Exact Parallelization of the Stochastic Simulation Algorithm for Scalable Simulation of Large Biochemical Networks

Arthur P. Goldberg,[1, a)] David R. Jefferson,[2, b)] John A. P. Sekar,[1, c)] and Jonathan R. Karr[1, d)]

[1)]*Icahn Institute for Data Science and Genomic Technology, and Department of Genetics and Genomic Sciences, Icahn School of Medicine at Mount Sinai*

[2)]*Lawrence Livermore National Laboratory*

(Dated: 22 May 2020)

Comprehensive simulations of the entire biochemistry of cells have great potential to help physicians treat disease and help engineers design biological machines. But such simulations must model networks of millions of molecular species and reactions.

The Stochastic Simulation Algorithm (SSA) is widely used for simulating biochemistry, especially systems with species populations small enough that discreteness and stochasticity play important roles. However, existing serial SSA methods are prohibitively slow for comprehensive networks, and existing parallel SSA methods, which use periodic synchronization, sacrifice accuracy.

To enable fast, accurate, and scalable simulations of biochemistry, we present an exact parallel algorithm for SSA that partitions a biochemical network into many SSA processes that simulate in parallel. Our parallel SSA algorithm exactly coordinates the interactions among these SSA processes and the species state they share by structuring the algorithm as a parallel discrete event simulation (DES) application and using an optimistic parallel DES simulator to synchronize the interactions. We anticipate that our method will enable unprecedented biochemical simulations.

[a)]Author to whom correspondence should be addressed: Arthur.Goldberg@mssm.edu

[b)]Electronic mail: drjefferson@gmail.com

[c)]Electronic mail: karr@mssm.edu

[d)]Electronic mail: john.sekar@mssm.edu

I. INTRODUCTION

Models of biochemical systems play a critical role advancing medicine and bioengineering. Technological advances in single-cell and genomic measurement are rapidly generating data that enable larger and more complex models of biochemical pathways and whole cells. But advances in the performance and accuracy of simulation techniques are needed for integrating models.

The Stochastic Simulation Algorithm (SSA) is a widely used method for predicting the time evolution of chemical systems transformed by chemical reactions.[1,2] In particular, SSA can model the variability exhibited by chemical systems with small species populations, conditions which often arise in small biological systems such as populations of individual cells.[3,4]

Our interest in creating tools to enable dynamical models of large biochemical systems has motivated us to parallelize SSA. We are especially interested in whole-cell modeling, which creates genome-scale models of the known biochemical pathways in individual cells.[5–8] A whole-cell model of a cell with a large genome, like a human cell, contains tens of thousands of species and tens of thousands of reactions. In addition, it contains numerous compartments that represent intracellular organelles such as the nucleus, mitochondria, lysosomes and others. Whole-cell models are typically integrated for an entire cell cycle.[7] The complexity of whole-cell models and the duration of their integrations cause their simulations run slowly. A highly parallel SSA algorithm could speed up simulations of models of large biochemical systems like whole-cell models.

There are multiple opportunities for parallelism in the simulation of biochemical networks due to the physical structure of cells, as well as due to artifacts of the limited ability of researchers to estimate their structure. First, the compartmentalization of cells into organelles means that the biochemical networks of a cell decomposes into a set of biochemical sub-networks of the organelles, weakly connected via edges that represent a comparatively small number of exchange reactions. Second, the subcompartmentalization of organelles into spatial domains such as chromosomal regions and individual mRNA means that the sub-networks of the organelles decompose into weakly connected cliques. Third, the chemical specificity of most enzymes means that the sub-networks of the organelles further decompose into cliques. Fourth, our limited knowledge of the interactions among cellular processes means that our in silico biochemical networks have higher clustering coefficients than real biochemical networks.

Exact implementations of SSA are computationally expensive when modeling systems that involve many reaction executions because they execute only one reaction per iteration. In addition, since SSA models stochastic behavior, multiple runs are needed to obtain distributions and moments of predicted properties.

While extensive effort has been devoted to improving the performance of SSA[9–12] an exact, parallel SSA algorithm that can concurrently leverage numerous processors to speed up an SSA simulation is not available.

We introduce a novel parallel algorithm for speeding up SSA. Our approach conceives of SSA as a discrete event simulation (DES) and leverages optimistic approaches for parallel DES (PDES).[13–18] In contrast to prior work, our parallel SSA algorithm takes advantage of more of the parallelism inherent in large reaction networks, exactly synchronizes changes to the species population state that is shared among components of the algorithm, and, assuming that random number generators are properly managed, can exactly reproduce the

predictions of a sequential SSA algorithm.

II.   THE STOCHASTIC SIMULATION ALGORITHM

A.   A summary of SSA

SSA solves the following problem. Consider a well-mixed container of chemical species, the chemical reactions that transform them, and a rate law for each reaction that provides its execution rate as a function of the species populations. Given an initial state of this system, one can model the dynamic probability that it occupies any population state by representing state transition probabilities as a system of coupled differential equations. Although this formulation, known as the Chemical Master Equation, is exact, it is not tractable for biologically interesting models because the state space is prohibitively large.

The Stochastic Simulation Algorithm models the dynamical behavior of this system by computing a sequence of reaction executions, choosing reactions and execution times so that the probability of generating a given trajectory equals the probability that would be provided from a solution of the Chemical Master Equation.

An algorithm for SSA, known as the Direct Method, was provided by Gillespie.[1]

1: **procedure** *SSA Direct Method*
2:      Set initial species populations
3:      $t \leftarrow 0$ ▷ simulation time
4:      $\tau, \mu \leftarrow$ PLAN NEXT REACTION EXECUTION
5:      **while** $t + \tau \leq$ max simulation time **do**
6:          $t \leftarrow t + \tau$
7:          Update species populations according to reaction μ
8:          $\tau, \mu \leftarrow$ PLAN NEXT REACTION EXECUTION
9:      **end while**
10: **end procedure**

11: **function** PLAN NEXT REACTION EXECUTION
12:      **for** r in reactions **do**
13:          Compute the rate (propensity) for r, a_r
14:      **end for**
15:      $a_0 \leftarrow \sum_r a_r$ ▷ the total propensity
16:                          ▷ let $U()$ randomly sample the standard uniform distribution
17:      $\tau \leftarrow (1/a_0) \ln(1/U())$ ▷ τ is the time the next reaction executes
18:      Choose μ by sampling $P[\mu] = a_\mu/a_0$ ▷ μ is the next reaction
19:      return τ, μ
20: **end function**

Detailed derivations for the Direct Method[1,2,19–21] rely on a key assumption of the Direct Method—that propensities a_r remain constant in between reaction executions (equation (1) in 21).

We use the Direct Method for pedagogical purposes in this paper, but base our parallel SSA algorithm on an optimized SSA algorithm, the Next Reaction Method.[19]

3

III. INTRODUCTION TO PARALLEL SSA

This section provides an overview of the parallel SSA algorithm.

A. Objectives

The parallel SSA algorithm we present has two primary objectives.

1. Speed up the performance of SSA on a large reaction network by partitioning the network into multiple sub-networks and integrating the sub-networks in parallel.

2. Exactly simulate reaction networks, so that the results of a parallel simulation are the same as the results of a sequential simulation. This objective is satisfied by the parallel SSA algorithm presented below.

B. Opportunities for parallelism

A reaction network in a large biochemical model like a whole-cell model provides two types of opportunities for parallelism. First, the sub-networks of reactions in separate compartments may be sufficiently decoupled from each other that they can be simulated in parallel. The amount of a compartment's coupling is measured by the mean sum of the rates of its reactions that exchange species with other compartments, relative to the mean sum of the rates of all reactions in the compartment. Multiple prior studies have taken this approach to parallelize SSA.[22–26] Second, reactions within a compartment may also be partitioned into sub-networks that are sufficiently decoupled that they can be simulated in parallel.

These two types of potential parallelism are complementary and can be combined in a single parallelization. Our parallel SSA approach below abstracts away the distinctions between them and simply partitions a reaction network into sub-networks. Nevertheless, we note these opportunities for parallelism because prior work has leveraged the first one, and this discussion may help readers develop intuition for parallel SSA.

C. Overview of the parallel SSA approach

To provide an overview of the parallel SSA algorithm we briefly summarize its high-level steps, broken down into its parallelization and simulation phases.

Parallelize

1. Read a model to simulate, including its reactions and rate laws.

2. Generate a graph G that characterizes the dependencies among reactions and species in the model.

3. Partition G into sub-networks of reactions that will be simulated in parallel.

4. Identify the species that are shared between multiple sub-networks of reactions, and partition them.

Simulate

1. Read a simulation configuration for the model, including the initial populations of species and other initial conditions, and a maximum simulation time.

2. Instantiate an optimistic object-oriented (OO) parallel discrete event simulation (PDES) environment on a parallel computer.[14–18,27–31]

3. Map each reaction sub-network of G onto an SSA simulation object in the optimistic PDES environment, instantiate the object, and send it an initialization event message.

4. Map each set of shared species onto a Shared Species Population (SSP) simulation object in the PDES environment, instantiate the object and send it an initialization event message.

5. Initialize all other objects in the simulation.

6. Run the parallel SSA simulation.

These steps and components are thoroughly described below.

IV.   THE PARALLEL SSA ALGORITHM

This section presents the concepts underlying the parallel SSA algorithm.

A.   Transforming a reaction network into a parallel simulation

1.   Encode the reaction network's dependencies into a directed graph

Let R and S represent all reactions and species, and r_i, s_j, and l_k represent individual reactions, species and rate laws, respectively. We map the dependencies between these components onto a bipartite directed graph $G(v, e)$, where $v = R \cup S$. G is constructed as follows. A directed edge (s_j, r_i) is added to G if species s_j participates in rate law l_i, thereby encoding the dependency of reaction r_i on species s_j. Similarly, a directed edge (r_i, s_j) is added to G if s_j has a non-zero stoichiometry in reaction r_i, thereby encoding the dependency of s_j on reaction r_i that arises because executing reaction r_i changes the populations of species s_j. Figure 1 illustrates G for a small example network.

2.   Partition the dependency graph

To identify reaction network components that can be simulated in parallel, we partition G into two types of sub-networks. The first type of sub-network contains reactions, and all of the species that participate only in reactions in the sub-network[1] (e.g., see partitions α and β in in Figure 2). The species contained in a reaction sub-network are called *local*

---

[1] More precisely, these are the species connected in G to reactions in the sub-network and not connected to any other reactions.

Reactions

r_1: A + B → C @ k_1 A B
r_2: C + D → E + F + D @ k_2 C D
r_3: F + F → G @ k_3 F(F-1)

Reaction depends on and changes species

Species changed by reaction

Reaction depends on species
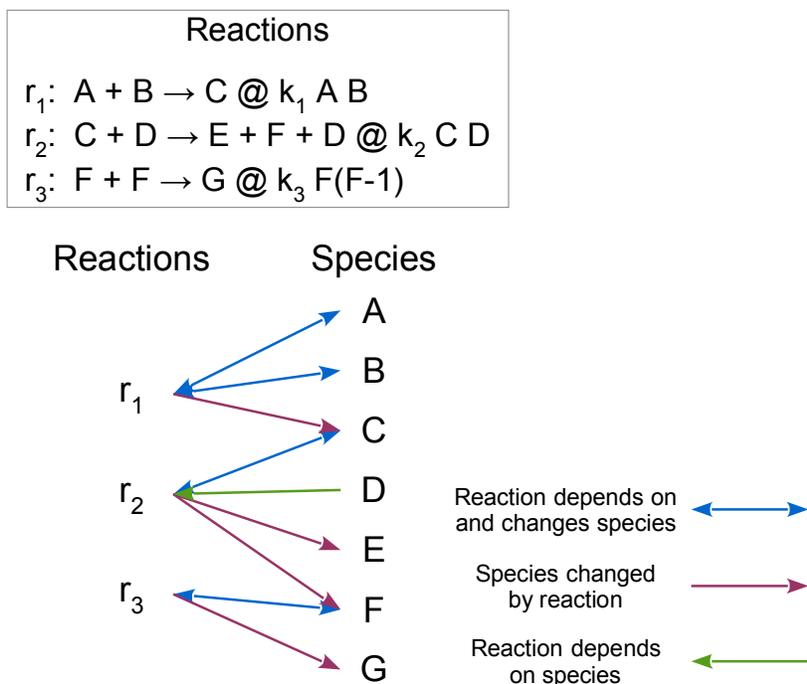
FIG. 1. Relationships between reactions and species encoded in dependency graph G. The relationships between reactions and species are encoded in a directed graph whose nodes are reactions and species. Directed edges indicate data dependencies. An edge from a reaction and its rate law to a species indicates that executing the reaction changes the species' population. An edge from a species to a reaction indicates that the reaction's rate law uses the species' population. For example, G contains an edge from enzyme D to reaction r_2 because the rate law for r_2 is a function of the population of enzyme D. Bi-directional edges indicate both of these dependencies.

The @ signs in the list of reactions separate each reaction from its rate law, a mathematical expression that computes the rate at which the reaction executes. In a rate law a species symbol represents the number of molecules of the species in the container being simulated.

species, because they're used only by the reactions in their local sub-network. For example, in Figure 2 species A and B are local to sub-network α.

The second type of sub-network created by a partition contains only *shared* species, which are used by reactions in two or more reaction sub-networks.

3. Designing the partitioning algorithm

The goal of the partitioning algorithm is to identify partitions into sub-networks that can be rapidly simulated in parallel. In the near-term, we plan to use linkage clustering algorithms to partition the dependency graph G. Intuitively, this will minimize the number of state changes that would need to be synchronized between the DES objects which are encoded into the sub-networks.

Long-term, we aim to develop an algorithm for identifying the optimal partitioning that minimizes the expected run-time on a parallel computer. Due to the challenges to analyzing the expected run-time of a parallel SSA simulation, the optimal partitioning will likely need

be to identify empirically by testing the performance of putative partitionings.

4. Encoding the reaction network into simulation objects

The two types of sub-networks produced by a partition are mapped into two types of DES objects.

Each sub-network of reactions defines the reactions that are integrated by one OO DES object that simulates SSA, called an *SSA object*. For example, in Figure 2 SSA object α integrates reaction r_1 and stores local species A and B, while SSA object β integrates reactions r_2 and r_3 and stores local species D, E, F, and G.

Each sub-network of shared species is mapped into one OO DES Shared Species Population (SSP) object which will coordinated the populations of the shared species during a simulation. For example, in Figure 2 shared species C is stored in an SSP object and is used by reactions r_1 and r_2, which are executed by two different SSA objects, α and β, respectively.
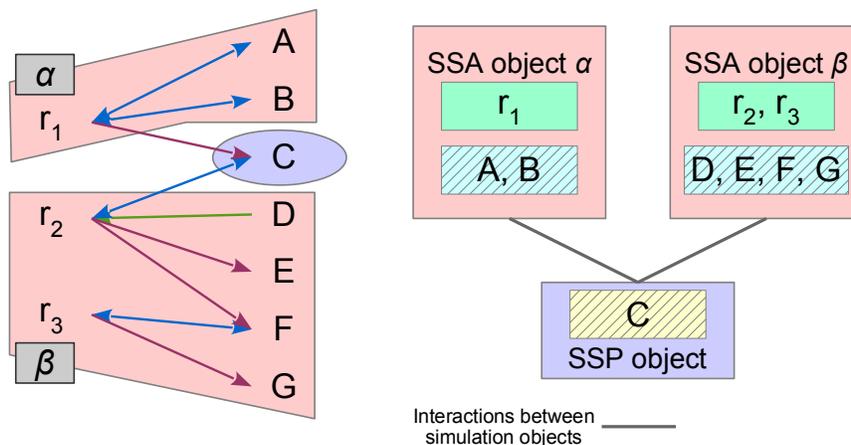
FIG. 2. A partition of the dependencies illustrated in Figure 1. The dependency graph G in Figure 1 is partitioned into two reaction sub-networks, α and β, and 1 shared species sub-network. These are then mapped onto the SSA objects α and β, and an SSP object. In this small network each SSA object interacts with the SSP object, but in reaction networks suitable for parallel execution by parallel SSA, each SSA object would interact with a small fraction of the SSP objects, and *vice versa*.

B. Architecture of parallel SSA

This parallel SSA algorithm has been designed to run as a parallel OO DES application on top of an optimistic PDES simulator.[13,14,16–18,27,28,30,32] The major advantage of this approach is that the optimistic PDES simulator will be responsible for running the objects in the parallel SSA algorithm in simulation time order. The major types of simulation objects in parallel SSA will be the SSA and SSP classes defined above. In addition, other utility classes will be added, such as classes for initialization and for checkpointing model predictions.

V.   THE PARALLEL SSA ALGORITHM

This section presents our approach for an exact parallel SSA algorithm. We begin by restructuring SSA as an OO DES application and defining the structure of DES application classes. In sub-section V B we illustrate the features and components of the algorithm, which we follow with a sub-section containing pseudocode for its classes. The final sub-section describes how the algorithm handles compartments.

A.   Structure SSA as an object-oriented discrete event simulation application

Many variations of the SSA algorithm have been published, beginning with two in Gillespie's original paper.[1] SSA variations have the characteristics of a DES application. First, their events, which simulate reaction executions, occur at a discrete time instants. And, second, these events are dynamically scheduled by computations at earlier simulation times. But published SSA variations do not use DES because they're written as free-standing algorithms.

Since an optimistic parallel OO DES simulator will be used to synchronize the parallel SSA algorithm, we begin the discussion of synchronization by recasting the Direct Method (presented in section II A) as an OO DES application. This implementation of the Direct Method OO DES class executes two event methods, *Initialize Direct Method* and *Execute and schedule reaction*:

1: **Event method** *Initialize Direct Method(0, self, self, initial populations, max simulation time)*
2:      Set initial species populations
3:      $t_m \leftarrow$ *max simulation time*
4:   ▷ Call Direct Method function defined in section II A
5:      $\tau, \mu \leftarrow$ PLAN NEXT REACTION EXECUTION()
6:      send *Execute and schedule reaction(τ, self, self, μ)*
7: **end Event method**

8: **Event method** *Execute and schedule reaction(t, self, self, μ)*
9:      **if** $t \leq t_m$ **then**
10:         Update species populations according to reaction μ
11:         $\tau, \mu \leftarrow$ PLAN NEXT REACTION EXECUTION()
12:         send *Execute and schedule reaction(t + τ, self, self, μ)*
13:      **end if**
14: **end Event method**

All OO DES application objects interact exclusively through event messages, a strict interface that makes it possible for parallel OO DES simulators to execute them.

As illustrated by this pseudo-code for the Direct Method class, all DES event messages in this paper have the form

Message type(time, sender, receiver, [arguments]),

where *Message type* is the type of the message, *time* is the simulation time at which the message will be received and executed, *sender* identifies the simulation object that sends the message, *receiver* identifies the simulation object that will receive the message, and the optional *arguments* are data carried by the message from its *sender* to its *receiver*.

The algorithm for a DES application class is defined by its message handler event methods, whose names and signatures must exactly match the types and fields of messages

received by the class, although the handler method names are written in SMALL CAPS to distinguish them from the messages. Throughout the execution of a handler the simulation time of the *receiver* object handling an event message is automatically *time*.

B.   An exact parallel SSA algorithm

This section presents an algorithm that focuses on the key logic needed to exactly parallelize SSA. Performance optimizations for the algorithm follow in section VI A.

The exact parallel SSA algorithm is a parallel OO DES application that executes two class types, SSA objects and Shared Species Population (SSP) objects. Updates and reads of local species stored in SSA objects always occur at the correct simulation times, so we focus on shared species. To exactly synchronize shared species, updates and reads of their populations must also occur at the correct times. More specifically, all updates to shared species populations by a reaction must change the populations in SSP objects at the time the reaction executes. And when an SSA object uses a shared species at time t it must obtain the SSP's value for the species' population at t.

Correct timing of updates and reads of shared species populations is achieved by explicitly accessing shared species at SSP objects. When a reaction executed by an SSA object updates the population of a shared species, the object updates the population at the SSP by sending an *Adjust populations* message to the SSP. And the SSP handles an *Adjust populations* message by sending a zero-delay *Populations* message to each SSA object that uses the shared species which were updated in the *Adjust populations* message.

Table I lists these messages and Figure 3 illustrates their dynamics. The interactions between SSA object α and the SSP over the time interval t_1 to t_3 (see section **A** of Figure 3) execute one reaction, r_1. At time t_1 three event messages are sent to schedule the reaction and the parallel coordination it requires:

TABLE I. Event message types used by Parallel SSA

Message type	Sender object type	Receiver object type	Arguments (not including the required arguments *time*, *sender*, and *receiver*)	Parallel SSA semantics and receiver object action
Execute reaction	SSA	SSA	μ	Execute reaction μ
Schedule reaction	SSA	SSA		Schedule the next reaction
Adjust populations	SSA	SSP	*pop_changes*	Convey the stoichiometry of a reaction's shared species to the SSP, which updates their populations
Populations	SSP	SSA	*shared_species_pops*	Provide populations of the requested species to the SSA

- *Execute reaction* schedules the execution of r_1 at SSA α at t_3. Every reaction execution requires this message.
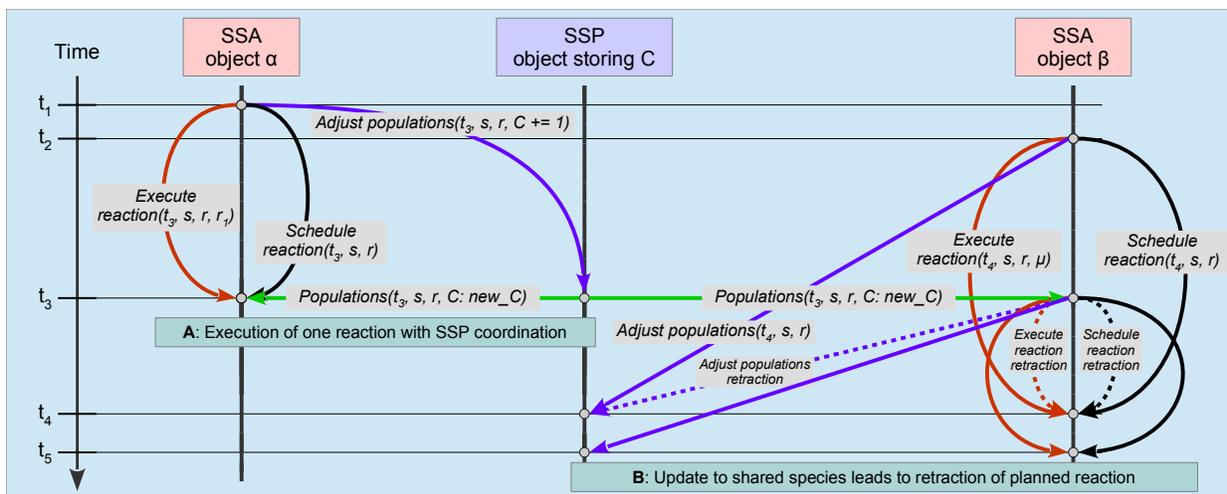
FIG. 3. Example event message interactions between SSA and SSP objects. To conserve space, the event message fields *time, sender, receiver* are abbreviated t_i, s, r. **A**: The interactions between SSA object α and the SSP show the messages involved in a single reaction execution coordinated with the SSP, as described in section V B. **B**: The *Populations* message that updates the population of shared species C at SSA object β at time t_3 causes SSA β to cancel its planned execution of reaction μ at time t_4, as discussed in section V B 1. Each *retraction* message is indicated by a dashed arrow that has the same color as the message being retracted, and points to the same object at the same time as it does.

- *Schedule reaction* causes SSA α to schedule the next reaction. Every reaction execution requires this message.

- *Adjust populations* schedules the SSP to increment the population of C at t_3, as per the stoichiometry of reaction r_1. Every reaction execution that changes the population of shared species requires this message.

At time t_3, in response to the *Adjust populations* message, the SSP sends a *Populations* message that contains the shared populations to SSA α.

The algorithms that send and handle these messages are detailed in sections V C 1 and V C 2.

1. Handling updates to shared species used by rate laws

A fundamental assumption SSA is that the propensities[2] $a_k(X(t))$ are constant except when reaction executions change species counts (equation (1) in.[21])[3] Parallel SSA revises this assumption by adding that **propensities can also change when the populations of shared species used by propensities are updated**. Changes external to an SSA object can thus invalidate the computation that scheduled a reaction execution.

---

[2] The standard notation $a_k(X(t))$ indicates the propensity of reaction k—computed using k's rate law—as a function of the species populations at time t, $X(t)$.

[3] Some SSA algorithms model time-varying propensities, for example to simulate containers with varying volume, as in section V of[21].

Figure 3 illustrates this situation in the interactions among 2 SSA objects and an SSP object. Focus on the SSP and SSA object β over the time interval t_2 to t_5 (section **B** of Figure 3):

1. At time t_3, in response to the *Adjust populations* it received from SSA α, the SSP sends a *Populations* message that updates the population of shared species C at SSA β.

2. At t_3 SSA β receives the message, updates the population of species C, determines that the population of C was used by the propensity calculations made when scheduling its pending reaction μ, and concludes that the assumption made when scheduling μ— that the propensities calculated at time t_2 would remain constant until time t_4—no longer holds. Thus, SSA β must now cancel the scheduled execution of reaction μ, and reschedule its next reaction.

3. SSA β cancels each of the messages it sent at t_2 to schedule the execution of reaction μ by sending a corresponding *retraction* message at t_3 (see the three messages labeled *retraction* and indicated by dashed arrows in Figure 3). *Retraction* messages[33] are application messages that completely cancel the effects of a corresponding retracted message, including the possible execution of the retracted message and that execution's side effects. They are a fully developed feature of the Virtual Time .[17,27,34] optimistic parallel synchronization theory which is implemented by Time Warp optimistic PDES simulators[14,16]

4. SSA β now reschedules its next reaction, which Figure 3 illustrates as executing at time t_5.

The algorithms for the parallel SSA classes that implement these actions are detailed in the next section.

C. Class algorithms for parallel SSA

This section presents event methods for the SSA and SSP classes that implement the parallel SSA algorithm. To simplify this section's presentation all shared species are stored in a single SSP object, and we consider a system with only one biological compartment, corresponding to a well-mixed container in the stochastic simulation's analytic framework .[1] These limitations are relaxed in section VI A below.

1. The SSP class

An SSP object executes two types of events: *Initialize* initializes the SSP, and *Adjust populations* updates the populations of specified shared species.

1: **Event method** *Initialize*(0, self, self, initial_populations, species_locs)
2:   ▷ initialize initial populations of all shared species
3:     self.populations = initial_populations
4:     self.SSA_map = species_locs   ▷ map shared species to their SSAs
5: **end Event method**

1: **Event method** *Adjust populations*(time, SSA_obj, self, pop_changes)
2:     add the changes in pop_changes to self.populations
3:                                     ▷ send updated populations to SSAs that depend on them
4:     **for** SSA in self.SSA_map **do**
5:         shared_species_pops = empty dictionary
6:         **for** species in pop_changes **do**
7:             **if** species in self.SSA_map[SSA] **then**
8:                 shared_species_pops[species] = self.populations[species]
9:             **end if**
10:         **end for**
11:         **if** shared_species_pops **then**
12:             send *Populations*(time, self, SSA, shared_species_pops)
13:         **end if**
14:     **end for**
15: **end Event method**

Adjust populations adjusts the populations of shared species in self.populations by the changes in pop_changes. In lines 4–14 the SSP acts like a write-through cache, forwarding updated shared species population values to the SSAs executing reactions that use the species in their rate laws.

2. The SSA class

This section defines the algorithms used by the SSA class. We adapt Gibson and Bruck's Next Reaction Method (NRM)[19,21] to determine the reactions that execute and their execution times. However, to simplify the presentation we do not incorporate NRM's optimization that avoids recomputing propensities which do not depend on changed species populations. Instead, in section VI A we identify it as an optimization that should be added to the algorithm.

An SSA object executes four types of event methods. They correspond 1-to-1 to the messages illustrated in Figure 3 and described in section V B: *Initialize* initializes the object[4], *Schedule reaction* schedules the next reaction, *Execute reaction* executes a reaction, and *Populations* receives updated values for the populations of the shared species used by the SSA object.

1: **Event method** *Initialize*(0, self, self, reactions, initial_populations)
2:     self.reactions = reactions   ▷ reactions and their rate laws
3:     self.local_species = initial_populations['local_species']
4:     self.cached_shared_species = initial_populations['shared_species']
5:     self.SSP = the single SSP
6:     self.next_rxn.time = 0   ▷ self.next_rxn tracks a pending reaction
7:     self.next_rxn.reaction = $\emptyset$
8:     μ, τ_μ = Select initial reaction(self)
9:     Send reaction events(μ, τ_μ, self)
10: **end Event method**

---

[4] Note that the *Execute and schedule reaction* event method used by the SSA class presented in section V A has been decomposed into two event methods, which is required by the reasoning at the end of section V D 1.

Initialize only runs once, at time 0.

1: **function** Select initial reaction(self)
2:                                  ▷ Select the first reaction, saving propensities and times
3:     **for** k in self.reactions **do**
4:         self.$a_k = a_k$ = propensity for k
5:         r_k = uniform(0, 1) random number
6:         self.$\tau_k = \tau_k = (1/a_k)\ln(1/r_k)$ ▷ execution time for k
7:     **end for**
8:     $\mu = k$ that satisfies $\min_k\{\tau_k\}$ ▷ Select next reaction
9:     **return** μ, τ_μ
10: **end function**

To support the event-driven model of OO DES, the reaction scheduling code must be separated into a different function for each event. Each function determines and returns the next reaction to execute, μ, and the time when it will execute, τ_μ. The first such function, Select initial reaction, which is called by Initialize, is mathematically equivalent to the initialization of NRM (lines 2–5 of Alg. 2 in 21).

1: **function** Send reaction events(μ, τ_μ, self)
2:     send *Execute reaction*(τ_μ, self, self, μ)
3:     send *Schedule reaction*(τ_μ, self, self)
4:     species_changes = shared species with stoichiometry $\neq 0$ in μ
5:     **if** species_changes **then**
6:         send *Adjust populations*(τ_μ, self, self.SSP, species_changes)
7:     **end if**
8:     self.next_rxn.reaction = μ
9:     self.next_rxn.time = τ_μ
10: **end function**

The Send reaction events function sends all of the event messages transmitted by in SSA object. It is called by each of the three event methods that send messages. Each call sends all the messages associated with the execution of one reaction. Lines 11–12 record the reaction and its execution time for later use.

1: **Event method** *Schedule reaction*(t, SSA_obj, self)
2:                                    ▷ Schedule a reaction after executing one
3:     μ = self.next_rxn.reaction
4:     μ, τ_μ = Select after reaction execution(self, t, μ)
5:     Send reaction events(μ, τ_μ, self)
6: **end Event method**

7: **function** Select after reaction execution(self, t, μ)
8:                        ▷ Select next reaction after executing μ; update self.a_k and self.τ_k
9:     s = self
10:     **for** k in self.reactions **do**
11:         $\bar{a}_k$ = new propensity for k
12:         **if** $k \neq \mu$ **then**
13:             self.$\tau_k = \tau_k = \frac{s.a_k}{\bar{a}_k}(s.\tau_k - t) + t$
14:         **end if**
15:     **end for**

16:        $r_\mu = \text{uniform}(0, 1)$ random number
17:        $\text{self.}\tau_\mu = \tau_\mu = (1/\ \overline{a}_\mu)\ln(1/r_\mu)$
18:        $\mu = k$ that satisfies $\min_k\{\tau_k\}$ ▷ Select next reaction
19:        **for** k in self.reactions **do**
20:            $\text{self.}a_k = \overline{a}_k$
21:        **end for**
22:        **return** μ, τ_μ
23: **end function**

Schedule reaction implements a DES version of the body of the NRM's iterative loop.

SELECT AFTER REACTION EXECUTION, which selects the next reaction after a reaction executes, is mathematically equivalent to the code in NRM's iterative loop (lines 5 and 7–10 of Alg. 2 in 21).

1: **Event method** *Populations*(t, SSP_obj, self, shared_species_pops)
2:        self.cached_shared_species = shared_species_pops
3:        **if** time ¡ self.next_rxn.time **then**
4:                              ▷ Cancel next reaction and schedule again
5:            CANCEL SCHEDULED REACTION(self)
6:            $\mu, \tau_\mu = $ SELECT AFTER REACTION CANCELLATION(self, t)
7:            SEND REACTION EVENTS(μ, τ_μ, self)
8:        **end if**
9: **end Event method**

10: **function** CANCEL SCHEDULED REACTION EVENTS(self)
11:                          ▷ Cancel events related to the execution of reaction self.next_rxn
12:        $\mu = $ self.next_rxn.reaction
13:        $\tau_\mu = $ self.next_rxn.time
14:        retract *Execute reaction*(τ_μ, self, self, μ)
15:        retract *Schedule reaction*(τ_μ, self, self)
16:        species_changes = shared species with stoichiometry $\neq 0$ in μ
17:        **if** species_changes **then**
18:            retract *Adjust populations*(μ, self, self.SSP, species_changes)
19:        **end if**
20: **end function**

21: **function** SELECT AFTER REACTION CANCELLATION(self, t)
22:                     ▷ Select next reaction after cancellation, updating self's a_k and τ_k
23:        $s = $ self
24:        **for** k in self.reactions **do**
25:            $\overline{a}_k = $ new propensity for k
26:            $\text{self.}\tau_k = \tau_k = \frac{s.a_k}{\overline{a}_k}(s.\tau_k - t) + t$
27:        **end for**
28:        $\mu = k$ that satisfies $\min_k\{\tau_k\}$ ▷ Select next reaction
29:        **for** k in self.reactions **do**
30:            $\text{self.}a_k = \overline{a}_k$
31:        **end for**
32:        **return** μ, τ_μ
33: **end function**

Populations records updated populations for the shared species used by an SSA object's rate laws. If a future reaction is pending then it must be cancelled and reaction scheduling must be redone, as previewed in section V B 1 above. This novel aspect of the parallel SSA algorithm is handled by a pair of functions: CANCEL SCHEDULED REACTION EVENTS cancels all of the events related to the previously scheduled reaction, and SELECT AFTER REACTION CANCELLATION then selects the next reaction to execute. In the former function, each "retract *Event message*" operation sends a *retraction* message that cancels the previously sent *Event message*.

SELECT AFTER REACTION CANCELLATION schedules the next reaction. We adapt the NRM to determine the reaction times. Because *no* reactions have executed since propensities were last calculated, all reactions—including the one that was cancelled—can be treated like the reactions that did not execute in the iterative NRM loop (see lines 4–9 of Alg. 2 in 21). Thus, lines 27–31 of SELECT AFTER REACTION CANCELLATION calculate a new propensity and execution time for each reaction, and select the reaction with the minimum time. No random numbers are needed.

1: **Event method** *Execute reaction*(t, SSA_obj, self, μ)
2:     update self.local_species according to the stoichiometry of μ
3: **end Event method**

Execute reaction updates the local species populations according to the stoichiometry of the reaction being executed.

D. Correctness of the parallel SSA algorithm

The algorithms for the SSP and SSA class methods define an exact parallel SSA algorithm because they maintain these invariants:

Read timing: Propensity calculations read species populations at the correct times.

Write timing: Updates to species populations are performed at the correct times.

Static propensities: The propensities used to schedule every reaction that executes remain constant between the time they are computed and the reaction's execution.

We consider only shared species, since locally stored species are trivially read and updated at the correct times. **Read timing** holds because the most recent populations of all shared species are used to calculate each propensity: whenever the SSP storing a shared species receives an *Adjust populations* message it responds by sending a zero-delay *Populations* message to each SSA object that uses the shared species which were updated in the *Adjust populations* message. **Write timing** holds because the populations of all shared species modified by a reaction are updated at the SSP via an *Adjust populations* message that is executed at the time the reaction executes. And **Static propensities** holds because whenever a species population used by a propensity calculation changes, the reaction that depends on the change is cancelled and scheduling is redone, as performed in lines 5–7 of *Populations*.

However, one final issue must be resolved to complete this informal correctness proof. The dependencies among *simultaneous* SSA event messages need to be addressed. The next sub-section defines these dependencies and describes how the parallel SSA algorithm ensures that simultaneous event messages execute in the correct order.

1. Dependencies among simultaneous parallel SSA event messages

When a reaction executes all four event messages associated with the reaction execute at the simulation time of the reaction (see section **A** of Figure 3). To achieve correctness parallel SSA must control the execution order of simultaneous event messages. Specifically, they should be executed in an order consistent with the logic of sequential SSA. This section presents the rationale for that order, and the simulation mechanisms that achieve the order.

Figure 4 illustrates logical dependency relationships between *simultaneous* parallel SSA event messages. These reasons explain the dependencies in Figure 4:

1. *Populations* is a response to *Adjust populations*, so *Adjust populations* must execute before *Populations*.

2. *Populations* must follow *Execute reaction* so that species populations are not altered before *Execute reaction* is executed.

3. *Populations* must precede *Schedule reaction* so that *Schedule reaction* sees fully updated populations, as occurs in sequential SSA algorithms.
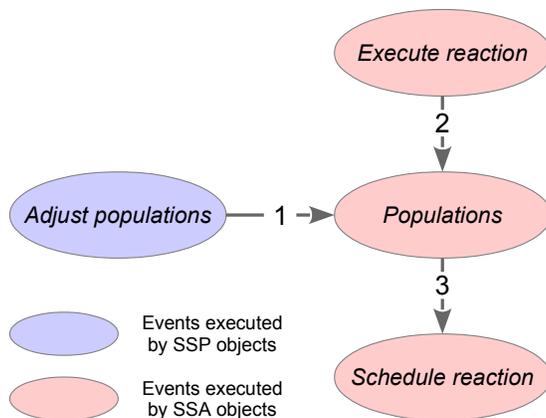
FIG. 4. Dependencies among *simultaneous* parallel SSA events associated with a single reaction execution. A directed edge from event message x to event message y means that the event that executes message x must occur before the event that executes message y. The rationale for each directed edge is provided in the text.

Parallel SSA achieves the ordering presented in Figure 4 in two levels. First, globally at any instant of time SSP objects must execute before SSA objects, as the alternative would be inconsistent with the dependencies in Figure 4.[5] And second, locally within each object type, messages must execute in the fully determined order shown in Figure 4.

Two standard PDES mechanisms can implement these two levels of ordering. Globally, simultaneous events at different PDES objects can be ordered by controlling a sub-time of the simulation time. SSP objects can be forced to execute before SSA objects when they execute simultaneously by always giving SSP objects a smaller sub-time than SSA objects.

---

[5] All simultaneous messages received by an object at a given simulation time must be passed to the object as an *event message set*.[34] Therefore, SSPs must precede SSAs at a given time to ensure that *Adjust populations* executes before *Populations*.

Locally, within a single simulation object the execution of simultaneous event messages must be ordered. To implement the ordering above SSA objects must execute their events in this order: *Execute reaction*, *Populations*, *Schedule reaction*. This ordering is straightforward to achieve. For example, a sequential OO DES engine called *DE Sim* that we wrote provides declarative support for controlling the execution order of simultaneous events.[35]

The requirement that *Populations* executes between *Execute reaction* and *Schedule reaction* forces parallel SSA to logically distinguish the event that executes a reaction from the event that schedules the next reaction.

E. Compartments

Multiple compartments are fully supported by the parallel SSA algorithm. Consider a system with two adjacent compartments, c_1 and c_2, and a single exchange reaction x that transfers a species from c_1 to c_2. Letting the species being transferred be named $s[c_1]$ and $s[c_2]$ in c_1 and c_2, respectively, reaction x can simply be $s[c_1] \rightarrow s[c_2]$. Let the reactions contained in c_1 and c_2 be mapped to SSA objects S_1 and S_2 respectively, with S_1 executing reaction x and $s[c_2]$ a shared species used by both S_1 and S_2. When x executes, the population of s is decremented in c_1 and incremented in c_2.

This exchange reaction is naturally handled by the parallel SSA algorithm. Since S_1 executes x, the population change in S_1 occurs at the time x executes, and the **Static propensities** invariant holds in S_1.

SSA object S_2 will receive a *Populations* message at the time x executes, which will cause S_2 to cancel its pending reaction and redo reaction scheduling, as performed by lines 5–8 of the *Populations* event method. Therefore, the **Static propensities** invariant holds in S_2 as well, and the parallel SSA algorithm handles reactions that transfer species between compartments.

Since each reaction execution is independent, this analysis extends to multiple exchange reactions between a pair of adjacent compartments, and to many compartments.

Lastly, we note that an exact sequential SSA simulation of a system containing multiple compartments must use an approach similar to ours because it will simulate each compartment independently and cannot be exact unless it cancels reactions which are interrupted by exchange reactions. However, unlike this parallel algorithm, it will not need to handle cascading cancellations.

F. Practical considerations for the PDES simulator

The parallel SSA algorithm application employs several features that must be supported by the OO PDES simulator on which it runs. First, because *Populations* is a *zero-delay* message[17,34] these must be allowed by the simulator. Second, the CANCEL SCHEDULED REACTION EVENTS function uses a "retract *Event message*" operation to send a *retraction* message that cancels a previously sent *Event message*. Retraction messages.[33,34] are supported by the ROSS optimistic PDES simulator[14,36]

VI. OPTIMIZATIONS AND EVALUATION

A. Optimizing parallel SSA

Multiple optimizations should be incorporated into the parallel SSA algorithm to improve its performance. These are all exact.

- Multiple SSP objects should be employed so that a) they can share computational load, and b) shared species can be mapped to SSP objects that are located on a parallel computer near the SSA objects that use them.

- The optimization introduced by NRM that recomputes only propensities which depend on species populations that have changed should be incorporated.[19] In parallel SSA, when a reaction executes these species are given by the union of the species with non-zero stoichiometry in the reaction with the shared species whose populations have been updated. When a reaction is cancelled, they are given by the updated species provided by the *Populations* message that triggered the cancellation. This optimization should be implemented using the indexed priority queue P defined in.[19] This requires that SSPs track the shared species population updates which have not been received by each SSA.

- Unnecessary data in *Populations* messages can be avoided by having SSA objects record updates to shared species locally and having SSPs not send an update to a shared species back to the SSA that reported the update. If all data in a *Populations* message is unnecessary, the message need not be sent.

These optimizations are all straightforward to implement.

In addition to these exact optimizations, approximate optimizations should also be considered.

B. Evaluation

While quantitative performance results are not available because parallel SSA has not been implemented yet, the parallel SSA algorithm achieves these conceptual objectives. It offers a method to accelerate SSA without sacrificing accuracy by partitioning reaction networks and simulating sub-networks in parallel on a supercomputer. In addition, it aims to reduce simulation run-times by maximizing the number of sub-networks in a simulation while minimizing the cost of synchronization.

The parallel SSA algorithm also faces several challenges. Partitioning will be performed on a static reaction network, whereas the characteristics of the network and its sub-networks will likely vary over the duration of a simulation. The performance of a parallel SSA simulation will depend on the rate of updates to shared species, and fraction of those updates that cause reactions to be cancelled. A partitioning algorithm that accurately estimates these rates needs to be developed.

VII.   NEXT STEPS

Much additional work must be completed before parallel SSA can become a standard tool for accelerating the simulation of large biochemical models. We plan these next steps.

- Implement the algorithm: Select a PDES simulator to use as a foundation, and implement the SSA algorithm and the optimizations from section VI A as an DES application that runs on the simulator.

- Implement partitioning: Develop a reaction-network partitioning algorithm. In addition, a related algorithm is needed to map SSA and SSP objects to processors and cores in a supercomputer when a parallel SSA simulation in initialized.

- Evaluate the implementation's performance: We will develop a benchmark reaction network, obtain a fast sequential SSA implementation such as[37], and evaluate parallel SSA's relative speedup. Initial construction has begun on a configurable generator of synthetic reaction networks.

- Integrate parallel SSA into a user-friendly modeling environment: a comprehensive environment for modeling whole-cells and other large networks needs a modeling language, a simulation experiment language, and a format for simulation results.[5]

- Combine parallel SSA with other integration algorithms in a multi-algorithmic simulator: because different pathways in cells are characterized at different levels, whole-cell models must be simulated with multiple integration algorithms, including dynamic Flux Balance Analysis and ODEs along with SSA.[5] To achieve this, we will merge parallel SSA with an existing multi-algorithmic simulator.

VIII.   CONCLUSIONS

We make important progress toward using parallelism to accelerate the Stochastic Simulation Algorithm (SSA) by presenting the first *exact* parallel algorithm for SSA. The algorithm parallelizes SSA with no loss of accuracy by partitioning a reaction network into multiple sub-networks that are simulated by independent but coordinated SSA instances. It exactly synchronizing accesses to species populations shared by the instances, and cancels pending reactions that are interrupted by population updates that invalidate prior propensity calculations. To recover from reaction cancellations the algorithm employs a modified Next Reaction Method[19] approach. All concurrent synchronization, including event timing, reaction cancellation and rollback, is achieved by leveraging the existing synchronization in optimistic parallel DES simulators. This will make the parallel SSA algorithm easier to implement and deploy. In addition, the algorithm exactly simulates systems that contain multiple compartments and transfer species between them. We present a plan for implementing, optimizing and evaluating it.

A high-performance, production parallel SSA algorithm would help enable simulations of comprehensive models of the entire biochemistry of cells, which would advance the treatment of disease and the engineering of useful microbes.

ACKNOWLEDGMENTS

REFERENCES

[1] D. T. Gillespie, The journal of physical chemistry **81**, 2340 (1977).

[2] D. T. Gillespie, Annu. Rev. Phys. Chem. **58**, 35 (2007).

[3] T. Maier, A. Schmidt, M. Güell, S. Kühner, A.-C. Gavin, R. Aebersold, and L. Serrano, Molecular systems biology **7** (2011).

[4] J. Yu, J. Xiao, X. Ren, K. Lao, and X. S. Xie, Science **311**, 1600 (2006).

[5] A. P. Goldberg, B. Szigeti, Y. H. Chew, J. A. Sekar, Y. D. Roth, and J. R. Karr, Current opinion in biotechnology **51**, 97 (2018).

[6] B. Szigeti, Y. D. Roth, J. A. Sekar, A. P. Goldberg, S. C. Pochiraju, and J. R. Karr, Current opinion in systems biology **7**, 8 (2018).

[7] J. R. Karr, J. C. Sanghvi, D. N. Macklin, M. V. Gutschow, J. M. Jacobs, B. Bolival Jr, N. Assad-Garcia, J. I. Glass, and M. W. Covert, Cell **150**, 389 (2012).

[8] A. P. Goldberg, Y. H. Chew, and J. R. Karr, in *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2016) pp. 259–262.

[9] V. H. Thanh, C. Priami, and R. Zunino, The Journal of chemical physics **141**, 10B602_1 (2014).

[10] D. T. Gillespie and L. R. Petzold, The Journal of Chemical Physics **119**, 8229 (2003).

[11] A. Auger, P. Chatelain, and P. Koumoutsakos, The Journal of chemical physics **125**, 084103 (2006).

[12] D. T. Gillespie, The Journal of chemical physics **115**, 1716 (2001).

[13] R. Fujimoto, Proceeding of the 2001 Winter Simulation Conference (Cat. No.01CH37304) **1**, 147 (2000).

[14] C. Carothers, D. Bauer, and S. Pearce, Proceedings Fourteenth Workshop on Parallel and Distributed Simulation **62**, 53 (2000).

[15] C. D. Carothers and K. S. Perumalla, in *Proceedings of the 2010 Winter Simulation Conference* (IEEE, 2010) pp. 678–687.

[16] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. DiLoreto, in *Proceedings of the eleventh ACM Symposium on Operating systems principles* (1987) pp. 77–93.

[17] D. R. Jefferson and P. D. Barnes, Jr, in *2017 Winter Simulation Conference (WSC)* (IEEE, 2017) pp. 786–797.

[18] E. Mikida, N. Jain, L. Kale, E. Gonsiorowski, C. D. Carothers, P. D. Barnes, Jr, and D. Jefferson, in *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2016) pp. 99–110.

[19] M. A. Gibson and J. Bruck, The journal of physical chemistry A **104**, 1876 (2000).

[20] Y. Cao, D. Gillespie, and L. Petzold, Journal of Computational Physics **206**, 395 (2005).

[21] D. F. Anderson, The Journal of chemical physics **127**, 214107 (2007).

[22] M. Jeschke, R. Ewald, A. Park, R. Fujimoto, and A. M. Uhrmacher, ACM SIGMETRICS Performance Evaluation Review **35**, 22 (2008).

[23] T. Mazza, P. Ballarini, R. Guido, and D. Prandi, IEEE/ACM transactions on computational biology and bioinformatics **9**, 911 (2012).

[24]L. Dematté and T. Mazza, in *International Conference on Computational Methods in Systems Biology* (Springer, 2008) pp. 191–210.

[25]B. Wang, Y. Yao, Y. Zhao, B. Hou, and S. Peng, in *2009 International Workshop on High Performance Computational Systems Biology* (IEEE, 2009) pp. 91–100.

[26]M. J. Hallock, J. E. Stone, E. Roberts, C. Fry, and Z. Luthey-Schulten, Parallel computing **40**, 86 (2014).

[27]D. R. Jefferson, ACM Transactions on Programming Languages and Systems **7**, 404 (1985).

[28]P. D. Barnes, Jr, C. D. Carothers, D. R. Jefferson, and J. M. Lapre, in *SIGSIM-PADS'13* (Association for Computing Machinery, Montrèal, 2013) pp. 327–336.

[29]C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto, ACM Transactions on Modeling and Computer Simulation (TOMACS) **9**, 224 (1999).

[30]M. Schordan, D. Jefferson, P. D. Barnes, Jr, T. Oppelstrup, and D. Quinlan, in *International Conference on Reversible Computation* (Springer, 2015) pp. 95–110.

[31]E. Mikida and L. Kale, in *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2018) pp. 189–200.

[32]R. M. Fujimoto, Communications of the ACM **33**, 30 (1990).

[33]G. Lomow, S. R. Das, and R. M. Fujimoto, ACM Transactions on Modeling and Computer Simulation (TOMACS) **1**, 219 (1991).

[34]D. R. Jefferson and P. D. Barnes, Jr, "Virtual time iii, part 1: Unified virtual time synchronization for parallel discrete event simulation," (2020), submitted.

[35]A. P. Goldberg, "Python framework for discrete event simulation," (2020).

[36]M. Schordan, T. Oppelstrup, D. Jefferson, P. D. Barnes Jr, and D. Quinlan, in *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2016) pp. 111–122.

[37]E. T. Somogyi, J.-M. Bouteiller, J. A. Glazier, M. König, J. K. Medley, M. H. Swat, and H. M. Sauro, Bioinformatics **31**, 3315 (2015).