time, all optimizations that a traditional AOT compiler can perform are possible with a JIT compiler—in fact, JIT compilers can often optimize better, as during runtime, certain variables can be treated locally as constants, which may then be optimized away. In practice runtime optimization is only limited by the compilation overhead, which offsets some of the execution efficiency to be gained by that transformation. JIT compilers are also better at dealing with dynamic languages such as Python and Lua, as they can emit runtime specialized code for variables that are dynamically typed. AOT compilers would typically use *boxed* structures to hold dynamically typed variables with specific fields indicating the types. This strategy would typically end up using more memory.

PyPy and LuaJIT are JIT compilers for the Python and Lua, respectively. In terms of language standards conformance, both at present are closest to their respective non-jitted counterparts, CPython and standard Lua interpreters. However, the manner in which PyPy and LuaJIT compilers operate is quite different. PyPy applies JIT at the interpreter loop, while LuaJIT applies JIT to the application script itself. Also, LuaJIT implements other advanced features, such as NaN-tagging (Gudeman 1993, Caro 1997), to speed up calculation that uses dynamically typed variables. Both PyPy and LuaJIT can turn off the JIT compiler feature, forcing the language to be interpreted on the fly, which is often slower.

3 THE SIMIAN (PARALLEL) DISCRETE EVENT SIMULATION ENGINE

Simian is a parallel discrete event simulation engine implemented in a functionally equivalent fashion in the two interpreted languages, Python and Lua. The main reason for choosing interpreted languages is ease of use and, in the case of Python, large application domain audiences. The reason for choosing Lua is its widely accepted best-in-class performance in just-in-time compilation and its generally minimalist design. In the following, we describe the design principles in more details. Simian is open-source software available at http://simian.lanl.gov and on Github at http://github.com/pujyam/simian.

3.1 Design Principles

Simian is a fairly radical, clean-slate re-design of a PDES engine in Python and Lua that nevertheless can trace its roots to a series of PDES engines developed at LANL: Transims (Nagel and Rickert 2001), EpiSims (Mniszewski, Valle, Stroud, Riese, and Sydoriak 2008), SimCore (Thulasidasan, Kroc, and Eidenbenz 2012), and SimX (Thulasidasan, Kroc, and Eidenbenz 2014) are PDES engines that have met changing users needs in the national security mission of LANL since the mid-nineties. The aggregation of the following features make Simian unique:

- **Simple:** Simian has a minimalistic design. For example, its Python implementation consists of only three source files with less than 550 lines of code (45 KB in total). Such a small size makes it easy to change the source code and add functionality that a user might need, although we have found that the functionality provided by Simian is sufficient for most tasks. Some MPI experts view PDES as a "simple wrapper around three MPI calls"; rather than feel insulted by this view, the Simian team embraces this as a design philosophy. Code simplicity leads to better maintainability.
- **User-friendly:** Simian is designed for domain experts with minimal programming requirements. Simian uses interpreted languages of Python and Lua. Particularly Python enjoys an immense user base from all walks of (professional) life. The large set of libraries for Python as well as its script-like approach to input handling and output visualization allow the user to quickly come to tangible results.
- **Pragmatically scalable:** Simian has full support for running on computing clusters of any size using MPI. In practice, genuine large-scale requirements are the exception rather than the rule. Simian adopts a simple barrier-based synchronization protocol. While adding more advanced features (such as entity migration, topology-based synchronization strategies, and more sophisticated model partitioning and load balancing schemes) is possible in the future, excluding them in the base version is a conscious decision to keep complexity at bay.
- **Portable:** On most platforms, Simian runs right out of the box. Simian minimizes its dependency on third-party libraries. The only dependencies are for supporting data communications (MPI) and user threads (greenlet for Python). Co-routines are a standard feature of Lua.

3.2 Application Programming Interface (API)

The Simian API consists of three modules `simian.lua`, `entity.lua`, and `process.lua` (replacing lua with py for the Python counterpart). A few additional modules are used for specific functions that differ between Python and Lua. We describe all modules in the following:

The Simulation Engine. The Simian simulation engine is defined in the `simian.lua` (or `simian.py`) file, which defines the main class Simian. The most important function is called `run`, which pops events in chronological order out of the event queue and calls the corresponding event handler functions. Setting up MPI and performing synchronization with other MPI ranks is also part of the function. This core part of Simian contains only 60 lines of code in Python.

There also are other functions in the module to create entities and attach services (i.e., event handlers) to entities. We also provide functions to randomly partition the user models among the MPI ranks. While we have found such a random scheme to work well in practice for most purposes, one may find it more useful to overwrite these functions to support more sophisticated partitioning schemes for specific models.

Simian's main loop is a no-frill call to `MPI_AllReduce` for the barrier synchronization. Message packing is done in Lua by using `luaMessagePack`, a third-party open source code (33KB) that is included in the Simian distribution. In Lua, the interface to MPI is done through Lua's effective foreign function interface. In Python, Simian offers two alternatives for MPI and message packing. One solution is to use the `mpi4py` package. This is simple but may add a dependency on a non-standard module whose maintenance we do not control. Alternatively, we can use ctypes as the foreign function interface to MPI coupled with the messagePack protocol. For the latter, we could use pickling to stay true to our philosophy of minimizing external dependencies; however, the loss of performance in this case due to larger resultant MPI payloads outweighs our dependency concerns.

Entities. Simian uses entities as objects that contain event handling functions. Simian entities can be distributed among the MPI ranks for parallel processing. The entity base class is defined in `entity.lua` (or `entity.py`). The key method of the class is `reqService`, which schedules a future event at the entity to be processed by an event handler. If the event is destined for the same entity or one in the same logical process, Simian inserts the event in the local event queue. Otherwise, a timestamped message is sent the appropriate LP. The `attachService` method associates an event handler that processes events destined to this entity.

The entity class also has several methods that deal with simulation processes. Method `createProcess` creates a new process, and `startProcess` starts to run the process. There are also methods for suspending, resuming, and killing processes.

Processes. Simian supports process-based simulation, which is defined in the process class in `process.lua` (or `process.py`). Simian processes are implemented using lightweight co-routines or user-land threads. In the Python version, this is through the use of the `greenlet` library; in Lua, standard language features are used. The arguments passed to the process class constructor include a unique name of the process and a function that the process needs to run. The process class implements functions, such as `sleep`, `wake`, `hibernate`, `spawn` and `kill`. Putting a process to sleep requires a simulation time as the argument, which is the amount of time before the process can resume execution. Waking up a process simply consists of switching to the corresponding co-routine. If a process puts itself into hibernation mode, it will be blocked until explicitly woken up by another process. The hibernate and wake methods are normally used for inter-process communication. One can also create new processes or kill existing ones.

Other modules. In Python, the basic simian, entity, and process modules are the only files in Simian when run using `mpi4py`. Simian without `mpi4py` dependency, needs an object serialization technique, for which we use the MessagePack protocol (MessagePack 2015). So in this case, we add a message packing module, and a ctypes module for interface with MPI, equivalents of which also exist in the Lua version. Additionally, the Lua version includes an implementation of a time-priority event-queue and a simple hashing function for model partitioning across LPs. In Python, corresponding functionality is achieved using the language's standard library. Overall, the Simian code is quite lean. Only 66KB is needed for the Lua version (half of which is used by the third party message packing), and 20KB for the Python version.

3.3 Simple Examples

The Simian distribution comes with a set of examples that illustrate the different modeling philosophies that are possible with Simian. We illustrate the use of Simian with a simple event-based example.

Listing 1: Simian basic example in Python: `hello_events.py`

```
1  # Filename hello_events.py
2
3  from SimianPie.simian import Simian
4  import random, math
5
6  simName, startTime, endTime, minDelay, useMPI = "HELLO_EVENTS", 0, 10, 0.1, False
7  simianEngine = Simian(simName, startTime, endTime, minDelay, useMPI)
8  count = 4
9
10 class Node(simianEngine.Entity):
11   def __init__(self, baseInfo, *args):
12     super(Node, self).__init__(baseInfo)
13
14   def generate(self, *args):
15     targetId = random.randrange(count)
16     offset = minDelay*(1+9*random.random())
17     self.out.write("Time " + str(self.engine.now) + ": Node " + str(self.num)
18       + " scheduling event in Node " + str(targetId) + " at " + str(offset) + " from
            now\n")
19     self.reqService(offset, "generate", None, "Node", targetId)
20
21 for i in xrange(count):
22   simianEngine.addEntity("Node", Node, i)
23 for i in xrange(count):
24   simianEngine.schedService(0, "generate", None, "Node", i)
25
26 simianEngine.run()
27 simianEngine.exit()
```

This example creates four entities each continuously sending events randomly to other entities (including itself). The simulation engine is initialized in Line 7. The initialization parameters are largely self-explanatory: the simulation name is used as file name for output; the start and end time are the simulated time window; minDelay is the minimum time advance that must be obeyed when sending events to other entities; and useMPI is a flag indicating whether to enable the MPI machinery. If the flag is not set, Simian does not require an MPI installation on the system in order to run.

The main part of the simulation starts at Line 21, where we first create `count` number of Node entities and at Line 23 where we seed the initial events at time zero, which would call the `generate` method of the `Node` entity with the proper entity index `i`. The `generate` method at Line 14 randomly chooses a target node id and a time offset, and schedules another event calling the `reqService` method (Line 17).

Output is written to `HELLO_EVENTS.0.out`, where 0 denotes rank. A few lines of the output are:

```
Time 0.331049487408: Node 1 scheduling event in Node 3 at 0.668142932925 from now
Time 0.597027344393: Node 3 scheduling event in Node 1 at 0.210857033655 from now
```

This example is kept as minimalistic as possible. Variations on this are certainly possible, including use of seed value to initialize random number generators, as well as using processes to simulate applications running on each entity. Our next example illustrates the use of processes.

In the example of Listing 2, we create a single entity (on Line 30). In the `__init__` constructor of the entity, we create and start `count` processes that have distinct string names. The process is the single function defined on Line 11. It puts itself to sleep for a random amount of time, and upon waking up resumes execution on the next line. Process based simulation is implemented using events and co-routines under the hood, but the single flow of instructions is a more natural fit for modeling processes. For SimianPie, this example requires the external `greenlet` package to be installed because of the use of co-routines.

Listing 2: Simian process based example in Python: `hello_procs.py`

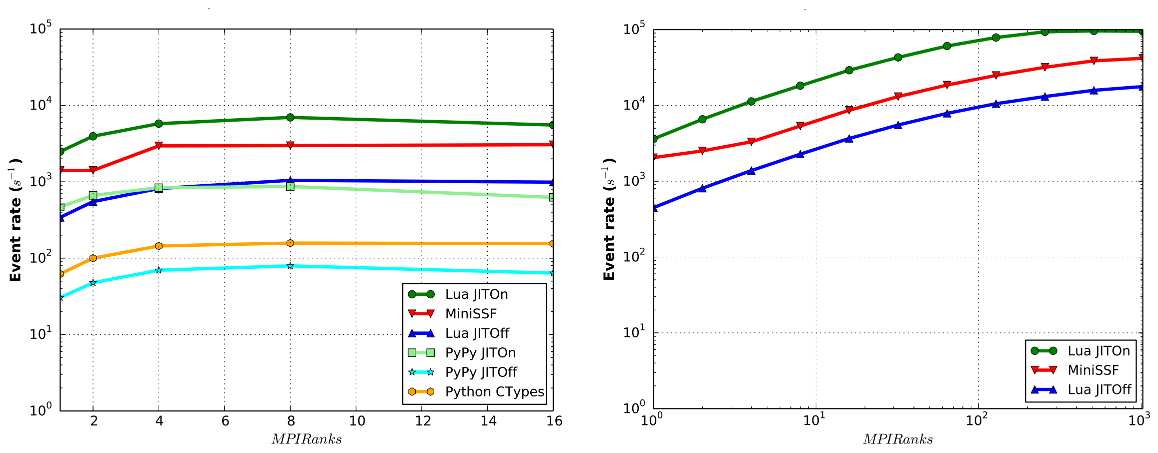
```python
# Filename hello_procs.py

#Initialize Simian
simName, startTime, endTime, minDelay, useMPI = "HELLO_PROCS", 0, 1000, 0.01, False
simianEngine = Simian(simName, startTime, endTime, minDelay, useMPI)

count = 10
maxSleep = 100

#Example of a process on an entity
def appProcess(this): #Here arg(1) "this" is current process
  entity = this.entity
  entity.out.write("Process App started\n")
  while True:
    x = random.randrange(0, maxSleep)
    entity.out.write("Time " + str(entity.engine.now) \
      + ": Process " + this.name + " is sleeping for " + str(x) + "\n")
    this.sleep(x)
    entity.out.write("Time " + str(entity.engine.now) \
      + ": Waking up Process " + this.name + "\n")

class Node(simianEngine.Entity):
  def __init__(self, baseInfo, *args):
    super(Node, self).__init__(baseInfo)
    for i in xrange(count):
      appName = "App" + str(i)
      self.createProcess(appName, appProcess) #Create "App[i]"
      self.startProcess(appName) #Start "App[i]" process

simianEngine.addEntity("Node", Node, 1)

simianEngine.run()
simianEngine.exit()
```

4 EXPERIMENTS AND RESULTS

We conducted experiments to study the performance of Simian. In this section, we present the performance results of the different versions of Simian, namely, SimianPie (a Python version), Pypy Simian (a Python version that allows JIT compilation), Lua Simian (a Lua version), and LuaJIT Simian (a Lua version that allows JIT compilation). We compare the performance against that of the C++-based MiniSSF implementation.

We used a variation of the La-PDES benchmark suite (Park, Eidenbenz, Santhi, Settlemyer, and Chapuis 2015), which consists of eight different cases (with sets of parameters) to focus on different aspects of a discrete-event simulation application. La-PDES is a single file with the eight different cases being defined as different parameter settings.

In brief, the La-PDES application simulates n_{ent} entities, each of which sends an average m_{ent} events to another entity. The events are distributed in time with an exponentially-distributed inter-arrival time with an additional min delay fixed at 1.0. The simulation ends at t_{end}. Each entity has an array of length l, which is randomly chosen but remained fixed throughout the simulation run. The array allows us to



(a) On an 8 core desktop machine (b) On a mid-range cluster with more than 64×16 cores

Figure 1: Parallel Scaling

mimic memory allocation for each entity. Upon receiving an event, an event handler executes on average ops_{ent} floating point operations by accessing the first k percent of the elements in the array and multiplying with some random weights. The average number of send events per entity is parameterized by s_{ent}. The events are distributed among the entities according to either a geometric or uniform distribution for both sending and receiving entities. The array length can also be distributed according to a geometric or uniform distribution. To test the event queue, each entity maintains q_{avg} of its future events in the event queue.

4.1 Parallel Scaling Experiments

As a first result, we present classic scaling runs to see what effect MPI parallelization has. The results are presented in Figure 1, where both panels show event rates on the y-axis and the number of MPI ranks on the x-axis. The left panel shows results from test runs performed on a desktop computer, whereas the right panel shows results from runs performed on a mid-range cluster.

For scaling studies on a desktop computer with 8 cores, we set the following La-PDES parameter values: $n_{ent} = 100$, $s_{ent} = 100$, $q_{avg} = 1$, $ops_{ent} = 100{,}000$, $t_{end} = 1000$. All other parameter values are set to default values according to the La-PDES specifications (Park, Eidenbenz, Santhi, Settlemyer, and Chapuis 2015). Looking at Figure 1 (a), we note that (i) parallelization indeed improves performance a little for all simulators, which is due to relatively large value of operations ops_{ent}: MPI communication latency can be more effectively hidden by the compute loads, resulting better parallel scaling as the problem size increases versus larger number of MPI ranks. (ii) LuaJIT is the clear winner in this scenario across all rank counts. LuaJIT outperforms its nonJIT-Lua by a factor 8. (iii) nonJIT-Lua and the jitted Python (Pypy) perform at about the same level. (iv) Unfortunately, and perhaps somewhat surprisingly, pure Python (PyPy and it's nonJIT cousin CPython) performs about another factor of 8 to 10 worse than nonJIT-Lua. The minimalist nature of Lua pays off here. Python appears to suffer from inefficient interface to the MPI library and its slower, more complex virtual machine. Better JIT compilers are perhaps the answer to Python's deficiencies. (iv) Most remarkably, LuaJIT outperforms MiniSSF. The slightly jagged shape of the MiniSSF curve is due to a different core discretization model: MiniSSF uses 3 threads per LP, and thus saturates the 8 core machine with 2 hardware threads per core at rank count 4.

The results in Figure 1 (b) are shown only for LuaJIT with jitting turned on and off, as well as MiniSSF. The La-PDES parameters are identical except for the number of entities n_{ent}, which we set to 1000. As seen from the plots, Simian Lua based benchmark tests are seen to scale well with more processor cores. The Simian performance stabilizes above $MPI\,Rank = 1000$, as that was the total number of entities in this test case. MiniSSF, continues to show performance gains till $MPI\,Rank = 1000$ as it uses separate threads for I/O, send and receive message buffering, which are able to utilize the additional processors above the number of simulation entities. The scaling behavior of all three simulation engines is as expected, showing big gains in event rate at lower MPI ranks, and tapering to stabilize as number of ranks exceeds number of LPs (more appropriately, total number of threads for MiniSSF). LuaJIT retains its dominant position, beating MiniSSF also at large rank counts.

We performed additional experiments according to the specification of La-PDES, where unevenly distributed computational as well as event loads are distributed across the entities. The relative positions of the different simulation engines do not change under these scenarios.

4.2 Computationally Intensive Loads

The performance gains of jitting rely on the computational load as defined by ops_{ent}. In this second experiment, we vary ops_{ent}, while setting the other parameters fixed to the same values as in the previous experiments with 100 entities and 100 expected send events, i.e. $n_{ent} = 100$, $s_{ent} = 100$,

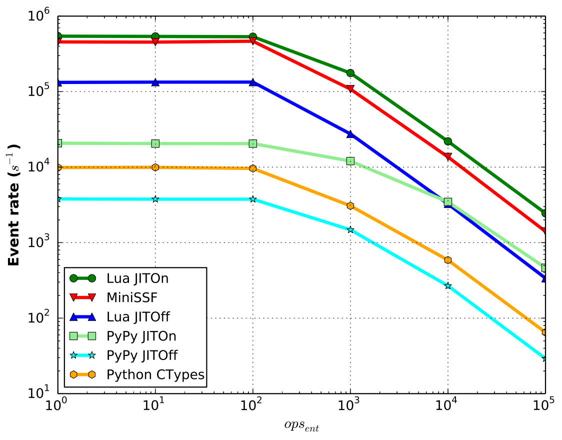
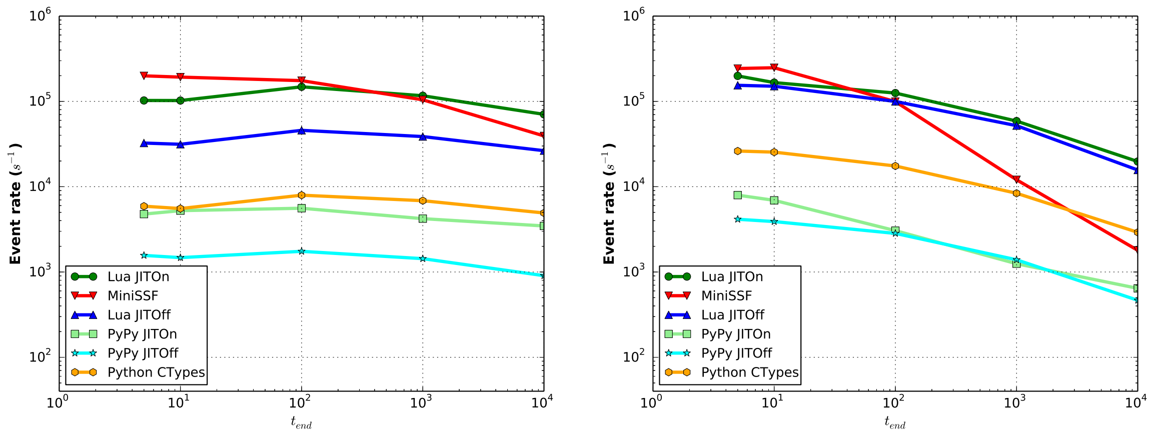


Figure 2: Computationally Intensive Loads

As we see in Figure 2, as ops_{ent} increases, ratio of the computation load due to the simulation model to the background load due to the simulation engine increases. The time spent in computation per event increases, as well. This means that the JIT compilers have more opportunities to optimize code, and more time is spent executing code than compiling it at runtime. While Luajit with JITing turned off initially performs better, Pypy with JIT turned on eventually succeeds in giving a better event-rate.



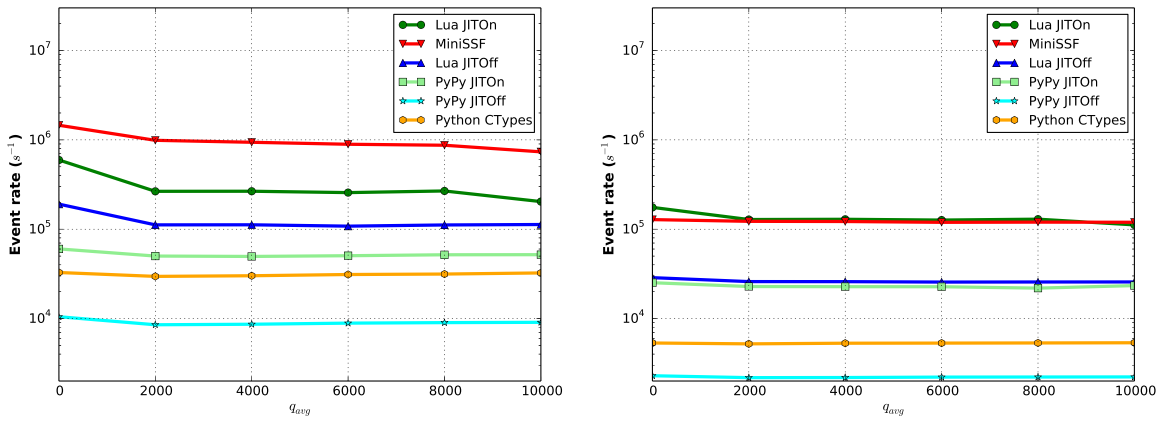
(a) $MPI\,Rank = 2$ (b) $MPI\,Rank = 16$

Figure 3: Quality of Message Buffering

4.3 Quality of Message Buffering

Because MPI message buffers can be a source of concern for PDES engines, La-PDES has a scenario that varies the end time t_{end} parameter without changing the number of total events. To be more precise, we decrease t_{end} from $n_{ent} * s_{ent}$ to 5 for various values of number of MPI ranks. Under this test, as the value of t_{end} decreases, the same number of events get scheduled in a shorter amount of time. This results in two things: (1) the remote messages get queued into the MPI buffers, increasing network communication pressure and increasing synchronization overhead, and (2) the received messages at each entity gets queued locally in its event-queue, potentially increasing the queue sizes and thereby increasing `push` and `pop` times. All other parameter values remain as in previous runs, except ops_{ent}, which we set to 1000 here.

MiniSSF initially performs better than Simian Lua because of the worker threads pushing messages using the extra processing resources available in the desktop machine. However, as the number of MPI ranks increases, the extra threads ($3 \times MPI Ranks$) tend to compete for valuable resources, and bogs down the event-rates. Pure Lua implementation of the priority-queue also shows its relative inefficiency compared to the C++ version at lower t_{end} values and higher number of MPI ranks. The statically typed C++ event-queues are better able to deal with the message buffering at these settings. However, at lower number of MPI ranks and/or higher t_{end} values, JITed application code tends to scale better.



(a) $ops_{ent} = 1$ (b) $ops_{ent} = 1000$

Figure 4: Efficiency of Queue Management

4.4 Efficiency of Event Queue Management

In order to test the efficiency of the event queue management, we run experiments varying the q_{avg} parameter, which indicates how many events per entity are in the event queue at the steady state of the simulation. Other parameter values are as follows: $n_{ent} = 100$, $s_{ent} = 10000$, $t_{end} = 1000$. The value of ops_{ent} was set to either 1 or 1000.

Efficient priority-queue data structures relying on heaps as their backbone, have `push` and `pop` times $O(\log n)$, where n is the number of elements in the queue. For larger values of q_{avg}, the priority-queue which stores the future events in the LP's timeline gets larger, and n increases, thereby also increasing the workloads of sorting the queue in place according to time-priority. We found that all language and compiler combinations behave in a similar fashion with varying q_{avg}. Since Simian Lua version has a pure Lua implementation of the event-queue, at larger q_{avg} values, we see a slightly decreased performance in JITed mode. At larger q_{avg} values, since the event-queues can become large, we also have side-effects due to the computer hardware's memory hierarchy and limited cache sizes. Notably, the two plots show clearly how the ordering of LuaJIT and MiniSSF depends on the computational load of ops_{ent}. At low computational load MiniSSF actually wins over Simian LuaJIT.

5 CONCLUSION

Our main insight is that the use of interpreted languages in combination with Just-In-Time compilation techniques allow for the same level of PDES performance as traditionally compiled simulation engines while still reaping the benefits of interpreted languages of being simple to handle. The trend towards JIT compilation is of course not limited to PDES; perhaps a bit boldly, we project that within 15 years, traditional AOT compilation will be found only in historical computer science texts as Just-In-Time compilation will have become the norm. In the short term, a Python PDES engine opens up parallel discrete event simulation to a much larger audience of application domain experts.

We encourage other researchers to challenge or validate our results. As for concrete future work, we plan to look at higher scaling regimes and extend our comparative performance studies to other PDES engines.

REFERENCES

Aycock, J. 2003. “A Brief History of Just-in-time”. *ACM Computing Surveys* 35 (2): 97–113.

Barnes, Jr., P. D., C. D. Carothers, D. R. Jefferson, and J. M. LaPre. 2013. “Warp Speed: Executing Time Warp on 1,966,080 Cores”. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS’13)*, 327–336.

Barr, R., Z. J. Haas, and R. van Renesse. 2005. “JiST: An Efficient Approach to Simulation Using Virtual Machines”. *Software Practice and Experience* 35 (6): 539–576.

Bauer Jr., D. W., C. D. Carothers, and A. Holder. 2009. “Scalable Time Warp on Blue Gene Supercomputers”. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation (PADS’09)*, 35–44.

Caro, A. 1997. “A Novel 64 Bit Data Representation for Garbage Collection and Synchronizing Memory”. MIT CSAIL Memo 396, April 1997. http://csg.csail.mit.edu/pubs/memos/Memo-396/memo-396.pdf. Last accessed: Feb 10, 2015.

Carothers, C. D., D. Bauer, and S. Pearce. 2000, May. “ROSS: A High-Performance, Low Memory, Modular Time Warp System”. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation (PADS’00)*, 53–60.

Luis Carvalho 2008. “Simulua: Discreate-Event Simulation in Lua”. http://simulua.luaforge.net/.

Cooper, K. D., and L. Torczon. 2003. *Engineering a Compiler*. Morgan Kaufmann.

Cordasco, G., R. De Chiara, A. Mancuso, D. Mazzeo, V. Scarano, and C. Spagnuolo. 2011. “A Framework for Distributing Agent-based Simulations”. In *Proceedings of the 2011 International Conference on Parallel Processing (Euro-Par)*, 460–470.

Cowie, J. H., D. M. Nicol, and A. T. Ogielski. 1999. “Modeling the global Internet”. *Computing in Science and Engineering* 1 (1): 42–50.

Das, S., R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. 1994. “GTW: A Time Warp System for Shared Memory Multiprocessors”. In *Proceedings of the 26th Conference on Winter Simulation (WSC’94)*, 1332–1339.

Fujimoto, R. M. 1990. “Parallel Discrete Event Simulation”. *Communications of the ACM* 33 (10): 30–53.

Fujimoto, R. M., K. Perumalla, A. Park, H. Wu, M. H. Ammar, and G. F. Riley. 2003. “Large-Scale Network Simulation: How Big? How Fast?”. In *Proceedings of the 11th Annual Meeting of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS’03)*.

Gudeman, D. 1993. “Representing Type Information in Dynamically Typed Languages”. TR 93-27, Department of Computer Science, University of Arizona, Tucson, AZ, October 1993.

Liu, J., D. Nicol, B. Premore, and A. Poplawski. 1999, May. “Performance Prediction of a Parallel Simulator”. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation (PADS’99)*, 156–164.

MessagePack 2015. “The MessagePack Protocol”. http://msgpack.org. Last accessed: June 20, 2015.

Mniszewski, S. M., S. Y. D. Valle, P. D. Stroud, J. M. Riese, and S. J. Sydoriak. 2008. “EpiSimS simulation of a multi-component strategy for pandemic influenza”. In *Proceedings of the 2008 Spring Simulation Multiconference, SpringSim 2008, Ottawa, Canada, April 14-17, 2008*, 556–563.

Nagel, K., and M. Rickert. 2001. “Parallel Implementation of the TRANSIMS Micro-simulation”. *Parallel Computing* 27 (12): 1611–1639.

North, M. J., N. T. Collier, and J. R. Vos. 2006. “Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit”. *ACM Trans. Model. Comput. Simul.* 16 (1): 1–25.

E.J. Park and S. Eidenbenz and N. Santhi and B. Settlemyer and G. Chapuis 2015. “Parameterized Benchmarking of Parallel Discrete Event Simulation Systems: Communication, Computation, and Memory”. LA-UR-15-22468, Los Alamos National Laboratory Technical Report.

Pecevski, D., T. Natschlger, and K. Schuch. 2009. “PCSIM: a parallel simulation environment for neural circuits fully integrated with Python”. *Frontiers in Neuroinformatics* 3 (11).

Perumalla, K. S. 2005. “μsik – A Micro-Kernel for Parallel/Distributed Simulation Systems”. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation (PADS’05)*, 59–68.

Perumalla, K. S. 2007. “Scaling Time Warp-based Discrete Event Execution to 104 Processors on a Blue Gene Supercomputer”. In *Proceedings of the 4th International Conference on Computing Frontiers (CF’07)*, 69–76.

Rong, R., J. Hao, and J. Liu. 2014. “Performance Study of a Minimalistic Simulator on XSEDE Massively Parallel Systems”. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment (XSEDE’14)*, 15:1–15:8.

SimPy 2015. http://simpy.sourceforge.net, Last accessed: May 2015.

Thulasidasan, S., L. Kroc, and S. Eidenbenz. 2012. “SimCore: A Library for Rapid Development of Large Scale Parallel Simulations”. In *SIMULTECH 2012 - Proceedings of the 2nd International Conference on Simulation and Modeling Methodologies, Technologies and Applications, Rome, Italy, 28 - 31 July, 2012.*, 71–76.

Thulasidasan, S., L. Kroc, and S. Eidenbenz. 2014. “Developing Parallel, Discrete Event Simulations in Python - First Results and User Experiences with the SimX Library”. In *4th International Conference On Simulation And Modeling Methodologies, Technologies And Applications, SIMULTECH 2014, Vienna, Austria, August 28-30, 2014*, 188–194.

Van Vorst, N., and J. Liu. 2012. “Realizing Large-Scale Interactive Network Simulation via Model Splitting”. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation (PADS)*, 120–129.

AUTHOR BIOGRAPHIES

NANDAKISHORE SANTHI is a Computer Research Scientist with the Information Sciences Group (CCS-3) at Los Alamos National Laboratory. He holds a PhD in Electrical and Computer Engineering from the University of California San Diego. His areas of research interests include parallel discrete event simulation, performance modeling of HPC systems, applied mathematics, communication systems and computer architectures. His email address is nsanthi@lanl.gov.

STEPHAN EIDENBENZ is the Director of the Information Science and Technology (ISTI) institute at Los Alamos National Laboratory. He obtained a PhD from the Swiss Federal Institute of Technology, Zurich (ETHZ) in Computer Science. His research interests include cyber security, computational codesign, communication networks, scalable modeling and simulation, and theoretical computer science. His email address is eidenben@lanl.gov.

JASON LIU is an Associate Professor at the School of Computing and Information Sciences, Florida International University. He received a Ph.D. degree from Dartmouth College in Computer Science. His research focuses on parallel simulation and high-performance modeling of computer systems and communication networks. His email address is liux@cis.fiu.edu.