Proceedings of the 2018 Winter Simulation Conference
M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, eds.

JUST-IN-TIME PARALLEL SIMULATION

Christopher Hannon
Dong Jin

Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616, USA

Nandakishore Santhi
Stephan Eidenbenz

Information Sciences (CCS-3)
Los Alamos National Laboratory
Los Alamos, NM 87545, USA

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

ABSTRACT

Due to the evolution of programming languages, interpreted languages have gained widespread use in scientific and research computing. Interpreted languages excel at being portable, easy to use, and fast in prototyping than their ahead-of-time (AOT) counterparts, including C, C++, and Fortran. While traditionally considered as slow to execute, advancements in Just-in-Time (JIT) compilation techniques have significantly improved the execution speed of interpreted languages and in some cases outperformed AOT languages. In this paper, we explore some challenges and design strategies in developing a high performance parallel discrete event simulation engine, called Simian, written with interpreted languages with JIT capabilities, including Python, Lua, and Javascript. Our results show that Simian with JIT performs similarly to AOT simulators, such as MiniSSF and ROSS. We expect that with features like good performance, user-friendliness, and portability, the just-in-time parallel simulation will become a common choice for modeling and simulation in the near future.

1 INTRODUCTION

We have only seen the tip of the iceberg that holds the potential of Discrete Event Simulation (DES) and its scalable cousin Parallel Discrete Event Simulation (PDES) as being an indispensable modeling tool for application domain scientists. DES as a tool in a researcher's toolbox should rival the popularity of machine learning tools. Alas, in reality, very few application domains heavily rely on DES, and even less for PDES. While the slow adoption of DES by most application domains may have multiple reasons, we believe that one key hurdle is the relatively high entry barrier that exists for DES and even higher for PDES. This may represent a loss of opportunity in many application domains. For example, the advent of exascale computing (expected by 2021) with its massive parallelism has put scale regimes for PDES within reach that in the past had been left to other more coarse-grained modeling paradigms, such as fluid approximations. Exascale computing will allow PDES to claim a much larger stake in inherently discrete or easily discretized domains, such as infrastructure modeling, social dynamics models, and even aspects of particle physics.

To grow the (P)DES user base, one needs to lower the barrier to entry by offering tools that are more straight-forward to use, especially by potential application domain scientists, without sacrificing performance. We postulate that developing simulators written with interpreted languages supported by Just-In-Time compilation techniques offer the solution. Just-in-Time (JIT) compilation refers to dynamically performing translations during program execution. As such, it allows program analysis and code optimizations (such as constant folding, branch elimination, and loop unrolling) to take advantage of runtime information that is otherwise unavailable to statically compiled languages.

In this paper, we present Simian, a full-fledged, open-source, parallel discrete event simulation platform with support for multiple languages (including Python, Lua, JavaScript, and their respective JIT versions, Pypy, LuaJIT and SpiderMonkey), multiple parallel simulation synchronization algorithms (both conservative and optimistic), and optimized data structures and functions for JIT compilation. We also contrast and evaluate various PDES design choices for JIT compilers and AOT compilers and their respective trade-offs. These improvements significantly expand upon the original bare-bone Simian implementation (Santhi et al. 2015), while keeping with the original lean design philosophy to remain scalable, simple, and easy to use. This design philosophy sets Simian apart from existing simulation engines. Additionally, our performance results demonstrate that simplicity and good performance do not need to be exclusive. The design features and use of just-in-time compilation techniques enable Simian to be both easy to use and efficient. Through our investigation into the performance of JIT compiled data structures and functions, we found that the pure Python and Lua priority queue implementations were faster than interfacing with an external C module. Additionally, we observe that just-in-time compiled data structure operations can sometimes outperform ahead-of-time compiled code. We also observe limitations of just-in-time compiled math functions in comparison to ahead-of-time math functions. To evaluate the performance of Simian, we compare the simulators event rates over various benchmark configurations.

The remainder of the paper is organized as follows. In Section 2, we present our Simian platform. In Section 3, we overview just-in-time compiled languages. In Section 4, we conduct experiments to reveal design options for just-in-time parallel simulation and their performance trade-offs. We conduct performance studies using a benchmark suite to evaluate our just-in-time simulator in Section 5. In Section 6, we discuss related work, including programming languages and existing state-of-the-art discrete-event simulators. We conclude the paper and outline future research directions in Section 7.

2 SIMIAN

Simian is a high-performance parallel discrete-event simulation (PDES) engine written in interpreted languages. Currently, Simian has been implemented in three languages: Python, Lua, and JavaScript. Two distinct features set Simian apart from other existing PDES engines. First, the primary design goal of Simian is to have a simulator that is simple-to-use, user-friendly, and portable. The main user base contains physicists, computational scientists, and other domain experts, who would use Simian to develop application models, such as particle simulations and fluid dynamics models, on high-performance computing platforms. As such, to maintain simplicity, Simian intentionally keeps its code base small and easy to understand. Using highly developed interpreted languages reduces the barrier for developing complex models and facilitates fast prototyping. Also, because interpreted languages do not require explicit recompilation when executed on different platforms, they provide easy portability. The second distinct feature of Simian is the use of state-of-the-art JIT techniques. We explore specific simulator design consideration to take advantage of JIT features. We discuss them in Section 4.

2.1 Language Support

The Simian simulation engine has currently been implemented in three interpreted languages: Python, Lua, and JavaScript. While the implementations have syntactic differences due to the language choices, they all share the same design. It is important to note that the overarching design principle of Simian is to keep a lean and modular design for simplicity and ease-of-use. In particular, to facilitate ease-of-use and portability, we require all implementations to minimize dependencies on external libraries and tools. As such, for all implementations of Simian, we prefer using language-native tools and packages that do not require additional user installation effort.

Simian's Python implementation works with the default CPython interpreter as well as Pypy (pypy.org), the just-in-time compiler for Python. Simian's Lua and Javascript implementations use the native JIT compiler provided by the specific languages, specifically LuaJIT (luajit.org) and SpiderMonkey (2018). The Luajit compiler can be used as is, because of the language's native support for co-routines, which are the Lua version for user-space micro-threads. Simian's Javascript implementation needs a customized version of SpiderMonkey, which includes native support for loading user-written modules at runtime, support for passing command line arguments to scripts, and also supports for native methods that implement a limited subset of MPI needed by Simian.

2.2 Parallel Synchronization

Simian supports both conservative and optimistic synchronization algorithms and allows the user to select the synchronization method at run-time. Currently, the conservative synchronization mechanism is implemented in Python, Lua, and JavaScript. The optimistic approach, however, has only been implemented in Python. We are currently developing the Lua and JavaScript implementations.

The conservative synchronization algorithm is based on the window-based YAWNS protocol (Nicol 1993). The implementation involves an MPI call to `MPI_AllReduce`, which forces a barrier synchronization between all parallel simulation instances and determines the size of the next synchronization window (defined by the lower bound of timestamps of all future events). The barrier synchronization moves the global simulation time to the next minimum timestamp across all pending event queues. The parallel simulation instances process local events independently within the synchronization window and then repeat the barrier synchronization.

The optimistic implementation is based on Time Warp (Jefferson 1985), which uses anti-messages and state saving to maintain temporal accuracy. For each event, an additional message is created as an inverse event. Before executing a pending event, the engine checks if an inverse event is also presented in the pending event queue. If so the two events cancel out and the next pending event is executed. This approach is slightly different than the classical approach. We take a lazy approach to the annihilation of events: When an anti-message or inverse event is created, instead of searching the pending event queue data structure and removing matching events, the event is added to the pending event queue. If the matching event has already been executed, the inverse event will be added to the front of the pending event queue; or in the case multiple inverse events are added concurrently, the one that rolls back the local engine's clock to the earliest timestamp will be the next event to be dequeued. Thus to annihilate, after dequeuing, the Simian engine checks to see whether the next pending event has the same timestamp. If the timestamps match, then the engine compares the events to see if they cancel out. This mechanism enables fast annihilation at the cost of a larger priority queue.

Each Simian engine instance maintains both a global clock that is synchronized according to the GVT approximation algorithm by Mattern (1993), and a local clock that is the timestamp of the last executed event. When an out-of-order event is processed, the state of the engine is rolled back to the time of the event, and for all events that were created after the timestamp, an anti-message or inverse event is sent. After GVT calculation is done all states that are before GVT can be reclaimed to free up memory.

3 JUST-IN-TIME COMPILATION

Just-in-Time (JIT) compilation, also known as the dynamic compilation, refers to the technique of performing translations dynamically during the program execution.

3.1 JIT Languages

JIT compilation has a long history with the earliest mention in McCarthy's LISP paper in 1960 (Aycock 2003). JIT compilation aims to achieve better performance when compared with either statically compiled programs (also known as Ahead-of-Time or AOT compilation) or interpreted programs.

The static compilation translates program source code, normally written in high-level programming languages, into executable code, either assembly code or machine instructions that can be directly executed on the target hardware. Statically compiled programs have the advantage of undergoing expensive program analyses and optimizations, possibly through several iterations of code optimizations, during the compilation time, and thus can produce efficient code before program execution.

Interpreted programs are highly portable. The programs can be first compiled into a machine-independent representation (such as Java bytecode), and on a target machine, only an interpreter is needed to run the program. Since interpretation is performed at runtime, an interpreter can access runtime information not available for static compilation. Consequently, interpreted languages can provide high-level programming semantics in their language design, such as dynamic typing, which can be difficult to implement for statically compiled languages.

JIT brings together the advantages of both static compilation and interpretation. On the one hand, dynamic compilation allows code analysis and optimizations to take place. On the other hand, the translations can take advantage of runtime information, such as control flow information, execution profile, and information of the target architecture. Several optimizations can be performed by a JIT compiler during runtime. For example, JIT compilers can generate constant addresses as opposed to references to variables (constant folding). JIT compilers can detect hot-path, perform branch prediction or branch elimination, and method inlining, based on the execution profile collected during runtime. The same techniques can be used to unroll loops, which often consist of instructions to increment a pointer or index to access array elements and perform end-of-loop tests. JIT compilers can generate code with pre-calculated offsets to the array elements and therefore avoid those arithmetic operations at runtime.

By default, Python is interpreted. Like Java, Python first compiles the source code into bytecode, which is a low-level platform-independent representation of the source code. The Python Virtual Machine (PVM) interprets bytecode instructions and performs the operations—one instruction at a time. CPython is the standard Python interpreter implementation (in C). There are ways to speed up Python execution including using an Ahead-of-Time (AOT) compiler, such as Cython (cython.org), or using a JIT compiler, such as PyPy (pypy.org). PyPy was originally launched as a project for improving the Python implementation and later evolved into an environment for implementing dynamic languages (Rigo and Pedroni 2006), although its primary focus still remains on efficient Python implementation. PyPy supports automated tracing JIT compiler generation (Bolz et al. 2009).

Lua is a scripting language more commonly used on embedded devices for its high performance and low memory footprint. LuaJIT is a JIT compiler, which is one of the fastest dynamic language implementations with a highly optimized trace compiler (luajit.org). There exist several JavaScript compilers.

Mozilla's SpiderMonkey (2018) is a JavaScript engine used in Firefox web browser. Internally, SpiderMonkey uses two JITs compilers: JaegerMonkey and IonMonkey. JaegerMonkey is a baseline general-purpose JIT used for normally hot traces. However, IonMonkey will be called for heavyweight optimization. This strategy allows the JIT compiler to gradually optimize code and balance between code optimization levels and compilation time.

3.2 Interfacing with C Code

While Just-in-time compilation has sped up the execution time of interpreted languages, we find that it is often necessary to invoke algorithms or modules written in C code. For example, to run a parallel simulation on distributed-memory machines, PDES engines usually need to perform communications through message passing. It is undesirable to implement some functions directly in Python, Lua, JavaScript, and alike.

Consequently, these languages would need to support the execution of C code within the interpreter or just-in-time compiler in order to utilize existing code and libraries written in C.

There are several techniques for calling C modules from interpreted languages: C code can be integrated directly into the interpreter or the just-in-time compiler; CTypes allows for languages to cast native data types as C data types, while foreign function interfaces provide a mechanism for native data types to be converted to C data types automatically.

Built-in modules are written in C inside the interpreter or compiler. This produces a high-performance interface but requires the most effort. For example, CPython interpreter implements some libraries and functions built into the interpreter, such as HeapQ, Math, Sys, and many more. Similarly, PyPy, a JIT compiler for Python, implements a subset of these libraries. Writing a C extension module requires an extended knowledge of the internals of the interpreter including rebuilding it after modification.

Foreign Function Interface (FFI) is the default mechanism in LuaJIT to interface with external libraries and functions (Rigo and M. Fijalkowski 2012). Similar to LuaJIT's FFI, Python includes CFFI, the C foreign function interface. One objective of FFI is to eliminate the need for users to learn the domain-specific languages and its complex APIs. One such common API for interpreted languages is the CTypes interface which provides C compatible data types. CTypes enables pure Python to call shared libraries or DLLs (Python Software Foundation 2010). The major difference between FFI and CTypes is the location of the code that bridges the gap from interpreted language to C. FFI implementations generate C code to accept Python data types, while CTypes provides the Python programmer the ability to cast their Python data into C compatible data types.

Other methods exist for integration of C code into Python, Lua, and JavaScript but typically require compilation of the interpreted language program ahead-of-time, or require run-time directives such as static types. Overall, we focus on the popular and most well-established techniques to interface C code with interpreted language code so as to be consistent with the Simian's minimalistic philosophy.

4 JIT SIMULATION

In this section, we investigate two core PDES components under JIT and evaluate the performance trade-offs between different implementation approaches and language choices. In particular, we explore the design and implementation approaches in the pending event queue of the PDES engine, and the pseudo-random number generators.

4.1 Pending Event Queue

Discrete event simulation advances through the execution of timestamped events. Events are executed chronologically in simulation time. In order to organize pending events, the simulation engine must maintain some internal data structures for storing pending events. A priority queue is a simple structure that can be implemented. A binary heap can push (insert) and pop (remove) elements in $O(log\ n)$, where n is the number of current pending events in the heap. If the simulation allows event cancellation, or when the simulation runs optimistic synchronization, the delete operation must also be supported. Since the pending event queue is a critical component of the PDES engine, there has been a lot of research on improving the performance of enqueue and dequeue operations using optimized priority queue data structures, such as the calendar queue (Brown 1988; Tan and Thng 2000) and the ladder queue (Tang et al. 2005). Multi-tiered data structures have also been explored (Higiro et al. 2017).

In order to evaluate the performance of interpreted and just-in-time compiled languages, we design three micro-benchmarks presented in Table 1 to evaluate the simulator's pending event queue data structure:

- Test 1 (Static Priorities): Enqueue 1,000,000 events with fixed time increments, followed by 1,000,000 dequeues.
- Test 2 (Random Priorities): Enqueue 1,000,000 events with random future times, followed by 1,000,000 dequeues.

Table 1: Performance of Data Structure Implementations (in Seconds).

Implementation	Static Priorities	Random Priorities	Interleaved Operations
CPython C Heapq	3.434	9.167	3.720
CPython Python Heapq	8.417	13.261	7.789
CPython CFFI (API Mode)	3.541	5.974	5.227
CPython Ctypes	6.349	12.4	10.2
Pypy heapq (default module)	**0.855**	2.246	**0.371**
Pypy Python Priority Queue	**0.873**	2.243	**0.345**
Pypy CFFI (API Mode)	1.903	2.832	1.660
Pypy Ctypes	18.4	18.5	18.5
Pypy CalendarQ (Pure Python)	1.049	3.158	15.371
Pypy Fibonocci Heap (Pure Python)	1.353	9.947	1.259
LuaJIT Pure Lua	0.933	2.271	0.574
Pure C code	**0.850**	**1.588**	0.663

- Test 3 (Interleaved Operations): Enqueue a random number of events with random future times, followed by a random number of dequeues. This process repeats until 1,000,000 events have all been enqueued and dequeued.

The various test results are shown in Table 1, we take the average of 100 runs, each run reports the best of 100 test executions. We compute the times in this manner to eliminate interference with OS scheduling, caching, etc. The executions are run on a single core, on a 2.3GHz Intel i5 processor running macOS High Sierra. Random priorities run slower than static priorities due to the additional overhead of random number generation as well as the random placement in the data structure. Interleaved operations often run fastest because interleaving the operations results in a smaller heap at all times in comparison to the first two tests. While the number of push and pop operations are the same throughout all tests, the time for each operation is proportional to the logarithm of the size of the heap.

CPython, the default Python interpreter, runs slower than Pypy, a Python JIT compiler. Specifically, CPython runs Tests 1 and 2 about 4 times slower than PyPy, and runs Test 3 about 10 times slower. CPython executes the built-in C module to maximize performance while Pypy executes native code faster than their corresponding C functions. Likely this is because the interface from Python to C creates a bottleneck. Additionally, within CPython, the CFFI interface to the C library actually executes faster than the default built-in heapq module on Test 2. Because Pypy outperforms CPython significantly, we further focus only on evaluating the JIT compilers.

Comparing the performance between Pypy, LuaJIT, and C, it is evident that just-in-time compiled languages can perform as well as or even better than pre-compiled languages. While Test 1 performed as well in Pypy as in C, LuaJIT performed slightly slower. C outperformed all others in Test 2 while LuaJIT and Pypy outperformed C in Test 3. In Test 3, there are a dynamic number of loops through enqueue and dequeue operations, this illustrates one of the advantages of JIT compilation. Specifically, jitted languages can utilize the run-time knowledge to unroll dynamic loops in the executing code.

The results show that for the binary heap data structure, the performance of native code is better than calling pre-compiled code. However, as discussed previously, it may not be possible to convert every package to native code, such as the message passing libraries needed for PDES. In such case, we see that the Ctypes mechanism seems to perform worse than the foreign function interface (CFFI) in Python. Thus if the PDES engine or an application developer requires C routines that cannot be easily translated to native code, we should prefer using the CFFI mechanism to interface with the existing C routines or libraries.

We also implemented a calendar queue data structure (CalendarQ) in Python and compared its performance to that of the binary heap. For the first two tests, the performance is relatively good while the third test shows that the performance becomes much worse. The calendar queue requires *resize* operations whenever the number of events currently in the queue goes across some threshold. While the C implementations of

Table 2: Performance of Pseudo-Random Number Generators (in Seconds).

Implementation	LCG	GFSR	Implementation	LCG	GFSR
CPython–Pure Python	0.33719	2.10156	Pypy–Pure Python	0.12245	0.47669
CPython–C Module	n/a	.64059	Pypy–C Module	n/a	0.15565
Lua–Pure Lua	0.09255	0.59783	Luajit–Pure Lua	0.0156	0.04367
Lua–C Module	0.06466	n/a	Luajit–C Module	**0.01109**	n/a
C code	**0.00813**	0.01082			

the calendar queue are able to preallocate space and efficiently reuse the available space, such fine-grained controls are not present in the Python implementation. Thus, while native code can run faster than the corresponding C module from within Python, it is not necessarily true for all cases, as may be evident in the calendar queue data structure. Future work is needed to investigate whether optimization techniques adopted by AOT compiled languages, such as C and C++, can achieve the same effect in JIT compiled languages.

In summary, interpreted languages can speed up execution of pending event queues by incorporating C modules when using the default interpreter. Just-in-time compilers can outperform C modules by executing native code.

We also found that Pypy and LuaJIT have similar performance advantage than that of C, which motivates us to conduct further validations at the full PDES level.

4.2 Pseudo-Random Number Generation

When investigating the pending event queue data structure implementations, we found that the implementations in Python and Lua were able to perform no worse than the C code implementation. Further, when we introduce random priorities for enqueued events in Test 2, we observed a large difference between Python/Lua and C. In this section, we investigate different language implementations for pseudo-random number generation, which uses math functions.

There are many pseudo-random number generators, which can be largely divided into two common categories: Linear Congruential Generators (LCG) and Generalized Feedback Shift Register (GFSR). In Table 2, we compare the performance of Python, Lua, and C with implementations of the two types of random number generators. We report the results of generating one million random numbers. Overall the LCG algorithm is faster than the GFSR algorithm, and the C module implementations outperform the pure code versions. Also, Luajit performs slightly worse than a pure C code implementation. It is possible to speed up the execution of Python by implementing a C module inside the JIT compiler. However, as we mentioned earlier, implementing a module in the JIT compiler requires extensive knowledge of the programming languages. One cannot expect all application developers to possess these skills.

From the experiments, we conclude that the C code still has the performance advantages for implementing the low-level functions (such as the pseudo-random number generation), although the gap is not as significant as one may have originally believed (such as the case for LuaJIT). The performance advantages of JIT compiled languages lies in dynamic run-time optimizations, which we explore in the next section.

5 BENCHMARK PERFORMANCE

To evaluate the performance of the simulator, we use the La-PDES benchmark suite (Park et al. 2015), which is designed to provide a configurable model for profiling simulation engines for a wide range of applications including communication networks, infection propagation, computing and more. In order to determine the scaling properties of Simian, and its parallel performance, we run the benchmark on a mid-range cluster using from 16 to 2048 cores.

5.1 La-PDES Benchmark Suite

There are ten primary input parameters for the benchmark that represent the model behavior and configuration of the simulation applications:

- **n_ent:** the number of entities (simulated processes)
- **s_ent:** the number of events created per entity throughout the life of the simulation
- **m_ent:** the average number of integers stored in entities memory to be used for event processing
- **p_send:** the parameter for the geometric distribution of **s_ent** over entities
- **p_receive:** the parameter for the geometric distribution of remote events
- **ops_ent:** the average number of operations calculated during each event
- **ops_sigma:** a parameter for normal distribution of **ops_ent**
- **p_list:** a geometric parameter affecting **m_ent** and **ops_ent**
- **cache_friendliness:** the percent of integers recalculated in **m_ent** every event
- **q_avg:** the number of concurrently pending events in the event queue per entity

The input parameters determine the size of the simulation, the location of the created events, the amount of work associated with processing each event, the destination of remote events among the entities, and so on. We compare the performance of Simian in different interpreted languages with that of ROSS (Carothers et al. 2002) and MiniSSF (Rong et al. 2014), the two simulators written in C/C++.

We determine the impact on overall simulation performance by varying the benchmark input parameters. We generate 24,000 Sobol sequences using the SALib library. Sobol sequences are pseudo-random numbers that better cover the parameter space than truly random numbers. We use quasi-Monte Carlo simulation of the benchmark and variance-based sensitivity analysis over 16 processors on the event rate to determine the impact of each variable on the execution of the simulator. We consider variables statistically significant if the total order sensitivity values are greater than 0.05. The simulations are run in the conservative mode with the priority queue heap data structure, on the Python version of Simian using Pypy. The results of the sensitivity analysis show that **ops_ent, p_list, p_receive, q_avg, s_ent,** and **n_ent** are statistically significant in contributing to the overall execution time of the simulator. To identify the impacting parameters of La-PDES, we compare the performance of Simian along with MiniSSF and ROSS on a partition of 4 nodes, 16 MPI ranks.

5.2 Benchmark Results

By varying the processing time required for handling each event, we can observe how the different simulators perform. We set the **ops_ent** parameter from 10^2 to 10^5 while keeping the other parameters constant. Figure 1 shows the event execution rate of the Simian implementations compared to MiniSSF and ROSS. ROSS running in optimistic mode outperforms Simian when **ops_ent** is 100; however, it drops below LuaJIT at 1000, and eventually falls below the Python implementations and MiniSSF. At lower computation levels the simulator experiences a higher number of rollbacks, while at higher levels spends more time processing events. LuaJit outperforms the other Simian implementations as well as MiniSSF, while the Conservative Python implementation of Simian performs about the same as MiniSSF.

The **p_list** parameter is related to the processing time and the amount of work associated with each event processing. Specifically, **m_ent** is a scalar factor that, together with **n_ent** (the number of entities), determines the size of the list that the event handler will access the state stored in memory. **p_list** is the parameter of the geometric distribution, which determines the size of the list after the previous calculation. Essentially entities will have a geometric distribution of state memory size. We set this geometric parameter from 0.0 to 0.6 and measure the event rate of Simian, MiniSSF, and ROSS. Interestingly, we see that the performance of Simian decreases slightly as the parameter increases. ROSS, on the other hand, has a significant performance boost as the size of **p_list** increases and then plateaus. MiniSSF remains constant and performs about the same as Simian using conservative synchronization. We also observe the same trend of LuaJIT, which has the best performance of all Simian implementations.

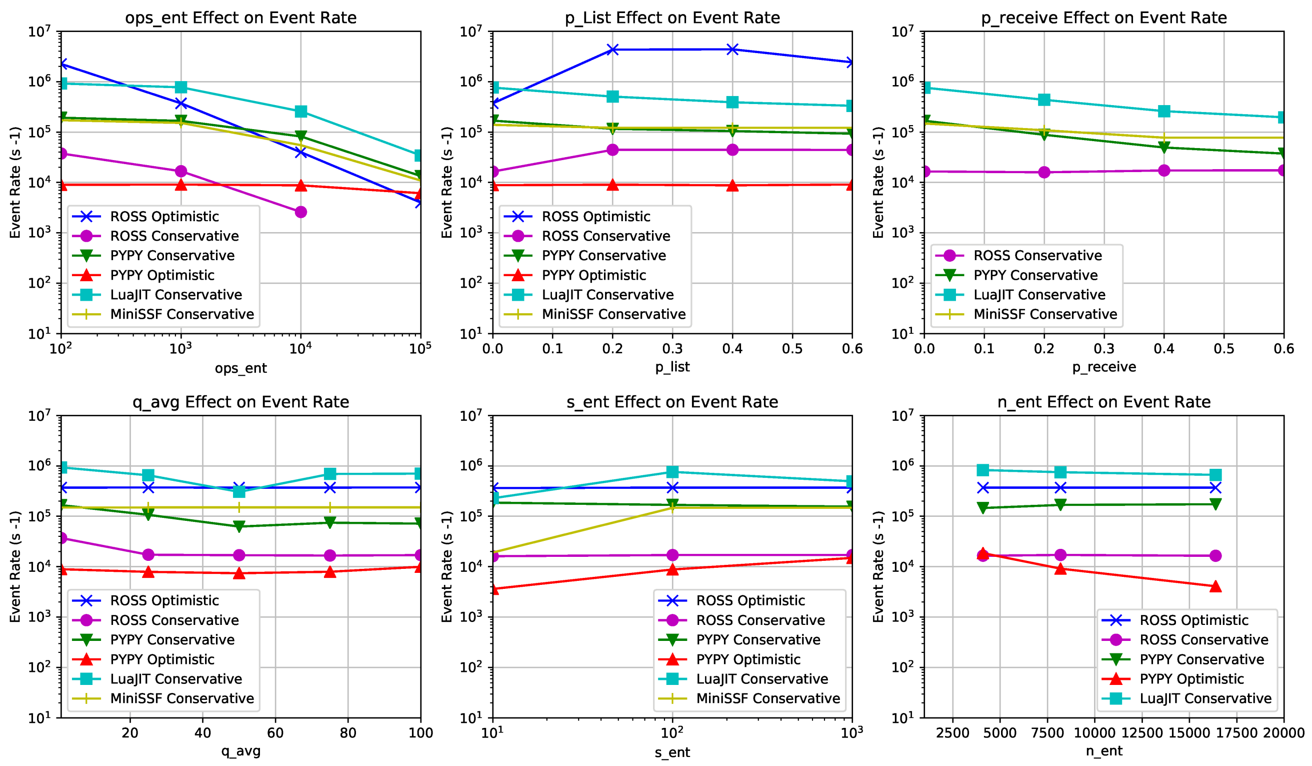

Figure 1: Performance Comparison Under Various Parameter Settings.

The **p_receive** parameter determines the recipients of created events. A value of 0 is a uniform distribution of event recipients while a value of 1 means that all entities send events to a single entity analogous to an all-to-one reduction. Simian performs best when the geometric distribution parameter is small. There is no noticeable performance change in ROSS. Because the sensitivity analysis measures the total order of effect of a parameter on the execution speed of the simulation, this includes the interaction of parameter settings that cannot be observed while varying only the **p_receive** setting. The number of pending events affects the performance of the PDES engines only mildly for possibly the same reason. The size of the pending event queue will determine the speed at which events can be enqueued and dequeued. The **q_avg** is the number of events that a given entity will have scheduled at all times. Interestingly, the event rate of LuaJIT drops at 50 but then increases again. The number of events created by each entity is specified by the **s_ent** parameter. Simian and MiniSSF increase performance significantly from 10 to 100 sent events, this is possibly due to the startup overhead of the simulators. More entities on a processor will only have a significant effect on the Simian optimistic mode.

5.3 Scaling Performance

To illustrate the performance gains through parallelization, we set La-PDES benchmark parameters to their default values and increase the number of MPI ranks from 2^4 to 2^{11}. Each simulation runs 8192 entities with 100 sent events per entity, and 10^6 operations per event. The distribution of events sent and received is set to be uniform.

We compare the performance of MiniSSF, ROSS, and Simian to see how the simulator implementations scale. For these runs, we utilize the Grizzly cluster at LANL which is a mid-range cluster with 53,640 cores with Xeon E5-2695v4 18C 2.1GHz processors. Our job utilizes 70 nodes of 36 cores and experiments are averaged over multiple runs. Figure 2 shows the comparison of the Simian implementations with the ROSS Simulator. We observe that Lua outperforms Python and ROSS. Python optimistic outperforms conservative when the number of ranks is larger than 2^7, while Python conservative is better at fewer ranks. Figure 2 also shows a comparison between the various Simian language implementations and shows that the performance of SimianLua leads SimianJS and SimianPie.

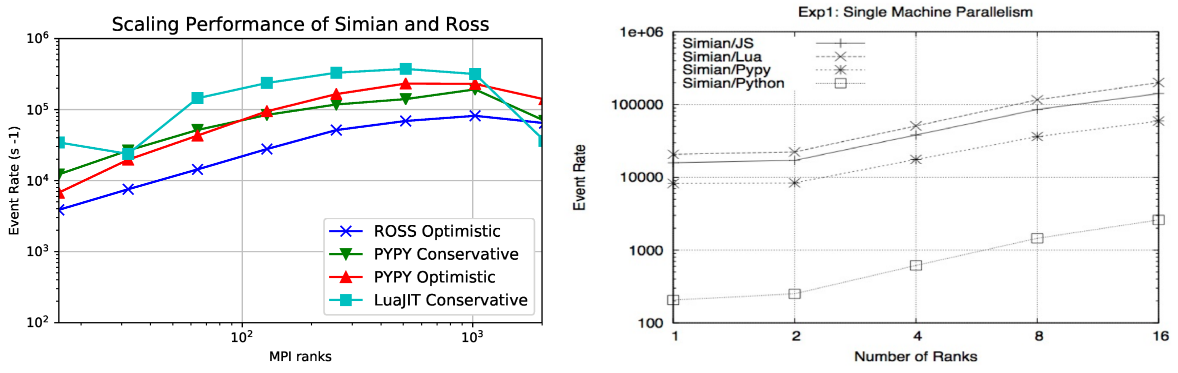

Figure 2: Performance of Simian platform against Ross from 4 to 2048 cores. Lua has the best performance from 16 to 1024 cores. Lua also outperforms JavaScript and Python.

6 RELATED WORK

Traditional scientific computing and high performance applications are commonly written in C, C++, or Fortran. Interpreted languages are typically dismissed as slow, however, advances in JIT compilers have boosted the performance of languages like Python, Lua, and JavaScript. As JIT compilers bridge the performance gap, we expect to start to see a shift in the language choices for scientific applications, including PDES applications running over a million cores (Barnes et al. 2013).

Existing discrete event simulators include ROSS (Carothers et al. 2002), MiniSSF (Rong et al. 2014), Parsec (Bagrodia et al. 1998), μSik (Perumalla 2005), and more. These parallel simulators are predominantly written in C and C++. There are simulators written in interpreted languages, such as SIM.JS (Vars‘hney 2011) in Javascript, JiST (Barr et al. 2004) in Java, SimPy (Team SimPy 2017) in Python, and Simulua (Carvalho 2008) in Lua. However, they do not support parallel execution. There is little existing work using interpreted languages for parallel simulation. SimX (Thulasidasan et al. 2014), and PCSIM (Pecevski et al. 2009) use interpreted languages as the application programming interface; however, they all use AOT compiled languages at the simulation core. As such, these approaches are quite different from Simian, whose PDES core is written directly in interpreted languages.

7 CONCLUSION AND FUTURE WORK

In this paper, we present the Simian family of just-in-time parallel discrete-event simulators. Advances in JIT compilation techniques enable Simian to perform comparably to existing state-of-the-art parallel simulators written in C/C++, such as ROSS and MiniSSF. There are inherent differences in the compilation methods which can affect the parallel simulator design; we investigate some of those differences.

More specifically, we study the performance of Simian using the La-PDES benchmark. We perform a sensitivity study to determine the effect of various model parameters on the overall performance of the simulation engines. In future work, we plan to further investigate how and why the simulation is influenced by the applications and how this impacts other applications. From the parallel performance results, we can observe that Simian outperforms ROSS. Interestingly, we also observe that the optimistic version of PyPy outperforms the conservative approach with higher ranks. We plan to further analyze the performance of Simian, including the Lua and JavaScript versions. If we can obtain a better understanding of the behavior of PDES applications, and how the PDES engine is affected, it will enable us to devise further optimizations for JIT compilation. A key design goal of Simian is to provide an easy-to-use, flexible, user-friendly, and high-performance simulation interface.

Our results show that simulators using JIT compilers can often outperform AOT compiled implementations with native data structures, while ahead-of-time compiled languages outperform just-in-time compiled languages for operations that invoke native code. Our investigation into the event queue data structures provides motivation to re-evaluate optimization techniques designed for AOT programming languages and how these techniques can be translated to performance gains or losses for JIT compiled languages. In future work, we would like to understand the extent of the differences between the two compilation paradigms and investigate the implications on parallel simulation, especially parallel synchronization and communication.

ACKNOWLEDGMENTS

This paper is partly sponsored by the Air Force Office of Scientific Research (AFOSR) Grant YIP FA9550-17-1-0240, the Department of Homeland Security (DHS) Grant 2017-ST-062-000002, the National Science Foundation (NSF) Grants CNS-1730488 and CNS-1563883, and the Maryland Procurement Office under Contract No. H98230-18-D-0007. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of AFOSR, DHS, NSF, and the Maryland Procurement Office.

REFERENCES

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

Bagrodia, R., R. Meyer, M. Takai, Y.-A. Chen, X. Zeng, J. Martin, and H. Y. Song. 1998, Oct. “Parsec: a parallel simulation environment for complex systems”. *Computer* 31(10):77–85.

Barnes, 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 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS ’13, 327–336. New York, NY, USA: ACM.

Barr, R., H. Zygmunt, and R. Van Renesse. 2004. “Jist: Embedding Simulation Time Into a Virtual Machine”. In *Eurosim Congress on Modelling and Simulation*. Paris, France.

Bolz, C. F., A. Cuni, M. Fijalkowski, and A. Rigo. 2009. “Tracing the Meta-level: PyPy’s Tracing JIT Compiler”. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, 18–25. New York, NY, USA: ACM.

Brown, R. 1988. “Calendar Queues: A Fast 0(1) Priority Queue Implementation for the Simulation Event Set Problem”. *Communications of the ACM* 31(10):1220–1227.

Carothers, C. D., D. Bauer, and S. Pearce. 2002. “ROSS: A High-Performance, Low-Memory, Modular Time Warp System”. *Journal of Parallel and Distributed Computing* 62(11):1648–1669.

Carvalho, L. 2008. “Simulua Discrete-Event Simulation in Lua”. http://simulua.luaforge.net/.

Higiro, J., M. Gebre, and D. M. Rao. 2017. “Multi-tier Priority Queues and 2-tier Ladder Queue for Managing Pending Events in Sequential and Optimistic Parallel Simulations”. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. Singapore.

Jefferson, D. R. 1985, July. “Virtual Time”. *ACM Transactions on Programming Languages and Systems* 7(3):404–425.

Mattern, F. 1993. “Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation”. *Journal of Parallel and Distributed Computing* 18(4):1–20.

Nicol, D. M. 1993. “The Cost of Conservative Synchronization in Parallel Discrete Event Simulations”. *Journal of the ACM* 40(2):304–333.

Park, E., S. Eidenbenz, N. Santhi, G. Chapuis, and B. Settlemyer. 2015. “Parameterized Benchmarking of Parallel Discrete Event Simulation Systems: Communication, Computation, and Memory”. In

Proceedings of the 2015 Winter Simulation Conference, edited by L. Yilmaz et al., 2836–2847. Piscataway, New Jersey: IEEE.

Pecevski, D., T. Natschläger, 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 *Workshop on Principles of Advanced and Distributed Simulation (PADS'05)*, 59–68. Monterey, CA, USA.

Python Software Foundation 2010. "A Foreign Function Library for Python". https://docs.python.org/2/library/ctypes.html.

Rigo, A., and S. Pedroni. 2006. "PyPy's Approach to Virtual Machine Construction". In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, 944–953. New York, NY, USA: ACM.

Rigo, A., and M. Fijalkowski 2012. "CFFI Documentation". https://cffi.readthedocs.io.

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. New York, NY, USA: ACM.

Santhi, N., S. Eidenbenz, and J. Liu. 2015. "The Simian Concept: Parallel Discrete Event Simulation with Interpreted Languages and Just-in-time Compilation". In *Proceedings of the 2015 Winter Simulation Conference*, edited by L. Yilmaz et al., 3013–3024. Piscataway, New Jersey: IEEE.

SpiderMonkey 2018. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

Tan, K. L., and L.-J. Thng. 2000. "SNOOPy Calendar Queue". In *Proceedings of the 2000 Winter Simulation Conference*, edited by J. A. Joines et al., 487–495. Piscataway, New Jersey: IEEE.

Tang, W. T., R. S. M. Goh, and I. L.-J. Thng. 2005. "Ladder Queue: An O(1) Priority Queue Structure for Large-scale Discrete Event Simulation". *ACM Transactions on Modeling and Computer Simulation* 15(3):175–204.

Team SimPy 2017. "Discrete Event Simulation for Python". https://simpy.readthedocs.io/en/latest/.

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 Int. Conf. on Simulation and Modeling Methodologies, Technologies and Applications*, 188–194. Vienna, Austria.

Vars'hney, M. 2011. "SIM.JS Discrete Event Simulation in JavaScript". http://www.simjs.com/.

AUTHOR BIOGRAPHIES

CHRISTOPHER HANNON is a Ph.D. student in the Computer Science Department at the Illinois Institute of Technology in Chicago, Illinois. His research focuses include simulation and modeling, critical infrastructure security, and blockchain technologies. His e-mail address is channon@iit.edu.

NANDAKISHORE SANTHI is a computer scientist at Los Alamos National Laboratory (LANL). His research interests include *Beyond Moore's Law* computing paradigms (ASIC hardware acceleration and quantum), algebraic error correction, and discrete event simulation. His mail address is nsanthi@lanl.gov.

STEPHAN EIDENBENZ is a computer scientist at Los Alamos National Laboratory (LANL). Stephan's research interests include scalable discrete event simulation, discrete algorithms, and quantum computing. He leads the Information Science and Technology Institute (ISTI) (isti.lanl.gov) at LANL. His mail address is eidenben@lanl.gov.

JASON LIU is an Associate Professor at the School of Computing and Information Sciences at Florida International University. His research focuses on parallel discrete-event simulation, high-performance modeling and simulation of computer systems and networks. His email address is liux@cis.fiu.edu.

DONG (KEVIN) JIN is an Assistant Professor in the Computer Science Department at the Illinois Institute of Technology. His research interests include simulation modeling and analysis, trustworthy cyber-physical critical infrastructures, software-defined networking, and cyber-security. His email address is dong.jin@iit.edu.