Proceedings of the 2024 Winter Simulation Conference
H. Lam, E. Azar, D. Batur, S. Gao, W. Xie, S. R. Hunter, and M. D. Rossetti, eds.

EFFICIENT PARALLEL SIMULATION OF NETWORKED SYNCHRONOUS DISCRETE-EVENT SYSTEMS

Neha Karanjkar[1], Madhav Desai[2], Akhil Kushe[3], and Anish Natekar[1]

[1]School of Mathematics and Computer Science, IIT Goa, Goa, INDIA
[2]Dept. of Electrical Eng, IIT Bombay, Mumbai, INDIA
[3]Dept. of Electronics and Telecomm Eng, Goa College of Engineering, Goa, INDIA

ABSTRACT

We present Sitar, an open-source, general-purpose modeling framework consisting of a custom modeling language and a simulation kernel, designed for efficient parallel simulation of networked Synchronous Discrete-event Systems (SDES) in applications such as communication networks and computer systems design. A unique feature of Sitar is its two-phase, cycle-based simulation algorithm that allows an efficient, race-free parallel execution on shared-memory systems. This is achieved by imposing a mild restriction on the set of SDES that can be described within the framework. The modeling language is designed for describing complex, networked systems with a static interconnection structure. We demonstrate Sitar's modeling capability, performance and scalability through a detailed performance evaluation and a comparison against SystemC and SimPy. Our results show that Sitar's single-threaded performance is better than that of SimPy and comparable to that of SystemC, whereas a multi-threaded execution shows near-linear scaling for several benchmark configurations.

1 INTRODUCTION

1.1 Overview

Synchronous Discrete-Event Systems (**SDES**) represent a subclass of discrete-event systems where all state-changes (*events*) occur at integer multiples of a fixed time unit. Systems across a variety of application domains such as communication networks, computer architecture models, synchronous digital systems and queueing networks are typically modeled as SDES. Given the complexity and scale of models in these domains, parallel simulation approaches are crucial. We show that imposing a mild restriction on the set of SDES that can be described within a framework (that is, restricting a framework's scope to a smaller, yet practically rich subset of SDES) presents the opportunity for a targeted simulation approach with an efficient parallel implementation. Yet, there is a gap in the availability of modeling frameworks specifically targeting this subset. Models in these application domains are often built, either as a standalone application-specific simulator bundling together custom code for time-advancement, *or* are built using general-purpose simulation frameworks such as SimPy or SystemC that are meant for a broader class of discrete-event systems, thus missing out on the potential for targeted, parallel simulation approaches.

Sitar is an open-source, general-purpose modeling framework that addresses this gap. It consists of a custom modeling language and a lightweight, C++ based simulation kernel with a shared-memory parallel implementation. The modeling language provides a rich set of constructs for describing complex, networked SDES with a static interconnection structure, typical in domains such as communication networks and computer systems. It supports modular and hierarchical descriptions of system structure as interconnected concurrent entities. The behaviour of each entity can be described in an imperative manner using constructs such as conditional and unconditional delays (‘`wait`’ statements), branches, loops, and fork-join concurrency in the control-flow and embedded (instantaneous) code blocks. A model description in Sitar is translated

into highly readable C++ code, which can be compiled with the simulation kernel to generate an executable, or linked with external libraries for co-simulation.

A unique feature of Sitar is its **two-phase simulation algorithm** that affords a straightforward, race-free parallel implementation. This is made possible by imposing a weak restriction on the set of SDES that can be modeled within the framework (detailed in Section 1.2). Sitar was initially built as an internal tool for computer architecture research. It has been used for building cycle-accurate multi-core processor models (Karanjkar and Desai 2015) and for modeling discrete-time queueing networks (Karanjkar, Desai, and Bhatnagar 2016). An updated version of Sitar, incorporating features for general-purpose use, was released as open-source software with an MIT licence (https://sitar-sim.github.io/sitar). A discussion of Sitar's modeling language constructs and a use-case in processor modeling are presented in (Karanjkar and Desai 2022).

1.2 Main Contributions

A key feature of Sitar is its two-phase simulation algorithm that enables a straightforward, race-free parallel implementation. This is made possible by limiting the framework's scope to a subset of SDES. Specifically, this subset consists of synchronous discrete-event systems that can be described as being composed of interconnected, communicating components (called *modules*) wherein, propagating an update from the module's inputs to its output incurs a delay of at-least one unit of time. In the domain of digital systems, this is akin to a model consisting of communicating Moore-type components. Any components that communicate instantaneously have to be encapsulated within a single module boundary. Modules can then be split across threads/processes for parallel simulation (using an algorithm described in Section 4) without the need for a model dependency analysis. Thus, the modeling framework implicitly aids in identifying potential model partition boundaries for work division during parallel execution. It should be noted that this subset is still practically sufficient to encompass system models across many application domains. We discuss the implications of this restriction on Sitar's modeling scope in Section 4.0.3.

In this paper, we present the context and the key ideas behind the two-phase simulation approach used by Sitar and demonstrate its effectiveness through a detailed performance and scalability evaluation using a parameterized benchmark model. We also present a comparison of Sitar with two popular open-source simulation frameworks: SystemC and SimPy in terms of simulation performance and modeling capability. Although Sitar is targeted for a subset of the systems that SimPy or SystemC can model, for benchmark models within this subset we find that Sitar's base (single-threaded) performance is significantly better than that of SimPy and comparable to that of SystemC, whereas a multi-threaded execution in Sitar shows near-linear scaling with a speed-up close to **50x** on a 40-core machine. (We present detailed results in Section 5.) This paper aims to position Sitar as a unique open-source framework filling a gap in targeted, parallel simulation of synchronous discrete-event systems.

2 CONTEXT AND RELATED WORK

We consider the modeling and simulation of SDES in application domains such as communication networks and computer systems. On one end of the modeling spectrum, standalone simulators used in these application domains provide the end-user with a pre-built configurable model that is directly written in a high-level programming language such as C++/Java, bundling together custom code for time advancement (Akram and Sawalha 2019; Naicken et al. 2007). While the time advancement algorithm in such simulators can be optimized for a specific application or model, it becomes difficult to reuse or extend the models for exploring alternate structural configurations or coupling them with other frameworks for co-simulation. On the other end of the spectrum, modelers may use commercial or open-source general-purpose Discrete Event Simulation (DES) frameworks such as AnyLogic, SimPy or SystemC to build their models. These general-purpose DES frameworks employ an **event-driven** simulation approach, because it is most flexible and suitable for modeling a broad class of discrete-event systems where event timestamps can be real-valued.

An event-driven approach employs a *Priority Queue* data structure for maintaining a list of all events scheduled for the future, sorted by their timestamps. The simulation progresses by repeatedly extracting and executing the events scheduled at the earliest timestamp, followed by updating the simulation-time variable *directly* to the next-earliest timestamp in the list. The event-driven approach can thus be thought of as *"taking shortcuts in time"* by directly jumping to timestamps of interest. An event-driven approach is efficient for systems where the frequencies of state updates can vary widely across components, for example in agent-based simulations in epidemiology and transportation. However this approach can be fundamentally difficult to parallelize, although there is a large body of research in this direction (Fujimoto et al. 2017). The difficulty in parallelization arises primarily because sharing or splitting the event queue across multiple processes/threads while maintaining causality of events can be difficult, and may yield marginal performance gains for some scenarios. Most general-purpose discrete-event modeling frameworks such as SimPy and SystemC use an event-driven simulation approach, and do not offer parallel simulation capability by default. Parallelization of the event-driven engine by partitioning the event-queue and model components across threads has been explored in some general-purpose frameworks such as SimX (Thulasidasan et al. 2014) and parallel variants of hardware modeling languages such as SystemC (Dömer et al. 2012).

Focusing solely on the *synchronous* subset (that is, SDES) opens up the possibility of a simpler, **cycle-based** simulation approach (also referred to in literature as **time-stepped** or **full-cycle** simulation), where state updates are performed at regular, fixed time intervals by executing/querying the state-update functions of all concurrent components in the model at every time-step. This approach is applicable for synchronous models or discrete-event models where a coarse-grained time-discretization is an acceptable approximation. Compared to an event-driven approach, this approach can be more suited for systems with time-dense activity, that is, systems where most components are likely to be active at every time-step (Tay et al. 2003). This is indeed the case for computer systems, clocked VLSI systems, communication networks etc. and several application-specific simulators in these domains use a cycle-based approach (Grossman et al. 2013). Despite the predictable time-advancement, parallelization of cycle-based simulation is still non-trivial because of the possibility of race conditions. Simulation may yield different results, depending on the order in which the individual components in the model are executed within each time-step. For race-free parallel execution, it may be necessary to first build a directed acyclic graph (DAG) representing the static dependency between model components, and to execute the components in sequence, respecting this model-specific ordering. As an example, consider the system shown in Figure 1 consisting of components A, B, C and D that need to be executed in the specific order: $A \rightarrow \{B,C\} \rightarrow D$, due to the dependence structure. This limits the scope for parallelism. The presence of zero-delay dependency loops in the model (for example, between components B and C in Figure 1) can further complicate the execution, necessitating multiple execution of components within each loop until convergence. Although such a cycle-based approach has been used in a few application-specific libraries and simulators, the possibility of dependency loops and the requirement of a static dependency analysis makes it difficult to partition a model for effective work division using multiple threads. Among the tools that use a cycle-based simulation approach, notable examples are Cascade (Grossman et al. 2013) which is a processor simulator, SystemCASS(Buchmann, Petrot, and Greiner 2004) a cycle-based implementation of SystemC and RepCut (Wang and Beamer 2023) which are both hardware (RTL) modeling frameworks. A cycle-based (time-driven) approach has also been used in applications such as agent-based and traffic simulation (Tan et al. 2021). In most of these tools and frameworks that specifically target the synchronous class, model dependency analysis and model partitioning are critical steps before the simulation can commence. In contrast, Sitar targets a subset of SDES by imposing a weak restriction on the set of models that can be described, leading to a simple, race-free parallel execution approach without the need for model dependency analysis. Although its underlying simulation algorithm is simple, the Sitar modeling language is rich with complex constructs, and well-suited for several application domains. To our knowledge there is currently no other general-purpose modeling framework that targets the same subset.

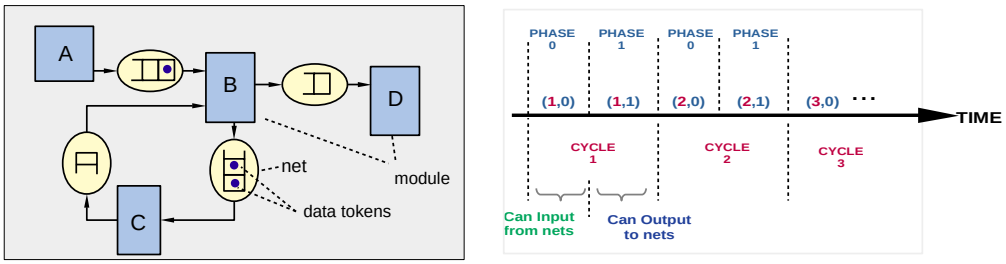


Figure 1: Basic elements in a sitar model, and the two-phase model of time in Sitar.

2.1 Differentiating aspects of Sitar

- **Modeling Scope and Simulation Approach:** In contrast to frameworks such as SystemC and SimPy that are meant for modeling the broad class of discrete-event systems, or frameworks targeting the SDES subclass using time-stepped simulation, Sitar's scope is limited to a *subset* of SDES (in particular, **Moore-type SDES**) which allows for a simple simulation algorithm with a straightforward and efficient, race-free parallel implementation. Notably this also eliminates the need for model dependency analysis or the need to compute possible model partitions for work division during parallel execution. Hornet, a multi-core simulator (Ren et al. 2012) reportedly uses a two-phase algorithm similar to that of Sitar. However Hornet is a specific configurable processor model and not a general-purpose modeling framework.
- **Application:** In contrast to application-specific simulators and frameworks such as Hornet, Cascade and RepCut, Sitar is a general-purpose modeling framework, with a language targeted for describing structured models in a wide range of application domains. To our knowledge there is currently no other general-purpose, open-source modeling framework that specifically targets this subset.
- **Language:** Unlike SystemC and SimPy, which are libraries in an existing high-level programming language, Sitar provides a custom modeling language with translation to C++ code. This makes it possible and easy to enforce the modeling restriction (for Moore-type components). Although a custom language with well-designed constructs aids in model comprehension, conciseness and validation, it necessitates learning yet another language for the modeler, which can be a disadvantage.

3 MODELING ASPECTS

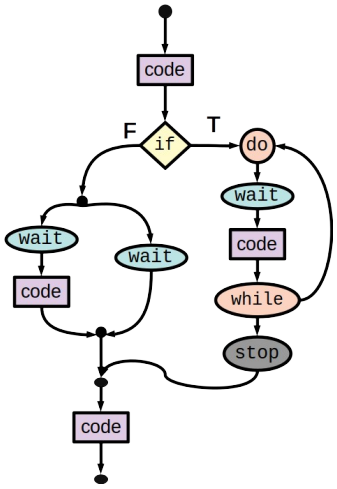
Sitar consists of a custom modeling language suited for systems that have a static, interconnected structure, such as networks and computer architecture models. The framework orthogonalizes the structural and execution semantics from the functional/behavioral aspects of the individual components in the model. A system can be described in a modular manner, as a set of behavioral entities (called *modules*) communicating over buffered channels (called *nets*). Modules can be hierarchical, containing instances of other modules. The language supports parameterized module descriptions and constructs like component arrays to model regular structures such as meshes conveniently. Module behavior can be described in an imperative manner as nested sequences of statements. A statement can consist of an atomic (instantaneous) C++ code block or model constructs such as conditional delays (*wait* statements), branches, loops, and fork-join type parallel blocks. A *procedure* construct is also provided to support modular descriptions. If a particular sequence of statements (including delays) occurs multiple times within a description, it can be encapsulated and described just once as a procedure, and invoked at multiple points. Procedures can also be parameterized. Figure 2 shows a minimal example illustrating some of the behavioral constructs and the corresponding control-flow and simulation output. Constructs for structural description are illustrated in Figure 4.

The system behavior is thus defined by the individual behaviors of the modules and their interconnections, described in an orthogonal manner. The Sitar parser translates model descriptions to highly readable C++ code. Each module description gets translated to a single class. This can be compiled together with the

```
module Top

  behavior
    decl $int x;$;
    $std::cout<<"\n Enter a number: ";$;
    $std::cin>>x;$;

    if($x%2==0$) then //branch
      do //loop
         wait(1,0); //delay
         $log<<endl<<"Even branch";$;
      while($current_time.cycle()<4$) end do;
      stop simulation; //simulation control
    else
      [//parallel block
        wait(1,0);
        $log<<endl<<"Odd branch";$;
      ||
        //conditional delay
        wait until ($current_time>=time(x,0)$);
      ];
    end if;
    $log<<endl<<"Done.";$;
  end behavior
end module
```



Output (with an even number as input)

```
Enter a number: 4

(1,0)TOP        :Even branch
(2,0)TOP        :Even branch
(3,0)TOP        :Even branch
(4,0)TOP        :Even branch
(4,0)TOP        :Done.
Simulation stopped at time (4,0)
```

Output (with an odd number as input)

```
Enter a number: 5

(1,0)TOP        :Odd branch
(5,0)TOP        :Done.
Simulation stopped at time (100,0)
```

Figure 2: A minimal example illustrating Sitar constructs for describing module behavior. In this example, the system consists of just a single module named `Top`. C++ code can be embedded within the dollar (`$`) symbols. The corresponding control flow and simulation output is also shown.

simulation kernel to generate a single executable or linked with external libraries for co-simulation. The Sitar framework also provides syntax highlighting, constructs for systematic, fine-grained logging and error reporting.

4 EXECUTION MODEL

4.0.1 Execution Model

The basic components in a Sitar system are *modules* and *nets*. Modules are behavioral entities in the system, and *nets* are buffered channels of communication between them (illustrated in Figure 1). All modules run concurrently on a single, global clock. Modules communicate via transfer of *data tokens* (which are packets of information) over nets. Nets are passive components, and their state can change only upon input or output actions by modules. A net can have at-most one module as a writer and at-most one module as a reader transferring tokens to/from it. Modules can be hierarchical, containing instances of other modules. The system description must contain a single module named *Top* which represents the top-level module in the hierarchy to be instantiated for simulation. The modeler can describe the behavior of each module in an imperative manner (as illustrated in Figure 2). The Sitar description of a module's behavior gets translated to C++ code into a member routine of the module's class called `run()` which needs to be executed at every time-step during simulation. To perform simulation, simply executing the `run()` function for each module once in every cycle can lead to race conditions, since the exact order of execution among modules can affect the final state. Further, there may be dependency loops within a cycle (for example, the loop through modules B and C in Figure 1), and therefore the `run()` function in each module may need to be executed multiple times until convergence. Even if the dependency graph among the modules were to be built statically, it does not offer a straightforward approach for parallelization since the execution of modules needs to be sequenced as per the dependency and cannot be performed in parallel. Sitar overcomes this challenge through a two-phase execution algorithm.

4.0.2 Two-Phase Simulation and Parallel Execution

In this algorithm, each time-step (or clock-cycle) is divided into two phases: *phase-0*, followed by *phase-1*. A module can input tokens from a net in phase-0 only and output tokens in phase-1 only. Simulation is performed by executing the `run()` function on each of the modules **exactly once** in every phase. Since

```
cycle = 0
while (cycle < simulation_end_time)
{
    phase=0;
    #pragma omp for //run all modules for phase 0
    for (m=0; m<num_modules; m++)
        module[m].run(cycle, phase);
    #pragma omp barrier

    phase=1;
    #pragma omp for //run all modules for phase 1
    for (m=0; m<num_modules; m++)
        module[m].run(cycle, phase);
    #pragma omp barrier
    cycle = cycle + 1
}
```

Figure 3: Pseudo-code showing the Sitar execution algorithm and its parallelization using OpenMP.

a net can have at-most one reader and writer, all state-updates to nets are performed in a deterministic manner, without the need of critical sections enclosing the updates. The simulation results do not depend on the order in which the individual modules are executed within a phase. This leads to a straightforward parallel implementation. To perform simulation in parallel, the set of all modules can be partitioned into groups. In each phase, the execution of a group of modules can be mapped to a single thread, and multiple such threads can run in parallel. All threads need to synchronize at the end of each phase (that is, twice within a single time step). The simple two-phase execution algorithm and its parallelization using OpenMP is illustrated by the pseudo-code in Figure 3. Race-free execution can be guaranteed if the modules that are mapped to different execution threads interact solely via nets, and not through any other shared variables. The partitioning of module instances and their mapping to OpenMP threads for execution can either be dynamically determined using OpenMP's default scheduler or statically specified by the modeler for a balanced work division.

4.0.3 Modeling Scope

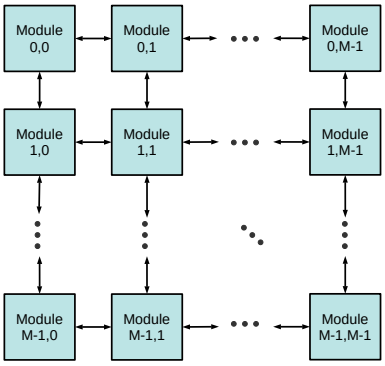
As a consequence of the two-phase execution, the propagation of information from a module's inputs to its outputs **incurs a delay of at-least one cycle**. In the domain of digital systems, this is akin to a model consisting of communicating Moore-type components. In such a model, the components have at-least one flip-flop in every path from the input to the output and there are no direct (combinational) paths running across component boundaries. This restricts Sitar's modeling capability to synchronous systems that can be described as interconnected Moore-type components. The framework enforces this restriction automatically through the two-phase execution algorithm, and no separate checks are necessary. To model instantaneous communication or zero-delay dependency loops between concurrent components in the system, Sitar provides other means. These can be modeled by placing the concurrent components *within a single module* as parallel branches of a fork-join section using Sitar's `parallel block` construct. In Sitar, these concurrent branches are automatically executed multiple times, if necessary, until convergence. The execution order of these branches in the fork-join section is fixed (determined by the order of declarations in the model code), and leads to deterministic execution, since they cannot be split across separate threads. In essence, the modeling framework leads the user to explicitly identify the possible parallelization boundaries (as modules), which makes parallelization straightforward.

5 PERFORMANCE EVALUATION

We present detailed results of a performance and scalability evaluation of Sitar using a parameterized benchmark model. We also present a comparison with SystemC and SimPy in terms of the modeling capability and performance.

5.0.1 Benchmark Model

A parameterized benchmark model (illustrated in Figure 4) was built with parameters representing the model size (**N**), the amount of computation within each module per time-step (**A**), and the extent of communication between the modules (**C**). The topology is representative of models occurring in systems



Benchmark Model

Parameter	Description
N	Model size (number of modules)
A	Amount of computation within each module per time-step
C	Extent of communication between modules
num_cycles	Total time-steps simulated
num_threads	Num of OpenMp execution threads

Parameters

```
module Top
    submodule M1:Mesh<4, 100, 2>
end module

module Mesh
    parameter int M = 1 //number of submodules(in each row/column)
    parameter int A = 1 //computation parameter
    parameter int C = 1 //communication parameter

    submodule_array node[M][M]  : Node<A,C>
    net_array   n_down[M-1][M]  : capacity 100 width 10
    net_array   n_up[M-1][M]    : capacity 100 width 10
    net_array   n_left[M][M-1]  : capacity 100 width 10
    net_array   n_right[M][M-1] : capacity 100 width 10

    //connect nets n_up and n_down
    for row in 0 to M-2
            for col in 0 to M-1
            //connect nets n_down
            node[row][col].outport_down => n_down[row][col]
            node[row+1][col].inport_up  <= n_down[row][col]
            //connect nets n_up
            //...
```

Sitar Description (partial)

Figure 4: Parameterized benchmark model.

such as networks-on-chip, computer mesh networks and many-core processor systems. To mimic some computation happening in every module in each cycle, each module has an integer array of size **A** which is populated and then sorted (using Bubble sort), once in every phase during simulation. The parameter **C** represents the extent of communication between the modules. In every cycle, each module generates **C** new data tokens with randomly assigned destination addresses and forwards them along its output ports. Each generated data token carries the destination module's address and some payload. We evaluate the performance and parallel speedups as a function of each of these model parameters (N, A and C), by varying the number of execution threads from 1 (single-threaded) to 64.

5.0.2 Host Configuration

The experiments were performed on a 40-core system running Ubuntu Linux 18.04 (64bit). The dual socket system has two Intel Xeon Gold 6148 processors, each with 20-cores and 27.5MB of L3 cache memory. Each core supports two threads via hyper-threading. The system has 384GB of DDR4 main memory.

5.0.3 Simulation time-steps and Model-size parameter (N)

We first present a basic validation case which shows that the total simulation time varies linearly with the number of cycles being simulated, as well as with the model size N (as one would expect) in Figures 5 and 6. We observe that the speedup is limited by the number of modules in the model, since each module is mapped to a single thread during execution and the computations for a single module cannot be divided across multiple threads. A higher speedup is observed when the number of modules N is an integer multiple of the number of threads. Comparing results in plots 5 and 6 obtained for slightly different configurations (C=8 and C=4) we note that a higher speedup is achieved when communication between modules is less frequent (a lower value of C). Next, we assess the scalability by sweeping the parameters N, C and A while simulating for a fixed number of cycles. For each experiment, we present two plots, the first showing the speedup versus the number of threads on the x-axis, and the second showing the speedup versus the model parameter value being swept, as the x-axis.

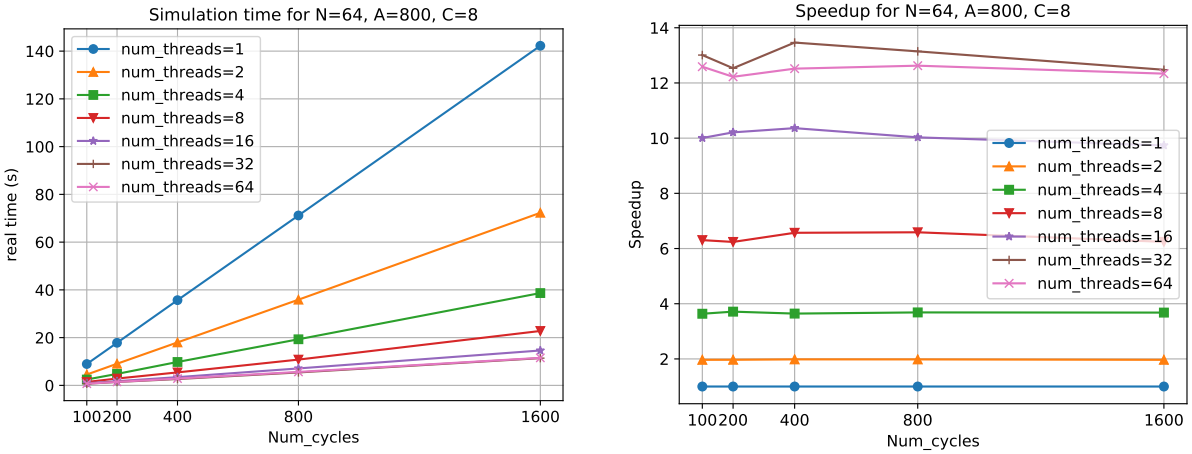


Figure 5: Total simulation time and speedup as a function of the number of cycles being simulated (for N=64, A=800, C=8).

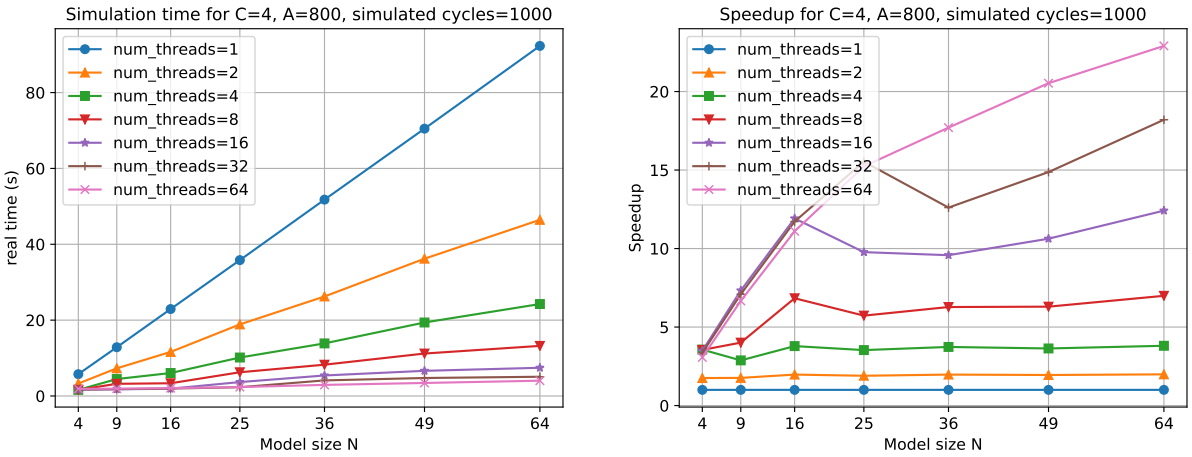


Figure 6: Total simulation time and speedup as a function of the model size N (for C=4, A=800, num_cycles=10^3).

5.0.4 Computation parameter (A)

We consider the case where C=0 and N and A are reasonable large. This represents an embarrassingly parallel scenario where the model size and the per-cycle computation on each module are very large and there is no communication between the modules. Figure 7 presents the speedup as a function of the number of threads and A. It is to be noted that while there is no communication between the modules, all OpenMP threads still have to synchronize at the end of each execution phase. Thus the slope of the speedup line is less than 1. The speedup increases with A and plateaus after a certain value.

5.0.5 Communication parameter (C)

Figure 8 presents the speedup as a function of C. We observe that a large value of C limits the maximum speedup achievable, though it does not have a significant impact on the total simulation time. While communication-related routines (for reading from or writing to nets by the modules) may contribute to the added workload of each thread, a large impact on performance may be caused by coherence-related cache evictions on each net. A net essentially acts like a shared variable between the modules. Multiple threads reading/writing to such a shared variable may increase the overhead associated with cache evictions

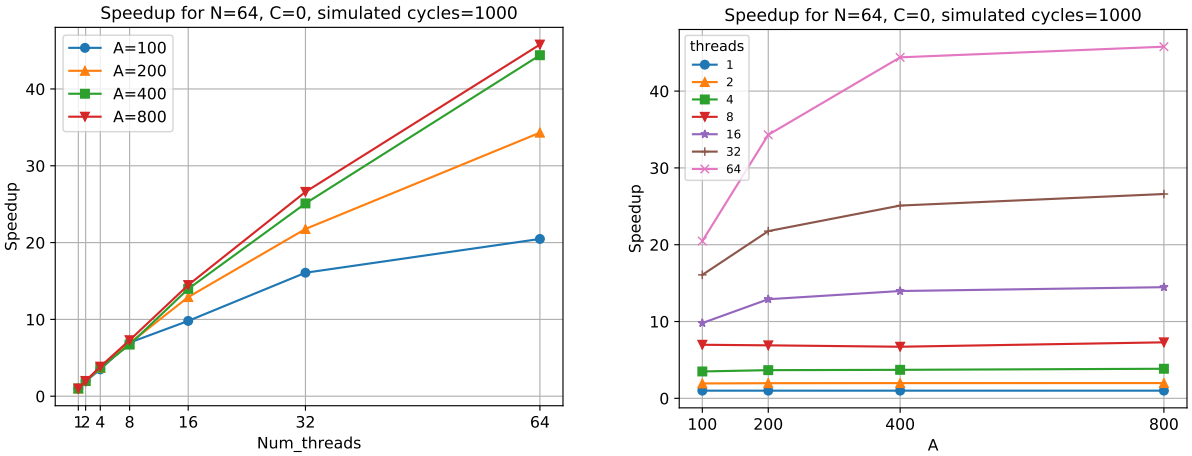


Figure 7: Speedup as a function of the number of threads and the model parameter A (for N=64, C=0, num_cycles=10^3).

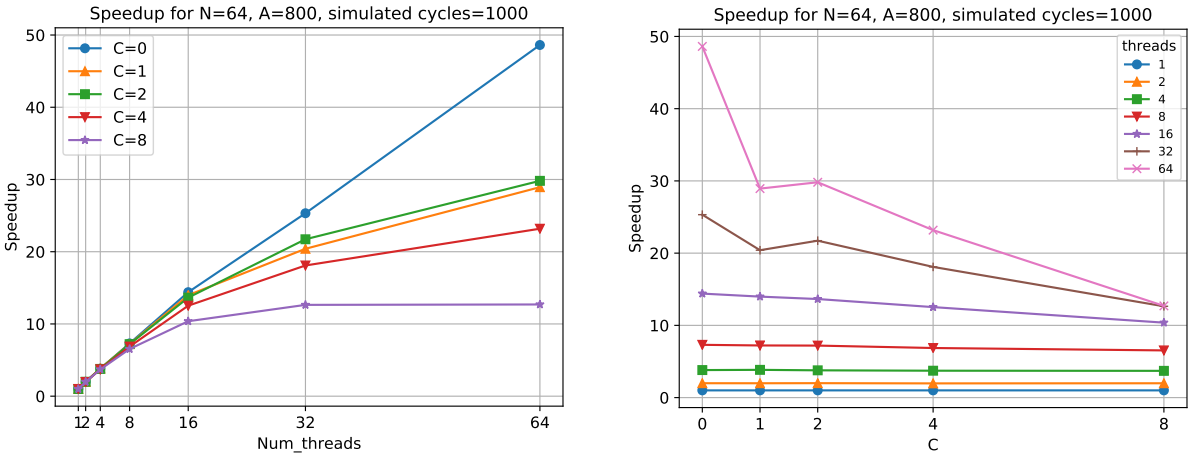


Figure 8: Speedup as a function of the number of threads and the model parameter C (for N=64, A=800, num_cycles=10^3).

(though the two-phase algorithm guarantees that this happens in a race-free manner without the need for using critical sections). For C=8 the maximum speedup possible is 12x-13x using 16 or more threads. In contrast, for C=0 the maximum speedup is found to be close to 50x for large values of A (A=800).

5.1 Comparison with SystemC and SimPy

We consider the same benchmark model as shown in Figure 4, modeled using Sitar, SimPy and SystemC. For each experiment reported in this section, the model parameter settings used for all three frameworks are identical and are listed on each plot heading. To give a comparison of the modeling effort, the source line counts (without comments) for modeling the same system in the three frameworks were as follows: **Sitar:** 219, **SimPy:**142 and **SystemC:** 212. The programmer effort is quite low for SimPy because of the higher level of abstraction offered by Python, with features such as dynamic type-casting and in-built data structures such as lists. For compiling the SystemC and Sitar models, we have used the gcc compiler with the optimization setting of `O3`. For the SimPy model, we present results for two simulation executions. The first is a simulation of the model using the default Python3 implementation. The second is simulation performed using PyPy (https://www.pypy.org/). PyPy is a highly optimized implementation of Python which provides

a Just-in-Time (JIT) compiler that translates Python code into machine-native assembly language. Since we are comparing the base (single-threaded) performance of the three frameworks, the simulations were performed on a desktop/laptop configuration (4-core system with the AMD Ryzen-5 processor with up-to 8 execution threads and 6GB of DDR4 RAM). Figures 9 and 10 present the total simulation time measured by sweeping various model parameters at different base parameter settings. In summary, we observe that the base performance (single-threaded) of Sitar is far better than that of SimPy, but lies in between those of the PyPy-execution of the SimPy model and SystemC. However, a multi-threaded execution using Sitar out-performs both. This is illustrated by the plots in Figure10.

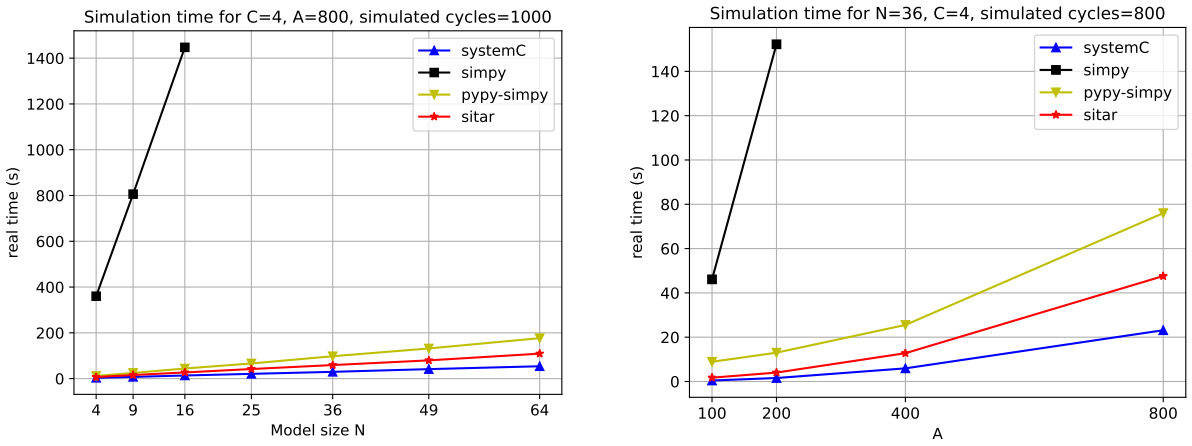


Figure 9: Total simulation time (for single threaded execution) plotted as a function of the model size N and the per-cycle computation parameter A.

5.2 Results Summary

The results indicate that the two-phase execution algorithm of Sitar is capable of efficiently leveraging multi-core platforms for significant speedups. When the model size is large or the computation carried out in each module per-cycle is large, a speedup of **close to 50x** on a 40-core machine using 64 threads was observed. Greater speedups can potentially be obtained by leveraging distributed platforms, and this can be a direction for future work. Further, there is no additional programmer effort or model dependency

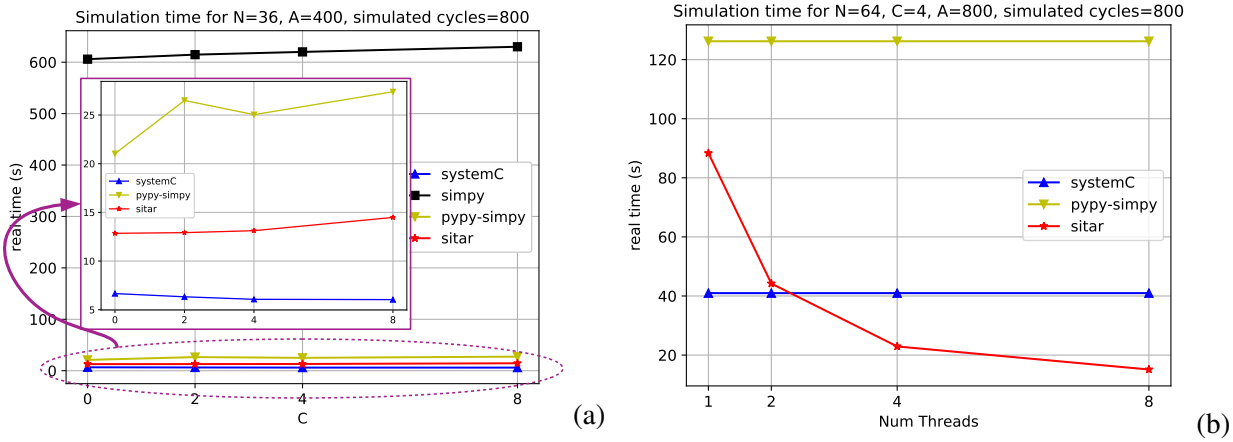


Figure 10: (a): Simulation time (for single-threaded execution) vs the communication parameter C (with a zoomed view in the Inset). (b): Simulation time vs number of threads when multi-threaded execution is enabled for Sitar.

analysis required to achieve these speedups. The single-threaded performance of Sitar is slightly lower than that of SystemC (due to the two phase execution overhead) and better than SimPy, but a multi-threaded execution can outperform both using just a few threads, and shows near-linear scalability for several model configurations. While the benchmark model considered here is homogeneous, for heterogeneous systems the workload balancing across threads can be effectively performed by OpenMP's default scheduler if the model is partitioned into a reasonably large number of modules. Further, it is possible for the modeler to explicitly specify the mapping between modules to execution threads for an effective work-division. The results also indicate that there is further scope in optimizing the communication routines in Sitar to improve performance, particularly in addressing the performance hit caused by cache coherence-related evictions of the communication channels (nets) that are accessed by multiple threads. The scalability demonstrated by this study makes this a promising direction for future work.

6 CONCLUSIONS

Most general-purpose discrete-event simulation frameworks use an event-driven simulation approach for modeling flexibility. We show that by restricting the modeling scope to a subset of discrete-event systems, (specifically synchronous systems that can be described as interconnected Moore-type components), it becomes possible to arrive at a simple two-phase simulation algorithm that affords an efficient, race-free parallel implementation. Models in several application areas typically fall within this subset. Yet, there is a notable gap in the availability of a modeling framework which specifically targets this subset.

This paper presents Sitar, an open-source, general-purpose framework we have developed to bridge this gap. By focusing on a subset of synchronous discrete-event systems and inferring parallelization boundaries directly from the model description, Sitar's two-phase execution algorithm can leverage shared-memory, many-core systems for efficient parallel simulation. We present detailed results from a performance and scalability evaluation study and also compare Sitar's base performance and modeling effort with that of SystemC and SimPy. Sitar's single-threaded performance is comparable to that of SystemC, and better than SimPy, while its multi-threaded execution across several model configurations demonstrates near-linear performance gains with respect to the number of execution threads. This makes Sitar well-suited for applications that involve extremely large models. The utility of Sitar as a general-purpose modeling framework lies in its expressive modeling language, which allows for a modular and hierarchical description of complex systems as interconnected concurrent entities and provides a variety of constructs, such as conditional delays, branches, loops, fork-join concurrency and embedded C++ code blocks for describing module behavior. Sitar also has built-in logging support, syntax highlighting, and systematic error reporting for easing modeling effort. Sitar descriptions get translated into modular, readable C++ code, which makes it easy to couple these models with other simulators for co-simulation. In summary, this paper positions Sitar as a useful open-source framework targeted for modeling and parallel simulation of synchronous discrete-event systems.

ACKNOWLEDGEMENT

Akhil Kushe and Anish Natekar worked on the performance evaluation study through an internship funded by the National Supercomputing Mission (NSM) India.

REFERENCES

Akram, A. and L. Sawalha. 2019. "A Survey of Computer Architecture Simulation Techniques and Tools". *IEEE Access* 7:78120–78145.

Buchmann, R., F. Petrot, and A. Greiner. 2004. "Fast Cycle Accurate Simulator to Simulate Event-driven Behavior". In *International Conference on Electrical, Electronic and Computer Engineering, 2004. ICEEC '04.*, 35–38: ICEEC.

Dömer, R., W. Chen, and X. Han. 2012. "Parallel discrete event simulation of Transaction Level Models". In *17th Asia and South Pacific Design Automation Conference*, 227–231.

Fujimoto, R. M., R. Bagrodia, R. E. Bryant, K. M. Chandy, D. Jefferson, J. Misra, *et al*. 2017. "Parallel Discrete Event Simulation: The Making of a Field". In *2017 Winter Simulation Conference (WSC)*, 262–291 https://doi.org/10.1109/WSC.2017.8247793.

Grossman, J., B. Towles, J. A. Bank, and D. E. Shaw. 2013. "The role of Cascade, a cycle-based simulation infrastructure, in designing the Anton special-purpose supercomputers". In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 1–9. Austin, TX: ACM.

Karanjkar, N. and M. Desai. 2015. "An Approach to Discrete Parameter Design Space Exploration of Multi-core Systems Using a Novel Simulation Based Interpolation Technique". In *23rd IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2015, Atlanta, GA, USA, October 5-7, 2015*, 85–88: IEEE Computer Society.

Karanjkar, N. and M. Desai. 2022, July. "Sitar: A Cycle-based Discrete-Event Simulation Framework for Architecture Exploration". In *Proceedings of the 12th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, 142–150. Lisbon, Portugal.

Karanjkar, N., M. P. Desai, and S. Bhatnagar. 2016. "On Continuous-space Embedding of Discrete-parameter Queueing Systems". *arXiv preprint arXiv:1606.02900*.

Naicken, S., B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman and D. Chalmers. 2007, mar. "The state of peer-to-peer simulators and simulations". *SIGCOMM Comput. Commun. Rev.* 37(2):95–98.

Ren, P., M. Lis, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, *et al*. 2012. "HORNET: A Cycle-Level Multicore Simulator". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31(6):890–903.

Tan, W. J., P. Andelfinger, D. Eckhoff, W. Cai and A. Knoll. 2021. "Causality and Consistency of State Update Schemes in Synchronous Agent-based Simulations". In *Proceedings of the 2021 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '21, 57–68. New York, NY, USA: Association for Computing Machinery.

Tay, Tan, and Shenoy. 2003. "Piggy-backed time-stepped simulation with 'super-stepping"'. In *Proceedings of the 2003 Winter Simulation Conference, 2003.*, Volume 2, 1077–1085 vol.2 https://doi.org/10.1109/WSC.2003.1261535.

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 *2014 4th International Conference On Simulation And Modeling Methodologies, Technologies And Applications (SIMULTECH)*, 188–194.

Wang, H. and S. Beamer. 2023. "RepCut: Superlinear Parallel RTL Simulation with Replication-Aided Partitioning". ASPLOS 2023, 572–585. New York, NY, USA: Association for Computing Machinery.

AUTHOR BIOGRAPHIES

NEHA KARANJKAR is an Assistant Professor in the School of Mathematics and Computer Science at the Indian Institute of Technology Goa (IIT Goa). Her research interests include discrete-event simulation, parallel simulation and hybrid (mixed discrete-continuous) simulation. She is a senior member of IEEE and currently serves as a member of the ACM India Education Committee and Chair of the ACM Goa professional chapter. Her email address is nehak@iitgoa.ac.in and her website is https://nehakaranjkar.github.io.

MADHAV DESAI is a Professor in the Department of Electrical Engineering at IIT Bombay, India. His research interests include VLSI Design, Design Automation, Circuits and Systems, Graph theory and Combinatorics. He is a co-founder of Powai Labs Technologies and has successfully led the design and development of an indigenous processor named 'Ajit'. He was formerly a Principal Engineer in the Semiconductor Engineering Group at the Digital Equipment Corporation in Hudson, MA. His email address is madhav@ee.iitb.ac.in and his website is https://www.ee.iitb.ac.in/wiki/faculty/madhav.

AKHIL KUSHE is a graduate of the Department of Electronics and Telecommunication at Goa College of Engineering, India, where he earned his Bachelor's degree. He is currently working as an Application Engineer at Synopsys (India) Pvt Ltd, Bangalore. His areas of interest include computer architecture and formal/static verification for digital systems. His email address is kusheakhil@gmail.com.

ANISH NATEKAR is an undergraduate student in Computer Science and Engineering at the Indian Institute of Technology Goa (IIT Goa), India. His research interests include Robotics, Drones, SITL simulations, and Computer Vision. Anish is a developer of the open-source project PicoW_Copter. His email address is anish.natekar.20031@iitgoa.ac.in.