Chinese Journal of Electronics
Vol.32, No.5, Sept. 2023

Towards Order-Preserving and Zero-Copy Communication on Shared Memory for Large Scale Simulation

LI Xiuhe, SHEN Yang, LIN Zhongwei, ZHAO Shunkai, SHI Qianqian, and DAI Shaoqi

(*College of Electronic Engineering, National University of Defense Technology, Hefei 230031, China*)

Abstract — Parallel simulation generally needs efficient, reliable and order-preserving communication. In this article, a zero-copy, reliable and order-preserving intra-node message passing approach named ZeROshm is proposed. This mechanism partitions shared memory into segments assigned to processes for receiving messages. Each segment consists of two levels of index L1 and L2 that recordes the order of messages in the host segment, and the processes also read from and write to the segments directly according to the indexes, thereby eliminating allocating and copying buffers. As experimental results show, ZeROshm exhibits nearly equivalent performance to message passing interface (MPI) for small message and superior performance for large message. Specifically, ZeROshm costs less time by 43%, 40% and 55% respectively in pure communication, communication with contention and real PHOLD simulation within a single node. Additionally, in hybrid environment, the combination of ZeROshm and MPI also shorten the execution time of PHOLD simulation by about 42% compared to pure MPI.

Key words — Zero-copy, Order-preserving, Shared memory, Parallel simulation, Message passing interface, Interprocess communicaton.

I. Introduction

Communication plays an essential role in any parallel simulation, where the main procedure is to process events (messages) in causal order. Connecting multi-core processors into a cluster (called multi-core cluster, MCC) has been a dominant computing architecture which is used to run parallel simulation. Compared to the cluster connecting single-core processors, MCC can provide deeper communication hierarchy, i.e., cores within a processor transfer data through special channels among them, which can achieve fast communication through proper layout of tasks. Threads within a process can share data without transforming addresses by operating system, thereby multi-threaded application has been the primary choice for communication-intensive applications, to make full use of fast channels among cores [1]–[4].

Parallel simulation proposes requirements to underlying communication, including 1) Reliability, it never lose any data, and a message reaches its destination before the end of simulation, which is a mandatory condition to the correctness of simulation; 2) Order-preserving, it receives messages in sent-out order between a pair of sender and receiver, which is a optional need to the correctness, as time management can control the process of disordered messages in causal order at the price of extra overheads, e.g. delay and computation of synchronizing messages; 3) Zero-copy, it never copies the whole body of the message, which is not mandatory but can reduce cost and overheads significantly.

In modern operating system (OS), a process generally uses different logic address, which prevents sharing data among processes directly even if those reside in an individual physical machine (node). As a result, interprocess communication (IPC) is employed to transfer data among processes.

The message passing interface (MPI) is the most-widely used IPC, and it employs the so called two-sided communication [5] to transfer data from one process to another, which copies data twice. Shared memory is proved to be the fastest IPC medium [6], [7] within a single machine. The present dominant communication mid-wares based on shared memory can be classified into two categories: 1) Simply employing shared memory

Manuscript Received Nov. 9, 2021; Accepted Feb. 11, 2022. This work was supported by the National Natural Science Foundation of China (61802422).
© 2023 Chinese Institute of Electronics. DOI:10.23919/cje.2021.00.393

as transient space, leading to copy operation between the source and destination process, e.g. MPI; 2) Data reside in shared memory, while are read and written in first-in-first-out (FIFO) order, which can lead to violation of data order when a process is switched during reading and writing the data, e.g. ZIMP [8] and shared memory based RTI [9]. Moreover, the FIFO policy may lead to damage to performance of parallel simulation, as demonstrated in Fig.1.

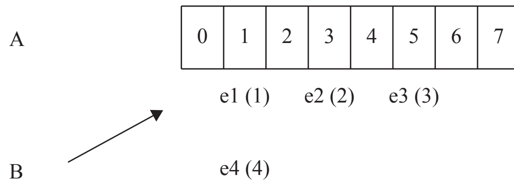

Fig. 1. Example of FIFO policy.

In Fig.1, the process B has sent 3 messages (with timestamp in brackets) to process A, after that message e1 is reclaimed and returned to contain e4, in this circumstance process A prefers to receive e4 in prior to e2 and e3, which in essence gives lower priority to messages with lower timestamp. As a result correct lower bound of time-stamp (LBTS) [10] can be computed only when the message with global minimum timestamp is received, which consequently increases the execution time of simulation.

In this work, a zero-copy, reliable and order-preserving mechanism (ZeROshm) was proposed on shared memory for large scale parallel simulation oriented to MCC. In ZeROshm, shared memory is partitioned into segments by which a process exchanges and stores messages, and each segment consists of two levels of index L1 and L2 that both control the use of space in the host segment and record the order of messages. Thereinto, L1 index marks the start and end point of L2 index in use, and L2 index directs to the space that contains pending messages, as a result it never copy message body. Sending and receiving message operates on L1 and L2 index, and a process reads from and writes to shared memory directly, thereby eliminating allocating and copying buffers. As experimental results show, ZeROshm exhibits nearly equivalent performance to MPI for small message and superior performance for large message. Specifically, ZeROshm costs less time by 43%, 40% and 55% respectively in pure communication, communication with contention and real simulation within a single node. In hybrid environment, the combination of ZeROshm and MPI also shows better performance by about 42% than pure MPI.

The rest of this article is organized as follows. Section II provides an overview of previous studies and their limitations. We describe the structure and procedures of ZeROshm in Section III, and then prove the correctness and analyze the cost of ZeROshm in Section IV. Experimental results and evaluation is represented in Section V, followed by conclusion in Section VI.

II. Related Work

Compared to general applications, parallel simulation is special in terms of requirements to communication, i.e., it needs reliable and order-preserving communication for correctness and performance [10]. In fact, order-preserving is not a rigid requirement on communication for correctness in modern simulation systems, as both conservative and optimistic policy can handle the disordered arrival of messages. In conservative policy, all logic processes (LPs) wait until a new LBTS is updated, and the calculation of LBTS must trace all messages, which prevents violation of causal order. Optimistic simulation rolls back to a past time point when violation of causal error is detected, e.g. a straggler message arrives. However those solutions to disordered message leads to significant degeneration in performance for simulation execution through prolonging the time of computing LBTS (keeping LPs idle meantime) in a conservative simulation and bringing in extra straggler message in optimistic policy. Zero-copy attribute of communication mechanism can significantly decrease the cost of transferring messages, thereby improve the performance of applications including parallel simulation.

As we know, transmission control protocol/internet protocol (TCP/IP) is the most typical reliable and order-preserving mechanism used in network communication, which is also effective for IPC within a node. However, copying buffer is always employed in TCP/IP protocol. Standard MPI is based on TCP/IP, while it is possible to avoid copying by skillfully-designed layout and usage of data to be transferred in special applications. Hoefler *et al.* [11] proposed zero-copy algorithms for fast Fourier transform and conjugate gradient using MPI datatypes and observed significant speedups up to a factor of 3.8. The MPI 3.0 standard [12] introduces new process-level interfaces to allocate space directly on shared memory (i.e., MPI_Win_allocate_shared) and to synchronize processes to access those data (i.e., MPI_Win_sync and MPI_Win_shared_query), making it possible to eliminating coping buffer. As tested and analyzed in [5] and [13], compared to the two-side version (namely MPI-2.x), MPI 3.0 promotes communication performance of transferring data up to 85%. The shared memory version MPI provides basic support to construct zero-copy communication, while extra control of receiving those data is necessary to achieve order-persevering.

Aublin *et al*. proposed ZIMP [8] which constructs channels (actually a circular buffer) on shared memory, and then a process can allocate memory from channels and others processes also read from channels without any copying. Moreover, it also achieves one-to-many communication. As tested and reported, ZIMP consistently improves the performance of the state-of-art mechanisms by up to one order of magnitude. However, messages in ZIMP are organized into a circular buffer and receivers read messages in arbitrary order, which leads to possibility of breaking down the message order-ZIMP is not order-preserving.

Swenson *et al*. [14] proposed a new approach to support zero-copy inter-process communication. Each logic process (LP) holds an allocator which can allocate memory (called "heap item") from shared memory, and smart pointer to heap item is sent to receiver. Each heap item holds the identifier of host LP which can be used to reversible allocation in case of rollback. They compared the performance among MPI full copy, Boost shared memory object (also zero-copy) and their approach, and concluded: 1) The execution time of zero-copy approach is nearly constant regardless of data size; 2) The general-purposed design and implementation of Boost leads to noticeable degeneration in performance; 3) MPI full copy performs better when massage size is small (approximately 3,000 bytes).

Each logic process in [14] holds an array of shared heap pointers (called "HeapCluster") to identify the sender LP of a message, whereas the memory consumption and the time of recognizing smart pointer are proportional to the number of LPs in the simulation. In other works, high memory consumption and long recognizing time should be inevitable as the number of LPs increases without extra optimization.

III. ZeROshm Design

We are achieving zero-copy, reliable and order-preserving communication on shared memory. To cover these requirements, we consider as follows:

1) Accessible: a shared memory block should be allocated and initialized properly in prior to simulation execution;

2) Zero-copy: all messages should reside in shared memory, and processes within a node write and read message directly by pointer or reference;

3) Reliable: messages can reach the destination and keeps unchanged before read;

4) Order-preserving: the order of sending message to a process should be recorded, and the receiver process reads messages according the recorded order.

Before the allocation of shared memory, it is necessary to know the number of processes in each node and the distribution of those processes which is used to calculate the size of shared memory block and distinguish local and remote messages.

1. Construction of routing map

As a process communicates to both local and remote processes, it needs route messages before actually sending them out. This can be done by querying routing map which tells the host node of an individual process. Suppose we have 2 nodes, node0 and node1, and create 8 processes in total and evenly distribute them to run simulation, which forms hybrid (distributed and shard memory) communication architecture as shown in Fig.2.

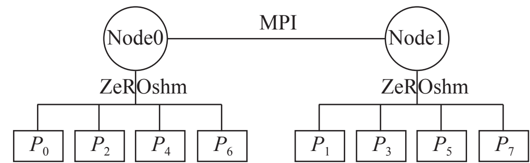

Fig. 2. Hybrid communication architecture.

At first, each process reports the hostname of its host node and identifier number (also named *rank* in MPI) to the controller, say P_0; and then P_0 groups the reports by hostnames; after that we can have n sets (n is the number of nodes), and each set contains the ranks of process in that node; and finally, the sets are serialized into a special message tagged *Topology* and sent to all processes. When another process receives the *Topology* message, it constructs a map $\boldsymbol{R}$ containing $\langle rank, path\rangle$, where *path* can be 0 (local) or -1 (remote), by comparing the local hostname and hostnames in the message. The process P_0 in Fig.2 can own a routing map item $\langle 1, -1\rangle$ and $\langle 2, 0\rangle$.

2. Allocation

For a node containing c processes numbered by $0, 1, \ldots, c-1$, we call the processes a family, and one process is assigned as family controller, which is in accordance with the design of NTW-MT [1], [2] and double-indexed shared memory [15]. Family controller can allocate a block of shared memory, and each process use a part of the block as so called a segment to contain messages sent to it, the access to which is controlled by the structure shm_entry as shown in Fig.3.

The structure shm_entry consists of three parts, including reading area prefixed by "r", writing area prefixed by "w", and message storage area. The reading area and the writing area have the same components, including a mutex, two integers, and an array of integers, which contribute to recording message orders. For each area, taking the reading area as an example, rMutex cares of exclusive access to the reading area, rBeginIndex and rEndIndex (L1 index) respectively indicates the start and end point of pending indexes in

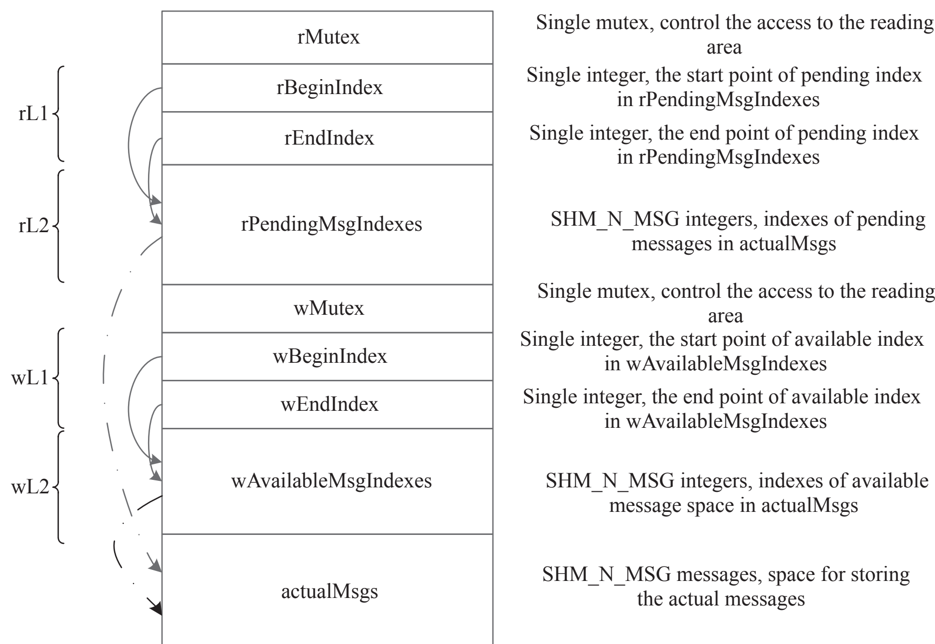

Fig. 3. Structure shm_entry for control the access to a segment.

rPendingMsgIndexes (L2 index) that stores the indexes of pending messages in actualMsgs, and actualMsgs contains SHM_N_MSG empty messages. Then the size of the block can be easily computed according to background platform. Suppose the operator sizeof(·) computes the size of variables, the size of an shm_entry equals $L_{\text{entry}} = 2 \times \text{sizeof(mutex)} + 4 \times \text{sizeof(integer)} + 2 \times \text{SHM_N_MSG} \times \text{sizeof(interger)} + \text{SHM_N_MSG} \times \text{sizeof(message)}$, then the family controller allocates $L_{\text{node}} = c \times L_{\text{entry}}$ memory in total. Once when the block is allocated and partitioned, family controller also decides the segment assignment $\langle rank, seg\rangle$, where seg indicates the segment number and can be $0, 1, \ldots, c-1$, and then each member can calculate the segment offset in the block $offset_{\text{seg}} = seg \times L_{\text{entry}}$. Finally, the family controller sends segment assignment to all members, and each member update its own routing map by replacing the second value in routing map item with the segment number of family members. In Fig.2, suppose P_0 and P_2 is assigned to used the first and second segment respectively, then P_0 updates the routing map items like $\langle 0, 0\rangle$ and $\langle 2, 1\rangle$.

3. Initialization

The family members attach to shared memory and initialize variables in their own shm_entry: construct and initialize mutex, set rBeginIndex, rEndIndex and wBeginIndex to 0, set wEndIndex to SHM_N_MSG−1, set all integers in rPendingMsgIndexes to 0, set integers in wAvailableMsgIndexes to 0, 1, . . . , SHM_N_MSG−1 in turn, at last clear all message space.

4. Path of sending and receiving messages

LPs interact with each other by message. Suppose an LP_x sends a message to LP_y, LP_x firstly queries the identifier of LP_y by name service, after that it continues to query the host process P_y of LP_y by partition service which controls distribution of LPs among processes, and then queries the communication path to process P_y according to the routing map which has been constructed, as demonstrated in Fig.4. As sending and receiving message via MPI has been fully introduced in wide literature, we are mainly representing the procedures of ZeROshm in this article.

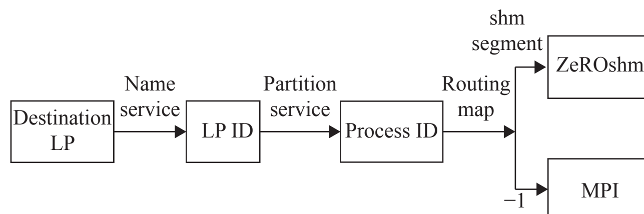

Fig. 4. Conversion path for sending a message.

5. Send messages

To send a message to family member P_r which uses the seg-th segment, the sender P_s directly writes to the seg-th segment as described in Algorithm 1, where all variables resides in the shm_entry of the receiver process, i.e. seg-th shm_entry.

Algorithm 1 Steps of sending messages

1: Lock wMutex, and load wBeginIndex and wEndIndex;
2: if wBeginIndex == wEndIndex then
3: Unlock wMutex {no available space};
4: Wait for available space and try again;
5: else

```
6:    pos ← wBeginIndex {L1 index};
7:    wBeginIndex++;
8:    wBeginIndex mod SHM_N_MSG;
9:    Unlock wMutex;
10:    index ← wAvailableMsgIndex[pos] {L2 index};
11:    e ← actualMsgs[index];
12:    e ->shm_pos ← index;
13:    Sender writes message to the space pointed by e;
14:    Lock rMutex;
15:    rPendingMsgIndexes[rEndIndex] ← index;
16:    rEndIndex++;
17:    rEndIndex mod SHM_N_MSG;
18:    Unlock rMutex;
19: end if
```

In Algorithm 1, the main steps of sending a message include finding space and submitting it to receiver. In L1 index, the index interval [wBeginIndex mod SHM_N_MSG, wEndIndex mod SHM_N_MSG] tells the start and end position which stores available indexes in wAvailalbeMsgIndexes, then the sender fetches the first available index, i.e. wBeginIndex, and records it by a temporary variable *pos*. In L2 index, the index interval available in wAvailableMsgIndexes, the sender fetches the integer with offset *pos* and records to a temporary variable *index*, then the actual space is the *index*-th element in actualMsgs. It should be noted that a message has a component shm_pos set by *index* for space reclaiming as represented in the algorithm in Section III.7.

After the space is found, the sender can write message content to shared memory, then submit it by writing the message index into the reading area, hence the message becomes perceptible to the receiver process.

6. Receive messages

Each process also looks up its own reading area and receives all perceptible messages as listed in Algorithm 2.

Algorithm 2 Steps of receiving messages

```
1: Lock rMutex, and load rBeginIndex and rEndIndex;
2: if rBeginIndex == rEndIndex then
3:    Unlock rMutex and return {no perceptible message};
4: else
5:    pos ← rBeginIndex;
6:    rBeginIndex++;
7:    rBeginIndex mod SHM_N_MSG;
8:    Unlock rMutex;
9:    index ← rPendingMsgIndexes[pos];
10:   e ← actualMsgs[index];
11:   Enqueue e into the destination LP;
12: end if
```

Receiving messages also includes twice access to indexes. It should be noted that algorithm of receiving messages in Algorithm 2 returns a pointer of message which locates at shared memory (thereby no buffer copy is needed), and the received message becomes pending event of the destination LP after properly enqueued.

7. Reclaim message space

The space occupied by messages is reclaimed in fossil collection (optimistic policy) or once when the message is processed (conservative policy). The main steps of reclaiming message space is to return index into wAvailableMsgIndexes as listed in Algorithm 3.

Algorithm 3 Setps of reclaiming message space

```
1: Lock wMutex;
2: wAvailableMsgIndexes[wEndIndex] = e.shm_pos;
3: wEndIndex++;
4: wEndIndex mod SHM_N_MSG;
5: Unlock wMutex.
```

Free space is added to the end of available messages, and thus can be allocated again in the algorithm of sending messages. In fact the access to actualMsgs is definitely controlled by L1 and L2 indexes, hence the order of empty messages does not matter the procedures of ZeROshm at all.

Here we summarize how ZeROshm satisfies the zero-copy, reliable and order-preserving requirements proposed at the beginning of Section III.

1) Accessible: All processes attach to shared memory and thereby can access all data there.

2) Zero-copy: Message resides in the shared memory block, hence reading and writing message is operated by pointers, as a result no data is copied during the whole procedure.

3) Reliable: Messages are tightly aligned, and all writing checks the bound of adjacent messages to get rid of any breakdown to message body.

4) Order-preserving: The r-prefixed indexes, i.e. rBeginIndex and rPendingMsgIndexes, forms a cyclic queue which records the orders of message sent to the host process; The w-prefixed indexes, i.e. wBeginIndex and wAvailableMsgIndex, also forms a cyclic queue and accepts free space at any time, which is suitable for asynchronous fossil collection in optimistic simulation.

IV. Correctness Proof and Cost Analysis

As analyzed in Section II, parallel simulation generally needs reliable and order-preserving message passing. In this section, we prove the reliability and order-preserving property, and also analyze the cost of ZeROshm.

1. Proof to reliability and order-preserving

Definition 1 A message passing mechanism is reliable if any message can arrive its specific destination and the content keeps unchanged.

Definition 2 A message passing mechanism is order-preserving if messages sent firstly are received firstly between a pair of sender and receiver.

Theorem 1 (Reliability) ZeROshm achieves reliable message passing.

Proof The reliability of ZeROshm highly depends on stable access to shared memory, here we assume operating system provides stable service to shared memory, lock and unlock operation in ZeROshm (error and fault can be detected).

- Reachability: In ZeROshm, any process can access the full space of shared memory, hence a sender process find room (equivalent to index) and push the index into the pending message queue (also resides in shared memory) of receiver, then the receiver process can read the message according to the indexes in queue.
- Fidelity: A message should not be broken down by another writing before reclaimed by the receiver, and this can be proved by contradiction. Note that space in actualMsgs is strictly aligned in message format and mutexes prevent concurrent operation on an individual segment, then a message, say e_{broken} is broken down only if the later writing of message, say e_{occupy}, gets the same index in actualMsgs, i.e. line 11 in Algorithm 1. And given that all L2 indexes are set to $0, 1, \ldots,$ SHM_N_MSG−1, which are different from each other, then the later writing gets the same L1 index, i.e. line 10 in Algorithm 1, and then it also gets the same value of wBeginIndex, i.e. line 6. However, wBeginIndex keeps increasing and moded by SHM_N_MSG: It reaches the same position until it exceeds SHM_N_MSG and turns back. As both wBeginIndex and wEndIndex are moded by SHM_N_MSG, wBeginIndex must stride across the wEndIndex position, which would be prevented by judgment between them (line 2 in Algorithm 1) .
- No loss of message: Given that all reading from and writing to the shared memory is reliable and the processes within a node would have the same view of orders of memory access, any message written by the sender process in actualMsgs and indexed at a position in rPendingMsgIndexes can be perceived and accessed by the receiver process in a reversed manner.

As modulus operator is used in ZeROshm, we use the concept "left" to mark position of indexes in cyclic queues. For modulus N, let B and E as the start and end position, for any integers $a, b \in [0, N)$, we say a is on the left of b if

$$\begin{cases} a < b, & B < E \\ B < a < b, a < b < E, b < E \& a > B, & B > E \end{cases}$$

and the symmetry (i.e., the concept "right") can be inferred.

Lemma 1 (L2 index position) For any segment, rBeginIndex (rEndIndex) is on the left (right) of any perceptible messages in rPendingMsgIndexes.

Proof Under the control of memory consistency [16], processes residing in the same node should hold the same order of writings to memory, which prevents disorder of readings. In Algorithm 1, each submission increases rEndIndex (line 16), then rEndIndex locates the right of the latest message in rPendingMsgIndexes, naturally. For reading messages from shared memory in Algorithm 2, rBeginIndex is initialized to 0 and increases only if it reads a message (line 6 in Algorithm 2), then rBeginIndex must locates the left of the earliest message in rPendingMsgIndex.

Theorem 2 (Order-preserving) ZeROshm is order-preserving.

Proof Proof by contradiction. Suppose process A sent message e_1 and e_2 in turn to process B, while process B received message e_2 first. The index of the two messages $e_1.pos$ and $e_2.pos$ in rPendingMsgIndexes can be in two cases as follows:

$e_1.pos < e_2.pos$: Process B received e_2 first, which indicates rBeginIndex passed the position $e_2.pos$ first, then we have rBeginIndex $> e_1.pos$ – rBeginIndex is on the right of e_1, which violates Lemma 1.

$e_1.pos > e_2.pos$: It follows the second branch of the left definition, and rBeginIndex also locates on the right of e_1.

2. Cost analysis

The primary cost of ZeROshm comes from the overhead of locking on mutex and integer operations. For sending message, it involves 2 locking and 2 unlocking of mutex, 5 integer assigning, 2 integer increment, 2 integer modulus and writing of message content, then the total time should be the sum of the above components. The cost of integer operations can be generally considered as constant in a specific run, while the time of locking depends on the number of processes that compete an individual mutex, in the worst case a process waits $\sum_{i=1}^{c} i \cdot O(\text{CAS}) = O(c^2)O(\text{CAS})$ until it locks the mutex successfully, where $O(\text{CAS})$ is the overhead of a single locking.

Receiving a message involves 1 locking and unlocking of mutex, 3 integer assigning, 1 integer increment and 1 modulus. Here we can infer that, for small messages, it is possible that ZeROshm costs more time than traditional copy-based communication in specific circumstances, e.g. MPI, since copying of small buffer is adequately optimized and message merging is generally in use.

V. Experiment and Evaluation

Attribute to ZeROshm, NTW-MT [1] now uses hybrid communication, i.e., intra-node message is sent via zero-copy access to shared memory while inter-node message is sent by MPI. In this section, we test and analyze the effect of ZeROshm to communication and real simulation.

1. Effect to communication

ZeROshm is designed for intra-node communication, thus we set a bunch of processes within a node to exchange messages and collect the total time, and then calculate the average time of receiving a single message. The hardware and software environment is listed in Table 1. We use non-blocking MPI interfaces, to be more specific, it uses MPI_Isend to send out the message and calls MPI_Wait to wait completion, and it firstly calls MPI_Iprobe to check if there are messages to receive and uses MPI_Irecv to receive data actually. All global parameters of MPI environment are set by default value.

Table 1. Environment used in intra-node communication

Item	Configuration
CPU	Intel i7 with 8 cores
Frequency	1.8 GHz
Cache	32 KB, 256 KB and 8192 KB
Memory	16 GB
OS	Linux 4.8.0 - 36 - generic x86_64
GCC	5.4.0
MPI	MPICH 3.3

1) Pure point-to-point message passing

To reduce the impact of contention on segments, we set only a pair of processes in this scenario, and each process sends and receives 10,000 messages and the time is averaged from 5 runs as demonstrated in Fig.5.

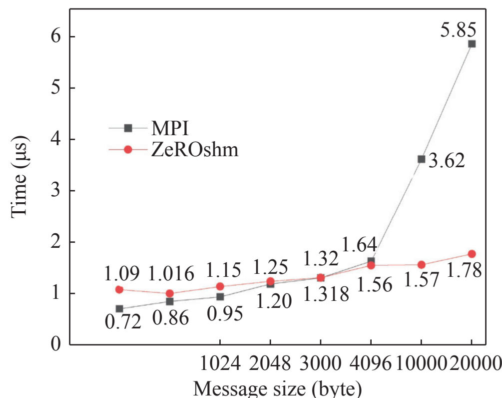

Fig. 5. Time of transferring a message.

As shown in Fig.5, MPI costs less time (about 20% on average) than ZeROshm for small message, e.g., 256 and 1024 bytes. As analyzed in Section IV, the primary cost of ZeROshm is the overhead of locking on mutex, and sending a message involves twice locking on mutex, which could introduce higher overhead than allocating small buffer by MPI that uses internal memory pool [17] to get rid of the overhead of small-sized (generally less than page size) allocation from heap. In this circumstances, the extra overhead of ZeROshm from OS is higher than MPI, as a result ZeROshm costs more time than MPI for small-sized messages. However, the time of transferring large message via MPI shows a sharp increase, i.e., it almost doubles as message size doubles up especially when message size exceeds page size (4096 bytes in our environment). The primary explanation to the sharp increase lies in that MPI takes much time to allocate large buffer for transiting message, and it can be naturally inferred MPI costs much more time for larger size than the ones in Fig.5. The time consumed by ZeROshm almost shows linear increase as message size. Note that this scenario involves only two processes, thus the contention on segments is low; therefore, the locking on mutex and integer operations cost constant time as described in Section IV, and the increased time is used for writing message content to shared memory.

2) Contention involved

When a few processes access shared memory concurrently, contention on segments can happen, which increases overhead of locking on mutexes directly. In this scenario, we vary the number of processes and message size to show the impact of contention as demonstrated in Figs.6 and 7. For simplicity, we only consider two message size, 1024 bytes and 16384 bytes, as representatives of small and large size. To create contention, all processes are organized into a virtual ring, and each process receives message from and sends message to one of its neighbor, i.e., the previous and next process in the ring, which leads to at most 3 processes accessing an individual segment simultaneously.

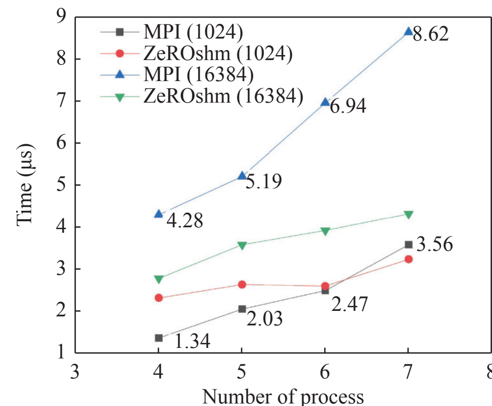

Fig. 6. Time of transferring a message when contention involved.

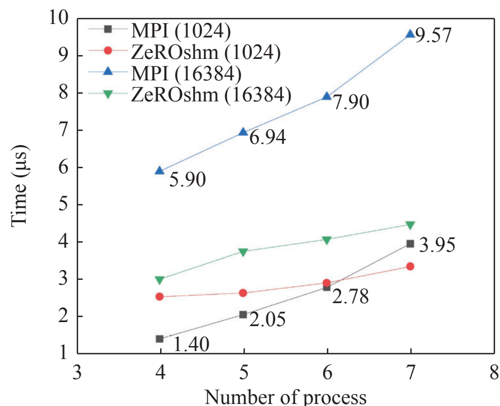

Fig. 7. Time of transferring a message in worst cases.

From Fig.6, both MPI and ZeROshm takes much more time to transfer a single message when contention happens. For small messages, MPI costs less time (about 13% on average) than ZeROshm when few processes are involved (consistent with that in the pure point-to-point communication circumstance), while the two curves shows a reverse case when a few processes, say 7 processes, are introduced. One primary explanation to the crossover is as follows, since allocation of buffer is critical operation, the overhead of simultaneous allocation of buffers increases as the number of processes increases in MPI, while ZeROshm never allocates buffers during the whole procedure. For large messages, MPI always takes more time than ZeROshm (about 40% on average, and shows higher difference when a few processes are included), since overhead of allocating large buffer and contention is much high as analyzed in Section V.

As stated in Section IV, uncertainty of ZeROshm lies in locking on mutex, i.e., the waiting time of access to segments can be long when a few processes tries to lock on an individual segment. Suppose n processes are used, process 0 sends message to the remaining $n-1$ processes in turn, while process numbered by $1, 2, \ldots, n-1$ sends message to process 0 simultaneously, which constructs a worst case for process 0 as shown in Fig.7.

The curves (worst case) in Fig.7 show similar trend to that in Fig.5, except for larger value which indicates that it takes more time to wait for exclusive control to mutexes.

2. Effect to simulation

1) Intra-node communication

Communication latency plays an essential role to performance of simulation [18]. In this section, we turns to evaluate the effect of ZeROshm to real simulation. The overall architecture of our simulator is same as NTW-MT [1], [2], while intra-node communication is improved by ZeROshm, thereby eliminating twice copy of intra-node messages between process space and the shared memory.

PHOLD is a classical model in parallel and distributed simulation for testing simulators and related techniques. Each PHOLD instance, namely LP in this simulation, receives and sends message to others. Selection of destination is determined by $(my_id + radius) \bmod N$, where N is the total number of LPs, $my_id \in [0, N-1]$ is identifier of an LP, $radius \in (0, N)$ is a parameter for controlling distance. The step between processing and sending message is $1 + time_scale$, where $time_scale \in (0, 1)$ is a parameter for controlling simulation rate.

The environment used in this experiment is same as in Section V.1 and listed in Table 1. We launch 3 processes which includes 1 controller process taking care of global control of simulation, e.g. synchronization among all processes, and 2 worker processes that contains 2 processing threads for processing events, and readers can refer to NTW-MT [1], [2] for more details about the use of process and thread. We create 10,000 PHOLD instances and distribute them evenly among all processing threads. The simulation employs conservative time policy and ends when global LBTS exceeds the end time which can be configured before execution. The parameter $radius$ is set to 200, 300, and 400 respectively, $time_scale$ is set to 0.8, and the experimental results are demonstrated in Fig.8.

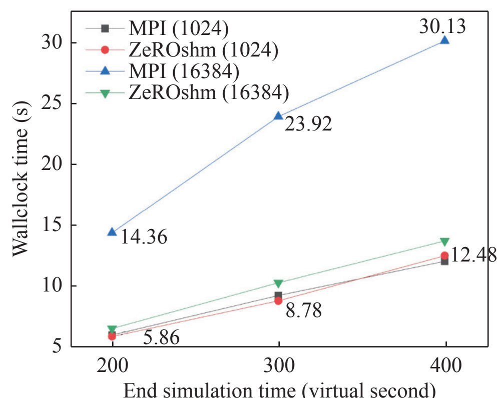

Fig. 8. Wallclock time of executing PHOLD model within node.

In Fig.8, for small message size, the wallclock time does not show noticeable difference between MPI and ZeROshm, which indirectly indicates both of them exhibits similar performance on communication. However, for large message size, MPI costs much more time than ZeROshm (about 55% on average), and the primary explanation includes: 1) MPI costs much more time than ZeROshm to transfer large messages as analyzed in Section V; and 2) high cost of MPI also leads to high communication latency which subsequently results in high latency for computing global LBTS and processing threads stay idle until latest LBTS is updated, thereby

it takes more total time to process all events. In real simulation, ZeROshm is not sensitive to message size either, i.e., the two curves (small and large message size) almost overlaps, whereas messages size appears to a key parameter on performance of MPI. The wallclock time increases linearly as the end time of simulation, which indicates our simulator architecture achieves fine scalability.

2) Hybrid communication

A bunch of physical nodes is generally employed to execute large scale simulation, which creates a hybrid environment of both intra-node and inter-node communication. In this section, we conduct experiments to test performance of ZeROshm in hybrid environment. We use multiple virtual nodes in Aliyun cloud environment (www.aliyun.com), and the configuration (elastic computing service ecs.c7.2xlarge) is listed in Table 2.

Table 2. Environment used in hybrid communication

Item	Configuration
CPU cores	8
CPU frequency	2.7 GHz
Memory	16 GB
OS	Linux 5.4.0-81-generic x86_64
GCC version	9.3.0
MPI	MPICH 3.4.2

We have two contrast configuration in this section, one is to compare the performance between pure MPI 2.x and the combination of MPI 2.x and ZeROshm, and the other is to compare the First-Fit policy (FFshm) and ZeROshm in hybrid communication. Each worker process contains 2 processing threads, thereby each node can hold 2 worker processes at most (3 threads per process $\times$ 2 processes = 6 threads < 8 cores), we varies the total number of threads by configuring the number of processes. The placement of process among nodes follows the default (evenly) distribution. As we employed 4 nodes, the number of processing threads can be 16 at most (4 threads per node) to get rid of extra overheads of scheduling threads from OS. Since real models may transfer many parameters among LPs, the message size is set to large, the parameter *radius* is set to 400, and the execution time is shown in Fig.9.

In Fig.9, ZeROshm exhibits superior performance than both of the other two contrast: 1) ZeROshm shows shorter execution time by about 42% on average compared to pure MPI 2.x, attribute to no need to allocating and copying buffer for each message. 2) The execution time of combination of MPI 2.x and FFshm is approximately 5.7 times of that of combination of MPI 2.x and ZeROshm, which agrees with the analysis in Section I. Messages with lower timestamp may get lower probability to be received, as a result it takes longer time to calculate a correct LBTS, consequently threads waits for the latest LBTS to advance simulation, wasting a lot of execution time. 3) When a few threads are used, the performance begins to decline as the latency of computing LBTS increases. The optimal number of threads for this experiment may be around 10 from visual observation.

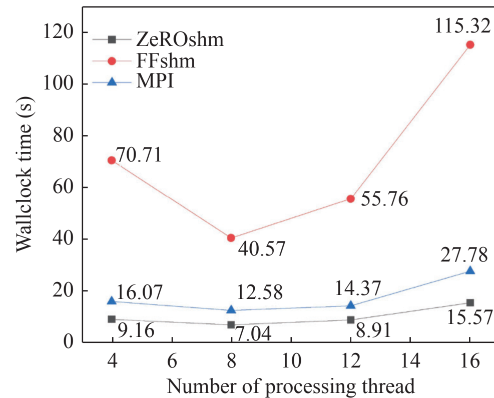

Fig. 9. Wallclock time of executing PHOLD model in hybrid environment.

VI. Conclusions

In this paper, we propose a zero-copy, reliable and order-preserving communication mechanism ZeROshm over shared memory for intra-node message passing to reduce overhead and latency, which also supports hybrid communication along with standard MPI implementation for very large simulation. ZeROshm partitions shared memory into segments by which a process stores and exchanges messages, and each segment consists of two levels of index L1 and L2 that both control the use of space in the host segment, i.e., L1 index marks the start and end point of L2 index in use and L2 index directs to the space that contains pending messages. Sending and Receiving message operates on L1 and L2 index, and process reads from and writes to shared memory directly, thereby eliminating allocating and copying buffers. We proved that ZeROshm is reliable and order-keeping under stable access to shared memory. We also tested the effect of ZeROshm to pure communication and real simulation within a single node and in hybrid environment. Compared to MPI, ZeROshm is not sensitive to message size, i.e., ZeROshm shows almost equivalent performance for small message size (less than page size of OS), and much lower cost for large message size. ZeROshm employs mutex to control exclusive operation on segments, which leads to possibility of contention, and the time of transferring a single message shows a linear increase as the number of process accessing an individual segment simultaneously. In real simulation of PHOLD model, ZeROshm also exhib-

its superior performance to MPI, it costs similar wallclock time for small message size and much less (about 55%) for large message size than MPI. For large scale simulation in hybrid environment, the combination of MPI and ZeROshm costs less execution time about 42% than the pure MPI.

ZeROshm pre-allocates space that can stores at most SHM_N_MSG messages in each segment, thereby intra-node communication can fail when the space is full at that moment. There can be two ways to resolve this failure: 1) Pre-allocate enough space according to the scale of real simulation, as reference [1] reported, the number of pending message is generally less than $N_{\rm msg} = 1.5 \times N_{\rm LP_per_process}$, where $N_{\rm LP_per_process}$ is number of LPs residing in a process, thus we recommend SHM_N_MSG $\geq N_{\rm msg}$ or even larger; 2) Wait until some space is returned. It should be noted, in some circumstance, e.g., processing of an event produces two or more new events (the total number of events keeps expanding), this failure can still happen in the first way. The second way can prevent this failure, whereas the waiting time can be long, which leads to degradation in performance.

The order-preserving and zero-copy attribute of communication can affect the performance of simulation that employs optimistic or hybrid time policy, which will be tested in future.

References

[1] Z. W. Lin, C. Tropper, M. N. I. Patoary, *et al.*, "NTW-MT: A multi-threaded simulator for reaction diffusion simulations in neuron," in *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, New York, NY, USA, pp.157–167, 2015.

[2] Z. W. Lin, C. Tropper, R. A. McDougal, *et al.*, "Multithreaded stochastic PDES for reactions and diffusions in neurons," *ACM Transactions on Modeling and Computer Simulation*, vol.27, no.2, article no.7, 2016.

[3] W. J. Tang, Y. P. Yao, and F. Zhu, "A hierarchical parallel discrete event simulation kernel for multicore platform," *Cluster Computing*, vol.16, no.3, pp.379–387, 2013.

[4] D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev, "Optimization of parallel discrete event simulator for multi-core systems," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, Shanghai, China, pp.520–531, 2012.

[5] T. Hoefler, J. Dinan, D. Buntinas, *et al.*, "Leveraging MPI's one-sided communication interface for shared-memory programming," in *Recent Advances in the Message Passing Interface*, J. L. Träff, S. Benkner, and J. J. Dongarra, Eds. Springer, Berlin, Heidelberg, pp.132–141, 2012.

[6] A. Venkataraman and K. K. Jagadeesha, "Evaluation of inter-process communication mechanisms," Technical. report, Department of Computer Science, University of Wisconsin-Madison, 2015. http://pages.cs.wisc.edu/~adityav/Evaluation_of_Inter_Process_Communication_Mechanisms.pdf

[7] D. Kranz, K. Johnson, A. Agarwal, J. *et al.*, "Integrating message-passing and shared-memory: Early experience," *ACM SIGPLAN Notices*, vol.28, no.7, pp.54–63, 1993.

[8] P. L. Aublin, S. Ben Mokhtar, C. L. Gilles, *et al.*, "ZIMP: Efficient inter-core communications on manycore machines," Technical report, Grenoble University, 2011. http://lig-membres.imag.fr/aublin/zimp/zimp_TR.pdf

[9] M. Li, "*Research and implement on adaptive communication mechanism of high-performance RTI*," *Master thesis*, Nat. Univ. Defense Technol., Changsha, China, 2011. (in Chinese)

[10] R. M. Fujimoto, *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[11] T. Hoefler and S. Gottlieb, "Parallel zero-copy algorithms for fast fourier transform and conjugate gradient using MPI datatypes," in *Recent Advances in the Message Passing Interface*, R. Keller, E. Gabriel, M. Resch, *et al.*, Eds. Springer, Berlin, Heidelberg, pp.132–141, 2010.

[12] M. P. Forum, *MPI: A Message-Passing Interface Standard*, University of Tennessee, Hall Knoxville, TN, USA, 2021.

[13] X. M. Zhu, J. C. Zhang, K. Yoshii, *et al.*, "Analyzing MPI-3.0 process-level shared memory: A case study with stencil computations," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Shenzhen, China, pp.1099–1106, 2015.

[14] B. P. Swenson and G. F. Riley, "A new approach to zero-copy message passing with reversible memory allocation in multi-core architectures," in *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, Zhangjiajie, China, pp.44–52, 2012.

[15] Z. W. Lin, X. H. Li, Y. Mao, *et al.*, "DISHM: A zero-copy intra-node communication approach in large scale simulation," in *2019 IEEE 19th International Conference on Communication Technology (ICCT)*, Xi'an, China, pp.578–582, 2019.

[16] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol.29, no.12, pp.66–76, 1996.

[17] T. L. Li, Y. P. Yao, W. J. Tang, *et al.*, "An efficient multi-threaded memory allocator for PDES applications," *Simulation Modelling Practice and Theory*, vol.100, article no.102067, 2020.

[18] R. M. Fujimoto, "Research challenges in parallel and distributed simulation," *ACM Transactions on Modeling and Computer Simulation*, vol.26, no.4, article no.22, 2016.

LI Xiuhe was born in 1975. He received the Ph.D. degree in electronic engineering from The PLA Electronic Engineering Institute. He is a Professor of National University of Defense Technology (NUDT). His current research interests include theoretical innovation and application of electromagnetic environment, computational system confrontation simulation and effectiveness evaluation, and multisource sensor information fusion technology. (Email: xhli75@163.com)

SHEN Yang was born in 1978. He received the Ph.D. degree in Operational Research from The PLA Electronic Engineering Institute. He is an associate professor of National University of Defense Technology. His research interests include modelling and simulation of electromagnetic environment and test evaluation. (Email: eeishy@163.com)

LIN Zhongwei (corresponding author) received the B.S., M.S. and Ph.D. degrees in computer science and technology from NUDT in 2009, 2011 and 2016 respectively, and visited the Department of Computer Science of McGill University, Montreal, Canada as joint Ph.D. candidate from 2013 to 2015. He is a Lecturer with College of Electronic Engineering, NUDT. His research interests include computer simulation, high performance computing, and artificial intelligence. (Email: zwlin@nudt.edu.cn)

ZHAO Shunkai was born in 1987. He received the B.E degree in network engineering and M.S. degree in military science from Electronic Engineering Institue of PLA. His research interests include simulation and electronmagnetism. (Email: 364393242@qq.com)

SHI Qianqian was born in 1996. She received the M.S. degree in electromagnetic field from Army Engineering University of PLA, China. Her research interests include dynamic spectrum management and reinforcement learning.
(Email: 1974604993@qq.com)

DAI Shaoqi was born in Hefei, China. He received the B.E. and M.S. degrees in electronic engineering from National University of Defense Technology, Changsha, China. His research interests include information and communication engineering and complex electromagnetic environment.
(Email: daishaoqi12@nudt.edu.cn)