Fine-Grained Local Dynamic Load Balancing in PDES

Anonymous Author(s)

ABSTRACT

We present a fine-grained load migration protocol intended for parallel discrete event simulation of spatially extended models. Typical models have domains that are fine-grained discretizations of some volume, e.g., a cell, using an irregular three-dimensional mesh, where most events span several subvolumes. Phenomena of interest in e.g., cellular biology, are often non-homogeneous and migrate over the simulated domain, making load-balancing a crucial part of a successful PDES. Our load migration protocol is local in the sense that it involves only those processors that exchange workload, and does not affect the running parallel simulation. We present a detailed description of the protocol and a thorough proof for its correctness. We combine our protocol with a strategy for deciding when and what load to migrate, which optimizes both for load balancing and inter-processor communication using tunable parameters. Our evaluation shows that the overhead of the load migration protocol is negligible, and that it significantly reduces the number of rollbacks caused by load imbalance.

KEYWORDS

Parallel Discrete-Event Simulation; PDES; Optimism control; Multicore; Spatial Stochastic Simulation; Load-balancing; Load Migration; Load Metric

ACM Reference Format:
Anonymous Author(s). 2018. Fine-Grained Local Dynamic Load Balancing in PDES. In *Proceedings of 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (ACM SIGSIM PADS).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Discrete Event Simulation (DES) is an important tool in a wide-ranging area of applications, such as integrated circuit design, systems biology, epidemics, etc. To improve performance and accommodate for large scale models, a vast repertoire of techniques have been developed for Parallel DES (PDES) during the last 30 years [15, 19, 24, 25]. New synchronization techniques have been triggered by the advent of multicore processors (e.g., [6, 26, 32]). Still, achieving good performance and speedup for larger number of processing elements has proven to be very difficult in the general case.

In PDES, the simulation model is partitioned onto logical processes (LPs), each of which processes timestamped events to evolve its partition along a local simulation time axis. Events that affect the state of neighboring LPs are exchanged to incorporate inter-LP dependencies. A number of synchronization techniques have been developed in order to guarantee that causally dependent events are processed in the right order, ranging from conservative [25] to optimistic [19] approaches, with many intermediate design choices (see, e.g., the surveys [8, 18]). Such intermediate protocols can reduce the performance loss caused by temporary variations in relative speed of different LPs. However, perhaps the most important prerequisite for high efficiency in PDES is that the simulation load is evenly partitioned over LPs This follows from the observation that the total simulation speed can never exceed that of the slowest LP (assuming a one-to-one correspondence between LPs and processors). For simulation models where the distribution of work does not vary over time, this can be achieved by a good static partitioning of the model before simulation starts. However, in many simulation models, the distribution of work varies with time. For instance, in systems biology, the phenomena of interest are often non-homogeneous and migrate over the simulation domain; examples include nerve signals, and the oscillation of proteins involved in the cell division of bacteria [14]. For such models, good parallel performance requires a dynamic load balancing mechanism, which detects when load imbalances arise and corrects them by migrating load between LPs.

Most existing approaches perform dynamic load balancing as a globally coordinated operation, at specified (often regular) time intervals [3–5, 12, 17, 22, 27–29, 33]. For models, where the load migrates continuously over the domain of the simulation model, load imbalances arise locally, and it seems more natural to let re-balancing be a local operation, which involves only those LPs that are affected by the migrating load, and is performed on-line. Such a mechanism must be designed carefully to preserve correctness of the underlying simulation.

In this paper, we present an online fine-grained dynamic load migration protocol, which is local in the sense that it concerns only those LPs that are affected by the migrated load. Our protocol is defined for simulation of spatial models that are discretized into *subvolumes* (also known as voxels) which are partitioned onto LPs. An example model is a bacterial cell, where the state of each voxel is represented by a number of entities of different proteins. Each LP is mapped to a core in a multicore processor. Our protocol migrates individual voxels between LPs, with low overhead for the underlying simulation algorithm. It assumes no restrictions on the topology of the simulation domain, nor on the number of LPs adjacent to a migrated element.

Our underlying implementation uses the time-warp protocol [19], with some optimizations. One of these is the use of *aggregated anti-messages*, each representing a set of inter-core anti-messages, that reduce inter-core communication to improve performance. These require extra care in the migration protocol, since their semantics may change as a result of voxel migration. We have therefore developed a proof of correctness of our time-warp protocol with voxel migration. The proof is inspired by that of [21], but adapted to our

ACM SIGSIM PADS, May 23–25, 2018, Sapienza University, Rome, Italy
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

version of time-warp with aggregated anti-messages, and thereafter extended to cover the migration protocol.

A load balancing mechanism must also include a load metric which represents the amount of load on an LP, and a mechanism for deciding which voxels to migrate where. Defining a good load metric in optimistic PDES is challenging, since natural metrics such as the average CPU load, are no good indicators of the actual simulation progress, since CPUs may be busy processing rollbacks. Many existing approaches [1, 10, 12, 27] to load balancing in PDES base their load migration choices on an LPs load metric related to some global computation including all LPs' load metrics. As our load migration protocol is local, there is a need for a local load metric. Here, our starting point is that observed synchronization costs, such as rollback processing, are very likely caused by load imbalance. In the ideal case, when the load is perfectly balanced, (almost) no messages arrive too late, and thus there should be no need for rollbacks. We therefore use the number of locally incurred rollbacks as a load metric. Even though it is not realistic to completely avoid rollbacks, it is reasonable to assume that by continuous dynamic rebalancing of the load, their incurred overhead should be substantially lower than the overhead of adding an optimism control protocol: our simulator therefore does not employ any additional optimism control.

We show the correctness of the load-balancing protocol, and evaluate the performance of our approach, together with two simple metrics for when and where load balancing is needed, on realistic benchmarks from computational cellular biology. Our evaluation shows that the overhead of the load migration protocol is negligible,that it significantly reduces the number of rollbacks caused by load imbalance, and that it achieves better speedup than a similar protocol without load balancing but with fine-grained mechanism for optimism control.

The paper is organized as follows. In the next section, we give an overview of related work. In Section 3, some necessary background and the main ideas of our load migration protocol is presented. In Section 4, a detailed description of the load migration protocol is given. In Section 5, we argue for the correctness of the migration protocol. In Section 6, we give a short outline of the load metric and load balancing algorithm that we've used for the evaluation of the migration protocol, in Section 7.

2 RELATED WORK

There are few works that has treated dynamic load balancing in spatial models. Deelman and Szymanski [10] consider a dynamic load balancing mechanism on a spatial two-dimensional circular model. The model is partitioned into segments, each assigned to an LP. Thus, each LP only has two neighbors. The load balancing mechanism moves columns of the spatial domain between adjacent processing elements.

Similar to Schlagenhaft et al. [30], who also have clustered the basic elements of computation, we do the load-balancing fully within the simulation application. However, they discard the option of moving single elements between partition, which we have opted for in this paper. They argue that no significant change in load balance is achieved with such a fine-grained protocol. However, that only depends on how often migrations are done.

A number of works outline their load migration protocol, albeit very concisely. Avril and Tropper [1] outlines how the load is transferred in two phases. First, a a message containing an encoding of the load (i.e., a set of LPs) is sent, and messages destined for any of the migrated LPs are temporarily stored. Later, the receipt of the LPs is acknowledged, and the new location of the load is broadcast. Received messages are forwarded from the old to the new owner of the data structure. Later messages for the load will also be forwarded. Burdorf and Marti [4] presents a rudimentary outline of their load migration protocol.

Sarkar and Das [29] evaluate two different mechanisms of load balancing of different granularity; a load transfer mechanism between LPs, and an LP transfer mechanism between physical processors. They use a total of 16 LPs, and limit the load balancing to occur only between the fastest and the slowest LP once per load balance cycle. They observe that load transfer is more effective for smaller grain sizes, than the process transfer mechanism. Load balancing through process migration has also been studied in Glazer and Tropper [17], Reiher and Jefferson [28]. It requires operating system support, and for very many small migration units the overhead of such an approach is substantial. The aforementioned arguments has been the reason in several cases for authors choosing another method [10, 30].

Another approach to load balancing is scattering. Thulasidasan et al. [31] explore a static spatial scattering partitioning technique. Adjacent LPs are assigned to different processors to minimize load imbalance, even when the load may be concentrated to a few hotspots in the spatial model. They report superior performance than other static partitioning techniques, even though message passing suffers from a high overhead due to the scattering.

Load Metrics. It is common to use some kind of relation between the simulation time of each LP and the wall-clock time to define load. The difference in progress between LPs. The *simulation advance rate* [12, 17, 29] (referred to using various names) is defined as

$$\text{rate} = \frac{\mathrm{ST}_{T_2} - \mathrm{ST}_{T_1}}{T_2 - T_1},$$

where ST denotes simulation time, and T denotes wall-clock time.

In [27, 30], the *total advance time* is defined as the integral of forward simulated time (thus the simulation of a rolled back event will count twice) during a period of wall-clock time. The total advance rate is then an inverse load metric. In [27], the metric is extended, so that the load is defined as the number of processed events divided by the total advance time.

Reiher and Jefferson [28] define *effective utilization* as the fraction of processor cycles spent on useful work, i.e., not on rollbacks. As the effective utilization is not possible to know (any work may be rolled back, at any time), they make an estimator, by recording the number of cycles spent on each event is recorded. When an event is rolled back, the corresponding number of cycles are subtracted from the estimator.

Lin and Yao [22] define the load as the weighted sum of the inverse to the distance to GVT, and the number of events committed, per round (i.e., rebalancing occurs at regular intervals).

The load metric in Deelman and Szymanski [10] is the number of future scheduled events, weighted by their imminence. They mention the advantage of basing load balance on future load instead of past load. Load metric data is gathered during the centralized GVT calculation, and then disseminated to all processes. The algorithm is evaluated on a basic model where two nodes have a high load, and two nodes have a light load.

Several works also tries to integrate a communication reducing element in the metric. Peschlow et al. [27] present a method where two separate metrics are used for the load balancing, a load metric and a communication metric. Communication and load is then balanced alternately. Xu et al. [34] devise a global partitioning method intended to reduce communication by reducing the number of neighbors of each partition.

Initiation of Load Balancing. Many papers on load balancing in PDES prescribe that the load balancing should be initiated at regular time periods, ranging from less than 5 seconds of wall-clock time to 10 minutes of simulation time [3, 5, 17, 22, 27–29, 33, 34], sometimes also expressed in multiples of GVT computations [4, 12].

Some papers have looked into a more dynamic approach to initiation of load balancing. In [1], a non-static interval is used, based on measurements taken every 3 seconds. In [7], a non-static interval is used, based on data updated every 500 events received. In [30] and [1], the authors try to account for the increased cost a migration carries, and their migration decision is based on when the predicted migration cost will improve over the predicted non-migration cost.

3 FRAMEWORK

Our load-balancing protocol is developed in the framework of spatial stochastic simulation. Since this framework has influenced some of our design decisions, we describe it here briefly.

3.1 Spatial Stochastic Simulation

The reaction-diffusion master equation (RDME) [16] describes systems where entities diffuse in some volume and may undergo transitions, or reactions, when in proximity to each other. The RDME is a popular tool for modeling biological systems where the population of entities is low and discrete effects therefore play an important role for the behavior of the systems.

In this framework, simulation models are defined over a spatial domain, which is discretized into *subvolumes*, also known as voxels. Each voxel contains a discrete *copy number* of entities of some set of species (e.g., proteins). The dynamics of the model is described by a spatially extended Markov process, in which two types of transitions are possible: (i) in a *reaction* a combination of species residing in a subvolume reacts and produces a new combination within the same subvolume, (ii) in a *diffusion* a single entity of a species moves to a neighboring subvolume. In general, each subvolume can host several types of reactions with different combinations of entities. The inter-event times between reactions and diffusions are stochastic, highly variable and without a lower bound. An important feature of typical models is that diffusions are significantly more common than reactions (by at least a factor of 10), and that the local state of each voxel is relatively simple (consisting of the copy numbers of all participating species).

Practically, the RDME is simulated using sampling methods that produce single trajectories from the relevant probability space. The most commonly used such method is the Next Subvolume Method (NSM) [13]. The NSM algorithm takes the form of DES, whose event queue contains the next occurrence time of the next event within each subvolume. The execution of that subvolume event first decides (by a random draw) which particular reaction or diffusion will occur, and then performs it: the the states of concerned subvolumes are updated, and the subvolume's next occurrence time is inserted into the event queue.

3.2 Parallelization using Standard Time Warp

Since the number of voxels is typically large, the natural approach to parallelization is to partition the set of voxels into subdomains, each of which is assigned to an LP, which is then mapped to a processing element. Each LP simulates the dynamics of its subdomain, using the NSM algorithm. We use the Time Warp synchronization mechanism [19] without optimism control. In order to support migration of voxels between LPs, voxel-specific information is stored in a self-contained structure, which for each voxel v maintains

- v.state, the local state of v,
- v.next, the time of its next event occurrence,
- v.history, the history of its processed events,
- v.routing, a routing table maps each voxel neighbor in the simulation model to the index of the LP on it is located.

We let LVT(v) be the time of the most recent event in v.history; intuitively, this is the local simulation time of voxel v. Each LP LP_i itself maintains LP-global information, viz.

- $\mathrm{LP}_i.\mathit{EventQueue}$, a time-sorted event queue, containing the occurrence time of the next event of each voxel in its subdomain,
- $\mathrm{LP}_i.\mathit{bnd}$, which maps each neighboring LP LP_j of LP_i to the set of voxels of LP_i that have some neighbor on LP_j. This is used to define the meaning of aggregated anti-messages (defined below), i.e., voxels in $\mathrm{LP}_i.\mathit{bnd}(\mathrm{LP}_j)$ will be rolled back when an aggregated anti-message is received from LP_j.

Each pair of neighbouring LPs exchange messages via unidirectional channels. We let $\mathit{chan}_{i\to j}$ denote the channel from $\mathrm{LP_i}$ to $\mathrm{LP_j}$ (there is then also a channel $\mathit{chan}_{j\to i}$ in the opposite direction). Each diffusion to a voxel residing on a different LP induces a message to that LP. We use $v \xrightarrow{t} v'$ to represent a message denoting that a diffusion from voxel v to voxel v' has occurred at time t (the message also contains the diffused species, which we ignore here). Channels also carry other types of messages: anti-messages, and control messages associated with the migration protocol, which are described later.

Each LP advances the simulation by finding the next event to process, either from the top of its event queue or from a message that is at the front of an incoming channel. Thereafter, the event is processed by (1) updating the states of affected subvolumes, (2) adding the event to the histories of affected subvolumes, and (3) if the event was taken from the event queue, determining a new next event time for the initiating voxel. If the event is a diffusion to another LP, a diffusion message is transmitted to the neighbor.

Selective Rollback. Whenever the next event to process is an incoming diffusion message of form $v \xrightarrow{t} v'$ such that $t \leq \mathrm{LVT}(v')$ (also called a *straggler*), a rollback is initiated. To do so, events are processed "backwards" until such a previous time is reached.

A simple approach would be to roll back all events performed by the LP of v' with a time stamp higher than t. However, it is typically less costly to perform a *selective rollback*, which only rolls back events that may be causally dependent on rolled back events involving v' [2, 9]. In addition, rollback of causally dependent events performed on other LPs must be initiated by sending anti-messages to concerned LPs. In our algorithm, anti-messages are aggregated per LP, i.e., in each rollback, at most one anti-message is sent per neighboring LP. An aggregated anti-message from LP_i to LP_j only carries a single timestamp, which is the minimum timestamp of a rolled back inter-LP diffusion at LP_i which also involves LP_j. Since the anti-message is only a single timestamp, it will initiate rollback of all inter-LP diffusions involving both LP_i and LP_j, even those that are not causally dependent on the initiating straggler.

More precisely, the rollback procedure on an LP_i can be described by specifying for each voxel v of LP_i a rollback time $RBT(v)$, and for each neighbor LP_j of LP_i, a time $AMT(\mathrm{LP}_j)$ for its anti-message. These times are the largest ones (including ∞) that satisfy the following constraints:

(1) if a straggler $v \xrightarrow{t} v'$ arrives to LP_i, then $RBT(v') < t$,
(2) if voxels v and v' of LP_i have performed a joint diffusion at time t with $RBT(v) \leq t$, then $RBT(v') \leq t$,
(3) if voxel v of LP_i has performed a diffusion $v \xrightarrow{t} v'$ such that $v' \in \mathrm{LP}_j$ and $RBT(v) \leq t$, then $AMT(\mathrm{LP}_j) \leq t$,
(4) if voxel v of LP_i has performed a diffusion $v \xrightarrow{t} v'$ such that $v' \in \mathrm{LP}_j$ and $AMT(\mathrm{LP}_j) \leq t$, then $RBT(v) \leq t$.

Intuitively, Conditions 1 and 2 describe the causality constraints for rolling back voxels on LP_i, Condition 3 specifies the timestamp of the aggregated anti-message to LP_j, and Condition 4 specifies the additional rollback that is caused by assuming that the anti-message represents all diffusions between LP_i and LP_j that are not earlier than $AMT(\mathrm{LP}_j)$.

When an aggregated anti-message from LP_j with time t arrives to LP_i, then a rollback is initiated with the same constraints as above, except that condition 1 is replaced by

(1') if voxel v of LP_i has received a diffusion $v' \xrightarrow{t'} v$ and $v' \in \mathrm{LP}_j$, with $t \leq t'$, then $RBT(v) < t'$.

4 MIGRATION ALGORITHM

In this section, we describe in detail the essential algorithms involved in the migration protocol. We start by describing the data structures being used, thereafter we describe the rollback and anti-message routines, and finally we describe the processing of messages and the core migration protocol.

4.1 Migration of a Voxel

A voxel v is migrated between two LPs by sending a control message $\overrightarrow{v}$ containing v. In the implementation, what is actually sent is a pointer to the structure representing v, which includes v.history, v.routing and v.next. Both the sending and the receiving LP will update relevant status information concerning v upon sending and receiving this message. The main challenge is that any messages being in flight to the migrated voxel v have to be rerouted, and must be guaranteed to arrive in the correct order. That is, to ensure correctness, it must be guaranteed that for any pair of voxels v_0, v_1, messages sent from v_0 to v_1, including anti-messages, are received in the same order as they are sent. A particular challenge is that anti-messages are aggregated, and that their meaning changes as a result of the migration. The protocol must therefore be carefully designed to consider also this complication.

In Algorithm 1 the procedure for sending a voxel v from $\mathrm{LP}_{\mathrm{src}}$ to $\mathrm{LP}_{\mathrm{dst}}$ is described. At line 2, the LOCK function ensures that no voxel neighbor to v is migrated simultaneously with v, details are described in Algorithm 3. Essentially, it atomically sets a *migration flag* on the channel to each neighboring LP that maintains a voxel neighbor to v. If one such flag already is set, i.e., the migration of some voxel neighbor v' to v has already been initiated, then the migration of v is aborted. At lines 3 and 4, v is removed from the local state. At line 6 internal routing information of v's neighboring voxels are updated to locate v on $\mathrm{LP}_{\mathrm{dst}}$. At line 7 the boundary list of voxels bordering $\mathrm{LP}_{\mathrm{dst}}$ is updated to contain all neighboring voxels of v. The function returns a set of neighbors Neigh on which v had neighbors. Since the neighbors in Neigh has voxels who are neighbors to v, these neighbors have to be informed about the migration, by means of message sent at line 10. The actual voxel is sent to $\mathrm{LP}_{\mathrm{dst}}$ at line 11.

Algorithm 1 Sending a voxel v from $\mathrm{LP}_{\mathrm{src}}$ to some $\mathrm{LP}_{\mathrm{dst}}$.

```
1  function SendVoxel($v$)
2    if Lock($v$) then return
3    $EventQueue \leftarrow \{\langle v_k, t\rangle \mid \langle v_k, t\rangle \in EventQueue \wedge v_k \neq v\}$
4    State ← State \ $v$
5    for each $v_{\mathrm{nbr}} \in \mathrm{LP}_{\mathrm{src}}$ that is a neighbor to $v$ do
6      $v_{\mathrm{nbr}}$.routing[$v$] ← $\mathrm{LP}_{\mathrm{dst}}$
7      $bnd(\mathrm{LP}_{\mathrm{dst}}) \leftarrow bnd(\mathrm{LP}_{\mathrm{dst}}) \cup v_{\mathrm{nbr}}$
8    for each $\mathrm{LP}_k \neq \mathrm{LP}_{\mathrm{src}}$ with a neighbor to $v$ do
9      remove $v$ from $bnd(\mathrm{LP}_k)$
10     Send($\mathrm{mv}_v(\mathrm{LP}_{\mathrm{dst}}), \mathrm{LP}_k$)
11   Send($\overrightarrow{v}, \mathrm{LP}_{\mathrm{dst}}$)
```

Our protocol for migrating a voxel v_m from $\mathrm{LP}_{\mathrm{src}}$ to $\mathrm{LP}_{\mathrm{dst}}$ makes use of the following control messages.

$\overrightarrow{v_m}$ is a control message, which transfers the migrated voxel v_m,
$\mathrm{mv}_{v_m}(\mathrm{LP}_{\mathrm{dst}})$ is sent by $\mathrm{LP}_{\mathrm{src}}$ to each LP (except $\mathrm{LP}_{\mathrm{dst}}$) whose domain contains neighbors of v_m, announcing that the voxel v_m has just been sent to $\mathrm{LP}_{\mathrm{dst}}$,
recv_{v_m} is sent by $\mathrm{LP}_{\mathrm{dst}}$ to each LP (except $\mathrm{LP}_{\mathrm{src}}$) whose domain contains neighbors of v_m, announcing that the voxel v_m has just been received at $\mathrm{LP}_{\mathrm{dst}}$,
forw_{v_m} is sent to $\mathrm{LP}_{\mathrm{dst}}$ by each LP (except $\mathrm{LP}_{\mathrm{src}}$) whose domain contains neighbors of v_m, announcing that they has received recv_{v_m} and is about to send normal messages (containing diffusion events) to v_m.

An overall description of the message protocol for migrating a voxel v_m follows.

- $\mathrm{LP}_{\mathrm{src}}$ initiates the migration by sending $\overrightarrow{v_m}$ to $\mathrm{LP}_{\mathrm{dst}}$. At the same time, $\mathrm{LP}_{\mathrm{src}}$ also sends the message mv_{v_m} to each neighbor LP_k (except $\mathrm{LP}_{\mathrm{dst}}$) whose domain contains some voxel neighbor of v_m, announcing that the voxel v_m has just been sent to $\mathrm{LP}_{\mathrm{dst}}$. For each such neighbor LP_k, and also for $\mathrm{LP}_{\mathrm{dst}}$, $\mathrm{LP}_{\mathrm{src}}$ creates a temporary data structure, denoted $log_{k \to \mathrm{src}}(v_m)$, in which it stores all received messages and anti-messages to v_m; these messages will thereafter be retrieved by LP_k in order to forward them to $\mathrm{LP}_{\mathrm{dst}}$.

- upon receipt of mv_{v_m}, each LP_k (except $\mathrm{LP}_{\mathrm{src}}$ and $\mathrm{LP}_{\mathrm{dst}}$) which formerly sent messages to v_m via $chan_{k\to\mathrm{src}}$, will first send the message forw_{v_m} to $\mathrm{LP}_{\mathrm{dst}}$, announcing that it will start to send messages to v_m over $chan_{k\to\mathrm{dst}}$. Thereafter, it retrieves all such messages sent, but not yet received, from $log_{k\to\mathrm{src}}(v_m)$ and $chan_{k\to\mathrm{src}}$, and thereafter forwards them (in order) to $\mathrm{LP}_{\mathrm{dst}}$ (over $chan_{k\to\mathrm{dst}}$).
- Upon receipt of $\overrightarrow{v_m}$, $\mathrm{LP}_{\mathrm{dst}}$ sends a message recv_{v_m} to LPs (except $\mathrm{LP}_{\mathrm{src}}$) whose domain contains neighbors of v_m, announcing that the voxel v_m has been received at $\mathrm{LP}_{\mathrm{dst}}$. Thereafter, it will (in the same manner as LP_k in the previous step) retrieve all messages to v_m sent, but not yet received, from $log_{\mathrm{dst}\to\mathrm{src}}(v_m)$ and $chan_{\mathrm{dst}\to\mathrm{src}}$. The events in these messages are then rolled back.

The control messages recv_{v_m} and forw_{v_m} block, i.e., prevent any message from being received, from the channels from which they were retrieved, unless $\mathrm{mv}_{v_m} < \mathrm{recv}_{v_m}$ and $\overrightarrow{v_m} < \mathrm{forw}_{v_m}$, respectively (where $<$ denotes order of reception).

In Algorithm 2, the detailed protocol logic for how an LP handles the different types of incoming messages is described. For each message type, the description refers to lines in Algorithm 2, unless stated otherwise. In the algorithm, the LP where a message is received is denoted LP_i, and the sending LP is denoted LP_k.

$\overrightarrow{v}$ The message $\overrightarrow{v}$ carries a migrating voxel v. Upon receipt of $\overrightarrow{v}$, LP_i first updates its bookkeeping information: v is added to the local state (lines 3 and 4), the routing tables of v's neighboring voxels are updated (lines 5 to 8), v is marked as located on LP_i in each neighbouring voxel's internal routing table (line 6), and v's local neighbors are also removed from the list of voxels bordering LP_k, if they no longer have any neighbours on LP_k (line 8). Thereafter, a message recv_v is sent to each LP (other than LP_k) with a neighbor voxel of v, informing that the migration has completed (line 11). If any incoming channel is closed due to the migration of v (i.e., a message forw_v has been received from some LP_j), it can now be reopened (line 12). Thereafter, LP_i retrieves all pending messages sent from LP_i to v, from the temporary structure $log_{i\to k}(v)$ stored at LP_k and from $chan_{i\to k}$; this retrieval also transforms aggregated anti-messages (with only a time t) into voxel-specific anti-messages (denoted $-t_v$). The retrieval from the channel is performed in the function PROJECT, described by (1) below. The sequence of retrieved messages are collected into the ordered list M_v (line 13). The projected diffusions are simultaneously removed from the corresponding channel, and the migration flag for v in $chan_{i\to k}$ is unset, indicating that no more messages should be added to the backlog $log_{i\to k}(v)$ when messages are received from $chan_{i\to k}$ (lines 15 and 16). Finally, the state of v and the messages in M_v are rolled back (lines 17 to 20), to avoid any ordering conflicts of future and already sent messages.

$\mathrm{mv}_v(\mathrm{LP}_j)$ This type of message is sent to LPs that have voxel neighbors to some migrating voxel $\overrightarrow{v}$ destined for LP_j, but which are not the recipient of $\overrightarrow{v}$ (at line 10 in Algorithm 1). Upon the receipt of the message at some LP_i, retrieves into M_v all pending messages sent from LP_i to v, from the temporary structure $logik(v)$ stored at LP_k and from $chan_{i\to k}$ in the same way as described for messages of type $\overrightarrow{v}$ (using the function PROJECT). The messages in M_v are forwarded to LP_j, the new LP of v, in the same order they previously were sent to LP_k, prefixed by a message of type forw_v (lines 26 and 27). Thereafter, for each voxel-neighbor v_n to v residing on LP_i, the LPs boundary information is updated as follows: If v_n has no other voxel neighbors on LP_k, then v_n is removed from the list of voxels representing the boundary to LP_k (line 31). The set of v's voxel neighbors is then stored in $mvbnd$, tagged with v (line 32). The voxels will later be added to the boundary of LP_k facing LP_j. If the channel from LP_j to LP_i is marked as closed, then it is reopened, and each voxel-neighbor v_n are immediately added to the list of voxels bordering to LP_j (lines 33 to 36).

recv_v This type of message is sent by the LP receiving a message $\overrightarrow{v}$ to LPs that are not the sender of the message $\overrightarrow{v}$, but have voxels neighboring v. Upon receipt of a message recv_v at LP_i, a check is done whether a corresponding message mv_v has been received, by checking if tag v exists in the set $mvbnd$ (line 38). If not, the channel to LP_i is closed to enforce that a message mv_v is received first for each migration of a voxel. Otherwise, the set of v's neighboring voxels on LP_k, previously bordering LP_i, are now included in the boundary towards LP_j (line 41).

forw_v This type of message is sent by an LP that has voxels neighboring some recently migrated voxel v, but was neither the sender nor the recipient of the corresponding migration message $\overrightarrow{v}$. It is sent to the LP to which v has been migrated. The message indicates that after the transmission of forw_v from LP_k to LP_j, LP_k starts forwarding messages destined for v to LP_j. Upon receipt of forw_w, LP_j checks if $\overrightarrow{v}$ has already been received. If not, the channel $chan_{k\to j}$ is closed (line 50). This enforces that $\overrightarrow{v}$ is received before any messages destined for v.

$-t$ When receiving an anti-message $-t$ from LP_k at time t, the history of each voxel v that may have received a diffusion from LP_k is scanned (line 43). If there is such a diffusion $w \xrightarrow{t'} v, w \in \mathrm{LP}_k$ which happened after t, rollback v_i to t' (line 44). Then, for all voxels that have been migrated to some other LP, say LP_j, and who previously were on the boundary to LP_k (and thus have an entry in $chan_{k\to i}$.flags), a copy of the anti-message is put in the corresponding backlog (line 45).

$w \xrightarrow{t} v$ When receiving a diffusion, if the receiving voxel v has migrated, the diffusion is added to the backlog of messages for v, located on $chan_{k\to i}$ (line 53). If the diffusion is a straggler, a rollback is performed (line 55), before processing the diffusion (line 56).

The PROJECT function (1) extracts all messages in a channel $chan$ destined for a voxel v. It should be noted, that the channel is protected by a lock, thus the effect of PROJECT is observed to take place instantaneously for any LP. The projection is done as follows: The messages in $chan$ are recursively traversed. Diffusions destined for v are returned, and anti-messages $-t$ are converted

Algorithm 2 Receipt of message from LP_k at LP_i.

```
function HandleMessage($m$)
  if $m = \overrightarrow{v}$ then
    $EventQueue \leftarrow$ insert $(EventQueue, \langle v.\text{next}, v\rangle)$
    $State \leftarrow State \cup v$
    for each $v_{\text{nbr}} \in \mathrm{LP}_i$ that is a neighbor to $v$ do
      $v_{\text{nbr}}.\text{routing}[v] \leftarrow \mathrm{LP}_i$
      if $v_{\text{nbr}}$ has no more neighbors on $\mathrm{LP}_k$ then
        remove $v_{\text{nbr}}$ from $bnd(\mathrm{LP}_k)$
    add $v$ to $bnd(\mathrm{LP}_k)$
    for each $\mathrm{LP}_j \neq \mathrm{LP}_k$ with a neighbor to $v$ do
      Send$(\text{recv}_v, \mathrm{LP}_j)$
      if $chan_{j\to i}$ closed by migration of $v$ then open $chan_{j\to i}$
    $M_v \leftarrow log_{i\to k}(v) \mathbin{+\!\!+} \textsc{Project}(chan_{i\to k}, v)$
    $log_{i\to k}(v) \leftarrow \emptyset$
    remove all diffusions destined for $v$ from $chan_{i\to k}$
    remove $\langle v, \mathrm{LP}_k\rangle$ from $chan_{i\to k}.\text{flags}$
    $t_{\min} \leftarrow \min\{\, t \mid w \xrightarrow{t} v \in M_v \lor -t \in M_v \lor -t_v \in M_v \,\}$
    Rollback$(v, t_{\min})$
    for each $w \xrightarrow{t} v \in M_v$ do
      Rollback$(w, t)$
  if $m = \text{mv}_v(\mathrm{LP}_j)$ then
    $M_v \leftarrow log_{i\to k}(v) \mathbin{+\!\!+} \textsc{Project}(chan_{i\to k}, v)$
    $log_{i\to k}(v) \leftarrow \emptyset$
    remove all diffusions destined for $v$ from $chan_{i\to k}$
    remove $\langle v, \mathrm{LP}_k\rangle$ from $chan_{i\to k}.\text{flags}$
    Send$(\text{forw}_v, \mathrm{LP}_j)$
    Send$(M_v, \mathrm{LP}_j)$
    $V_{\text{bnd}} \leftarrow \{v_{\text{nbr}} \mid v_{\text{nbr}} \in \mathrm{LP}_i \land v_{\text{nbr}}$ neighbor to $v\}$
    for each $v_{\text{nbr}} \in V_{\text{bnd}}$ do
      if $v_{\text{nbr}}$ has no more neighbors on $\mathrm{LP}_k$ then
        remove $v_{\text{nbr}}$ from $bnd(\mathrm{LP}_k)$
    $mvbnd \leftarrow mvbnd \cup \langle v, V_{\text{bnd}}\rangle$
    if $chan_{j\to i}.\text{closed}$ then
      $bnd(\mathrm{LP}_j) \leftarrow bnd(\mathrm{LP}_j) \cup V_{\text{bnd}}$ s.t. $\langle v, V_{\text{bnd}}\rangle \in mvbnd$
      if $chan_{j\to i}$ closed by migration of $v$ then
        open $chan_{j\to i}$
  if $m = \text{recv}_v$ then
    if $\nexists\, V_{\text{bnd}}.\langle v, V_{\text{bnd}}\rangle \in mvbnd$ then
      close channel $chan_{k\to i}$
    else
      $bnd(\mathrm{LP}_k) \leftarrow bnd(\mathrm{LP}_k) \cup V_{\text{bnd}}\text{s.t.}\langle v, V_{\text{bnd}}\rangle \in mvbnd$
  if $m = -t$ then                        ▷ Aggregated anti-message
    for each $v \in bnd(\mathrm{LP}_k)$ do
      Rollback$(v, t)$                    ▷ According to (1'), Section 3
    for each $\langle v, \mathrm{LP_j}\rangle \in chan_{k\to i}.\text{flags}$ do
      $log_{j\to i}(v) \leftarrow log_{j\to i}(v) \cdot -t_v$
  if $m = -t_v$ then                      ▷ Local anti-message
    Rollback$(v, t)$                      ▷ According to (1'), Section 3
  if type = $\text{forw}_v$ then
    if $v \notin State$ then close $chan_{k\to i}$
  if $m = w \xrightarrow{t} v$ then       ▷ Regular diffusion to $v$
    if $v \notin State$ then              ▷ $v$ is migrated to some other LP
      $log_{k\to i}(v) \leftarrow log_{k\to i}(v) \cdot m$
    if $t \leq \text{LVT}(v)$ then        ▷ Handle straggler
      Rollback$(v, t)$
    process $w \xrightarrow{t} v$
```

to local anti-messages $-t_v$, only affecting v. Other messages are filtered out.

$$\textsc{Project}(chan, v) = \begin{cases} m \cdot \textsc{Project}(chan, v) & \text{if } m = w \xrightarrow{t} v \\ -t_v \cdot \textsc{Project}(chan, v) & \text{if } m = -t \\ \textsc{Project}(chan, v) & \text{otherwise} \end{cases} \quad (1)$$

In Algorithm 3, we see the mechanism for preventing simultaneous migrations of neighboring voxels. For all channels coming from LPs where v has a neighbor, a migration flag is set marking that an attempt migrating v is undertaken (line 5). If, after having set the flag, a flag of the corresponding neighbor is seen on the outgoing channel, all flags already set on the channels are unset, and the lock procedure returns false (line 8). If the procedure manages to set all flags without seeing a flag set on a corresponding outgoing channel, it has succeeded the migration may take place, and returns true.

Algorithm 3 Prevent simultaneous migration of a voxel v from LP_i and any of its neighbors v_n.

```
1 function Lock($v$)
2   $V_n \leftarrow \{v_n \mid v_n \notin \mathrm{LP}_i \land v_n$ neighbor to $v\}$
3   for each $v_n \in V_n$ do
4     $m \leftarrow v.\text{routing}[v_n]$
5     $chan_{m\to i}.\text{flags} \leftarrow chan_{m\to i}.\text{flags} \cup v$
6     if $v_n \in chan_{i\to m}.\text{flags}$ then
7       unset all set flags and abort
8       return false
9   return true
```

5 CORRECTNESS

In this section, we give a proof for the correctness of our migration protocol. It is inspired by that of [21], but adapted to cover our version of time-warp with aggregated anti-messages and voxel migration. We will focus on the property of safety: that any parallel execution produces the same simulation run as the corresponding sequential simulation algorithm. The notion of "produced simulation run" is well-defined if the simulation model is deterministic, so that a simulation run is uniquely determined by the initial state and the transition rules, which in our case are fixed (we make the assumption that no two events in a voxel have exactly the same time-stamp). In the presence of random events, the simulation is made deterministic by using deterministic pseudorandom number generators, whose states are included in the local states of the corresponding voxels, and which are reverted together with the voxel state when performing rollbacks.

As described in Section 3, our simulation models consist of voxels, which evolve the state of the model by performing reactions, which are local, and diffusions that also affect the state of a neighbouring voxel. Within a voxel v, the simulation run is defined by the ordered sequence $v.\text{history}$ of its processed time-stamped events. We define $\text{In}^{v'}(v)$ as the sequence of processed incoming diffusion from v' in $v.\text{history}$, and let $\text{Out}^{v'}(v)$ be the sequence of sent diffusions to voxel v' in $v.\text{history}$. A sequential simulation run satisfies the following two properties.

(i) The history $v.\text{history}$ in a sequential simulation is the one that is uniquely determined by following the simulation rules from the initial voxel state and the sequences $\mathrm{In}^{v'}(v)$ of processed incoming diffusion messages from each neighbour v' in the simulation model.

(ii) For each pair v, v' of neigbour voxels in the simulation model, $\mathrm{In}^{v}(v') = \mathrm{Out}^{v'}(v)$, i.e., the sequence of incoming diffusions to v' from v is the same as the sequence of outgoing diffusions from v to v'.

Property (i) uses the assumption that no two events in v.history have exactly the same timestamp. Property (ii) follows by noting that in a sequential simulation, a diffusion from v to v' is simultaneously added both to v.history and to v'.history.

Conversely, any completed simulation run, which is constructed from voxel histories that satisfy (i) and (ii) is the same as the uniquely defined simulation run. We can then establish safety of the parallel simulation by proving that it satisfies properties (i) and (ii).

Property (i) for the parallel simulation algorithm can be established by checking that each step (both forward simulation steps and rollbacks) in the simulation algorithm respect the rules of the simulated model. This is not difficult, an we omit the details.

Property (ii) is less straightforward: in general it does not hold during the simulation run, since diffusion messages and anti-messages may be in transit between v and v' when they reside on different LPs. We will therefore replace property (ii) by a set of invariants that hold during the simulation, and which imply property (ii) when the simulation is completed and channels are empty. In the remainder of this section, we will formulate and prove this set of invariants. In Subsection 5.1 we formulate and prove them for our version of the underlying Time Warp algorithm. Then, in Subsection 5.2, we will extend them to our migration protocol.

Notation. We say a message is of the form $v \longrightarrow v'$ if it is sent from v and destined for v'. We write a timestamp ordered sequence of messages $m_0 \cdot m_1 \cdots m_n$, where we let m_0 be the oldest message in the sequence, and m_n the most recent. We use $+\!\!+$ to denote concatenation of sequences. We let $\mathrm{In}^{v}(v')$ be the sequence of processed diffusion messages of form $v \longrightarrow v'$ in v'.history, and let $\mathrm{Out}^{v'}(v)$ be the sequence of sent diffusion messages of form $v \longrightarrow v'$ in v.history. In particular, no anti-messages occur in the histories. The time of the earliest anti-message affecting a voxel v (aggregated or local), in a channel *chan*, is denoted *chan*.min(v). If there is no such anti-message, *chan*.min(v) is ∞.

5.1 Correctness for Time Warp

Intra-LP Consistency. For all pairs v, v' of voxel, that reside on the same LP, the incoming and outgoing diffusion histories are always consistent.

$$\mathrm{In}^{v}(v') = \mathrm{Out}^{v'}(v). \tag{IV1}$$

Proof sketch of (IV1). Since each LP processes messages in timestamp order, all diffusions from v to v' are added in timestamp order. At each processing of some event with a timestamp t, both $\mathrm{In}^{v}(v')$ and $\mathrm{Out}^{v'}(v)$ will be extended with the same diffusion. A local rollback caused by a straggler or received anti-message will, by rule (2) of the rollback operation, remove all diffusions from both $\mathrm{In}^{v}(v')$ and $\mathrm{Out}^{v'}(v)$ whose timestamps are greater than some common time t, thereby preserving (IV1). □

Inter-LP Consistency. In the case where the two voxels reside on different LPs, messages reside in a channel *chan* connecting the two LPs, before being received. Thus we must characterize the relationship between in- and out-histories at voxels and the messages in the corresponding channels. To this end, we define the *effect* with respect to voxels v and v' of the messages in a sequence *chan*, denoted $\mathrm{effect}_{v\to v'}(chan)$, as follows.

$$\underset{v\to v'}{\mathrm{effect}}(chan \cdot m) = \begin{cases} \underset{v\to v'}{\mathrm{effect}}(chan) \cdot m & \text{if } m = v \xrightarrow{t} v' \\ \mathrm{rm}_t(\underset{v\to v'}{\mathrm{effect}}(chan)) & \text{if } m = -t \text{ or } m = -t_{v'} \\ \underset{v\to v'}{\mathrm{effect}}(chan) & \text{otherwise} \end{cases} \tag{2}$$

where $\mathrm{rm}_t(chan)$ is obtained by removing all messages with timestamp $\geq t$ from *chan*. Intuitively, $\mathrm{effect}_{v\to v'}(chan)$ is the subsequence of diffusions from v to v' in *chan* that will not be reverted by a later anti-message in *chan*. This property can be derived from (2) and expressed as the following property of $\mathrm{effect}_{v\to v'}(chan)$.

$$\underset{v\to v'}{\mathrm{effect}}(m \cdot chan) = \begin{cases} m \cdot \underset{v\to v'}{\mathrm{effect}}(chan) & \text{if } m = v \xrightarrow{t} v' \text{ and} \\ & (\nexists m' \in chan. \exists t' \geq t. \\ & \quad (m' = -t' \text{ or } m' = -t'_{v'})) \\ \underset{v\to v'}{\mathrm{effect}}(chan) & \text{otherwise} \end{cases} \tag{3}$$

Let v and v' be two neighboring voxels on two different LPs, communicating through a channel *chan*, and let $t = chan.\mathrm{min}(v')$. Then the sequences of diffusions from v to v' in the two voxel histories are related by the following invariant.

$$\mathrm{rm}_t(\mathrm{In}^{v}(v')) +\!\!+ \underset{v\to v'}{\mathrm{effect}}(chan) = \mathrm{Out}^{v'}(v) \tag{IV2}$$

Proof sketch of (IV2). We establish the invariant by induction over the length of a simulation run. Initially, $\mathrm{In}^{v}(v')$ and $\mathrm{Out}^{v'}(v)$ and the channel are empty, and the invariant holds trivially. Assume non-empty $\mathrm{In}^{v}(v')$ and $\mathrm{Out}^{v'}(v)$, and a state of the channel *chan* such that Invariant (IV2) holds. We will let $\mathrm{In}'^{v}(v')$, $\mathrm{Out}'^{v'}(v)$, and *chan'* be their states after the performed action. Let $t = chan.\mathrm{min}(v')$. Line numbers refer to Algorithm 2. We get the following cases depending on the performed action.

- v's LP sends a diffusion m of form $v \xrightarrow{t} v'$ to v''s LP.
  Then $\mathrm{In}'^{v}(v') = \mathrm{In}^{v}(v')$, and $\mathrm{Out}'^{v'}(v) = \mathrm{Out}^{v'}(v) \cdot m$, and $chan' = chan \cdot m$. We get
  $\mathrm{Out}'^{v'}(v) = \mathrm{Out}^{v'}(v) \cdot m =$ (by (IV2))
  $\mathrm{rm}_t(\mathrm{In}^{v}(v')) +\!\!+ \mathrm{effect}_{v\to v'}(chan) \cdot m =$ (by (2))
  $\mathrm{rm}_t(\mathrm{In}^{v}(v')) +\!\!+ \mathrm{effect}_{v\to v'}(chan \cdot m) =$
  $\mathrm{rm}_t(\mathrm{In}'^{v}(v')) +\!\!+ \mathrm{effect}_{v\to v'}(chan \cdot m)$.
- v's LP sends an aggregated anti-message $-\hat{t}$ to v''s LP, while reverting diffusions in $\mathrm{Out}^{v'}(v)$ with a timestamp $\geq \hat{t}$, since $v \in \mathrm{LP}_{\mathrm{dst}}.bnd(\mathrm{LP}_{\mathrm{src}})$ (Condition 4 in the the calculation of RBT in the rollback operation in Section 3). Let $t' = \min(t, \hat{t})$. Then $t' = chan'.\mathrm{min}(v')$.
  We have $\mathrm{Out}'^{v'}(v) = \mathrm{rm}_{\hat{t}}(\mathrm{Out}^{v'}(v)) =$ (by (IV2))
  $\mathrm{rm}_{\hat{t}}(\mathrm{rm}_t(\mathrm{In}^{v}(v'))) +\!\!+ \mathrm{rm}_{\hat{t}}(\mathrm{effect}_{v\to v'}(chan)) =$ (by (2))
  $\mathrm{rm}_{t'}(\mathrm{In}^{v}(v')) +\!\!+ \mathrm{effect}_{v\to v'}(chan \cdot -\hat{t}) =$
  $\mathrm{rm}_{t'}(\mathrm{In}'^{v}(v')) +\!\!+ \mathrm{effect}_{v\to v'}(chan')$.

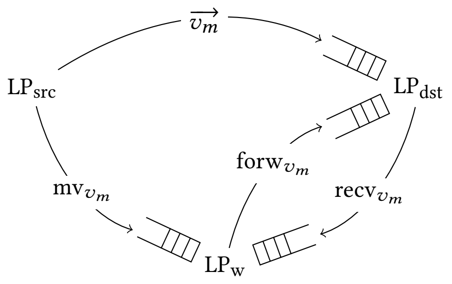

Figure 1: Schematic of LPs involved in the migration of voxel v_m, and the types of messages, $\overrightarrow{v_m}$, mv_{v_m}, recv_{v_m} and forw_{v_m}, being involved in the migration.

- v'''s LP receives a diffusion m of form $v \xrightarrow{\hat{t}} v'$ from v's LP (line 51). Details are analogous and omitted.
- v'''s LP receives an anti-message of form $-\hat{t}$ or $-\hat{t}_{v'}$ from v's LP, reverting diffusions in $\text{In}^v(v')$ with a timestamp $\geq \hat{t}$ (lines 42 and 47). Again, details are analogous.

□

5.2 Correctness for the Migration Protocol

In this section, we show that the algorithm maintains consistency when extended with migrations of voxels between two LPs. The migration protocol introduces several types of control messages, viz., $\overrightarrow{v_m}$, mv_{v_m}, recv_{v_m} and forw_m. Since the migration protocol does not modify the histories of voxels directly, effect is defined to ignore such control messages c, i.e., $\text{effect}_{v\to v'}(chan \cdot c) = \text{effect}_{v\to v'}(chan)$.

An aggregated anti-message $-t$ changes in meaning when a voxel v_m is migrated from LP$_{\text{src}}$ to LP$_{\text{dst}}$. If $-t$ is sent from LP$_{\text{src}}$, after $\overrightarrow{v_m}$, then it does not affect messages sent from v_m to v on LP$_{\text{dst}}$, which is achieved by the removal of v_m from *bnd* in the algorithms (line 9 in Algorithm 1 and line 8 in Algorithm 2). To capture this, we define an operator $\text{cut}_{m'}$, which simply truncates a sequence after the first occurrence of m', by

$$\text{cut}_{m'}(m \cdot chan) = \begin{cases} m & \text{if } m = m' \\ m \cdot \text{cut}_{m'}(chan) & \text{otherwise} \end{cases}$$

5.2.1 Consistency between sender and receiver of migrated voxel. Now, we are ready to define the invariants. Let v_m and v be two neighboring voxels, where v_m, previously located on LP$_{\text{src}}$, is being migrated to the same LP as v, LP$_{\text{dst}}$. We have to show the consistency of messages in both directions relative the migration direction. After the transmission of $\overrightarrow{v_m}$, and before $\overrightarrow{v_m}$ has been received, we have the following two invariants.

- For diffusions from v_m to v:
  Let $t = (\text{cut}_{\overrightarrow{v_m}}(chan_{\text{src}\to\text{dst}})).\min(v)$. Then we have

$$\text{rm}_t(\text{In}^{v_m}(v)) \mathbin{+\!\!+} \underset{v_m\to v}{\text{effect}}\left(\text{cut}_{\overrightarrow{v_m}}(chan_{\text{src}\to\text{dst}})\right) = \text{Out}^v(v_m). \quad \text{(IV3)}$$

  This invariant is analogous to (IV2), but considers only the part of $chan_{\text{src}\to\text{dst}}$ that precedes $\overrightarrow{v_m}$.

- For diffusions from v to v_m:
  Let $t = (log_{\text{dst}\to\text{src}}(v_m) \mathbin{+\!\!+} chan_{\text{dst}\to\text{src}}).\min(v_m)$. Then

$$\text{rm}_t(\text{In}^v(v_m)) \mathbin{+\!\!+} \underset{v\to v_m}{\text{effect}}(log_{\text{dst}\to\text{src}}(v_m) \mathbin{+\!\!+} chan_{\text{dst}\to\text{src}}) = \text{Out}^{v_m}(v). \quad \text{(IV4)}$$

  This invariant extends (IV2) by considering that diffusions from v to $\overrightarrow{v_m}$ are collected in $log_{\text{dst}\to\text{src}}(v_m)$.

We note that v_m neither can send nor receive any messages while in transit, thus the proof of Invariant (IV3) is only concerned with v receiving a message, and the proof of Invariant (IV4) is only concerned with v sending a message.

5.2.2 Correctness of receipt of migrated voxel. During migration of a voxel v_m, consistency is defined by Invariants (IV3) and (IV4). When the migrated voxel v_m has been received and processed by LP$_{\text{dst}}$, these invariants are replaced by Invariant (IV1), stating that v_m is locally consistent with all its neighbors $v \in \text{LP}_{\text{dst}}$. Again, we omit the details.

5.2.3 Consistency between voxel being migrated and neighboring voxels on other domains. Let v_m, v be two neighboring voxels, where v_m is being migrated from LP$_{\text{src}}$ to LP$_{\text{dst}}$ and v is located on a third LP, LP$_{\text{w}}$, as depicted in Figure 1. At the transmission of $\overrightarrow{v_m}$, a message mv_{v_m} is sent to all neighboring LPs except the receiver of $\overrightarrow{v_m}$ (line 10 in Algorithm 1). At the receipt of $\overrightarrow{v_m}$, a message recv_{v_m} is sent to all neighbors, except the sender of $\overrightarrow{v_m}$ (line 11 in Algorithm 2). Thus, in the following, we have to take into account if the control messages mv_{v_m} and recv_{v_m} have been received or not, since they affect the protocol. We split the argument into the two possible directions in which messages may be sent, viz. $v_m \longrightarrow v$ and $v \longrightarrow v_m$.

Case $v_m \longrightarrow v$: We let $-t$ be the earliest anti-message preceding the message mv_{v_m} in $chan_{\text{src}\to w}$ or succeeding the message recv_{v_m} in $chan_{\text{dst}\to w}$. The cut operator reflects that anti-messages change in meaning upon receipt of mv_{v_m} (line 31 in algorithm 2). After the transmission of $\overrightarrow{v_m}$, but before mv_{v_m} has been received by LP$_{\text{w}}$, we have

$$\text{rm}_t(\text{In}^{v_m}(v)) \mathbin{+\!\!+} \underset{v_m\to v}{\text{effect}}(\text{cut}_{\text{mv}_{v_m}}(chan_{\text{src}\to w}) \mathbin{+\!\!+} chan_{\text{dst}\to w}) = \text{Out}^v(v_m). \quad \text{(IV5)}$$

Intuitively, this invariant extends (IV2) by considering that diffusions from v_m to v are found preceding mv_{v_m} in $chan_{\text{src}\to w}$. After the reception of v_m by LP$_{\text{dst}}$, they are then transmitted over $chan_{\text{dst}\to w}$. We also note, that after the reception of mv_{v_m} by LP$_{\text{w}}$, any outstanding diffusions from v_m to v are in $chan_{\text{dst}\to w}$, obeying the corresponding instance of (IV2).

Case $v \longrightarrow v_m$: In this direction, we have two cases, depending on whether LP$_{\text{w}}$ has yet observed the migration or not:

- mv_{v_m} has been received by LP$_{\text{w}}$.
  We let t be the time of the earliest anti-message in $chan_{w\to\text{dst}}$ following forw_{v_m}; if forw_{v_m} is not in $chan_{w\to\text{dst}}$, let t be the time of the earliest anti-message in $chan_{w\to\text{dst}}$. Then

$$\text{rm}_t(\text{In}^v(v_m)) \mathbin{+\!\!+} \underset{v\to v_m}{\text{effect}}(chan_{w\to\text{dst}}) = \text{Out}^v(v_m) \quad \text{(IV6)}$$

  This invariant is essentially the same as (IV2).

- mv_{v_m} has not been received by $\mathrm{LP_w}$.
  We let t be the time of the earliest anti-message in $log_{\mathrm{w}\to\mathrm{src}}(v_m)$ and $chan_{\mathrm{w}\to\mathrm{src}}$.

$$\mathrm{rm}_t(\mathrm{In}^v(v_m)) \mathbin{+\!\!+} \underset{v\to v_m}{\mathrm{effect}}(log_{\mathrm{w}\to\mathrm{src}}(v_m) \mathbin{+\!\!+} chan_{\mathrm{w}\to\mathrm{src}}) = \mathrm{Out}^v(v_m) \tag{IV7}$$

  This invariant reflects that messages from v to v_m are stored in $log_{\mathrm{w}\to\mathrm{src}}(v_m)$ until mv_{v_m} is received by $\mathrm{LP_w}$.

Invariants (IV6) and (IV7) are also established by induction over the steps of the algorithm. Details are omitted for space reasons.

6 LOAD BALANCING

In this section, we describe the load and communication metrics, and the load balancing algorithm, that we use for the evaluation of the load migration protocol in the next section.

Load balancing can be seen as an online local rebalancing technique for data partitioning. In our case, the simulator is given an initial partition of the model, generated offline by the METIS library [20]. Due to factors not available to the offline partitioning, such as a dynamic or variable load, the partitions may need continuous rebalancing to ensure that the load is balanced.

Data partitioning and rebalancing algorithms typically try to optimize for a balanced workload and minimize communication. For the evaluation of the migration protocol we selected a load balancing mechanism consisting of two parts: a load metric based on the number of rollbacks caused by stragglers received at a particular voxel, which is used to initiate a load balancing locally, and a communication minimizing step that selects some voxel in the vicinity whose migration minimizes communication.

More specifically, we define the inverse voxel load for a period of wall-clock time Δt and voxel v as

$$L_v = \frac{\Delta t}{\Delta r_v},$$

where r_v is the total number of rollbacks at time t, including secondary, that are incurred due to stragglers received at v. The measure could be seen as the mean distance, in wall clock time, between two consecutive rollbacks. If the *inverse rate of rollbacks*, i.e., the limit of L_v as Δt goes to 0, surpasses a threshold, R, then a request for load balancing is sent to the LP $\mathrm{LP_{src}}$ that sent the last straggler to v (a simplification). Upon receipt of the request, the voxel neighbor to v with the highest *gain*, i.e., the highest communication load, is selected for migration. We define the gain of a voxel v on LP LP_i relative a neighboring LP LP_j as

$$g_{ij}(v) = \frac{||\{E(v,v') \mid v' \in \mathrm{LP}_j\}||}{||\{E(v,v') \mid v' \in \mathrm{LP}_i\}||},$$

where $E(v,v')$ denotes an edge, i.e., a communication channel, between v and v'. Thus, the gain is defined as the *external degree* towards LP_j over the *internal degree*. The gain as defined above describes how well connected a voxel is to its domain. To limit bad migrations that do not improve the load or communication balance, we introduce a *gain threshold*, G, so that only voxels with a gain superior to G may be migrated.

The successful migration of one or more of the neighboring voxels to v, based on the load metric defined above, would serve two purposes. First, migrating voxels balances the amount of work between two LPs, and potentially reduces the number of future rollbacks. Second, a high rate of rollbacks at the boundary also means a high rate of local communication between two or more voxels located on different LPs. The migration thus also reduces inter-LP communication, if the voxels to migrate are chosen wisely, e.g., by selecting to migrate the voxel that minimizes the communication.

7 EVALUATION

In this section, we evaluate the performance of the migration protocol coupled with a load measure and a refinement technique. We try to specifically understand the performance of the migration protocol and its shortcomings, as it can be used together with many different load measures and refinement techniques.

We look at the following questions:

- *What is the overhead of the migration protocol?* (Section 7.4)
- *How to tune the threshold for the load measure?* (Section 7.5)
- *How well does the protocol, together with a load-balancing algorithm, adapt to a changing load?* (Section 7.6)
- *How does load balancing perform compared to optimism control?* (Section 7.7)

7.1 Algorithms

For the evaluation of the load-balancing algorithm, we compare the results of the load-balancing parallel NSM algorithm to a corresponding sequential NSM implementation, of the URDME framework [11], and a variation of our protocol without load balancing, but with sophisticated optimism control: for this we use techniques employed in the so-called Refined PNSM algorithm of [23], and also include optimizations that are possible when the load does not migrate. One such optimization is that there is only a single rollback history per LP. In contrast, the load-balancing parallel NSM algorithm of this paper has arranged the data so that each voxel easily can be moved from one LP to another; in particular does each voxel has its own rollback history. We use the following names for the rest of the evaluation:

NSM The sequential NSM implementation.
R-PNSM The Refined PNSM implementation with optimism control.
PNSM-LB The load-balancing parallel NSM implementation.

7.2 Benchmarks

To evaluate the algorithm, we use two types of benchmarks:

- Simulation of the Min-protein system in a three-dimensional unstructured (i.e., an irregular mesh) model of the *E. coli* bacterium [14]. The Min-protein has a central role in the cell division of the bacterium, where the oscillation of the min-proteins help determine the position of the septum, the new cell wall. The model consists of 5 species, and the dynamics are described by 5 reactions. It is complicated by some reactions only taking place on the membrane of the cell. We present data for two different stages of cell division, short and long, comprising 1500 and 2700 voxels, respectively. The two models are denoted mincde[s] and mincde[L], respectively. Due to the oscillation of the proteins, the load is highly dynamic. Since the number of voxels are small, the amount of available disjoint level parallelism is limited.

- Simulation of a reversible isomerization, where two species randomly are transformed into each other, in a sphere model. The load is well balanced, and the model comprises ∼ 13000 voxels. The model is denoted sphere.

7.3 Experimental Setup

The experiments are run on a 4 socket Intel Sandy Bridge E5−4650 machine. Each processor has 8 cores and 20 MB L3-cache. Hyper-threading was turned off, and threads were pinned to cores. The operating system of the machine is Linux 4.9.0, and the binaries were compiled using GCC version 6.3.0. The models were initially partitioned using the multilevel k-way partitioning method from the Metis library [20].

7.4 Overhead of Migration Protocol

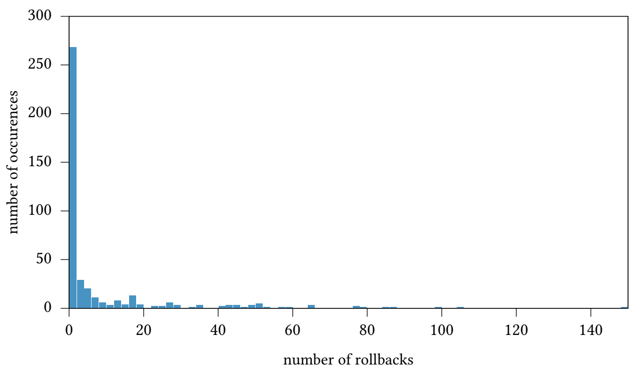

Figure 2: Frequency of the number of rollbacks caused by a single migration on the mincde[s]**, 3 threads. On average, there is 1.3 rollbacks per migration.**

The migration of a voxel may cause rollbacks, which in turn may cause anti-messages. We would like to understand how much overhead the migration of a single voxel represent. Therefore, we trace the rollbacks and anti-messages that are caused by a migration of a voxel. For this, we repeatedly run the mincde[s] benchmark on 3 threads, only letting a single migration occur during the simulation run. In Figure 2, we see the frequency of the number of rollbacks, including secondary rollbacks and rollbacks caused by anti-messages that were caused by the migration in the first place. We see that the most of the time, very few rollbacks occur due to the migration, with a mean of 1.3 rollbacks. Hence, the secondary effects of the protocol are marginal.

7.5 Tuning of the Rate and Gain Parameters

In this section, we tune the inverse rollback rate parameter R for locally initiating a migration of a voxel and the gain threshold parameter G, as described in Section 6.

In Figures 3 and 4, results for tuning of the inverse rollback rate parameter R and the gain threshold parameter G of the migration mechanisms is presented. In the figures, each line represents one value of the gain threshold. R is varied from 0 (no migration) to 80, and the gain threshold is varied from 0.5 to 0.7. The left hand sub-figure within each figure show the speedup relative not using migration, and the right hand sub-figure show the total number of rollbacks occurring during the simulation. In Figure 3 the tuning is done on the mincde[s] benchmark, 3 threads. The maximum speedup over no migration is reached with a rollback rate threshold of 40, and a gain threshold of 0.6–0.7. In Figure 4, we get similar results, but a low gain threshold results in worse performance. Initially, increasing the parameter R increases performance, but beyond a value of 20–30, performance tend to become worse again. I.e., when reducing the threshold for performing a rollback, more time tend to be spent on migrating voxels, without any positive effects on performance.

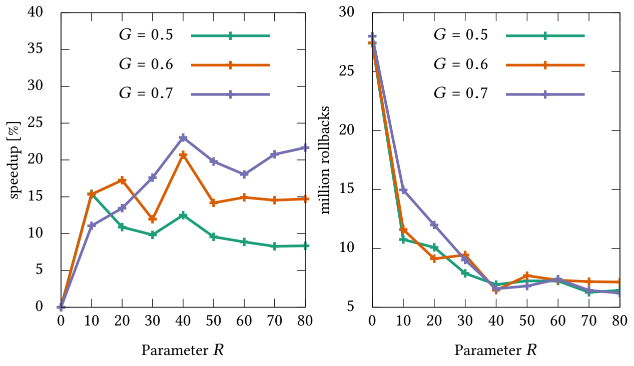

Figure 3: Inverse rollback rate parameter R impact on the number of rollbacks and speedup. Evaluated on the mincde[s] **benchmark, 3 threads.**

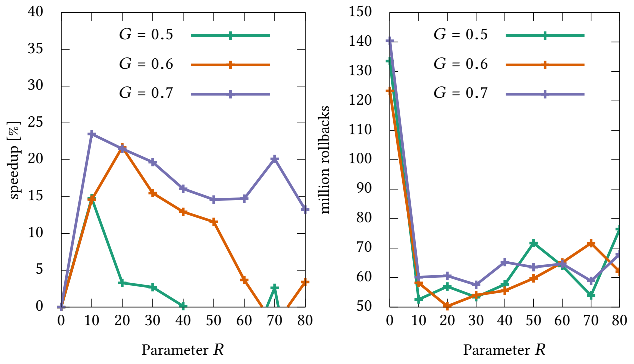

Figure 4: Inverse rollback rate parameter R impact on the number of rollbacks and speedup. Evaluated on the mincde[s] **benchmark, 16 threads.**

In general, the run time improvement over not using migration is modest, peaking at about 25% improvement with a threshold parameter of 10–50. However, the number of rollbacks is greatly reduced, by a factor of 2–4.

7.6 Effective Utilization

In this section, we look at the effect load balancing has on the amount of time individual LPs spend on rollback overhead, e.g., processing rollbacks and redoing the rolled back time interval (roughly

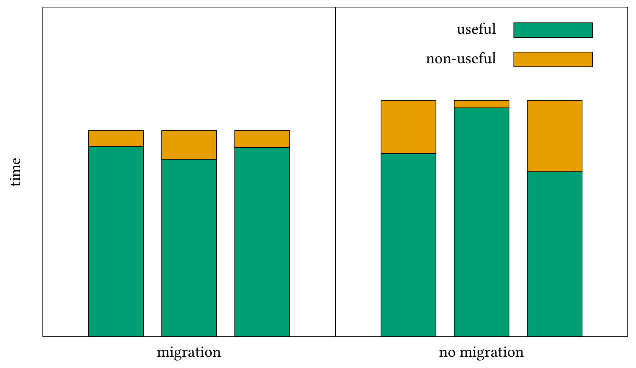

Figure 5: Instrumentation data for each thread of the R-PNSM and PNSM-LB algorithms on the mincde[s] **benchmark, 3 threads.**

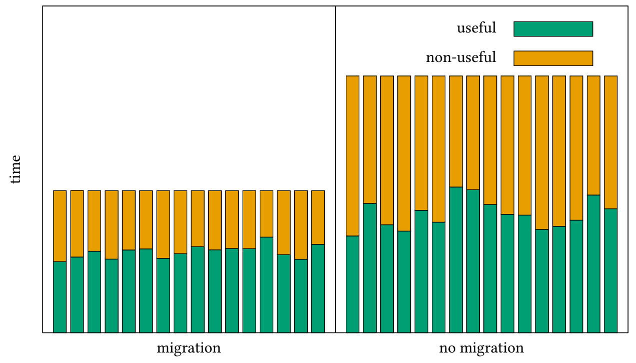

Figure 6: Instrumentation data for each thread of the R-PNSM and PNSM-LB algorithms on the mincde[s] **benchmark, 16 threads.**

estimated to take the same amount of time as the rollbacks themselves) and anti-messages, denoted *non-useful* work. In Figure 5, we see a simulation run of the mincde[s] benchmark on 3 threads. Each bar represents the time allocation of one individual LP, divided into useful and non-useful work. We see that one thread has a surplus of work, spending little time on non-useful work. At the same time, it incurs a high cost in rollbacks to its two neighbors. In Figure 6, we see the same benchmark on 16 threads. The variance of the amount of time spent on non-useful work is more than halved when using migration. However, as can be seen by the amount of non-useful work even with load-balancing, we can conclude that the load imbalance is still momentarily high during the simulation.

To illustrate in more detail, we try to visualize the rollbacks and the behavior of the load balancing mechanism during a simulation. In Figures 7 and 8, a timeline of the simulation of the mincde[s] benchmark with the PNSM-LB algorithm on 3 threads is depicted. For each thread, one individual timeline is depicted, horizontally. In each timeline, the total population, the number of voxels, and the number of rollbacks is shown. In Figure 7, no migration is used. We see how the population varies with time, and how the number of rollbacks increases when the population difference between LPs is big. We also see how the middle LP always has more work to do

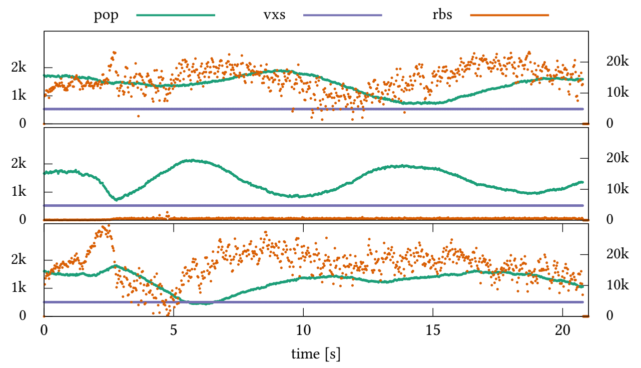

Figure 7: Time series of population, number of voxels and rollbacks per LP when running the mincde[s] **benchmark on 3 cores without migration.**

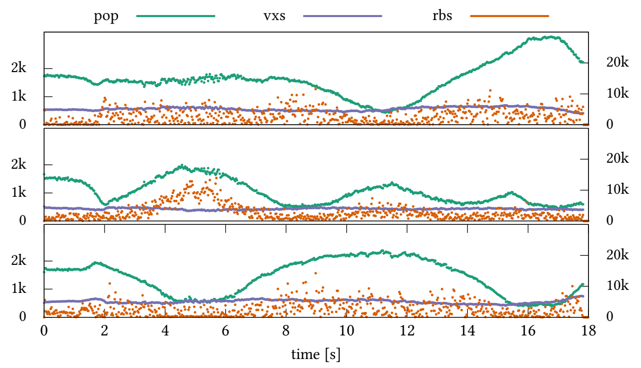

Figure 8: Time series of population, number of voxels and rollbacks per LP when running the mincde[s] **benchmark on 3 cores with migration.**

than its two neighbors, and suffers from practically no rollbacks. In Figure 8, migration is used. We see that there are continuously much fewer rollbacks that are more evenly distributed over the LPs. The population still varies, but in general the middle LP has a lower population than in figure Figure 7, i.e., the load is better balanced.

7.7 Performance

In this section, we compare the scaling of the PNSM-LB algorithm with and without load migration, and also compare it to the R-PNSM algorithm.

First, in Figure 9, we see the speedup over NSM for the parallel algorithms, evaluated on the sphere[s] benchmark. The benchmark is uniform in load, and we see as expected that the R-PNSM algorithm, which does not perform any load balancing, perform much better. In Figure 10a, we see the speedup over NSM for the parallel algorithms, evaluated on the mincde[s] benchmark. We see that for up to 8 cores, the PNSM-LB with migration has the best scaling. Only for 16 cores, the R-PNSM algorithm has better performance. Similar results are also found in Figures 9 and 10b. We believe this to be due to a suboptimal load balancing algorithm, that does not make the best decision on where to move and when to move the voxels.

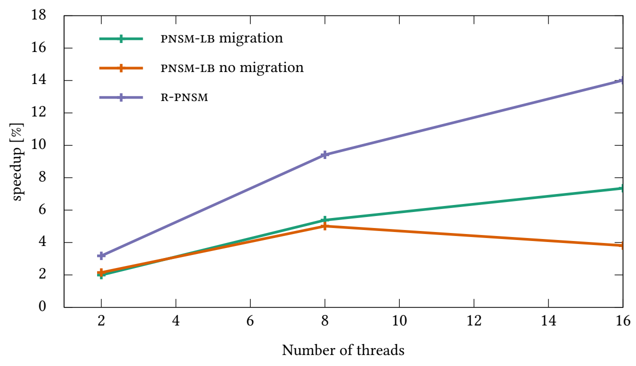

Figure 9: scaling over sequential nsm on the sphere[small] benchmark.

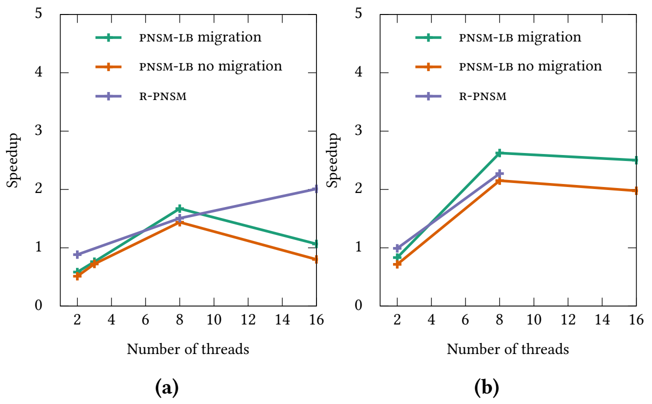

Figure 10: Scaling over sequential NSM on the mincde[s] **(a) and the** mincde[l] **(b) benchmarks.**

In all cases, PNSM-LB with load migration has better scaling than the PNSM-LB without load balancing.

8 CONCLUSIONS

We have described a load migration protocol aimed at fine-grained dynamic load balancing. The protocol works across aggregated anti-messages, a technique used to reduce inter-LP communication. The protocol has been evaluated together with a local load metric and load-balancing algorithm. The load balancing is shown to reduce the number of incurred rollbacks drastically, by a factor of 2–4, and the speedup is improved by up to 25%, to a large extent due to a reduced number of rollbacks. The balancing scheme achieves better speedup than a similar protocol without load balancing but with fine-grained mechanism for optimism control. Possible future work would be to improve the strategy for which load to migrate when, and the choice of metrics for load and communication.

REFERENCES

[1] Hervé Avril and Carl Tropper. 1996. The Dynamic Load Balancing of Clustered Time Warp for Logic Simulation. *SIGSIM Simul. Dig.* 26, 1 (July 1996), 20–27.
[2] Pavol Bauer, Jonatan Lindén, Stefan Engblom, and Bengt Jonsson. 2015. Efficient Inter-Process Synchronization for Parallel Discrete Event Simulation on Multicores. In *Proc. SIGSIM PADS.* ACM, 183–194.
[3] Azzedine Boukerche and Sajal K. Das. 1997. Dynamic load balancing strategies for conservative parallel simulations. *SIGSIM Simul. Dig.* 27, 1 (June 1997), 20–28.
[4] Christopher Burdorf and Jed Marti. 1993. Load Balancing Strategies for Time Warp on Multi-User Workstations. *Comput. J.* 36, 2 (1993), 168–176.
[5] Christopher D. Carothers and Richard M. Fujimoto. 1996. Background Execution of Time Warp Programs. *SIGSIM Simul. Dig.* 26, 1 (July 1996), 12–19.
[6] Li-li Chen, Ya-Shuai Lu, Yi-Ping Yao, Shao-Liang Peng, and Ling-Da Wu. 2011. A Well-Balanced Time Warp System on Multi-Core Environments. In *Proc. 25th Workshop on Parallel and Distributed Simulation.* IEEE, 1–9.
[7] M. Choe and C. Tropper. 1999. On learning algorithms and balancing loads in Time Warp. *Proc. 13th Workshop on Parallel and Distributed Simulation*, 101–108.
[8] Samir R. Das. 2000. Adaptive Protocols for Parallel Discrete Event Simulation. *J. Oper. Res. Soc.* 51, 4 (2000), 385–394.
[9] Ewa Deelman and Boleslaw K. Szymanski. 1997. Breadth-First Rollback in Spatially Explicit Simulations. In *Proc. 11th Workshop on Parallel and Distributed Simulation.* IEEE, 124–131.
[10] Ewa Deelman and Boleslaw K. Szymanski. 1998. Dynamic Load Balancing in Parallel Discrete Event Simulation for Spatially Explicit Problems. *SIGSIM Simul. Dig.* 28, 1 (July 1998), 46–53.
[11] Brian Drawert, Stefan Engblom, and Andreas Hellander. 2012. URDME: a modular framework for stochastic simulation of reaction-transport processes in complex geometries. *BMC Syst. Biol.* 6 (2012), 76.
[12] Khalil El-Khatib and Carl Tropper. 1999. On Metrics for the Dynamic Load Balancing of Optimistic Simulations. In *Proc. 32nd Annual Hawaii International Conference on System Sciences*, Vol. 8. IEEE.
[13] J. Elf and M. Ehrenberg. 2004. Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. *Syst. Biol.* 1, 2 (2004), 230–236.
[14] David Fange and Johan Elf. 2006. Noise-Induced Min Phenotypes in E. coli. *PLOS Comput. Biol.* 2, 6 (June 2006), e80.
[15] Richard M. Fujimoto. 1990. Parallel Discrete Event Simulation. *Commun. ACM.* 33, 10 (1990), 30–53.
[16] Crispin W. Gardiner. 2007. *Handbook of stochastic methods for physics, chemistry, and the natural sciences.* Springer.
[17] David W. Glazer and Carl Tropper. 1993. On Process Migration and Load Balancing in Time Warp. *IEEE Trans. Parallel Distrib. Syst.* 4, 3 (1993), 318–327.
[18] S. Jafer, Q. Liu, and G.A. Wainer. 2013. Synchronization methods in parallel and distributed discrete-event simulation. *Simul. Model. Pract. Th.* 30 (2013), 54–73.
[19] David R. Jefferson. 1985. Virtual Time. *ACM Trans. Program. Lang. Syst.* 7, 3 (1985), 404–425.
[20] G. Karypis and V. Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (Jan. 1998), 359–392.
[21] Jonathan I. Leivent and Ronald J. Watro. 1993. Mathematical Foundations of Time Warp Systems. *ACM Trans. Program. Lang. Syst.* 15, 5 (1993), 771–794.
[22] Zhongwei Lin and Yiping Yao. 2015. Load Balancing for Parallel Discrete Event Simulation of Stochastic Reaction and Diffusion. In *Int'l Conference on Smart City/SocialCom/SustainCom.* IEEE, 609–614.
[23] Jonatan Lindén, Pavol Bauer, Stefan Engblom, and Bengt Jonsson. 2017. Exposing Inter-Process Information for Efficient Parallel Discrete Event Simulation of Spatial Stochastic Systems. In *Proc. SIGSIM PADS.* ACM, 53–64.
[24] Boris D. Lubachevsky. 1988. Efficient parallel simulations of dynamic Ising spin systems. *J. Comput. Phys.* 75, 1 (1988), 103–122.
[25] Jayadev Misra. 1986. Distributed Discrete-Event Simulation. *ACM Comput. Surv.* 18, 1 (1986), 39–65.
[26] Alessandro Pellegrini, Roberto Vitali, Sebastiano Peluso, and Francesco Quaglia. 2012. Transparent and Efficient Shared-State Management for Optimistic Simulations on Multi-core Machines. In *Proc. MASCOTS.* IEEE, 134–141.
[27] Patrick Peschlow, Tobias Honecker, and Peter Martini. 2007. A Flexible Dynamic Partitioning Algorithm for Optimistic Distributed Simulation. In *Proc. 21st Workshop on Principles of Advanced and Distributed Simulation.* IEEE, 219–228.
[28] Peter L. Reiher and David R. Jefferson. 1990. Virtual Time Based Dynamic Load Management in the Time Warp Operating System. *Trans. Soc. Comput. Simul. Int.* 7, 9 (1990), 103–111.
[29] Falguni Sarkar and Sajal K. Das. 1997. Design and implementation of dynamic load balancing algorithms for rollback reduction in optimistic PDES. In *Proc. MASCOTS.* IEEE, 26–31.
[30] Rolf Schlagenhaft, Martin Ruhwandl, Christian Sporrer, and Herbert Bauer. 1995. Dynamic Load Balancing of a Multi-cluster Simulator on a Network of Workstations. *SIGSIM Simul. Dig.* 25, 1 (July 1995), 175–180.
[31] Sunil Thulasidasan, Shiva Prasad Kasiviswanathan, Stephan Eidenbenz, and Phillip Romero. 2010. Explicit Spatial Scattering for Load Balancing in Conservatively Synchronized Parallel Discrete Event Simulations. In *Proc. 24th Workshop on Principles of Advanced and Distributed Simulation.* IEEE, 150–157.
[32] Jingjing Wang, Deepak Jagtap, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. 2014. Parallel Discrete Event Simulation for Multi-Core Systems: Analysis and Optimization. *IEEE Trans. Par. Distr. Syst.* 25, 6 (2014), 1574–1584.
[33] Linda F. Wilson and Wei Shen. 1998. Experiments in Load Migration and Dynamic Load Balancing in SPEEDES. In *Proc. 30th Winter Simulation Conf.* IEEE, 483–490.
[34] Yadong Xu, Wentong Cai, David Eckhoff, Suraj Nair, and Alois Knoll. 2017. A Graph Partitioning Algorithm for Parallel Agent-Based Road Traffic Simulation. In *Proc. SIGSIM PADS.* ACM, 209–219.