Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

AN ASYNCHRONOUS GVT COMPUTING ALGORITHM IN NEURON TIME WARP-MULTI THREAD

Zhongwei Lin

State Key Laboratory of High Performance Computing
National University of Defense Technology
17 Yanwachizheng, Changsha
Hunan, 410073, CHINA

Yiping Yao

College of Information System and Management
National University of Defense Technology
17 Yanwachizheng, Changsha
Hunan, 410073, CHINA

ABSTRACT

Multi-threaded Parallel Discrete Event Simulation (PDES) is promising to achieve high performance. Generally employing a collection of multi-core nodes is necessary to accomplish large scale PDES, which makes it running in a hybrid of distributed and shared memory platform. Present Global Virtual Time (GVT) computing algorithms are suitable for pure distributed or shared memory platform. In this paper we present an asynchronous GVT computing algorithm in Neuron Time Warp-Multi Thread (NTW-MT) simulator for stochastic simulation in NEURON project. GVT is computed asynchronously both within and among processes, which is the first try in multi-threaded PDES as far as we know. Then we prove this algorithm can compute a valid GVT at any wall clock time, and conclude it has less computational cost through analysing the cost and delay . Finally we show results of simulating a calcium wave model in an unbranched apical dendrite of a hippocampal pyramidal neuron.

1 INTRODUCTION

1.1 Stochastic Simulation of Reaction and Diffusion in Neurons

The human brain may be viewed as a sparsely connected network of neurons (Carnevale and Hines 2006) containing approximately 10^{14} neurons. 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 (a) diffusion from a higher concentration of ions or (b) by pumps which are dependent on the voltage drop across the membrane. Electrical models for neurons (Lytton 2002) 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 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 (Carnevale and Hines 2006, Lytton 2002). However, a continuous model is not appropriate for a small number of molecules. Stochastic model is a far more realistic and accurate representation (Sterratt, Graham, Gillies, and Willshaw 2011, Ross 2012) 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 (Sterratt, Graham, Gillies, and Willshaw 2011). In general, it is very difficult to solve this equation. In (Gillespie 1977) a Monte Carlo simulation algorithm called 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) (Elf and Ehrenberg 2004) is an extension of SSA which incorporates the diffusion of molecules into the model. The NSM partitions the whole space into cubes called sub-volumes, and represents the diffusion of ions between these cubes by events.

As previously indicated, the number of cells involved in a realistic simulation of a network of neurons is immense. Hence it is necessary to make use of a cluster of computers for such a simulation. Since each cube in NSM can interact with other cubes, it can be represented by a Logical Process (LP) in PDES (Wang, Hou, Xing, and Yao 2011), and diffusion of ions between neighbouring sub-volumes is represented as event, then PDES techniques can be introduced in order to speed up the simulation. We previously developed a process based simulator, Neuron Time Warp (NTW) (Patoary, Tropper, Lin, McDougal, and Lytton 2014), which does not use threads. NTW was verified and its performance examined on a Calcium buffer model and a predator-prey (Schinazi 1997) model.

NEURON (Carnevale and Hines 2013, Carnevale and Hines 2006) is a widely used simulator in neuroscientist community. It makes use of deterministic simulators for reaction-diffusion models (McDougal, Hines, and Lytton 2013) and electrical models. We are collaborating with the NEURON group, and our intention is to develop parallel discrete event simulators suitable for simulating reaction diffusion models within and among neurons. It is intended that our simulators will be integrated into NEURON.

1.2 Multi-threaded Extension to NTW

Communicational latency is the main bottleneck of PDES systems (Fujimoto 1999). The emergence and widespread use of multi-core processor presents a promising opportunity to PDES, for the communicational cost is significantly reduced by very fast channels among cores on a multi-core chip. Alam (Alam, Barrett, Kuehn, Roth, and Vetter 2006) observed a significant benefit (approximately 8% to 12%) when communicating between processes running within a multi-core processor as opposed to between cores on different processors.

Two constrains make it impractical to have very large scale PDES in a single cluster node: (a) the number of threads should not exceed the number of physical cores in a cluster node (Alam, Barrett, Kuehn, Roth, and Vetter 2006); (b) the processing rate of all cores in a cluster node is enslaved to the bandwidth of memory bus, for the cores still share the same memory. Hence to run a simulation in a collection of cluster nodes is needed, necessitating corresponding architecture and algorithms.

The architecture of our multi-threaded simulator, Neuron Time Warp-Multi Thread (NTW-MT), is depicted in figure 1. One process is the controller, exercising global control functions (GVT computing and load balancing etc.). The remaining processes are worker processes that process events at the LPs residing in each process. Each worker process contains a *communication* thread and a bunch of *processing* threads. The overhead of communication is high, thus we remove message receiving and sending in the main loop of event-processing and employ a *communication* thread to receive and send messages for those *processing* threads within the same process instead. All worker processes have the same number of threads.

The communication thread sends and receives messages for an individual process. Processing threads can neither send nor receive messages. After initialization, the communication thread receives messages from shared memory if the message is from a *family process* that resides in the same cluster node or via MPI from remote cluster nodes. LPs then schedule external events by first placing them into the send buffer of the corresponding communication thread. To avoid contention on 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 also scans the segments in this buffer and 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 in each worker process. Each subset is mapped to a processing thread. Each processing thread includes a *LP List* that stores the LPs associated with the thread, a *Thread Event Queue* (TEQ), a

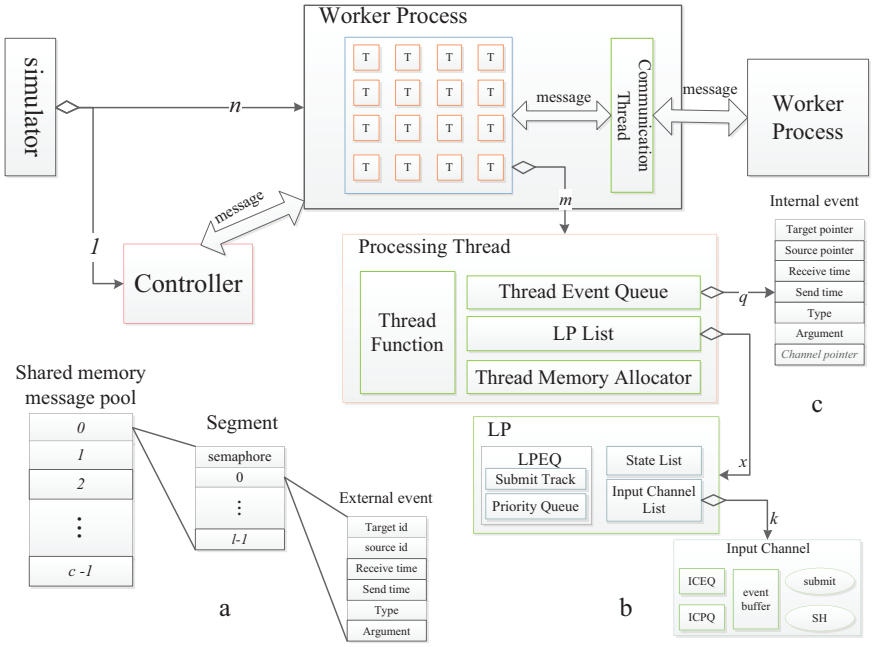


Figure 1: Architecture of NTW-MT simulator.

Thread Memory Allocator involved in Memory Management and a *Thread Function*. The Thread Function is responsible for processing events.

An event has two timestamps, the receive time and the send time (Jefferson 1985). Events in priority queues are sorted by their receive time. There are two types of events: *internal event* scheduled by internal processing threads and *external event* from external processing threads. As the processing threads within a worker process share the same memory space, internal events are managed by pointers (eliminating copying messages), and they use pointers to identify LPs, then a thread can use this pointer to access the associated LP directly. External events use an integer identifier to represent LPs. They are converted into internal events upon receipt by communication threads and then inserted into LPs. We extend the Multi-Level-Queue (MLQ) algorithm and RB-message mechanism in (Xu and Tropper 2005) to reduce contention on TEQs and overhead of roll-back respectively.

NTW uses Mattern's algorithm (Mattern 1993) to compute GVT. As processing threads in NTW-MT can neither send nor receive external events, Mattern's algorithm cannot work, while employing the communication thread alone to compute GVT would cause simultaneous reporting problem. In this paper we are computing GVT asynchronously both within and among processes by combining distributed-memory and shared-memory GVT computing algorithms.

The remainder of this paper is as follows. Section 2 is devoted to related work. We analyse the simultaneous reporting problem in computing local GVT asynchronously within a process, present and prove our asynchronous algorithm, and analyse the computational cost and delay of our algorithm in section 3. Section 4 contains a description of our experiments as well as the results and section 5 contains our conclusion and future work.

2 RELATED WORK

The Time Warp protocol (Jefferson 1985) includes local control mechanism and global control mechanism. Computing GVT is one essential operation of global mechanism. A GVT value t at wall clock time T tells the fact that no LP will roll back to a time point prior to t after T, then all states and processed events

with timestamp less than t can be reclaimed, i.e. *fossil collection*. Obviously, GVT should be computed frequently to save memory consumption, and efficiently to achieve high performance.

In distributed memory environment two problems, the *transient message* problem and the *simultaneous reporting* problem (Samadi 1985), make it non-trivial to compute GVT. Both Samadi's (Samadi 1985) and Mattern's (Mattern 1993) algorithm, two classical GVT computing algorithms, compute GVT by two rounds of message passing, while Mattern's algorithm does not require acknowledgement for each message, which turns out better performance. Fujimoto (Fujimoto and Hybinette 1997) proposed an efficient algorithm for shared memory multiprocessors. In this algorithm, suppose p processes participate in a simulation, any process can trigger GVT computing by setting a variable $GVTFlag$ to p, other processes periodically check this variable, report its local GVT and decrease $GVTFlag$ by 1, then the latest GVT comes out when $GVTFlag$ equals to zero. However this algorithm requires all of the processes reside within one shared memory cluster, which constrains the scale of simulation.

For computing GVT in multi-threaded PDES systems, Ross-MT (Jagtap, Abu-Ghazaleh, and Ponomarev 2012) uses an optimized barrier to block all threads at wall clock time point T, which can cause high overhead in waiting for all threads if there are a few threads within one process or the duration for processing a single event is long. Each thread in (Chen, Lu, Yao, Peng, and Wu 2011) can receive and send message, thus Mattern's algorithm can work properly. In Threaded WARPED (Miller 2010), a manager thread is responsible to calculate local GVT in a *threadedWarped* node. When a GVT computing requirement arrives at a node, the manager thread suspends all of the simulation objects in that node (to make sure no creation of new messages within that node) and then sets the Least Time-Stamped Event (LTSE) as the present local GVT of that node. The computing of local GVT in Threaded WARPED still operates like a barrier. In (Pellegrini and Quaglia 2014) the authors propose a wait-free GVT computing algorithm by splitting the GVT computing phase in (Fujimoto and Hybinette 1997) into three phases and achieve better performance than Fujimoto's original algorithm, whereas this algorithm only works in shared memory platform due to the need of memory consistency.

Overall the present GVT computing algorithms are suitable for either pure distributed or pure shared memory platform. NTW-MT runs in a hybrid platform and thereby needs corresponding algorithms.

3 ASYNCHRONOUS GVT COMPUTING IN NTW-MT

3.1 Problem Analysis

There is no *transient message* problem within a worker process, for any internal message should be in the queuing system of that process in either the sender thread side or the receiver thread side. In a barrier based GVT computing algorithm, the processing threads in a worker process are blocked when this process is computing GVT, thus there will be no creation of new messages before a local GVT value is computed. This fact eliminates the probability of *simultaneous reporting* problem within worker processes. In NTW-MT, a local GVT is calculated asynchronously among processing threads in a process to avoid cost for synchronizing threads, then the *simultaneous reporting* problem can occur. Consider the example in figure 2, the number on a local GVT computing arrow is the wall clock time of that operation, and the number on an event scheduling arrow is the wall clock time of that scheduling, while the number in parentheses is the timestamp of that event and the number in a processing thread rectangle is the least timestamp of events in that TEQ (note that all number is given as an example). There are concurrent operations to TEQs (the hosting thread dequeues event from the corresponding TEQ while other threads can enqueue events into that TEQ simultaneously.) within a worker process, thus each access and operation to a TEQ is exclusive, and each individual TEQ is protected by a *lock*.

In figure 2, in order to compute a local GVT meeting the definition in (Fujimoto 1999) the communication thread begins to check TEQ_i at wall clock time T_1 and leaves TEQ_i at wall clock time $T_1^{'}$, then it goes to check TEQ_j at wall clock time T_4, thus we have $T_1 < T_1^{'} < T_4$. Processing thread j schedules an event stamped at 10 to thread i at wall clock time T_2, thread k schedules an event stamped at 15 to thread j

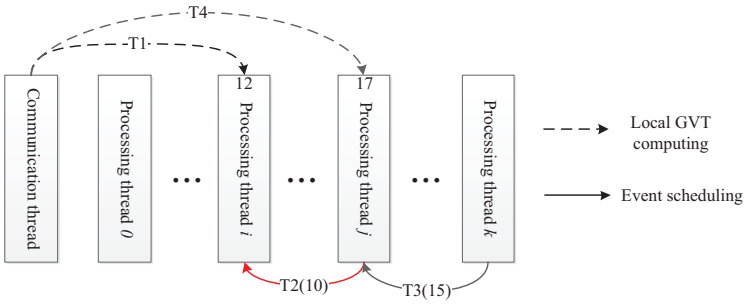


Figure 2: An example of simultaneous reporting problem within a worker process.

at wall clock time T_3, here T_2 and T_3 can be in arbitrary order. Suppose these time points satisfy the order $T_1 < T_1^{'} < T_2 < T_3 < T_4$ (this circumstance is possible due to arbitrary order of locking and unlocking TEQs), then the event stamped at 10 is not accounted for, resulting in an incorrect local GVT value 12.

The key to this problem is that the inter-thread event that is sent during GVT computing in that process should be accounted for. Fujimoto (Fujimoto and Hybinette 1997) uses a variable *GVTFlag* to indicate the number of processes in GVT computing, and a process should trace the minimum timestamp when it sends out events if *GVTFlag* is greater than zero.

3.2 Proposed GVT Computing Algorithm in NTW-MT

From figure 1, we can see that NTW-MT runs on a hybrid of distributed memory and shared memory platform, thus both distributed-memory and shared-memory GVT computing algorithms are used. The message flow and corresponding data structure is depicted in figure 3. Every *worker* process holds two sets of variables for GVT computing, i.e. the set (color, whiteCount, minRed) is to trace interprocess communication, and the set (GVTFlag, localGVT) along with the *minSend* and *localMin* variable in each *processing* thread is to find local GVT in each *worker* process.

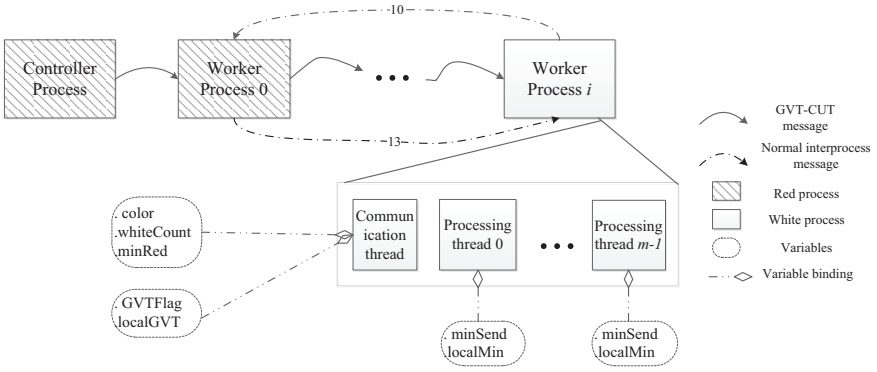


Figure 3: Message flow and data structure of asynchronous GVT computing in NTW-MT.

In Mattern's algorithm, a process can be in either *red* or *white* color, and a *red* process indicates it is in calculation of GVT. At the beginning, all of the processes are in *white* color, and a *white* process becomes *red* when it receives a GVT-CUT message, while a *red* process becomes *white* when it receives a GVT-broadcast message that notifies the latest GVT value. Meanwhile, normal messages are also coloured by the sender process. A GVT-CUT message is a special control message to notify every *worker* process prepare for GVT computing, and it has two fields (tempGVT, count), where *tempGVT* indicates a temporal value of the latest GVT and *count* indicates the number of *white* messages sent but not yet received since the last broadcasting of GVT. The *controller* process triggers a GVT computing round by sending a GVT-CUT

message (∞, 0) to the first *worker* process in every T_{GVT} physical second. Upon receiving an external message, a *worker* process counts the number of *white* messages received since the last GVT broadcasting, and follows the steps in figure 4(a) to prepare GVT computing if the received message is a GVT-CUT message. In Fujimoto's original algorithm, a process should check the *GVTFlag* variable before processing every event and update its local GVT value if *GVTFlag* is greater than zero, because it is not aware of whether there are some other processes computing GVT at that time. In NTW-MT a process is computing GVT if that process is in *red* color, then it inserts a GVT-CMP message to the TEQ of each *processing* thread. A GVT-CMP message is a special control message to require each *processing* thread report its local GVT. Hence each *processing* thread only modifies *GVTflag* and updates local minimum timestamp once in each round of GVT computing.

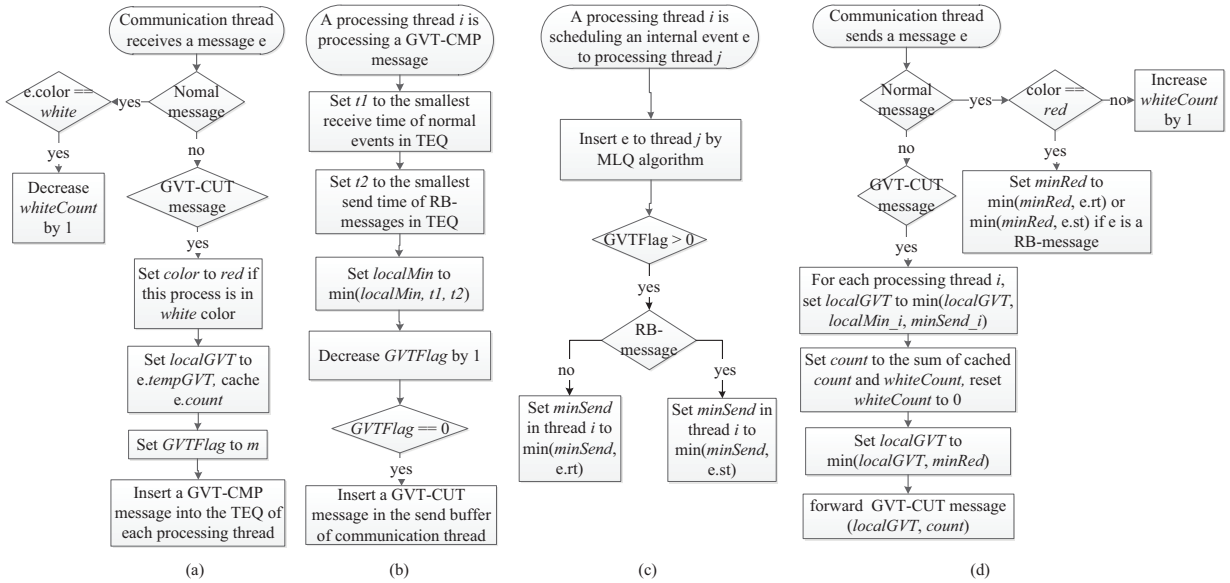


Figure 4: Steps for computing GVT by communication and processing threads in worker processes, rt (st) is receive time (send time) of an event. Control branches that have no impact on GVT are omitted.

A *processing* thread reports its local GVT value during processing GVT-CMP messages in its TEQ as depicted in figure 4(b), and records this value by its *localMin* variable. A local GVT value in a *processing* thread is the minimum between the least receive time of pending events in TEQ and the least send time of RB-messages in the TEQ (note that the current event being processed by this thread is the GVT-CMT event, thus there is no partially processed normal events in this thread). After that this *processing* thread updates the *localMin* value and decreases *GVTFlag* by 1. It implies that all of the *processing* threads in a process have reported their local GVT value when *GVTFlag* equals to zero. The *processing* thread that decreases *GVTFlag* to zero inserts a GVT-CUT message into the send buffer of the *communication* thread to notify local computing has completed, and the steps are shown in figure 4(b). When a *processing* thread schedules normal events to other *processing* threads within the same process, it should check *GVTFlag* and update its *minSend* variable if *GVTFlag* is greater than zero, as illustrated in figure 4(c).

A *communication* thread follows the steps in figure 4(d) to compute local GVT when sending interprocess messages. For normal messages, it counts the number of *white* messages sent and received since the last GVT broadcasting and updates the minimum among receive time (send time of RB-message) of *red* messages sent since the latest round of GVT computing. The final value of local GVT in a process is calculated when the *communication* thread sends out a GVT-CUT message. The *localGVT* value is the minimum among the received *localGVT* and *localMin*, *minSend* in each *processing* thread. Then it forwards a GVT-CUT

message (localGVT, count) to the next *worker* process, resets *count* to zero and *localMin*, *minSend* in each *processing* thread to ∞.

GVT-CUT messages are passed from one *worker* process to another, traverse all of them, and finally return to the *controller* process. The *controller* process checks whether the *count* field in the received GVT-CUT message equals to zero, if it dose a new GVT value *tempGVT* in the received GVT-CUT message is obtained and the *controller* process broadcasts this value to all of the *worker* processes; otherwise (some transient messages were missed in this round) the *controller* process forwards this GVT-CUT message to the first *worker* process and triggers a new round. In the example in figure 3, a GVT-CUT message arrives at *worker* process i in the wall clock time T, the message stamped at 13 is in *red* color, while the *white* message stamped at 10 is sent before T thereby a transient message. The *count* field in The GVT-CUT message that returns to the *controller* process should equal to 1 (implying one transient message is out of consideration), thus another round is needed and then triggered. Since the underlying communication is reliable, all transient messages should be received finally.

A *worker* process becomes *white* when it receives a GVT broadcast message, then it updates present GVT value to the received value, resets *minRed* to ∞, *whiteCount* to zero.

3.3 Correctness Proof

We use the definition of GVT in (Fujimoto 1999).

Definition 1 GVT at wall clock time T is defined as the minimum time stamp among all unprocessed and partially processed messages and anti-messages in the system at wall clock time T.

Definition 2 An event e can be observed by a process at wall clock time T if e is already in the pending events sets in that process at T or e was sent by that process prior to T.

Theorem 1 At any wall clock time T, this asynchronous GVT computing algorithm can determine a GVT value G that meets definition 1.

Suppose p processes numbered by 0 to $p-1$ participate in a simulation, process 0 is the *controller* process, and each *worker* process contains m processing threads. Each worker process can be viewed as a subsystem that employs the shared-memory algorithm in (Fujimoto and Hybinette 1997) to compute local GVT (note that a worker process satisfies the observability condition in (Fujimoto and Hybinette 1997).), the unique difference is that this subsystem can exchange normal messages with other worker processes, which breaks down the closure of this subsystem.To prove theorem 1, we first show lemma 1. The controller process triggers a round of GVT computing at wall clock time T, and assume a new GVT value comes out at wall clock time T_{last}.

Lemma 1 Suppose process i determines a local GVT value GVT_i at wall clock time T_i by this asynchronous algorithm, then any message or anti-message sent by process i after T_i must have a timestamp greater than or equal to GVT_i if no LP in process i would roll back to a time point prior to GVT_i after T_i.

PROOF Proof by contradiction. Assume there is one or more messages or anti-messages with timestamp less than GVT_i that were sent by process i after T_i. Among all such messages, let M be the one containing the smallest timestamp, $TS(M) < GVT_i$, where $TS(e)$ refers to the timestamp of the event e. According to Lemma 1 in (Fujimoto and Hybinette 1997), GVT_i is the minimum timestamp among unprocessed events and anti-messages that can be observed at wall clock time T_i, and no LP in process i would roll back to a time point prior to GVT_i after T_i, then for any LP j in process i $LVT_j \geq GVT_i$, where LVT is the Local Virtual Time of a LP, the inequality $TS(M) < GVT_i$ indicates a LP scheduled an event to its past, which violates Time Warp protocol.

Corollary 1 In lemma 1, any message or anti-message sent by process i after T_i must have a timestamp greater than or equal to GVT_i if no LP receives an message with timestamp less than GVT_i after T_i.

Corollary 1 is direct result form lemma 1, since a LP rolls back to a time point prior to t if and only if it received an event with timestamp less than t and t is less than its local virtual time.

Lemma 2 In lemma 1, process i sends an event with timestamp less than GVT_i after T_i only if at least one LP in process i received an external event with timestamp less than GVT_i after T_i.

PROOF Proof by contradiction. Assume no LP in process i received any external event with timestamp less than GVT_i, process i sends an event M with timestamp less than GVT_i. According to lemma 1 at least one LP LP_x rolled back to a time point prior to GVT_i, according to corollary 1, LP_x received an internal event $M^{'}$ with timestamp less than GVT_i (say it is sent by LP_y). According to Lemma 1 in (Fujimoto and Hybinette 1997) and the above assumption, GVT_i is the minimum timestamp among unprocessed events and anti-messages after T_i, again for any LP j in process i $LVT_j \geq GVT_i$, $LVT_y \geq GVT_i > TS(M^{'})$, that is LP_y scheduled an event to its past.

Obviously, Any event sent after T_i and before T_{last} is in *red* color, its timestamp has been included in $minRed_i$.

Assume process i receives an external event Ex with timestamp less than GVT_i after T_i, $TS(Ex) < GVT_i$, then event Ex can be in either *red* or *white* color.

- If it is in *red* color, $TS(Ex)$ has been included in *minRed* of the sender process;
- If it is in *white* color, it is a transient message, then the *count* value must be greater than 0 when the GVT-CUT message arrives at the controller process, and a new round would be triggered. Note that the underlying communication is reliable, and all of the processes are in *red* color after the first round, thus no *white* message would be created after the fist round, then there must exit a wall clock time T_x after which all of the transient messages are received, thus there is no this case in the round after T_x.

Finally, in the round after T_x, G=min(GVT_i, $minRed_i$), $i = 1, 2, \cdots, p-1$, is a valid GVT value that satisfies definition 1.

3.4 Cost Analysis

As GVT computing is a frequent operation during the whole simulation, it has direct and profound impact on performance, here we analyse two major metrics, computational cost and delay, of our algorithm. According to the number of worker processes in a simulation and the number of threads in each worker process, there are three cases:

Case 1: one worker process and $m(m > 0)$ processing threads in the worker process

This case regresses to the shared-memory algorithm, there are no normal messages between processes, GVT can be calculated in one round. Assume a GVT-CUT message arrives at the worker process at wall clock time T, then the m threads begin to report the minimum timestamp in their respective TEQ (the cost is no more than $X \cdot O(cmp)$, where X is the maximum number of events in the TEQs at that time, $O(cmp)$ is the overhead of comparing an event in TEQ), then update $GVTFlag$ (the cost of updating is $O(1)$, the principle overhead is due to contention, in the worst case the m threads update $GVTFlag$ in series, the cost is $\sum_{i=1}^{m} i \cdot O(CAS) = O(m^2)O(CAS)$, where $O(CAS)$ is the cost of a single lock operation), and compute local GVT by comparing $localGVT$ and $localMin$,$minSend$ in each processing thread ($O(m)$). Hence the total cost is no more than $m \cdot X \cdot O(cmp) + O(m^2)O(CAS) + O(m)$. Another cost comes from updating *minSend* in each thread when a thread sends internal events, and the cost of updating *minSend* is $O(1)$, whereas the principle overhead is due to contention for reading $GVTFlag$. Since $GVTFlag$ can change in the wall clock time interval [T_1, T_2], where T_1 is the wall clock time at which the communication thread receives a GVT-CUT message, T_2 is the wall clock time at which this process complete computing local GVT, using a read-write type lock on $GVTFlag$ can highly decrease this overhead.

Since the processing threads calculate local GVT in parallel, the GVT delay is enslaved to the last thread which completes reporting local value. The delay includes four parts: (a) GVT-CMP events have higher priority than normal events, then the delay of processing GVT-CMP events is $T(e) + X \cdot T(compare)$, where $T(e)$ is the time of processing a normal event, $T(compare)$ is the time of comparing an event in TEQ;

(b) delay of waiting for updating $GVTFlag$, it is no more than $m \cdot T(CAS)$, where $T(CAS)$ is the time of a single lock operation; (c) delay of computing local GVT, i.e. finding minimum among received $tempGVT$ and $localMin$, $minSend$ in each thread; (d) delay of transferring GVT value between the controller and worker process. The sum of the maximum of these four parts gives an upper bound of GVT delay.

Case 2: $n(n > 1)$ worker processes and one processing thread in each worker process

This case regresses to the distributed-memory algorithm. There is no inter-thread messages within a worker process, then the cost and delay of computing GVT is proportional to the number of rounds of GVT-CUT message passing. A new GVT value can be computed only if all of the transient messages sent after last updating of GVT are received, whereas it entirely depends on the underlying communication. Hence in the worst case, it needs a few rounds until all of the transient messages arrive at receiver. However it is not worse than Mattern's original algorithm.

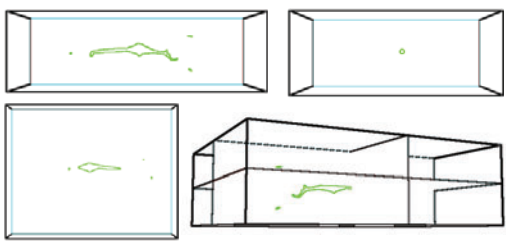
Case 3: $n(n > 1)$ worker processes and $m(m > 1)$ processing threads in each worker process

This case is a hybrid of case 1 and 2 (and also NTW-MT designed for), according to the analysis in case 1 we know a worker process can calculate a local GVT value in a bounded period, while analogous to case 2 a new GVT value can be computed only if all of the inter-process transient messages are received, thereby it may take several rounds of GVT-CUT message passing.

Assume the total number of processing threads is a fixed value, there is less inter-process communication in case 3, thus it is promising to compute a GVT value in fewer rounds of GVT-CUT message passing, compared to case 2.

4 EXPERIMENTAL STUDY

We simulate the intracellular Ca^{2+} wave (a brief introduction can be found in (Neymotin, McDougal, Sherif, Fall, Hines, and Lytton 2015)) in an unbranched apical dendrite of a hippocampal pyramidal neuron (length: 1000 μm, diameter: 1 μm) as shown in figure 5(a). 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. As the real Ca^{2+} wave model is complex we simplified it by assuming (a) a IP_3 receptor opens when the concentration of IP_3 and Ca^{2+} are both higher than some respective threshold (b) an opening IP_3 receptor channel will close in a period of time determined by an exponential distribution. The reactions are shown in figure 5(b), where Ca^{2+}_{er} refers to Ca^{2+} in Endoplasmic Reticulum (ER), Ca^{2+}_{cyt} refers to Ca^{2+} in cytosol, $[\bullet]$ refers to the concentration of the corresponding species $\bullet$, $m = [IP_3]/([IP_3] + k_{IP_3})$, $n = [Ca^{2+}_{cyt}]/([Ca^{2+}_{cyt}] + k_{act})$, k_{IP_3}, k_{act}, ν_{IP3R}, ν_{leak}, ν_{SERCA} and k_{SERCA} are given constant parameters, the value can be found in (Neymotin, McDougal, Sherif, Fall, Hines, and Lytton 2015). Ca^{2+}_{er} can only diffuse within ER, while cytosolic Ca^{2+} and IP_3 can only diffuse within cytosol.



(a) Geometry.

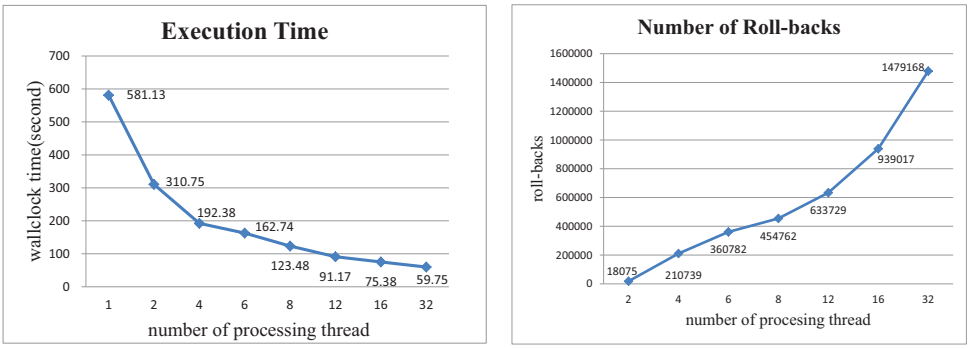
$$\text{ER Release: } Ca^{2+}_{er} \xrightarrow{k_{release}} Ca^{2+}_{cyt},$$
$$k_{release} = \nu_{IP3R}\, m^3\, n^3\, ([Ca^{2+}_{er}] - [Ca^{2+}_{cyt}])$$
$$\text{ER Leak: } Ca^{2+}_{er} \xrightarrow{k_{leak}} Ca^{2+}_{cyt},$$
$$k_{leak} = \nu_{leak}\, ([Ca^{2+}_{er}] - [Ca^{2+}_{cyt}])$$
$$\text{SERCA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

(b) Calcium wave reactions.

Figure 5: A three-dimensional view of the pyramidal neuron and the calcium wave reactions.

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, Red Hat Enterprise Linux

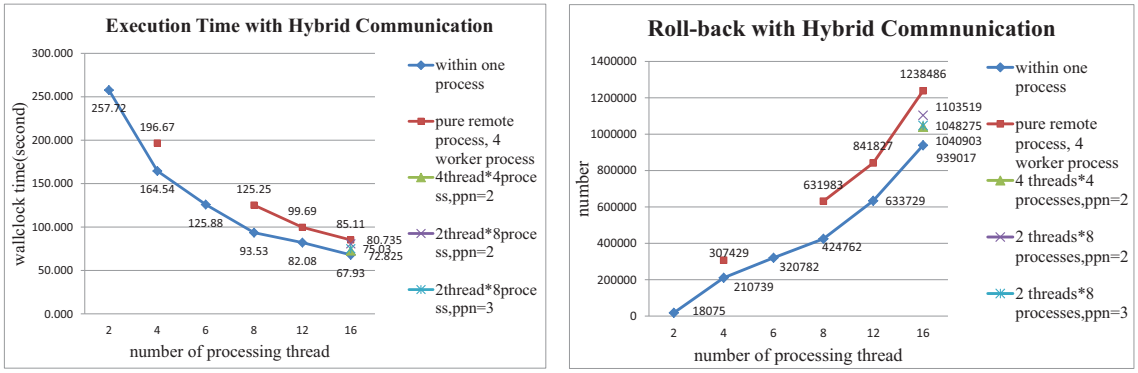
Server release 6.4 (Santiago). The other is the SW2 node (of Guillimin at McGill HPC center), consisting of two Dual Intel(R) Sandy Bridge EP E5-2670 2.6 GHz CPUs, 8 cores per processor, 8 GB of memory per core, and a Non-blocking QDR InfiniBand network with 40 Gbps between nodes. The node runs Linux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



(a) Execution time. (b) Roll-back.

Figure 6: Execution time and roll-back of simulating the calcium wave model in the PEPI machine.

In figure 6(a) and 6(b), all processing threads are placed in one process, which is the case 1 in section 3.4. A valid GVT can be computed in a bounded duration by one round of GVT-CUT message passing as analysed above.



(a) Execution time. (b) Roll-back.

Figure 7: Execution time and roll-back of simulating the calcium wave model in the Guillimin machine, ppn refers to process per node.

The within one process mode in figure 7(a) and 7(b) is the case 1 in section 3.4 (same as the PEPI scenario). The execution time is the shortest along with the lowest roll-back due to short communicational latency compared to the other modes. As pointed out above, a GVT value can be computed in a bounded duration, which also contributes to shortening the overall execution time.

The data point with 4 processing threads in the pure remote process mode in figure 7(a) and 7(b) is the case 2 in section 3.4, in which each worker process hosts only one processing thread. Only the distributed memory algorithm is used in this scenario, and a valid GVT value can be computed until all transient message sent since last GVT broadcasting arrives at the receiver, thus it may takes more than two rounds of GVT-CUT message passing.

The other curves in figure 7(a) and 7(b) belong to case 3 in section 3.4, in which both the shared and distributed memory algorithm is used. Since inter-process communication is used, it may take several rounds

of GVT-CUT message passing to compute a GVT value. Considering the data points with 16 processing threads, we can conclude placing more threads in the same process results in better performance, and this is reasonable for less interprocess communication is used. As analysed in section 3.4, fewer rounds of GVT-CUT message passing are used, which also improves the overall performance.

5 CONCLUSION AND FUTURE WORK

We introduced multi-threaded extension to NTW (Patoary, Tropper, Lin, McDougal, and Lytton 2014) to produce a high performance PDES simulator for reaction and diffusion simulation in NEURON project (Carnevale and Hines 2006). To avoid overhead of blocking threads in a worker process, it is intended to compute a local GVT asynchronously among threads within a process, which gives the probability of simultaneous reporting problem. We combine the shared-memory algorithm in (Fujimoto and Hybinette 1997) (to compute a local GVT value in each worker process) and Mattern's (Mattern 1993) algorithm (to trace inter-process messages) to compute GVT in NTW-MT. GVT is computed asynchronously both within and among processes, which is the first try in multi-threaded PDES as far as we know. We proved that our proposed GVT computing algorithm can determine a valid GVT value G that meets definition 1 at any wall clock time during simulation. We also analysed the computational cost and delay of this algorithm: (a) it can compute a GVT value in a bounded period if all of the processing threads are in a single process; (b) it may take more time to compute a GVT value if more than one worker process are used, and the overhead and delay highly depends on the underlying communication.

Our future work on the this GVT algorithm is to have larger scale experiments. As far as the project goes, we are planning to implement a more detailed calcium wave model and develop load balancing algorithms for NTW-MT. A hybrid (deterministic-stochastic) model is another future effort.

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).

REFERENCES

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

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

Carnevale, Nicholas T and Hines, Michael L 2009-2013. "NEURON, for empirically-based simulations of neurons and networks of neurons". http://www.neuron.yale.edu. Last access on May 1st 2015.

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

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

Fujimoto, R. M. 1999. *Parallel and Distribution Simulation Systems*. 1st ed. New York, NY, USA: John Wiley & Sons, Inc.

Fujimoto, R. M., and M. Hybinette. 1997. "Computing global virtual time in shared-memory multiprocessors". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 7 (4): 425–446.

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

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

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

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

Mattern, F. 1993. "Efficient algorithms for distributed snapshots and global virtual time approximation". *Journal of Parallel and Distributed Computing* 18 (4): 423–434.

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

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

Neymotin, S. A., R. A. McDougal, M. A. Sherif, C. P. Fall, M. L. Hines, and W. W. Lytton. 2015, March. "Neuronal Calcium Wave Propagation Varies with Changes in Endoplasmic Reticulum Parameters: A Computer Model". *Neural Computation* 27 (4): 898–924.

Patoary, M. N. I., C. Tropper, Z. Lin, R. McDougal, and W. W. Lytton. 2014. "Neuron Time Warp". In *Proceedings of the 2014 Winter Simulation Conference*, WSC '14, 3447–3458. Piscataway, NJ, USA: IEEE Press.

Pellegrini, A., and F. Quaglia. 2014, October 22-24. "Wait-Free Global Virtual Time Computation in Shared Memory TimeWarp Systems". In *IEEE 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 9–16. Paris, France: IEEE.

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

Samadi, B. 1985. *Distributed Simulation, Algorithms and Performance Analysis (Load Balancing, Distributed Processing)*. Ph. D. thesis. AAI8513157.

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

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

Wang, B., B. Hou, F. Xing, and Y. Yao. 2011. "Abstract Next Subvolume Method: A logical process-based approach for spatial stochastic simulation of chemical reactions". *Computational biology and chemistry* 35 (3): 193–198.

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

AUTHOR BIOGRAPHIES

ZHONGWEI LIN is a Ph.D. candidate of College of Computer at National University of Defense Technology. He received M.S. degree in Computer Science and Technology from National University of Defense Technology. His research area is High Performance Simulation.His email address is zwlin@nudt.edu.cn.

YIPING YAO is a Professor of Computer Science and Technology in College of Information System and Management at National University of Defense Technology. His research interests include: Parallel and Distributed Simulation, Agent-based and Component-based modeling and simulation and hard-in-the-loop realtime simulation. His email address is yipingyao@qq.com.