NTW-MT: a Multi-threaded Simulator for Reaction Diffusion Simulations in NEURON

Conference Paper · January 2015

CITATIONS
7

READS
74

6 authors, including:

Zhongwei Lin
National University of Defense Technology
5 PUBLICATIONS **11** CITATIONS
SEE PROFILE

Carl Tropper
McGill University
84 PUBLICATIONS **980** CITATIONS
SEE PROFILE

Mohammad Nazrul Ishlam Patoary
McGill University
9 PUBLICATIONS **18** CITATIONS
SEE PROFILE

Robert A Mcdougal
Yale University
21 PUBLICATIONS **112** CITATIONS
SEE PROFILE

NTW-MT: a Multi-threaded Simulator for Reaction Diffusion Simulations in NEURON

Zhongwei Lin*
National University of Defense Technology
Changsha, Hunan, China
zwlin@nudt.edu.cn

Carl Tropper
School of Computer Science
McGill University
Montreal, Quebec, Canada
carl@cs.mcgill.ca

Mohammand Nazrul Ishlam Patoary
School of Computer Science
McGill University
Montreal, Quebec, Canada
nazrul.eis@gmail.com

Robert A. McDougal
Department of Neurobiology
Yale University
333 Cedar St. New Haven, Connecticut, USA
robert.mcdougal@yale.edu

William W. Lytton
SUNY Downstate Medical Center
Brooklyn, NY, 11203, USA
blytton@downstate.edu

Michael L. Hines
Department of Neurobiology
Yale University
New Haven, Connecticut, USA
michael.hines@yale.edu

ABSTRACT

This paper describes a parallel discrete event simulator, Neuron Time Warp-Multi Thread (NTW-MT), developed for the simulation of reaction diffusion models of neurons. The simulator was developed as part of the NEURON project and is intended to be included in NEURON. It relies upon a stochastic discrete event model developed for chemical reactions. NTW-MT is optimistic and thread-based, in which communication latency among threads within the same process is minimized by pointers. We investigate the performance of NTW-MT on a reaction-diffusion model for the transmission of calcium waves in a neuron. Calcium plays a fundamental role in the second messenger system of a neuron. However, the mechanism by which calcium waves are transmitted is not entirely understood. Stochastic models are more realistic than deterministic models for small populations of ions such as those found in apical dendrites. To be more precise, we simulate a stochastic discrete event model for calcium wave propagation on an unbranched apical dendrite of a hippocampal pyramidal neuron. We examine the performance of NTW-MT on this calcium wave model and compare it to the performance of (1) a process based optimistic simulator and (2) a threaded simulator which uses a single priority (SQ) queue for each thread. Our multi-threaded simulator is shown to achieve superior performance to these simulators.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*parallel, distributed, discrete event*

General Terms

Algorithms, Design, Performance

Keywords

Stochastic Neuronal Simulation, PDES, Multiple Thread

1. INTRODUCTION

The human brain may be viewed as a sparsely connected network of neurons [3] containing approximately 10^{14} neurons. Each neuron receives inputs from thousands of dendrites and sends outputs to thousands of neurons by means of its axon.

The membrane of a neuron contains channels which selectively control the flow of ions (primarily sodium, potassium, and calcium) through the membrane. Movements of ions through these channels is by (1) diffusion from a higher concentration of ions or (2) by pumps which are dependent on the voltage drop across the membrane. Electrical models for neurons [14] can be constructed using the well-known laws of electricity (Ohm, Kirchkoff, capacitance). However, these electrical models only provide a limited view of neuronal activity since there are ions (notably calcium) which

*This author is affiliated with State Key Laboratory of High Performance Computing at National University of Defense Technology.

SIGSIM-PADS'15, June 10–12, 2015, London, United Kingdom.

DOI: http://dx.doi.org/10.1145/2769458.2769459.

function as information messengers. In order to develop realistic models of a neuron, it is necessary to develop models which account for the movement and functioning of these messengers.

The combination of chemical reactions within a cell with the diffusion of ions through the membrane can be modelled as a reaction diffusion system and simulated by (parabolic) partial differential equations [3, 14]. However, a continuous model is not appropriate for a small number of molecules. Stochastic model is a far more realistic and accurate representation [23, 20] for this sort of situation.

It is well known that a system consisting of a collection of chemical reactions can be represented by a chemical master equation, the solution of which is a probability distribution of the chemical reactants in the system [23]. In general, it is very difficult to solve this equation. In [9] a Monte Carlo simulation algorithm called the Stochastic Simulation Algorithm (SSA) is described. Under the assumption that the molecules of the system are uniformly distributed, the algorithm simulates a single trajectory of the chemical system. Simulating a number of these trajectories then gives a picture of the system. The Next sub-volume Method (NSM) [8] is an extension of the SSA which incorporates the diffusion of molecules into the model. The NSM partitions space into cubes called sub-volumes, and it can be applied to PDES by representing sub-volumes as Logical Processes (LP) [24]. Diffusion of ions between neighboring sub-volumes is represented as events. NTW-MT relies on the NSM algorithm.

NEURON [4, 3] is a widely used simulator in the neuroscience community. It makes use of deterministic simulators for the reaction diffusion model [15] and for electrical models. We are collaborating with the NEURON group; our intention is to develop parallel discrete event simulators suitable for simulating the reaction diffusion models. It is intended that our simulators will be integrated into NEURON. We previously developed a process based simulator, NTW [18], which makes use of a multi-level queue. We verified and examined its performance on a Calcium buffer model and a predator prey [21] model. The queueing structure described in this paper and the one in [18] are outgrowths of the multi-level queue described in XTW [27].

The remainder of this paper is organized as follows. Section 2 describes the related work, section 3 is devoted to the architecture and algorithms of our simulator, section 4 describes our experimental results. The conclusion and future work are presented in section 5.

2. RELATED WORK

[6] points out that a conservative synchronization algorithm for parallel simulation will perform poorly because of the zero-lookahead property of the exponential distribution and the fact that a dependency graph of the reactions is likely to be highly connected and filled with loops. This indicates that an optimistic synchronization algorithm such as Time Warp is preferable. [25] presents an experimental analysis of several optimistic protocols for the parallel simulation of a reaction-diffusion system. [13] uses NSM in an optimistic simulation along with an adaptive time window. [12] compares the performance of spatial τ-leaping with that of NSM and Gillespie's Multi-particle Method (GMPM) in terms of speedup and accuracy.

There are two main schemes for the storage of the pending events for each thread, a global queue and separated (distributed) queue. In a global queue, a number of threads (Processing Elements, PE) share a single priority queue. This achieves load sharing at the expense of contention at the queue, e.g. threaded WARPED [16]. The main drawback of this scheme is too much contention on the global queue [5]. In a separated queue, each worker thread has its own priority queue and only processes events from this queue. However, there are still concurrent operations on the priority queue arising from other threads in the same process. In order to avoid locking the contents of individual LPs, each LP is mapped to only one specific thread. This may lead to a load imbalance, so it is necessary to balance the workload for threads and processes. [5] uses a separated queue and proposes a global scheduling mechanism to balance the workload between the threads in the same process. A global scheduling mechanism was employed for load balancing, decreasing the contention and improving performance. However, a global schedule still allows simultaneous processing of events at individual LPs, necessitating a locking mechanism. ROSS-MT [10] also employs a separated queue and uses an input queue to store the events sent from other threads. Contention on the input queue remains high. A hybrid scheme combines the above two schemes by creating several priority queues within one process and mapping a subset of the threads to a single queue, as described in [7].

3. ARCHITECTURE

We employ a separated priority queue scheme in our architecture. The simulator architecture is depicted in figure 1. One of the processes is a controller, exercising global control functions (GVT computation (Mattern's algorithm) and load balancing). The remaining processes are worker processes which process the events at the LPs which reside in the process. Each worker process contains a *communication thread* and several *processing threads*. All of the worker processes have the same number of threads.

The communication thread sends and receives messages for a process. Processing threads cannot send or receive messages. After initialization, the communication thread receives messages from shared memory, in which case the the message is from a *family process* residing in the same node (see section 3.3) or via MPI from remote nodes. Control messages are the first messages to be processed. LPs then schedule external events by placing them into the send buffer of the communication thread. To avoid contention for this buffer, it is partitioned into m segments, where m is the number of processing threads. The ith processing thread can write only into the ith segment. The communication thread scans the segments in the buffer and then sends out the messages. At present, the communication thread sends only one message per segment (a fairness policy).

LPs are partitioned into $m \times n$ subsets, where n is the number of worker processes and m is the number of processing threads. Each subset is mapped to a processing thread. Each processing thread includes a *LP List* which stores the LPs associated with the thread, a *Thread Event Queue* (TEQ), a *Thread Memory Allocator* which is involved in the Memory Management and a *Thread Function*. The Thread Function is responsible for processing the events, and is portrayed in figure 2.

An event has two timestamps, the receive time and the send time [11]. The events in the priority queue are sorted by their receive time. There are two types of events: *internal*

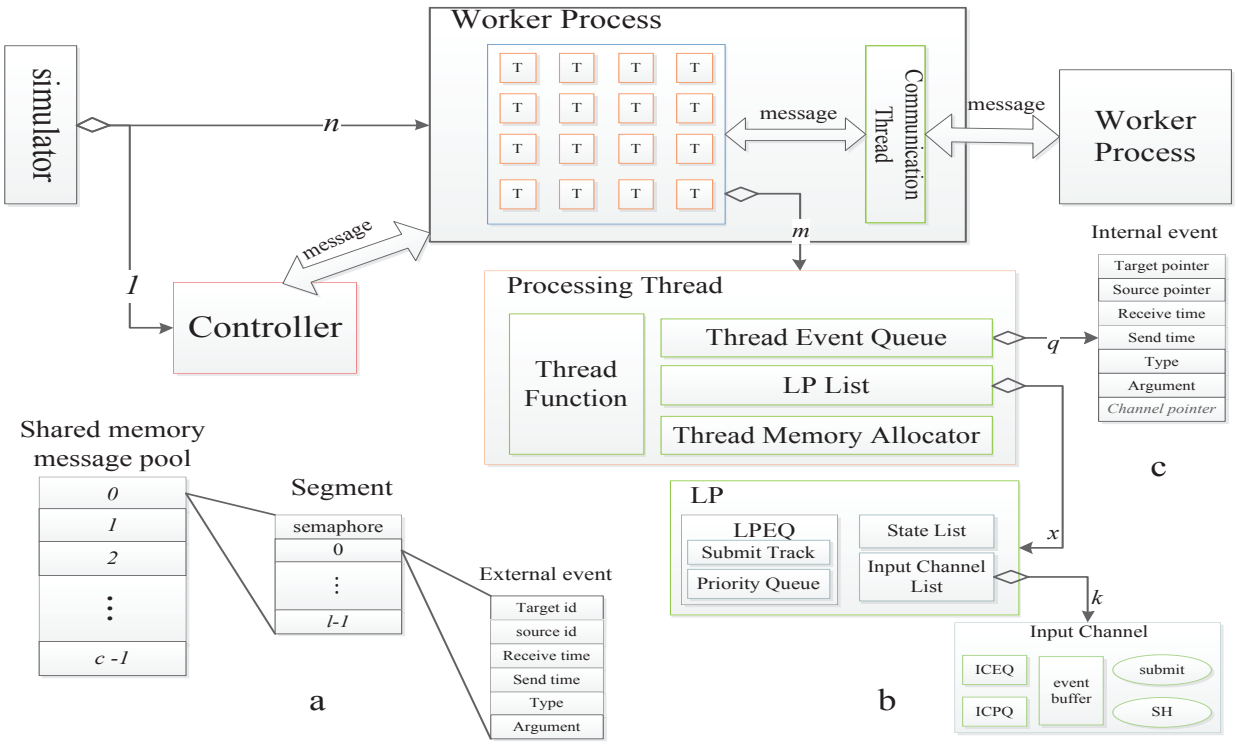

Figure 1: Architecture of NTW-MT simulator.

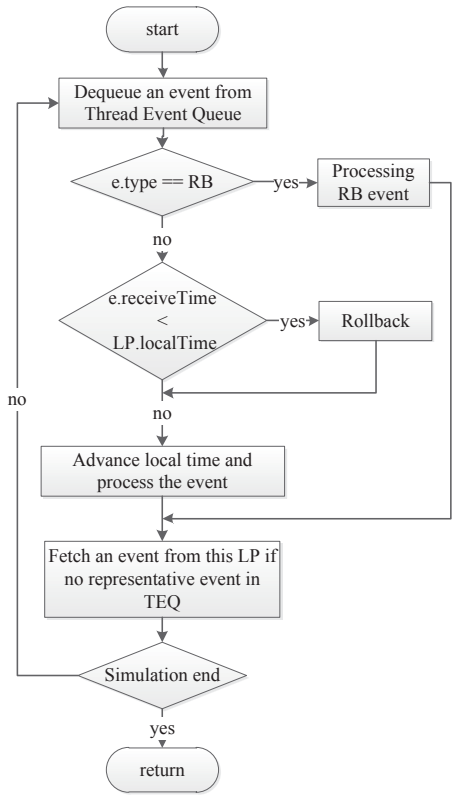

Figure 2: Processing sequence of a PE.

events which are scheduled by internal processing threads and *external events* from external processing threads. As the processing threads share the same memory space, internal events use a pointer to identify LPs. A PE can use this pointer to access the LP directly. External events use an integer identifier to represent LPs. They are converted into internal events upon receipt by a communication thread and are then inserted into LPs.

3.1 Multi-Level Queuing

Every thread in a process can access any priority queue in the same process, which can cause excessive contention [7]. To further aggravate matters, during a roll-back processed events with a timestamp greater than the receive time are re-enqueued in the priority queue. Two ways are used to alleviate this contention (1) decreasing the probability by which a few threads can access the same queue and (2) decreasing the cost of a single operation on a queue.

Consider the example in figure 3. In a stochastic neuronal simulation, the virtual time increment can vary from 0 to a large value. The diagram in figure 3 depicts 6 LPs (rounded rectangles) which are distributed over three processes (dashed rectangles). The number in each LP is the Local Virtual Time (LVT) of the LP (also used to identify the LP). Each arrow represents an event. The number on each arrow is the timestamp of the event (receive time, send time).

In figure 3 event (12,10) is a straggler at LP 15, and the dashed-lined-events at LP 16 are pending events. LP 16 has four pending events which would be stored in a queue. One may notice that it is not necessary to store and sort all four events in the TEQ- only event (18,14) needs to stay in the

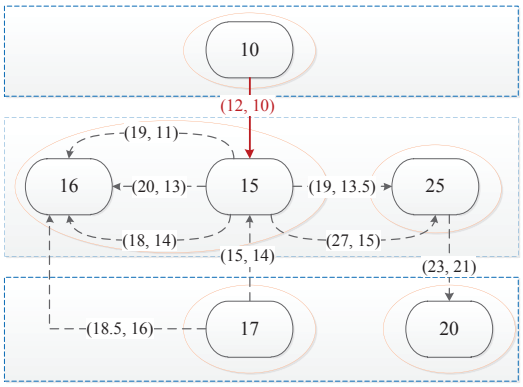

Figure 3: This example shows 6 LPs (rounded rectangles) which are distributed over three processes (dashed rectangles). Some are located in the same processing thread(an ellipse). The number in each LP is the local virtual time of the LP, and is also used to identify the LP. Each arrow indicates an event.The number on each arrow is the timestamp of the event (receive time, send time). Some LPs and events are omitted.

TEQ. There are several reasons for this: (1) some insertions in the TEQ can be omitted, decreasing the probability of contention. (2) the size of the TEQ can be controlled and the cost of an operation eliminated. Events (20,13) and (18,14) would be cancelled due to the roll-back of LP 15, hence sorting them is wasteful. (3) to cancel an event which would not access the TEQ decreases the probability of contention. Suppose that (1) a PE holds x LPs, numbered from 0 to $x-1$ (2) S_i is the Pending Event Set (PES) of LP_i and that the events in S_i are in non-decreasing order in receive time. The minimum pending event of LP_i is $minPE_i$. It is easy to prove that $minPE$ in PES is equal to the $min\{minPE_i,\ i = 0, 1, \cdots, x-1\}$. Therefore the PE only needs to compare each $minPE_i$ to find $minPE$. This indicates that any LP only needs to have one representative event in TEQ. The remaining events are stored in the LP Event Queue (LPEQ). Each LP has a LPEQ and all the LPEQs are linked to the corresponding TEQ.

In a three-dimensional environment, space is partitioned by a mesh grid, resulting in a maximum of 6 neighbors for each LP. Molecules diffuse through channels between these neighbors. We use an input channel to receive events from a neighbor and store them in an Input Channel Event Queue (ICEQ). All of the ICEQs are linked to the corresponding LPEQ.

To control the size of a queue, a lower level queue can only submit an *urgent event* (definition 1 below) to an upper level queue. A lower level queue records the unprocessed events which have been submitted to an upper level queue, and checks to see if an event is urgent when it receives it. Every input channel has a variable *submit* of event pointer which refers to the lower bound of unprocessed events which have been submitted to the LPEQ. Every LPEQ has a stack structure *submitTrack* which traces the unprocessed events which have been submitted to its TEQ. At the input channel level, urgency is checked by comparing the receive time of event e_x and *submit* of this channel. Because the LPEQ receives events from several input channels, it may submit several events to the TEQ, and there may be several pointers in *submitTrack*, hence urgency is checked by comparing the receive time of event e_x and the *top* element of this stack. In a reaction-diffusion simulation an event can have smaller receive time then a predecessor, thereby leading to the creation of an urgent event (see the example below).

Definition 1. An event e_x^i in a queue q_x^i at level i is urgent if its receive time is less than the receive time of any events e_y^j in its upper level queue q_y^j at level j $(j > i)$.

At this point, we have three level queuing architecture, as shown in part b of figure 1. The queuing works as follows.

- In the initialize phase, the TEQ, LPEQ and input channels are constructed, *submitTrack* of LPEQ is empty and *submit* of input channel is set to 0. Each LP schedules an *initial* event to itself and adds it to its TEQ and then add the *initial* event to *submitTrack*.
- Any processing thread and communication thread can insert events into a LP. To insert an event e, a thread first identifies the target LP_x by routing and checks if it is located in the same process. If not, this event will be added to the send buffer of the communication thread. If so it is inserted in the target LP as follows.
  1. Apply memory, find the target LP LP_x, fill in the $Targetpointer$ field of this event by the pointer to LP_x.
  2. Determine the input channel $channel_x$, fill in the *Channel Pointer* field of this event by the pointer to $channel_x$, check whether it is urgent. If it is not urgent, insert it into ICEQ. Otherwise submit it to the LPEQ and update *submit*.
  3. At the LPEQ level, check urgency. If it is not urgent, insert it into LPEQ. Otherwise submit it to TEQ and insert its pointer to the *top* of *submitTrack*.
  4. At the TEQ level, insert this event into the TEQ.
- After processing an event from input channel $channel_s$, a PE fetches an event from $channel_s$ to make sure that there is a representative in the LPEQ, updates *submit* of $channel_s$, then inserts the new event in the LPEQ. At the level of the LPEQ, this PE checks *submitTrack* and the smallest event in LPEQ at that time to determine whether to submit an event to the TEQ or not.

In the example in figure 3, LP 16 has two neighbors LP 15 and 17. Event (19, 11) comes from LP 15, arriving at channel [16, 15] (a channel is marked in the format [host LP, source LP]), finds *submit* of this channel to be 0, and is submitted to the LPEQ of LP 16, *submit* is set to (19, 11); *submitTrack* of LPEQ 16 is empty. Then this event is submitted to TEQ, the pointer to (19, 11) is pushed at the *top* of *submitTrack* and the insertion of (19, 11) now ends. It is in the TEQ, where it at time $T1$. Then event (20, 13) arrives at channel [16, 15]. It is not urgent and thus stays in ICEQ. This insertion ends at time $T2$. Event (18, 14) arrives, and is found to be urgent for channel [16, 15], then it is submitted to the LPEQ; it is also urgent for LPEQ 16, and is submitted to the TEQ. The *top* of *submitTrack* becomes the pointer to event (18, 14) at time $T3$. Event (18.5, 16) from LP 17 arrives at channel [16, 17], finds *submit* to be 0, and is submitted to LPEQ 16. *submit* is set as a pointer to (18.5, 16) at time $T4$. The successive insertions depend upon the relationship of the above time points.

- $T4 < T1$, this implies this event arrives before any events from LP 15. (18.5, 16) will be submitted to TEQ, (19, 11) will stay at LPEQ 16, (18, 14) will be submitted to TEQ, (20, 13) stays at ICEQ [16, 15]. *top* of *submitTrack* is pointer to (18, 14) followed by pointer to (18.5, 16).
- $T1 < T4 < T2$, event (18.5, 16) is urgent and will be submitted to TEQ. Event (18, 14) is also urgent when it arrives at LPEQ 16. In this case, there are three events, (18, 14), (18.5, 16) and (19, 11), in TEQ.
- $T2 < T4$, event (18.5, 16) is not urgent when it arrives at LPEQ 16, thus stays at LPEQ 16. Two event, (19, 11) and (18, 14) are submitted to TEQ.

3.2 RB-Message

XTW [27] uses one RB-message to cancel incorrect events instead of sending a series of anti-messages, eliminating the need for an output queue at each LP thereby reducing the overhead of a roll-back.

When a LP receives a straggler, it rolls back to a point in time prior to the time of the straggler and then processes the straggler, as shown in figure 4. A LP does not store the processed events which were scheduled by itself. Processed events scheduled by other LPs are stored in the appropriate Input Channel Processed Queue (ICPQ). Note that the straggler may be not the processed event with smallest event because a processed event may have been sent from the TEQ.

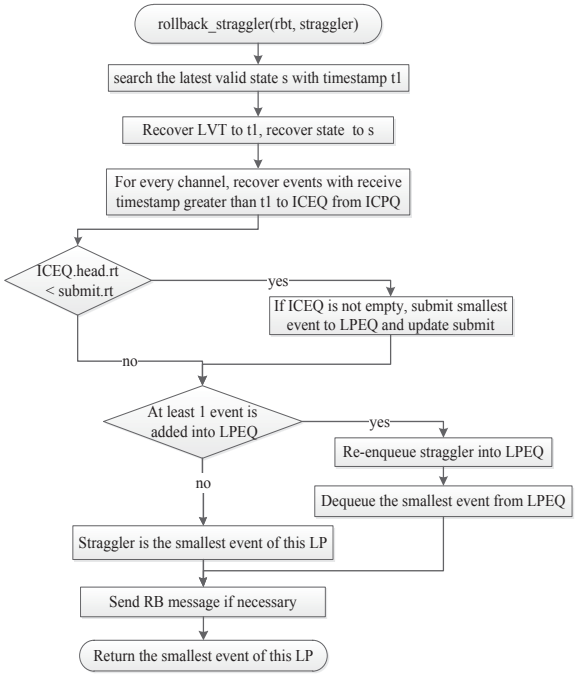

Figure 4: Steps for roll-back caused by straggler, *head* is the smallest event in any queue. rt(st) is receive time (send time).

In the above example, the message (12, 10) sent by LP 10 to LP 15 is a straggler, so LP 15 should roll back to a point in time before time 12. Suppose a state at 11.5 is found, and the states with timestamp greater than 11.5 are released. After that, LP 15 recovers processed events with timestamp greater than 11.5, finds the smallest event and sends RB-messages to its neighbors, LP 16 and LP 25. However, it is not really necessary to send a RB message to every neighbour; we use a variable *ScheduleHistory* (SH) [18] in the input channel in figure 1. SH records the upper bound of the send times of events sent to a LP. The use of SH can avoid sending unnecessary RB messages. For example, the local virtual time of LP i is 100, its input channel[0].SH is 80 (this indicates that it did not schedule any event to this neighbour after 80) and input channel[1].SH is 90. Assume that LP i receives a straggler and needs to roll back to 88. It is easy to assert that a RB message should be sent to the neighbour related to input channel[1] whereas there is no need to send a RB message to the neighbour related to input channel[0].

RB-messages has a higher priority *RB_PRIORITY*, a negative real constant, than other normal events, the send time of RB-message is set as the present LVT of the LP which sends it. A RB-message (*RB_PRIORITY*, t_{rb}) sent from LP x to LP y announces that LP x has rolled back to t_{rb}, then the pre-sent events with send time greater than t_{rb} becomes invalid. LP y follows the steps in figure 5 to process RB-messages.

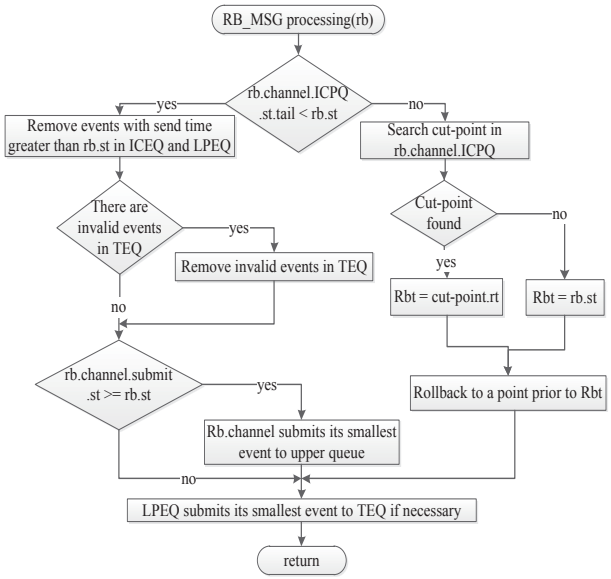

Figure 5: Steps for processing a RB-message, *tail* refers to the element with the greatest timestamp in the queue.

The processing of a RB-message depends upon whether or not invalid events have been processed.

- Left branch. The invalid events have not been processed. These events are removed and submit event to upper level queue if necessary. In the above example, a RB-message (*RB_PRIORITY*, 12) will sent by LP 15 to LP 16. Events (20, 13) and (18, 14) become invalid, while event (19, 11) is valid. The invalid events are removed from the queue in LP 16. This kind of RB-message does not interfere other LPs, for no successive roll-back will be triggered, thus we call it *friendly* RB-message.
- Right branch. The invalid events have been partly or totally processed. A roll-back is triggered. The send time

of the RB-message is used to find the $cut - point$ in the ICPQ, such that all events with a send time larger than the send time of the RB-message are after the cut-point. The LP sets the *rollbacktime* equal to the receive time of the first event after the $cut - point$. In the above example, a RB-message (*RB_PRIORITY*, 12) will be sent by LP 15 to LP 25, the events (19, 13.5) and (27, 15) become invalid while (19, 13.5) has been processed. A secondary roll-back is triggered and LP 25 follows steps in figure 6 to handle the roll-back.

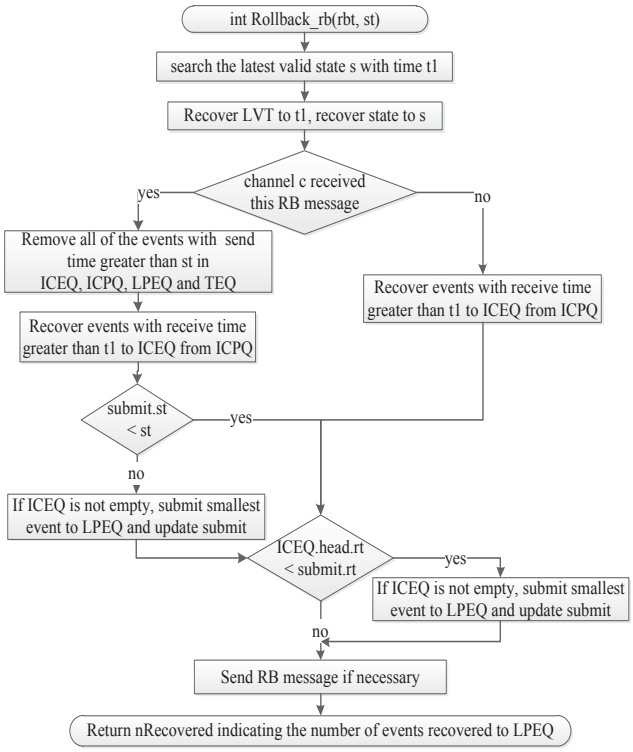

Figure 6: Steps for roll-back triggered by a RB-message.

The basic operation of RB-rollback is the same as that of a roll-back triggered by a straggler, except for the recovery of processed events. The channel which received the RB-message removes all of the events with a send time greater than the send time of the RB-message. Otherwise it moves the processed events with receive time greater than *rollback time* from ICPQ to ICEQ. Sending a RB-message is necessary for either type of roll-back. The $ScheduleHistory$ of channel [25, 20] is 21. Suppose that the first event after the $cut - point$ is (22, 10.5). LP 25 will roll back to time 22, because $22 > 21$ and no RB-message will be sent from LP 25 to LP 20. If the *rollback time* is 19, a RB-message (*RB_PRIORITY*, 19) will be sent to LP 20 by LP 25, and LP 20 applies the above steps to process this RB-message again. Our use of multi-level queue and RB-message is an extension of [27].

3.3 Hybrid Communication

We employ shared memory to shorten the delay of message passing because communication is the main performance bottleneck for PDES applications. We call the worker processes located in the same node a family. As shown in part a of figure 1, suppose there are c processes in a node, the shared memory is partitioned into c segments, numbered by $0, 1, \cdots, c-1$, each family process uses one of the segments as its receive buffer. Each segment employs a semaphore to control the access to it. For example, when process 0 is about to send data to its family member process 2, process 0 needs to hold the semaphore for process 2, then find room to write the data, after which it releases the semaphore when writing is finished. When receiving data, process 0 must first hold the semaphore and then receive the data, after which it releases the semaphore.

Together, we have three-level communication mechanism, communication between threads within same process is completed by pointers, by shared memory for processes in the same node, and by MPI for remote processes.

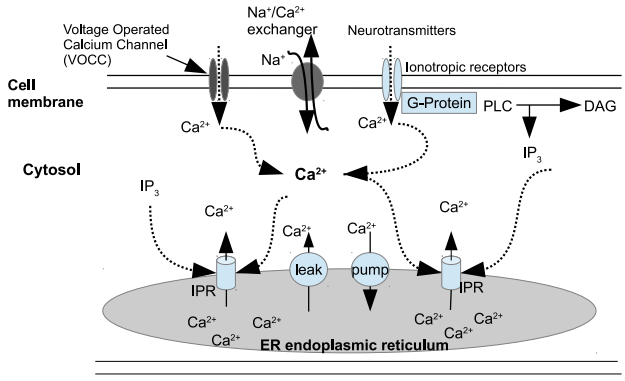

Figure 7: Intracellular calcium dynamics.

4. EXPERIMENTAL STUDY

4.1 Model

As previously mentioned, Calcium plays an important role in regulating a great variety of neuronal processes. A notable example of intracellular calcium dynamics is the Ca^{2+}-induced-Ca^{2+}-release (CICR) [2, 19] which controls a diverse array of cellular processes including fertilization, gene transcription, muscle contraction and even cell death. Figure 7 describes the CICR flow.

There is high level of calcium stored in the endoplasmic reticulum (ER) of a neuron. Inositol 1,4,5-triphosphate (IP_3) receptors are distributed on the surface of the ER. These receptors can be activated when the Ca^{2+} and IP_3 in the cytosol reaches a certain concentration. The IP_3 receptor acts to open a channel, and the ER then releases Ca^{2+} into the cytosol through this channel, thereby elevating the concentration of Ca^{2+}. Both the IP_3 and Ca^{2+} can diffuse freely. Generally the concentration of cytosolic Ca^{2+} and IP_3 are low and the channels are closed. If a neuron receives a signal from adjacent neurons, the G-protein on the cell membrane releases IP_3 into cytosol, some receptors are activated, leading to a local elevation of cytosolic Ca^{2+}. The IP_3 and the newly-generated Ca^{2+} diffuse to an adjacent region and activate the IP_3 receptor channels there. The concentration of cytosolic Ca^{2+} increases, and a wave begins to spread. Because of a difference in the concentra-

tion (the concentration gradient) of calcium along the ER, calcium ions leak into the cytosol at a low rate, increasing the concentration of calcium in the cytosol. There are pumps on the surface of the ER (known as SERCA pumps) which pump the calcium back into ER at a rate related to the concentration gradient, causing the opened channels to close.

A deterministic model has been developed in NEURON [17]. Based on this model we developed a discrete event model and simulated it. In our experiments we only take the IP_3 and Ca^{2+} into account. As the real CICR model is complex we simplified it by assuming (1) the $IP3$ receptor opens when the concentration of $IP3$ and Ca^{2+} are both higher than some respective threshold (2) an opening $IP3$ receptor channel will close for a period of time determined by an exponential distribution. The reactions include:

$$\text{ER Release: } Ca_{er}^{2+} \xrightarrow{k_{release}} Ca_{cyt}^{2+},$$
$$k_{release} = \nu_{IP3R}\, m^3\, n^3\, ([Ca_{er}^{2+}] - [Ca_{cyt}^{2+}])$$
$$\text{ER Leak: } Ca_{er}^{2+} \xrightarrow{k_{leak}} Ca_{cyt}^{2+},$$
$$k_{leak} = \nu_{leak}\, ([Ca_{er}^{2+}] - [Ca_{cyt}^{2+}])$$
$$\text{SERCA Pump: } Ca_{cyt}^{2+} \xrightarrow{k_{pump}} Ca_{er}^{2+},$$
$$k_{pump} = \frac{\nu_{SERCA}\,[Ca_{cyt}^{2+}]^2}{k_{SERCA}^2 + [Ca_{cyt}^{2+}]^2}$$

where Ca_{er}^{2+} refers to Ca^{2+} in ER, Ca_{cyt}^{2+} refers to Ca^{2+} in cytosol, $[\bullet]$ refers to the concentration of the corresponding species $\bullet$, $m = \frac{[IP_3]}{[IP_3]+k_{IP_3}}$, $n = \frac{[Ca_{cyt}^{2+}]}{[Ca_{cyt}^{2+}]+k_{act}}$, k_{IP_3}, k_{act}, ν_{IP3R}, ν_{leak}, ν_{SERCA} and k_{SERCA} are given constant parameters, the value can be found in [17]. Ca_{er}^{2+} can only diffuse within ER, cytosolic Ca^{2+} and IP_3 can only diffuse within cytosol.

We made use of the following scenario. At first, both cytosolic Ca^{2+} and IP_3 concentrations are low. Hence most of the IP_3 channels are inactive, and only leaks and SERCA pumping take place. IP_3 molecules are injected into some sub-volumes in the middle of a dendrite. The cytosolic Ca^{2+} concentration achieves a threshold level due to the leak, the IP_3 receptor channels begin to open and the ER release is triggered.

4.2 Geometry

We simulate the intracellular Ca^{2+} wave in an unbranched apical dendrite of a hippocampal pyramidal neuron (length: 1000 μm, diameter: 1 μm). An introduction to pyramidal neurons can be found in [22], Watanabe et al. modulate the calcium wave propagation in the dendrites and to the soma of rat hippocampal pyramidal neurons [26]. The hippocampus is a small region of the brain that resembles a seahorse and plays a role in learning and memory, figure 8 shows a three-dimensional view of the neuron. The neuron is partitioned into mesh grids, and each grid is taken to be a sub-volume. We select 14749 sub-volumes with a distance of less than 50 μm from the middle, and the length of each sub-volume to be 0.5 μm. The sub-volumes are evenly distributed among the processing threads.

4.3 Platform

We use two platforms. One machine (PEPI) is a cluster with 4 Intel(R) Xeon(R) E7 4860 2.27 GHz, 10 cores per processor, 1 TB memory, with Linux 2.6.32-358.2.1.el6.x86_64,

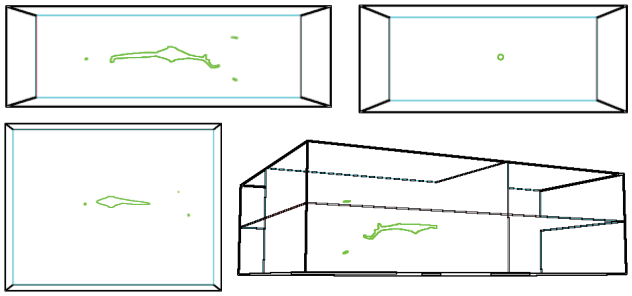

Figure 8: Pyramidal neuron in a three-dimensional view.

Red Hat Enterprise Linux Server release 6.4 (Santiago). The other is the SW2 node (of Guillimin), consisting of two Dual Intel(R) Sandy Bridge EP E5-2670 2.6 GHz CPUs, 8 cores per processor, 8 GB of memory per core, and a Non-blocking QDR InfiniBand network with 40 Gbps between nodes. The node runs Linux 2.6.32-279.22.1.el6.x86_64 GNU/Linux.

4.4 Performance

The performance of NTW-MT is compared to two other simulators. One is a process-based parallel simulator which uses a controller process to calculate GVT. Memory operations employ the standard *new* and *delete* mechanism. A thread+SQ version uses threads but does not use the MLQ algorithm. Each thread uses a single priority queue to hold the pending events. We know from [27] that RB messages result in superior performance when compared to anti-messages and do not compare the simulator to one with anti-messages.

We use an STL multi-set as the implementation of the priority queue. The mean access time to this multi-set is proportional to its size.

In the Thread+SQ case, when 32 processing threads are used, a roll-back avalanche occurs. This phenomenon is much more serious for the process-based version, which essentially cannot get beyond 8 processes. We consider these results inaccurate in terms of performance and do not include them.

The placement of processing threads is an important issue, it affects the memory usage and communication. We consider three placements- (1) *within process* in which all of the processing threads are in the same process, thereby no interprocess events; (2) *within node* in which processes exchange messages via shared memory; (3) *hybrid* respectively which makes use of MPI for remote processes and the preceding techniques otherwise.

4.4.1 within Process Mode

We run this experiment in the PEPI machine by starting up two processes, one *controller* and one *worker* process, and creating the *processing* threads within the *worker* process.

From figure 9, we can see that the execution time decreases with an increase in the number of processing threads. The process-based version is slowest because each process receives and sends events in the main processing loop and communication is time-consuming.

When fewer then 8 processing threads are involved in the simulation, the thread+MLQ version is slower then the thread+SQ version. The greatest difference (about 13%)

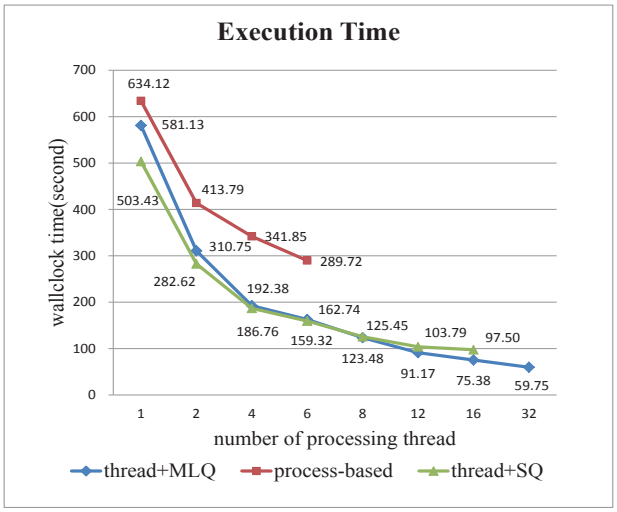

Figure 9: Execution time with all of the processing threads running within one process in the PEPI machine.

occurs when one processing thread is used. Because the essence of MLQ is the dispersion of contention on a single queue, it is of no use if there is no contention.

Consider the 1 processing thread case-almost all of the insertion of events end at the TEQ level, figure 12 illustrates this. Checking urgency in the input channel and the LPEQ level cause unnecessary overhead. When more threads are used, contention for the TEQ takes place. One LP may receive more than one event, the probability of contention at the SQ increases and the MLQ becomes more efficient. The thread+MLQ version is superior to the thread+SQ version when more than 8 processing threads are used, and finally achieves a speedup of 9 with 32 processing threads used, compared to the 1 processing thread case.

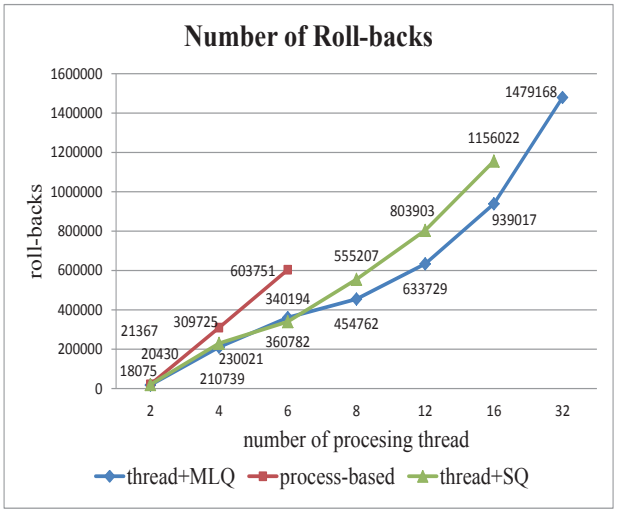

Figure 10: Roll-backs in within process mode in the PEPI machine.

Roll-backs increase for all of the versions in figure 10. The process-based version suffers more roll-backs (25%-35%) than the other two versions. The roll-back of the two thread version is almost same in the few thread cases, while the MLQ version experienced fewer (around 18%) roll-backs than the SQ version. The events are inserted into thread queue directly in the SQ version resulting in a greater delay.

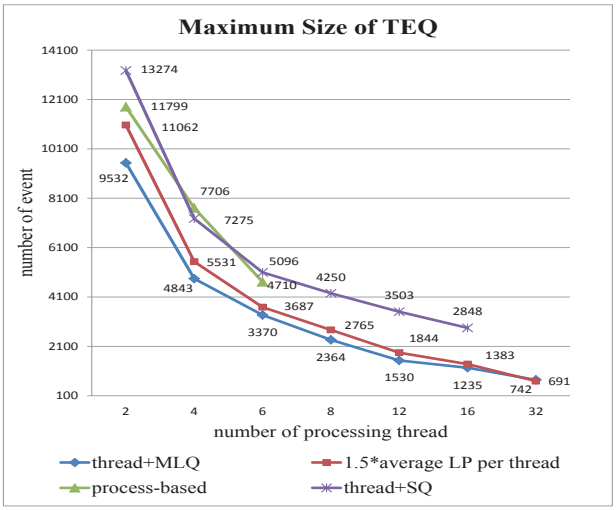

Figure 11: Maximum size of TEQ in within process mode in the PEPI machine.

The size of the TEQ scales well-it contains no more than 1.5 times the average number of LPs per processing thread. Hence the access time for the TEQ is well controlled, see figure 11.

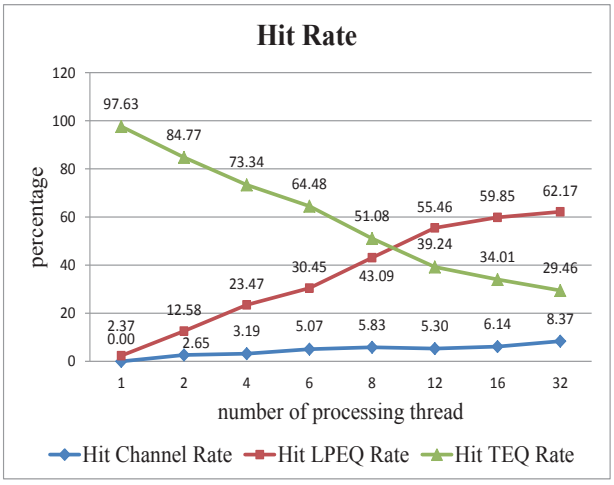

Figure 12: Hit rate in within process mode in PEPI machine.

In the MLQ algorithm, an event insertion may end at different levels of the queuing system-the input channel, the LPEQ and the TEQ. Define the hit rate of a level to be the proportion of insertions ending at each level. In the process-based and thread+SQ version, all of the events are inserted into the thread queue. From figure 12, we can see that most insertions end at the LPEQ when more than 8 threads are used, suggesting that the contention is dispersed.

4.4.2 *within Node Mode*

We assume that each worker process has the same number of processing threads, and vary the total number of processing threads in the simulation by starting up variant number of worker processes. All of the processes reside in the same node, they transfer external messages via shared memory. This experiment was done in the PEPI machine.

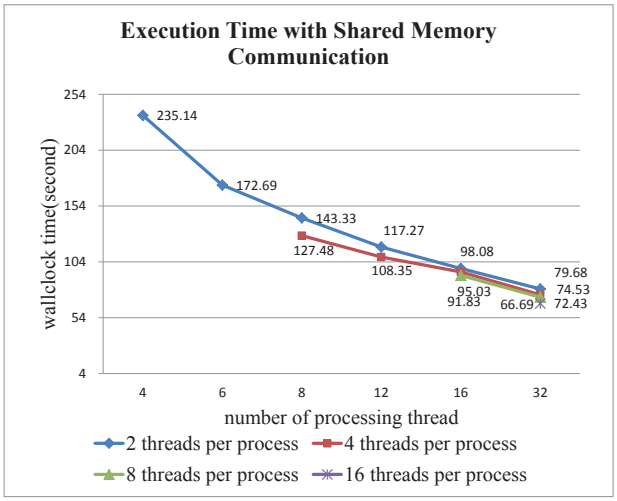

Figure 13: Execution time with shared memory communication in within node mode in the the PEPI machine.

From figure 13 and 14, we see that placing more threads in the same process results in better performance, this is reasonable for less interprocess communication is used. However, comparing these results to those obtained by placing all of the processing threads in the same process, we do not see a great difference. The combination of communication threads and shared memory results in a short latency.

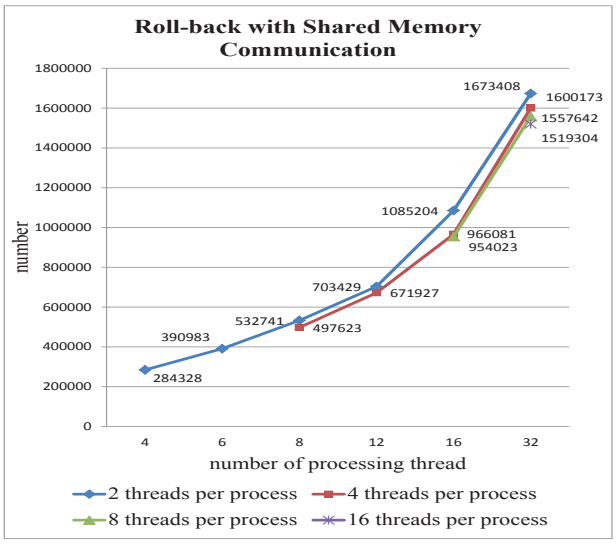

Figure 14: Roll-back with shared memory communication in within node mode in the PEPI machine.

4.4.3 *Hybrid Mode*

The number of threads is limited by the number of physical cores in a node, and the number of threads should not exceed the number of physical cores [1], which indicates employing several nodes to have large scale simulation is inevitable. We had this experiment in the Guillimin machine and used the MPI option ppn (process per node) to dispatch *worker* processes to nodes.

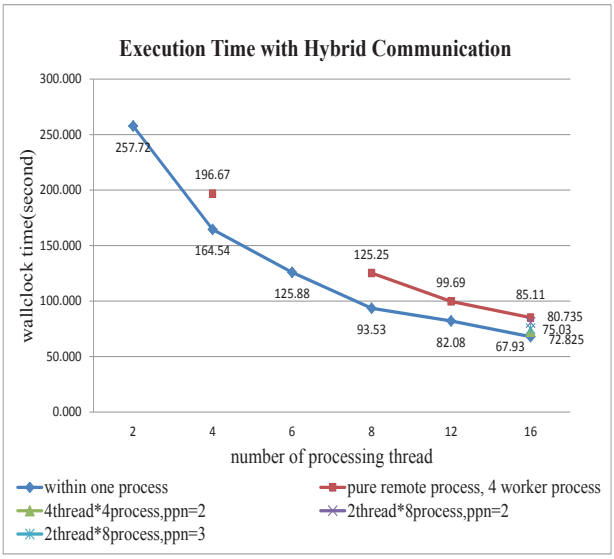

Figure 15: Execution time with hybrid communication in the Guillimin machine, ppn refers to process per node.

The Guillimin machine uses a well-optimized infiniBand for remote communication. From the results in figure 15 and 16, we again see that placing all of the processing threads in the same process results in the best performance. Once when the remote communication is involved in, the performance goes down without any surprise, the roll-back increases sharply, the execution time is enslaved to the longest latency.

In the purely remote communication case, the send buffer of the communication thread overflows when more than 8 threads are in the same process. One communication thread cannot accommodate 8 processing threads and more. This indicates the hybrid mode is the general mode for large scale simulation, and the number of processing threads in one worker process should be properly determined.

5. CONCLUSION AND FUTURE WORK

This paper is concerned with the development of a parallel discrete event simulator for reaction diffusion models used in the simulation of neurons. The research was done as part of the NEURON project (www.neuron.yale.edu). It is our intention to include NTW-MT in NEURON for use by the general neuroscience community.

We simulate a discrete event model for calcium wave propagation on an unbranched apical dendrite of a hippocampal pyramidal neuron. It is known that calcium plays a fundamental role in the second messenger system of a neuron. However, the mechanism by which calcium waves are transmitted is not completely understood. Our model is based

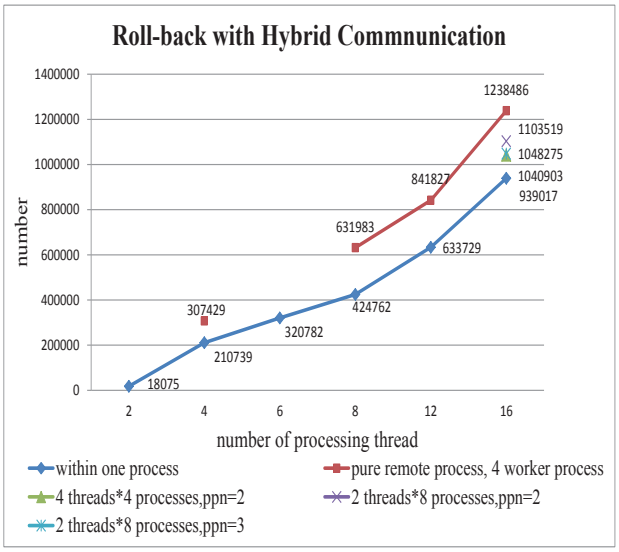

Figure 16: Roll-back with hybrid communication in the Guillimin machine.

on a deterministic calcium wave model described in [17]. Because stochastic models are more realistic than deterministic models for small populations, it may be possible to shed more light on the transmission mechanism.

Our parallel simulator is optimistic and thread based. It makes use of the NSM algorithm [8]. The use of threads is an attempt to capitalize on multicore architectures used in high performance machines, communication latency among threads within the same process is minimized by pointers. It makes use of a multi-level queue for the pending event set and a single rollback message in place of individual anti-messages. We examined its performance on the calcium wave model and compared it to the performance of (1) a process based optimistic simulator and (2) a threaded simulator which uses a single priority queue for each thread. The multi-level queue simulator exhibited a superior performance when all of the threads were placed in the same process. The effects of shared memory and MPI based communication were also investigated; the multi-level queue simulator proved to be scalable. However, the need for load balancing algorithms was very clear in our experiments.

Our future work on the calcium wave model includes (1) implementing a more detailed model and larger reaction diffusion model and examining the performance of the multi-level queue algorithm on this model (2) developing load balancing algorithms for NTW-MT. A hybrid (deterministic-stochastic) model is another future effort.

6. ACKNOWLEDGMENTS

This work is funded by China Scholarship Council and in part by the National Natural Science Foundation of China (No. 61170048), Research Project of State Key Laboratory of High Performance Computing of National University of Defense Technology (No. 201303-05) and the Research Fund for the Doctoral Program of High Education of China (No. 20124307110017). This work is also funded by NIH R01MH086638 and NIH T15LM007056.

7. REFERENCES

[1] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. S. Vetter. Characterization of scientific workloads on systems with multi-core processors. In *Proceedings of the 2006 IEEE International Symposium on Workload Characterization*, pages 225–236, San Jose, California, USA, October 25-27 2006. IEEE.

[2] M. J. Berridge. Neuronal calcium signaling. *Neuron*, 21(1):13–26, 1998.

[3] N. T. Carnevale and M. L. Hines. *The NEURON book*. Cambridge University Press, New York, USA, 2006.

[4] N. T. Carnevale and M. L. Hines. Neuron, for empirically-based simulations of neurons and networks of neurons. `http://www.neuron.yale.edu`, 2009-2013. Last access on May 1st 2015.

[5] L.-l. Chen, Y.-s. Lu, Y.-p. Yao, S.-l. Peng, and L.-d. Wu. A well-balanced time warp system on multi-core environments. In *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, pages 1–9, Nice, France, June 14-17 2011. IEEE Computer Society.

[6] L. Dematté and T. Mazza. On parallel stochastic simulation of diffusive systems. In M. Heiner and A. M. Uhrmacher, editors, *Computational methods in systems biology*, volume 5307 of *Lecture Notes in Computer Science*, pages 191–210. Springer Berlin Heidelberg, 2008.

[7] T. Dickman, S. Gupta, and P. A. Wilsey. Event pool structures for pdes on many-core beowulf clusters. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '13, pages 103–114, New York, NY, USA, 2013. ACM.

[8] J. Elf and M. Ehrenberg. Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. *Systems biology*, 1(2):230–236, 2004.

[9] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The journal of physical chemistry*, 81(25):2340–2361, 1977.

[10] D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev. Optimization of parallel discrete event simulator for multi-core systems. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 520–531, Shanghai, China, May 21-25 2012.

[11] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.

[12] M. Jeschke, R. Ewald, and A. M. Uhrmacher. Exploring the performance of spatial stochastic simulation algorithms. *Journal of Computational Physics*, 230(7):2562–2574, 2011.

[13] M. Jeschke, A. Park, R. Ewald, R. Fujimoto, and A. M. Uhrmacher. Parallel and distributed spatial simulation of chemical reactions. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, PADS '08, pages 51–59, Washington, DC, USA, 2008. IEEE Computer Society.

[14] W. W. Lytton. *From Computer to Brain*. Springer-Verlag, New York, USA, 2002.

[15] R. A. McDougal, M. L. Hines, and W. W. Lytton. Reaction-diffusion in the neuron simulator. *Frontiers in Neuroinformatics*, 7(28), 2013.

[16] R. J. Miller. *Optimistic parallel discrete event simulation on a beowulf cluster of multi-core machines*. PhD thesis, University of Cincinnati, 2010. http://secs.ceas.uc.edu/p̃aw/research/theses/ryan_miller.pdf.gz, last access on May 1st 2015.

[17] S. A. Neymotin, R. A. McDougal, M. A. Sherif, C. P. Fall, M. L. Hines, and W. W. Lytton. Neuronal calcium wave propagation varies with changes in endoplasmic reticulum parameters: A computer model. *Neural Computation*, 27(4):898–924, Mar. 2015.

[18] M. N. I. Patoary, C. Tropper, Z. Lin, R. McDougal, and W. W. Lytton. Neuron time warp. In *Proceedings of the 2014 Winter Simulation Conference*, WSC '14, pages 3447–3458, Piscataway, NJ, USA, 2014. IEEE Press.

[19] H. Roderick, M. J. Berridge, and M. D. Bootman. Calcium-induced calcium release. *Current Biology*, 13(11):R425, 2003.

[20] W. N. Ross. Understanding calcium waves and sparks in central neurons. *Nat Rev Neurosci*, 13(3):157–168, Mar. 2012.

[21] R. B. Schinazi. Predator-prey and host-parasite spatial stochastic models. *The Annals of Applied Probability*, 7(1):1–9, 1997.

[22] N. Spruston. Pyramidal neuron. *Scholarpedia*, 4(5):6130, 2009.

[23] D. Sterratt, B. Graham, A. Gillies, and D. Willshaw. *Principles of computational modelling in neuroscience*. Cambridge University Press, New York, USA, 2011.

[24] B. Wang, B. Hou, F. Xing, and Y. Yao. Abstract next subvolume method: A logical process-based approach for spatial stochastic simulation of chemical reactions. *Computational biology and chemistry*, 35(3):193–198, 2011.

[25] B. Wang, Y. Yao, Y. Zhao, B. Hou, and S. Peng. Experimental analysis of optimistic synchronization algorithms for parallel simulation of reaction-diffusion systems. In *Proceedings of the 2009 International Conference on High Performance Computational Systems Biology*, pages 91–100, Trento, Italy, October 14-16 2009.

[26] S. Watanabe, M. Hong, N. Lasser-Ross, and W. N. Ross. Modulation of calcium wave propagation in the dendrites and to the soma of rat hippocampal pyramidal neurons. *The Journal of Physiology*, 575(2):455–468, 2006.

[27] Q. Xu and C. Tropper. Xtw, a parallel and distributed logic simulator. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 181–188, Monterey, California, USA, June 1-3 2005. IEEE Computer Society.