IEICE TRANS. INF. & SYST., VOL.E96–D, NO.5 MAY 2013

PAPER

Deterministic Message Passing for Distributed Parallel Computing

Xu ZHOU[†a)], **Kai LU**[†], **Xiaoping WANG**[†], **Wenzhe ZHANG**[†], ***Nonmembers***, **Kai ZHANG**[†], ***Student Member***, **Xu LI**[†], ***and*** **Gen LI**[†], ***Nonmembers***

SUMMARY The nondeterminism of message-passing communication brings challenges to program debugging, testing and fault-tolerance. This paper proposes a novel deterministic message-passing implementation (DMPI) for parallel programs in the distributed environment. DMPI is compatible with the standard MPI in user interface, and it guarantees the reproducibility of message with high performance. The basic idea of DMPI is to use logical time to solve message races and control asynchronous transmissions, and thus we could eliminate the nondeterministic behaviors of the existing message-passing mechanism. We apply a buffering strategy to alleviate the performance slowdown caused by mismatch of logical time and physical time. To avoid deadlocks introduced by deterministic mechanisms, we also integrate DMPI with a lightweight deadlock checker to dynamically detect and solve these deadlocks. We have implemented DMPI and evaluated it using NPB benchmarks. The results show that DMPI could guarantee determinism with incurring modest runtime overhead (14% on average).

key words: *determinism, message-passing, debugging, distributed computing*

1. Introduction

The message-passing programming model is widely used in distributed parallel programs to perform inter-process communication. The most common message-passing model is the Message Passing Interface (MPI), which is developed as a standard of message-passing [1]. For performance reason, MPI allows asynchronous transmission operations and promiscuous receiving operations. However, these operations do not guarantee *reproducibility of messages*, which means the messages (message content and message order) should be exactly the same between two different runs. Due to this reason, a user program running on MPI is often nondeterministic, i.e., even given the same inputs, the program may have different execution paths and produce different outputs in two runs. This nondeterminism brings challenges to MPI programs in the fields of debugging, testing and fault tolerance [2], [3]. For example, a bug may only appear in certain message orders. Simply re-executing the program could not reproduce the bug, since the same message order is unlikely to occur. Hence, the traditional cyclic debugging approach will be invalid for this circumstance.

Different from the nondeterministic message-passing models, a *deterministic message-passing model* ensures the reproducibility of messages. Deterministic message-passing achieves this by constraining the freedom of how messages are sent and received. For example, Kahn's network is a deterministic message-passing model which leverages FIFO pipes to make communication. This message-passing model requires that all receiving operations should wait for a specified sending message [4], [5]. However, these models are too strict, thus they are easy to cause performance slowdown or introduce deadlocks if programs are not carefully designed. Besides, there is no general-purpose implementation of such deterministic message-passing model like MPI currently.

In this paper, we propose a novel Deterministic Message Passing Implementation (DMPI) for distributed computing. DMPI is compatible with MPI in user interface which is well accepted by programmers, and it guarantees the reproducibility of messages in program execution. The basic design of DMPI is to control message transmissions according to logical time as opposed to physical time. For any asynchronous transmission operation, DMPI forces the transmission of the message to finish at a specific logical time so that the process will always perceive the finishing of transmission at deterministic program point. For any promiscuous receiving operation, DMPI orders the incoming messages according to logical time to solve message races. The difficulty of this design is that each process should see the distributed logical time consistently and timely to make a correct and efficient decision. To overcome this difficulty, we design a *world clock* and use the small-sized control messages to propagate clock values throughout the distributed system quickly. The constraints on sending and receiving messages in DMPI may affect performance and introduce deadlocks. We design two improvements to solve these problems. First, we adopt a buffering strategy to alleviate the performance problem introduced by the mismatch of physical time and logical time. Second, we integrate DMPI with a lightweight deadlock checker to detect and solve extra deadlocks.

We implemented DMPI based on MPICH-2 and evaluated it using the NPB benchmarks. Our evaluation results show that DMPI could guarantee determinism of message-passing with incurring only a modest runtime overhead (14% on average) and memory overhead (around 50%).

The rest of this paper is organized as follows. Section 2 analyzes the nondeterministic sources of the message-passing communication. Section 3 presents our deterministic message-passing design. Section 4 describes the solu-

Manuscript received October 19, 2012.
Manuscript revised December 18, 2012.

[†]The authors are with the School of Computer, National University of Defense Technology, Hunan, 410073, P.R. China.
a) E-mail: zhouxu@nudt.edu.cn

DOI: 10.1587/transinf.E96.D.1068

tions for deadlocks and performance slowdown. Section 5 is evaluation and Sect. 6 related work.

2. Nondeterminism of MPI

There are two major sources of nondeterminism in the message-passing model of MPI: (1) the asynchronous transmission operations and (2) the wildcard receiving operations [3], [6]. Asynchronous transmission operation indicates the operation is nonblocking, i.e., it does not have to wait for acknowledgement of the message receiving. In MPI, the function names of these operations are usually prefixed with 'MPI_I', like *MPI_Isend*, *MPI_Irecv*, etc. For example, the calling of *MPI_Isend* will send a message to a remote process. However, this function is returned once the MPI runtime receive this request, and it does not wait for the remote process to get the message and send back an ACK. For the sending process, the transmission may be finished at a nondeterministic program point depending on the timing of message transmissions. Since the finishing point of transmission could change process states (e.g., the calling of *MPI_Test* will return different values), this kind of nondeterminism may be exposed to the user program.

The asynchronous sending operations have different modes, like the standard mode (*MPI_Isend*), the buffered mode (*MPI_Ibsend*) and the synchronous mode (*MPI_Issend*). The difference of them is that they apply different buffering strategies to store the user messages. The buffered-mode send should store the user message in a runtime buffer and return directly once the message is buffered. The synchronous-mode send does not use the runtime buffer, and it must wait until the receive operation in the remote process is posted. In the standard mode, it is the runtime to decide whether to use a runtime buffer to store the user message. However, all the three operations should use testing functions to determine whether the message transmission is completed. Since the user interfaces of these operations are the same, they expose the same nondeterminism to the user application. Due to this reason, they are handled in the same way in our implementation.

The wildcard receiving operation indicates the receiving operation could accept any incoming message from any source. In MPI, a receiving operation with the wildcard parameter is such an operation, e.g., *MPI_Recv(..., MPI_ANY_SOURCE, ...)*. Since messages from different sources could race to arrive, a wildcard receiving operation may accept an arbitrary incoming message depending of the timing of their arrivals. Hence, the program states after a wildcard receiving will be nondeterministic, which also cause problem to the user program.

3. Design

The design aim of DMPI is to eliminate the two nondeterministic sources from MPI, thus providing a deterministic message-passing mechanism for distributed parallel programs. The two sources of nondeterministic factors are both related to physical time, which is a nondeterministic metric as the executing environment changes (network transmission states, operating system states, hardware states, etc.). Hence, the basic idea of DMPI is to design a deterministic *logical time* to direct the message transmissions.

3.1 Design Overview

The nondeterminism of MPI is due to the fact that every decision is made according to physical time. For example, the wildcard receiving will accept a message which is the first to arrive. The asynchronous sending will wait for an uncertain period of time (depending on how fast the message is transferred) before it knows the message is received.

Since the decision on physical time causes nondeterminism, the basic design principle of DMPI is to use logical time to guide the sending and receiving of messages. Following this principle, we design DMPI as shown in Fig. 1. We set a local logical clock in each process to time stamp every message. Hence, each message could be identified by a vector of <src_rank, timestamp>. In DMPI, every decision of message transmission is made according to logical time. For example, the red rectangles in Fig. 1 show three decisions for two asynchronous transmissions and a wildcard receiving. In this example, we expect any asynchronous message will be received in a certain logical time units (10 logical time units in Fig. 1), no more, no less. For the wildcard receiving operation, DMPI will choose an unaccepted message with the smallest logical timestamp (<rank0, msg0, 11> in Fig. 1) no matter when the message arrives.

Based on logical time, we propose two deterministic mechanisms for the two nondeterministic sources: the *deterministic waiting* mechanism and the *deterministic mapping* mechanism. The deterministic waiting mechanism sets a *Deterministic Transmission Point* (DTP) for each asynchronous transmission operation. DTP is a fixed program point in the instruction stream of a process. The position of DTP is defined by logical time to ensure determinism. The deterministic waiting mechanism forces the transmission of asynchronous message to finish exactly at its corresponding DTP, which hides the nondeterministic time of asynchronous message transmission. The deterministic mapping mechanism dominates the message mapping of wildcard receiving operations according to logical timestamp of messages. For each wildcard receiving operation, the deterministic mapping mechanism forces it to accept the *earliest message* (message with the smallest logical timestamp among all the unaccepted messages at now and in the future). Therefore, no matter when the message arrives, the wildcard receiving operation will always accept a determinate message. In this way, the deterministic mapping mechanism resolves the nondeterminism of message races.

Since every decision about message transmission is made according to logical time instead of physical time, we only need to ensure that the logical time is deterministic between runs. The logical time is measured by a local logical clock in each process which is implemented by compile-

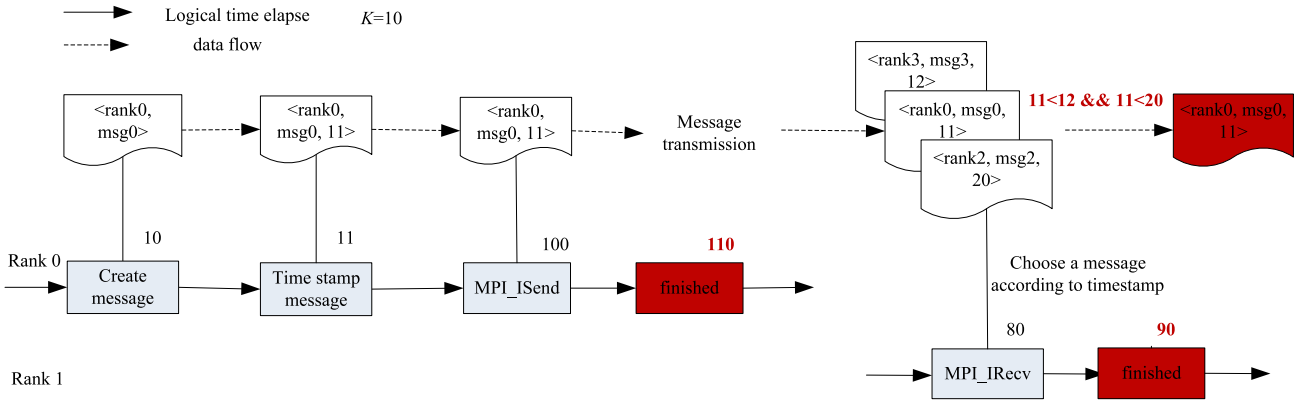


Fig. 1 The design overview of DMPI.

time instrumentation. As the instrumented codes are statically written into the original program, the logical time will be deterministic.

The problem of using logical time to make decisions is that it may cause performance slowdown and deadlocks. The performance problem is due to the reason that speed of logical time may mismatch with the speed of physical time, which may introduce extra wait time for processes. To preserve performance maximally, we make logical time highly relevant to physical time. Hence the decision on logical time is similar to the decision on physical time, which preserves performance. We also leverage a buffering strategy to alleviate the wait time when logical time does not match physical time. The deadlock problem is caused by the extra wait-for dependencies introduced by the two deterministic mechanisms. We integrate DMPI with a lightweight deadlock checker to solve this problem.

3.2 Clock and Logical Time

To maintain a deterministic logical time, We propose the *world clock* which is a hybrid of Lamport clock [7] and vector clock [8], [9]. World clock is a clock vector set in each process to monitor the time of all processes (including the local process). Each slot in the clock vector is corresponding to a process in the process network, describing the time of the process already seen by the local process. The time values in the vector could be divided into two parts: 1) the *local logical time* which is the i^{th} value in the clock vector of rank i, and 2) the *shadow time values* which describe the time of the remote processes. We apply different time-driven methods for these two kinds of time values.

First, we use messages to drive the shadow time values. Each message is time stamped with the local logical time of the sending process. The timestamp piggybacked in the message will be used to refresh the corresponding vector slot in the receiving process if the timestamp is bigger. Different from vector clock, we only store a scalar value (the local logical time) in each message, which is implemented by adding an 8-byte field to all kinds of MPI message packets. This design simplifies the original vector clock and reduces the communication overhead, which provides an efficient implementation for a deterministic time metric. However, it also makes our logical time a little weaker than the original vector clock. Since only a scalar time value is propagated in a message, our logical clock cannot be used to describe the partial order of all events in a distributed system. Nevertheless, this weakness does not affect our deterministic mechanisms for message transmissions.

Second, we use local events (e.g., calling of MPI operations, instruction issuing, system calls, etc.) to drive the local logical time. The increasing of local logical time must meet two requirements: (1) local logical time must be precise enough to distinguish messages, and (2) the speed of local logical time should reflect the physical time of the process execution well. The first requirement is achieved by interposing all the MPI APIs (e.g., *MPI_Send*, *MPI_Recv*, etc.). The local logical clock is increased by 1 when an MPI function is called. By doing so, no two MPI function calls would happen at the same logical time. Hence, each message will be created in a different logical time, which grants the message a unique timestamp.

The second requirement is achieved by compile-time instrumentation and the using of empirical statistics. There are many kinds of events in the execution of the user program (e.g., a function call, an instruction execution or a system call). The execution time for these events may be different (e.g., a load instruction may only takes a nanosecond while a Sleep(1) call could take 1 seconds to finish). To reflect the physical time well, we must catch the differences of these events and assign them a corresponding number of logical time units. To this end, we first divide the events into two categories: 1) the issuing of normal instructions, and 2) the execution of system events (e.g., system calls), then we measure them in two different ways.

Measuring normal instructions. First, we define 1 logical time as 1000 instructions issuing, which will be a standard for the mapping between the deterministic logical time to the nondeterministic physical time. According to this definition, logical time of normal instruction execution is measured by counting the number of issued instructions. Since the execution time of normal instructions is propor-

tional to the number of the issued instructions, our measuring method could reflect the physical time well. To count the number of issued instructions, we leverage the LLVM compile framework [26] to instrument user programs. The LLVM framework allows users to write their own *Passes* to perform certain codes transformation. LLVM provides the functionality of typical code analysis (e.g., CFG analysis) which could be used in user's pass. We write an LLVM pass to instrument user programs based on the CFG analysis provided by the LLVM framework. In the LLVM pass, we insert into each basic block a *time_tick* call, whose parameter is set as the instruction number of that basic block. At runtime, when the *time_tick* function is invoked, the number of instruction is accumulated. As a result, the local logical clock ticks as the program runs.

Measuring system events. The above method works well for normal instructions, but is not suitable for system calls as their execution time is hard to predict (e.g., the calling of *Sleep(1)* and *Sleep(3)* will differ much in execution time). To reflect the execution time well, we carried out an empirical study for the underlying hardware platform. The empirical study is to build an approximate mapping between typical system calls and the logical time of their execution. In this study, we assume that the execution time of a certain system call is stable in a certain hardware platform. This empirical study tests the execution time of typical system calls, and translate them into logical time based on the logical time unit definition. After the empirical study, we set a table in our runtime to describe the logical time values of the typical system calls. For example, the calling of *Sleep(1)* is corresponding to 1000 logical time unit for our experiment hardware. Hence, we interpose these typical system calls in DMPI, and increase a corresponding value of logical time when such an event happens. Note that there is a tradeoff between determinism and performance when we want to port DMPI across different hardware platforms. If we want better performance, we should carry out an empirical study for each specific hardware platform. However, if we want determinism over performance, we should use the same empirical study table even in a different platform.

3.3 Deterministic Waiting

The deterministic waiting mechanism forces each asynchronous operation to wait for the finishing of message transmission at its corresponding DTP. DTP is a function call that is dynamically inserted into a fixed code point of the user program during runtime. We use logical time to identify different code points so that we could determine the positions of DTPs dynamically. For any asynchronous transmission operation, the corresponding DTP is inserted at the code point which is **K** logical time units after the function call. Note that the value of **K** should be a constant value so as to ensure determinism (In our implementation, **K** = 10, as shown in Fig. 1).

The algorithm of deterministic waiting for asynchronous receiving is show in Fig. 2 as an example. DMPI

```
   MPI_Irecv_Hook:
1  waiting_queue.add_request(current_logical_time + K, request);
   Time_Tick:
2  inc_local_time();
3  foreach request in waiting_queue
4     if(request.logical_time <= current_logical_time)
          /* wait for the finishing of transmission */
5        wait_for_request(request)
6        waiting_queue.dequeue(request)
   MPI_Test_Hook:
7  if(request in waiting_queue &&
8     current_logical_time < request.logical_time)
9     return false; /*postpone declaration*/
10 else if(request in waiting_queue)
11    return true; /*forward declaration*/
   else
      /*Normal test*/
   MPI_Wait_Hook:
12 if(request in waiting_queue)
13    waiting_queue.dequeue( request )
```

Fig. 2 The deterministic waiting algorithm for asynchronous receiving.

enforces that the transmission of message be finished right at its corresponding DTP. When an asynchronous request is sent (e.g., *MPI_Irecv*), DMPI first adds a DTP for that request (line 1). If the message arrives earlier, DMPI postpones the declaration of its arrival until the process reaches the DTP (line 7-9). Therefore, the issued testing function (e.g., *MPI_Test* and *MPI_Probe*) before the DTP could not perceive the arrival of the message. On the contrary, if a message is late, DMPI forces the process to wait at the DTP until the message arrives (line 3-5) and declares the arrival of the message (line 10-11). Therefore, the testing function after the DTP will always detect the message. The DTP is removed once the process has received the message (line 6 and line 13). Note that the design of DTP only constrains the declaration of the finishing of message transmissions, and the underlying transmission is not affected. In this way, DTP creates a deterministic environment for the upper applications.

3.4 Deterministic Mapping

The deterministic mapping mechanism requires that the incoming messages are ordered according to logical time as opposed to physical time, and only the earliest message can be accepted by the wildcard receiving (as shown in Fig. 1).

When a message arrives, DMPI tries to map it with a posted receiving request. If the receiving request is not a wildcard receiving, normal mapping mechanism is used. Otherwise, we check if this message is the earliest by its timestamp. The earliest message is the unaccepted incoming message that has a smaller timestamp than any other unaccepted incoming messages (including the incoming messages that is on the way or have not been sent). We first choose an incoming message with the smallest timestamp among all the messages that have arrived, then we leverage

the world clock to determine whether this is the earliest message currently. The world clock records the logical time of all processes, thus DMPI simply compares the timestamp of the message with the values in the clock vector. If the timestamp is smaller than or equal to all the values in the clock vector, the message is considered to be the earliest and it is accepted directly. However, if the timestamp is larger than at least one time value in the clock vector, indicating there is a potential message which may be earlier than this message, the wildcard receiving must wait until the earliest message appears. Since the world clock only refreshes time values of other processes upon messages arrivals, the time values seen by the local process may be out-of-date, which may cause a wildcard receiving to be blocked for a long time. To mitigate this problem, we introduce the small-sized *control message* to exchange information between MPI processes.

Control messages are sent and received by the runtime, and they are transparent to the user programs. To implement control message, we add a new packet type *PKT_CONTROL* in MPICH, and associate it with a handling function *PacketHandler_ControlMessage*. When we send a control message, we tag the type of the message as *PKT_CONTROL*. When the message arrives at the remote process, the underlying transmission engine of the process will handle the message. If the transmission engine discovers that it is a control message, it will call the *PacketHandler_ControlMessage* callback function. Since the work in *PacketHandler_ControlMessage* is done in an independent thread, it does not block the user program.

The typical use of control message is to update clock vectors. We design an *on-demand strategy* to minimize communication overhead. The on-demand strategy works as follows. If the checking of the earliest message fails because of a smaller value in the clock vector, the process will send a *Request* (a control message used to ask for the logical time of another process) to the corresponding process to refresh its time. The process that receives the *Request* will schedule an *Answer* based on the local logical time. As the *Request* contains the expected logical time, the replying process only needs to send the *Answer* when its logical time is greater than the expected value.

4. Improvement

The two deterministic mechanisms in DMPI could ensure deterministic message-passing. However, it may also bring performance problem and correctness problem to the user program. This section discusses these two problems and presents our improvements of DMPI.

4.1 Performance

DMPI decreases performance due to the additional wait time caused by the deterministic waiting mechanism and the deterministic mapping mechanism. Theoretically, DMPI could not totally eliminate the wait time as logical time could not be the same fast as physical time. Otherwise, logical time will be nondeterministic as physical time. In other words, wait time is an inevitable overhead for determinism that can only be mitigated.

In DMPI, most of the wait time is caused by the wildcard receiving operations which requires identifying the smallest logical time globally. Our idea is to use buffering to reduce this *determinism-induced wait time*. We maintain an additional *accepting buffer* in the MPI runtime. The accepting buffer stores incoming messages which cannot be accepted by the user application due to the deterministic mapping mechanism. Once a message is stored in the accepting buffer, we claim that the message is successfully accepted (although it has not be seen by the user application). Our runtime acts as a proxy to store this message temporarily for the user application. Hence, the runtime could send an acknowledgement message to the sender so as to release the sending process. Afterwards, when this stored message is copied by the user application, we delete the message from the accepting buffer. In our current implementation, each process maintains a 16 MB accepting buffer, which could store most blocking messages. However, it is possible that a message is too big to be stored in the accepting buffer. In this case, the message should wait until it can be stored in the buffer provided by the user application.

This buffering strategy brings two advantages. First, it decreases the wait time caused by the deterministic mapping mechanism, which improves performance. Second, this strategy also mitigates the deadlock problems caused by deterministic mapping, which we discuss in the next subsection.

4.2 Deadlock

As our deterministic mechanisms (the deterministic waiting mechanism and the deterministic mapping mechanism) put constraints on normal message transmissions, they may introduce extra deadlocks. These determinism-inducing deadlocks are caused by the extra wait-for dependencies enforced by DMPI, which should be eliminated to enable practical use.

4.2.1 Typical Deadlocks

One typical deadlock situation caused by the deterministic waiting mechanism is shown in Fig. 3. These codes are from a real world application (the BT benchmark in the NPB suite). In this implementation, each process will issue

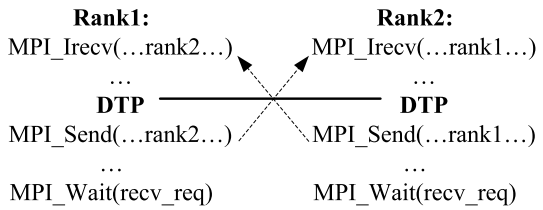


Fig. 3 A typical deadlock caused by the deterministic waiting mechanism.

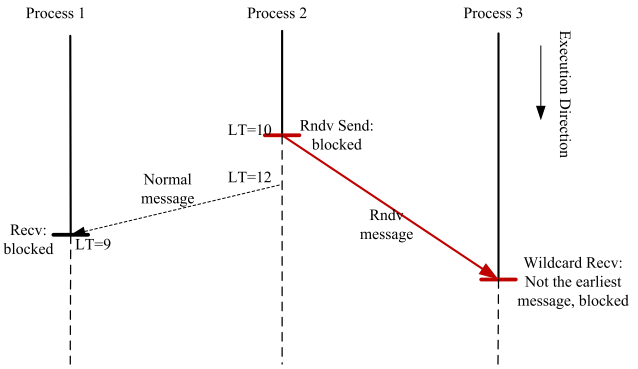


Fig. 4 A typical deadlock caused by the deterministic mapping mechanism.

an asynchronous receiving operation before sending a message to others. Since the asynchronous receiving will not block the process, and the message will be sent eventually, all the receiving operations will be satisfied with messages. Therefore, these codes will not cause deadlock with the normal MPI. However, if DMPI happens to insert a DTP for *MPI_Irecv* before *MPI_Send*, then the program will wait at the DTP for a message which will be sent in the future. This will cause the waiting in the DTP cannot be satisfied, resulting in a deadlock.

Another kind of deadlock is caused by the deterministic mapping mechanism. If the receiving message is a rendezvous message, which should be stored directly in the buffer provided by the user program, the wildcard receiving may block a sending process, which in turn blocks other process, finally resulting in a deadlock. We show a typical situation of this kind of deadlock in Fig. 4. In this situation, the wildcard receiving of Process 3 cannot accept the message from Process 2 because Process 1 has a smaller logical time. Hence the deterministic mapping mechanism of Process 3 blocks the sending procedure of Process 2. Meanwhile, Process 2 blocks Process 1 because it cannot move on to send a message to Process 1. In turn, Process 1 blocks Process 3 as it cannot increase its logical time. As a result, the cyclic dependence causes a deadlock.

We solve the deadlock problem in two levels. First, we alleviate the deadlock problem by using message buffering and removing unnecessary DTPs, which we call the *mitigating strategy*. The mitigating strategy filters out most of the determinism-inducing deadlocks. Second, we use a lightweight deadlock checker (LW-DC) to detect and solve the rest of the determinism-inducing deadlocks which cannot be avoided by the mitigating strategy.

4.2.2 Mitigating Strategy

Our deadlock-mitigating strategy works for the two deterministic mechanisms respectively. First, we reduce the number of inserted DTPs for the deterministic waiting mechanism. We do not insert DTPs in arbitrary program points. Instead, we only insert DTPs in testing functions (e.g., *MPI_Test*, *MPI_Probe*, etc.). If there is no testing function, meaning the user program does not care when the message is transmitted, we do not need the waiting codes. As testing functions are added by the programmers to expect the arrival of messages, it is unlikely that the messages have not been sent at these points. In our tests, this improvement could prevent all the deadlocks of this kind. Second, the buffering strategy described in Sect. 4.1 could be used to avoid deadlocks induced by the deterministic mapping. In the deadlock example of Fig. 4, the problem is that Process 3 cannot accept the rendezvous message from Process 2 (the red line in Fig. 4), because Process 3 cannot decide whether this message is the earliest message in logical time. Our mitigating strategy solves this problem by buffering this message and telling Process 2 that the message is successfully accepted by the user codes (in fact, it will be accepted in the future). Hence, Process 2 will continue to execute, which breaks the deadlock.

The mitigating strategy could avoid most of the determinism-inducing deadlocks. However, there are some special deadlocks which may pass the mitigating strategy. For example, it is possible that a programmer test the coming of messages before sending it, though it does not make any sense. Meanwhile, if there are too many wildcard receiving operations, it may run out of buffering memory, which is still possible to cause a deadlock. Therefore, we rely on our lightweight deadlock checker to solve these deadlocks totally.

4.2.3 Deadlock Checker

Different from existing general-purpose deadlock detectors [10]–[13], LW-DC is supposed to be a special deadlock detector for deadlocks introduced by our two deterministic mechanisms only. Hence, LW-DC is lightweight and fast. LW-DC is a hybrid approach of timeout deadlock detector and dependency analysis detector. Since each wildcard receiving in DMPI will accept a determinate message, it greatly simplifies the design of LW-DC [10].

LW-DC makes each process start to check for deadlock when the process stops for a while (100 ms in our implementation). Then LW-DC will collect the wait-for dependency using control messages. Once LW-DC detects a cyclic dependency, it simply relaxes the deterministic mechanism to break the deadlock. Note that the deadlocks are solved in a deterministic manner.

The procedure for checking deadlock is as follows. First, the stopped process (**A**) checks for its stopping reason and its dependent process (**B**). Then **A** sends a request to its dependent process **B** to confirm this dependence. If **B** is still running, which means it may release the dependence, then **B** will send a *cancel message* (a type of control messages) to **A**. Otherwise, it is possible that there is a deadlock, so **B** will send a *confirm message* to **A**. When the dependence **A**→**B** (**A** is dependent on **B**) is confirmed, **A** stores this information in the dependent list and updates all its dependent information to **B**. This procedure is repeated, and the dependent information is propagated. Once a process (**C**) detects

a dependence cycle in its dependent list, it finds out a deadlock candidate. This deadlock is a candidate because there may be some cancel messages on the way. Therefore, **C** will send a deadlock confirm message to all the processes in dependent cycle to confirm that deadlock. If the deadlock is confirmed by all (all the cancel messages are flushed), the process will propagate the information of the dependent cycle to all other processes in the cycle to declare the deadlock.

The lightweight deadlock checker could quickly identify a determinism-inducing deadlock. Then it solves the deadlock without affecting determinism. To solve the deadlock, we first find out all the non-normal dependency in the dependent cycle. Non-normal dependency means the dependency is caused by our deterministic mechanism: waiting at DTPs set for asynchronous operations or being blocked by the deterministic mapping mechanism. We choose the non-normal blocking process with the smallest rank ID as the breaker to release the deadlock. If the breaker is blocked by DTP of an asynchronous operation, we postpone this waiting to the next DTP. This could solve the deadlocks caused by our DTP constraints. If the breaker is blocked by the deterministic mapping mechanism, we match the wildcard receiving with the existing earliest incoming message. This could solve the deadlocks caused by our deterministic mapping mechanism.

5. Evaluation

This section evaluates DMPI on determinism, runtime overhead and memory overhead.

5.1 Methodology

We implemented DMPI based on MPICH2 (version 1.4) [14]. Our evaluation hardware consisted of 2 AMD servers connected with a 1 GB/s switch. Each server has a 2.2 GHz CPU with 4 cores, 1 GB memory and a 1 GB/s Ethernet card, running Fedora 12 with Linux kernel version 2.6.31.5. We used the NPB3.2 benchmarks [15] to test DMPI. Since the benchmarks in NPB3.2 do not use testing functions, we also add two other benchmarks: *mpiGraph* and *mpiTest* in our experiment. *mpiGraph* is a bandwidth benchmark from the Phloem suite [27], while *mpiTest* is a synthesized program which performs a stress test for *MPI_Irecv* and *MPI_Test*. Among these applications, *mg* and *lu* are nondeterministic due to the using of wildcard receiving operations, while *mpiTest* and *mpiGraph* are nondeterministic due to the using of asynchronous operations and testing functions. Each benchmark is configured with 8 processes which are equally distributed among the two servers.

We verified the determinism of DMPI by (1) examining the outputs of programs, (2) checking the return value for each testing function, and (3) checking the accepted message for each wildcard receiving. We ran each benchmark 100 times, and recorded the above information. We compared the results of different runs for each benchmark. Note that although DMPI ensures deterministic execution of MPI programs, it does not ensure deterministic performance. Meanwhile, DMPI does not address the nondeterministic events caused by system calls (e.g., gettimeofday). Therefore, when comparing the normal program outputs, we ignored the time-related data.

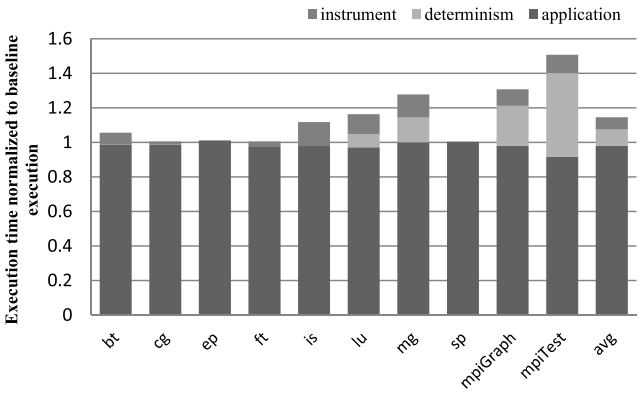


Fig. 5 The execution time of DMPI compared with the normal MPI library.

Table 1 The detailed information of programs running with DMPI. WC indicates the number of wildcard receiving operations. DL indicates the number of deadlocks.

	DMPI profiling			Normal message		Control message	
APP	DTP	WC	DL	Num	Size (KB)	Num	Size (KB)
bt	0	0	0	2436	283918	0	0
cg	0	0	0	1683	46658	0	0
ep	0	0	0	9	0.7	0	0
ft	0	0	0	40	201330	0	0
is	0	0	0	128	92358	0	0
lu	0	510	13	31661	122780	789	3
mg	0	714	9	867	32057	622	2
sp	0	0	0	4836	493724	0	0
mpiGraph	35000	0	0	48698	573655	0	0
mpiTest	32768	0	0	65536	262114	0	0
avg	6776	122	2.2	15589	210859	141.1	0.5

To evaluate the performance, we setup a baseline execution which is the performance of the original MPICH2. We compared the performance of DMPI with the baseline execution to show the overhead of DMPI. For each benchmark, we ran it 10 times to collect the mean value as the final result.

5.2 Performance

The performance overhead of DMPI is shown in Fig. 5. We divide the execution time of each benchmark into three parts: (1) the application time which is the execution time of normal program codes; (2) the instrument overhead, which is the execution time of the instrumented codes for logical time; (3) the determinism overhead, which is caused by the enforcing of our deterministic mechanisms. Overall, the average overhead of DMPI is small (below 14%).

In Table 1, we provide the detailed profiling data of programs running under DMPI, including the number of inserted DTPs (Column 2), the number of wildcard receiving

operations (Column 3) and the number of additional deadlocks (Column 4). We also show the information of control messages compared with normal application messages (Column 5-8). Note that DMPI incurs little overhead for applications that do not trigger the two deterministic mechanisms.

5.3 Memory Footprint

DMPI also consumes more memory than the original MPI implementation. The memory overhead comes from two aspects: (1) the buffering strategy allocates memories to store messages that cannot be accepted by the deterministic mapping mechanism temporarily and (2) the control messages that are used to propagate logical time and detect deadlocks consume memories. We tested the memory footprint of DMPI using the NPB benchmarks and show the memory overhead in Fig. 6. As we can see, DMPI introduces around 30% overhead in memory allocation and 70% overhead in peak memory usage compared with the normal MPI implementation.

5.4 Scalability

We also test how DMPI scales as the number of processes increases. We totally choose 6 benchmarks (as shown in Fig. 7) to stress the two deterministic mechanisms in DMPI.

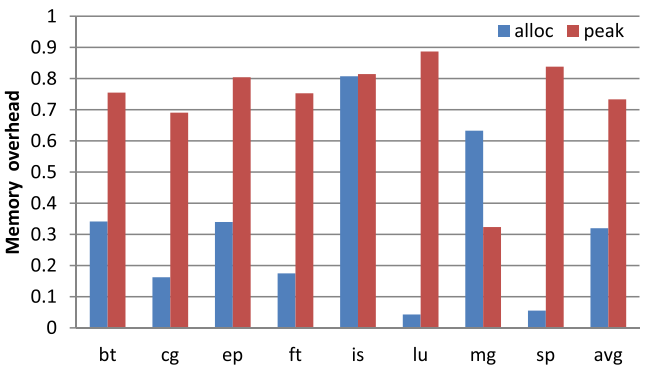


Fig. 6 Memory overhead for different benchmarks. 'alloc' indicates the totally allocated memory. 'peak' indicates the peak memory in use.

Each of the benchmark is configured with 1, 2, 4, 8, 16, 32 and 64 processes respectively. We record both the runtime overhead and the memory overhead of the benchmarks. The results are shown in Fig. 7. As we can see, most programs (*cg*, *ft*, *mpiGraph*, *mpiTest*) incur a stable runtime overhead, while the programs that triggers the deterministic mapping mechanism (*lu* and *mg*) will incur a relatively large runtime overhead as the number of processes increases. The reason is that the deterministic mapping mechanism requires each wildcard receiving operation to wait for the smallest logical time of all processes, thus this overhead becomes large as the number of processes increases. The memory overhead is also highly affected by the deterministic mapping mechanism (see *mg* and *lu* in Fig. 7 (b)) as the number of processes increases. The reasons is that memory overhead is mostly caused by control message processing and message buffering, and both of them are related to the deterministic mapping mechanism. Hence, we conclude that the scalability of DMPI is limited by its deterministic mapping mechanism currently. This problem could be solved by dividing processes into communication groups, which is our future work.

6. Related Work

Record & replay is a typical method to eliminate the nondeterminism of MPI [2], [3], [6], [16]–[19]. These approaches record the nondeterministic effects of messages, and deterministically replay the MPI programs according to the recorded logs. Order-replay [6], [19] and data-replay [18] are two different ways to achieve this aim. Order replay records the nondeterministic message order in the recording phase. In the replaying phase, order-replay regenerates messages and forces these messages to follow the order in the recorded logs. Data-replay, on the other hand, records the contents of messages instead of their orders. Therefore, each process could regenerate messages directly from the recorded logs. A hybrid method MPIWiz [3] could achieve better performance and smaller log size. Distinct from these works, we provide a solution for determinism without any

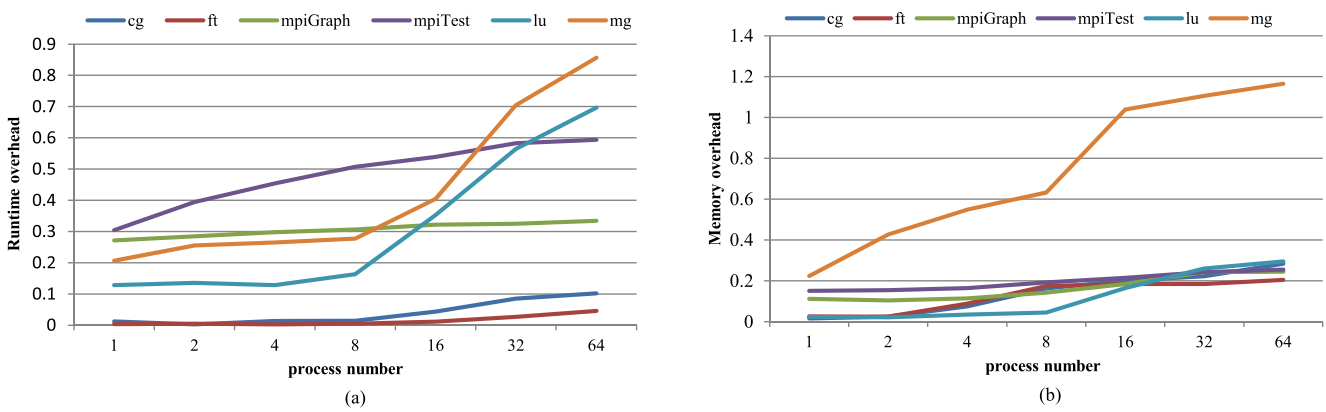


Fig. 7 The runtime and memory overhead as the number of processes increases.

external supports (e.g., logs), which is efficient in performance and is convenient to deploy.

Deterministic Multi-threading has been introduced in the shared-memory multi-processing field to facilitate the debugging of multi-threaded programs [20], [21]. They eliminate the nondeterminism of thread concurrency of parallel programs on shared-memory architecture. Unlike these works, DMPI is to address the nondeterministic problem of the message-passing mechanism in the distributed environment.

Some deterministic message-passing programming models are designed to ensure determinism of parallel programs. They achieve determinism by using a restricted message-passing model. For example, the SHIM language [5], [22], [23] leverages FIFO pipes to pass messages between processes, and prevents the using of asynchronous messages and wildcard receiving operations. However, these programming models are usually domain-specific. Unlike these works, DMPI is compatible with the well-defined MPI in user interface, which makes it a general-purpose solution for parallel programs.

The systems of PDES (Parallel Discrete Event Simulation) are often used for performance simulation and debugging [24], [25]. Different from these works, DMPI is used as a debugging tool for correctness.

7. Conclusion and Future Work

This paper designed and implemented a deterministic message-passing system (DMPI) for distributed parallel computing. DMPI could be used to facilitate debugging, testing and fault tolerance. DMPI ensures determinism by using logical time to control the finishing points of asynchronous transmissions and the accepted messages of wildcard receiving operations. Further, we design a lightweight deadlock checker to avoid deadlocks caused by DMPI, and a buffering strategy to mitigate the performance slowdown. We implemented and evaluated DMPI to demonstrate its practicality. The evaluation results on the NPB benchmarks show that the runtime overhead of DMPI is practical and low for small-scale applications. In the future, we will use speculation and process grouping mechanisms to further mitigate the performance impact of DMPI and test it in large-scale distributed environment.

Acknowledgment

This work is partially supported by National High-tech R&D Program of China (863 Program) under Grants 2012AA01A301 and 2012AA010901, by program for New Century Excellent Talents in University and by National Science Foundation (NSF) China 61272142, 61103082, 61003075, 61170261 and 61103193.

References

[1] M. Snir, S.W. Otto, D.W. Walker, J. Dongarra, and S. Huss-Lederman, Eds., MPI: The Complete Reference. MIT Press, Cambridge, MA, USA, 1995.

[2] M. Elwakil and Z. Yang, "Deterministic replay for message-passing-based concurrent programs," ACM Trans. Design Automation of Electronic Esystems, vol.17, pp.1–30, 2012.

[3] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. Voelker, "MPIWiz: Subgroup reproducible replay of MPI applications," PPoPP, pp.251–260, 2009.

[4] G. Kahn, "The semantics of a simple language for parallel programming," Information Processing 74: Proc. IFIP Congress 74, Stockholm, Sweden, 1974.

[5] S.A. Edwards and O. Tardieu, "SHIM: A deterministic model for heterogeneous embeded systems," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.14, no.8, pp.854–867, 2006.

[6] J. de Kergommeaux, M. Ronsse, and K. De Bosschere, "MPL: Efficient Record/Replay of nondeterministic features of message passing libraries," Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp.673–673, 1999.

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. ACM, vol.21, pp.558–565, 1978.

[8] C.J. Fidge., "Partial orders for parallel debugging," ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, vol.24, no.1, pp.183–194, Jan. 1989.

[9] M. Singhal and A. Kshemkalyani, "An efficient implementation of vector clocks," Inf. Process. Lett., vol.43, no.10, pp.47–52, Aug. 1992.

[10] T. Hilbrich, B.R.D. Supinski, M. Schulz, and M.S. Muller, "A graph based approach for MPI deadlock detection," International Conference on Supercomputing, pp.296–305, Yorktown Heights, NY, USA, 2009.

[11] S. Lee, "Fast, centralized detection and resolution of distributed deadlocks in the generalized model," IEEE Trans. Softw., vol.30, no.9, pp.561–573, 2004.

[12] M. Singhal, "Deadlock detection in distributed systems," Computer, vol.22, no.11, pp.37–48, 1989.

[13] G.R. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva, "Deadlocks detection in MPI programs," Concurrency and Computation: Practice and Experience, pp.14:911–932, 2002.

[14] MPICH2. Available: http://www.mcs.anl.gov/research/projects/mpich2/

[15] D.H. Bailey, "The NAS Parallel Benchmarks," Supercomputer Application, vol.5, no.3, pp.63–73, 1991.

[16] C. Clmen on, J. Fritscher, M. Meehan, and R. Rhl, "An implementation of race detection and deterministic replay with MPI," EURO-PAR'95 Parallel Processing, pp.155–166, 1995.

[17] A. Bouteiller, G. Bosilca, and J. Dongarra, "Retrospect: Deterministic replay of MPI applications for interactive distributed debugging," Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp.297–306, 2007.

[18] M. Maruyama, T. Tsumura, and H. Nakashima, "Parallel program debugging based on data-replay," PDCS, pp.151–156, 2005.

[19] D. Kranzlmller, C. Schaubschläger, and J. Volkert, "An integrated record&replay mechanism for nondeterministic message passing programs," Recent Advances in Parallel Virtual Machine and Message Passing Interface, pp.192–200, 2001.

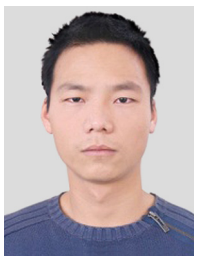
[20] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems, pp.97–108, 2009.

[21] D. Joseph, L. Brandon, C. Luis, and O. Mark, "DMP: deterministic shared memory multiprocessing," Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems, Washington, DC, USA, 2009.

[22] S.A. Edwards, N. Vasudevan, and O. Tardieu, "Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads," DATE, March 2008.

[23] O. Tardieu and S.A. Edwards, "Scheduling-independent threads and

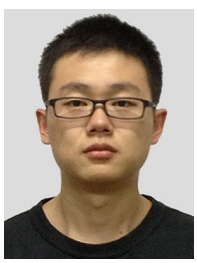
exceptions in SHIM," EMSOFT, pp.142–151, Oct. 2006.
[24] K. Perumalla, "Parallel and distributed simulation: Traditional techniques and recent advances," Winter Simulation Conference, 2006.
[25] S. Boehm and C. Engelmann, "xSim: The extreme-scale simulator," HPCS, 2011.
[26] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," CGO, 2004.
[27] Phloem Benchmarks. Available: https://asc.llnl.gov/sequoia/benchmarks/



Xu Zhou received the B.S. and M.S. degrees in 2007 and 2009, respectively, from the School of Computer Science, National University of Defense Technology, Changsha, China, where he is currently pursuing the Ph.D. degree. His research interests include operating system and parallel computing.

Kai Lu received the B.S. and Ph.D. degrees in 1995 and 1999, respectively, from the School of Computer Science, National University of Defense Technology, Changsha, China. He is now a Professor in the School of Computer Science, National University of Defense Technology. His research interests include operating system, parallel computing and security.

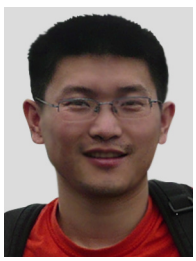
Xiaoping Wang received his B.S., M.S., and Ph.D. degree in the School of Computer Science from National University of Defense Technology, China, in 2003, 2006, and 2010, respectively. He is now an assistant professor in the School of Computer Science at National University of Defense Technology. His research interests include operating system and sensor networking.



Wenzhe Zhang received the B.S. degree from the School of Computer Science, National University of Defense Technology in 2010. He is currently pursuing the M.S degree. His research interests include parallel and distributed computing.

Kai Zhang received the B.S. degree from National University of Defense Technology in 2007, where he is currently working for his Ph.D. degree. His research interests include high performance processor architecture, HPC, embedded system, and VLSI design.

Xu Li received the B.S. and M.S. degrees in 2006 and 2008, respectively, from the School of Computer Science, National University of Defense Technology, Changsha, China, where he is currently pursuing the Ph.D. degree. His research interests include operating system and parallel computing.



Gen Li received his B.S., M.S., and Ph.D. degree in the School of Computer Science from National University of Defense Technology, China, in 2004, 2006, and 2010, respectively. He is now an assistant professor in the School of Computer Science at National University of Defense Technology. His research interests include operating system and software testing.