dMPI: Facilitating Debugging of MPI Programs via Deterministic Message Passing

Xu Zhou, Kai Lu, Xicheng Lu, Xiaoping Wang, Baohua Fan

dMPI: Facilitating Debugging of MPI Programs via Deterministic Message Passing

Xu Zhou, Kai Lu, Xicheng Lu, Xiaoping Wang, Baohua Fan

School of Computer, National University of Defense Technology
Changsha, Hunan, China, 410073

Abstract. This paper presents a novel deterministic MPI implementation (dMPI) to facilitate the debugging of MPI programs. Distinct from existing approaches, dMPI ensures inherent determinism without using any external support (e.g., logs), which achieves convenience and performance simultaneously. The basic idea of dMPI is to use deterministic logical time to solve message races and control asynchronous transmissions, thus we could eliminate the nondeterministic behaviors of the existing message passing mechanism. To avoid deadlocks introduced by dMPI, 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 overhead (8% on average).

1 Introduction

Parallel programs written in the Message Passing Interface (MPI) are inherently nondeterminism due to message races and asynchronous transmission of messages. Nondeterminism hampers the debugging of MPI programs and makes cyclic debugging a particularly challenging task [1-5].

Currently, researchers adopt the approaches of *record & replay* to eliminate the nondeterminism of MPI. These approaches record the nondeterministic events of messages, and deterministically replay the MPI programs according to the recorded logs. Since determinism is not an inherent character, these systems have limitations. First, these systems do not guarantee *first-run determinism* (default determinism), i.e., a program can be deterministic only after it is recorded. Moreover, one set of logs only make a certain execution of a program deterministic. When the inputs change, the program will undergo a different execution path, which is still nondeterministic. Second, deploying a record & replay system in the production environment is difficult, as they are constrained by maintaining logs. In some systems, the logging rate could be nearly 1 GB per minute for large-scale programs [3], which presents a great challenge for the deployment of these systems. Third, these approaches are prone to cause large recording overhead, which may be unacceptable for some performance-critical programs. For example, the state-of-the-art hybrid record & replay system reported an average of 27% runtime overhead [3].

The major problem of record & replay systems is they rely on external support to implement determinism instead of making determinism as a default property. To improve this weakness, we propose a novel *deterministic Message Passing Interface* (dMPI) that guarantees inherent determinism of MPI program without using any external support (e.g., logs). By using dMPI, programs are deterministic even at their first execution, thus any heisenbugs related to the message passing mechanism could be captured and debugged.

The basic design of dMPI is to conduct message transmissions and mappings according to a deterministic logical time. For any asynchronous transmission operation, dMPI forces the transmission of the message to be finished 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 the nondeterminism of message races. The difficulty of this design is each process should see the distributed logical time consistently and timely to make a correct and efficient decision. To overcome this difficulty, we design the *world clock* and use the small-sized control messages to propagate clock values. As we put constraints on the actions of sending and receiving messages, it may cause deadlocks. To make a practical implementation, we design a lightweight deadlock checker to detect and solve these deadlocks. We have implemented dMPI as a compatible MPI library—programs only need to be linked to the dMPI library to gain the deterministic feature.

2 Design and Implementation

2.1 Nondeterminism of MPI

There are two major sources of nondeterminism in MPI: (1) the asynchronous operations and (2) the wildcard receiving operations [1, 2]. The calling of asynchronous operations (e.g., *MPI_Isend*, *MPI_Irecv* and *MPI_Iprobe*, etc.) may have nondeterministic effects on program states depending on the timing of message transmissions. The wildcard receiving operations (e.g., the calling of *MPI_Recv* with the wildcard parameter) may accept an arbitrary message if several messages are racing to arrive.

The design of dMPI is to eliminate these nondeterministic effects while preserving performance maximally. Note that there may be other kinds of non-determinism in MPI programs (e.g., the nondeterminism of thread concurrency inside one process). As these problems could be addressed by existing approaches (e.g., deterministic scheduling [6]), they are not discussed in this paper.

2.2 Design Overview

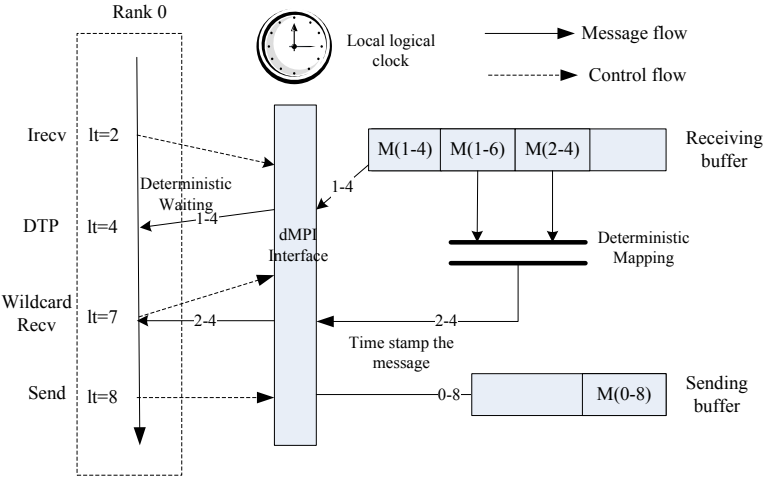

Fig. 1. The design overview of dMPI. "M(1-4)" indicates a message whose source is rank 1 and whose logical time is 4. "lt=2" indicates current logical time is 2.

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 Figure 1. We set a local logical clock in each process to time stamp every concerning event (e.g., the calling of *MPI_Send*). We add two mechanisms to ensure determinism: the *deterministic waiting* mechanism and the d*eterministic 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 positions of DTPs are defined by logical time to ensure determinism. The deterministic waiting mechanism forces the transmission of asynchronous message to be finished exactly at its corresponding DTP (as shown in Figure 1), which hides the nondeterminism 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, it should only 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 (as shown in Figure 1, the wildcard receive always accepts M(2-4)). In this way, the deterministic mapping mechanism resolves the nondeterminism of message races.

2.3 Clock and Logical Time

Logical time is used to order messages and label DTPs in dMPI. To maintain a deterministic logical time, we propose the *world clock* which is a hybrid of Lamport clock [12] and vector clock [11]. 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. Different from vector clock [11], world clock only time stamp messages with a scalar value of the local time (we add an 8-bytes field to all kinds MPI packets). This design simplifies the mechanism of normal vector clock and reduces the communication overhead. In the clock vector, only the local time (the logical time of the local process) is precise, as it is driven by local events (e.g., calling of MPI operations, etc.). The times of other processes may be old values, as they are driven by external messages—when an incoming message is accepted, the timestamp piggybacked in this message will be used to refresh the corresponding clock vector slot if the timestamp is bigger.

To provide a correct and efficient support for message ordering and DTP creation, the driven of local logical time must meet two requirements: (1) local logical time must be precise enough to distinguish DTPs and 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 MPI APIs (e.g., *MPI_Send*, *MPI_Recv*, etc.). By doing so, no two MPI function calls will happen at the same logical time, thus messages and DTPs are easily distinguished by logical time.

The second requirement is achieved by compile-time instrumentation and the using of empirical statistics. At compile-time, we insert a *time_tick* event for each function call to increase the local logical time. The increased value of this function is derived from the function length (the predicted execution time of the function is based on instruction count). The compile-time instrumentation works well for memory-only codes, 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 set an empirical studying table to describe the logical time values of special operations. For example, the calling of Sleep(1) is corresponding to 1000 logical time unit in our empirical studying.

2.4 Deterministic Waiting

The logical time is used to setup DTPs to confine the finishing points of transmission operations. For any asynchronous transmission operation (e.g., *MPI_Isend*), the corresponding DTP is inserted at the program point that is K logical time units after the function call. Note that the value K should be constant so as to ensure determinism (In the example of Figure 1, $K = 2$).

dMPI enforces that the transmission of message be finished right at its corresponding DTP. When an asynchronous request is posted, dMPI adds a DTP for that request. If the message arrives earlier, dMPI postpones the declaration of its arrival until the process reaches the DTP. Therefore, the issued testing function (e.g., *MPI_Test* or *MPI_Probe*) before the DTP could not perceive the arriving of the message. In the contrary, if a message is late, dMPI forces the process to wait at the DTP until the message arrives. Therefore, the testing function after the DTP will always detect the message. Note that the design of DTP only constrains the declaration of the finishing of message transmissions, while the underlying transmission is not affected. In this way, DTP creates a deterministic environment for the upper applications. To mitigate

the performance degradation caused by DTP waiting, we also adopt a buffer-strategy to pre-receive messages that cannot be accepted at once.

2.5 Deterministic Mapping

The deterministic mapping mechanism solves the non-determinism of message race. 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 an 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 leverage the world clock to determine the earliest messages. 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. Otherwise, the message is not the earliest and the process 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 process being blocked for a long time. To mitigate this problem, we introduce the small-sized *control messages* to exchange information between MPI processes. 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 need to send the *Answer* when its logical time is greater than the expected value.

2.6 Dead Locks

The deterministic mechanisms in dMPI may introduce *deterministic deadlocks* which are caused by the extra wait-for dependencies. We propose a *lightweight deadlock checker* (LW-DC) to detect and solve these deadlocks. Different from existing general deadlock detectors [10], LW-DC is supposed to be a special deadlock detector for deterministic deadlocks only. LW-DC is a hybrid approach of timeout deadlock detector and dependency analysis detector. Since the wildcard receiving will accept a determinate message in dMPI, it greatly simplifies the design. The working procedure of LW-DC is as follows. Each process will start to check for deadlock when it stops for a while (100ms 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. Due to space limitation, we do not discuss LW-DC in detail in this paper.

3 Evaluation

3.1 Methodology

We implemented dMPI based on MPICH2 (version 1.4) [7]. Our evaluation hardware is consisted of 2 AMD servers connected with a 1GB/s switch. Each server has a 2.2GHz CPU with 4 cores, 1GB memory and a 1GB/s Ethernet card, running Fedora 12 with Linux kernel version 2.6.31.5. We used the NPB benchmarks [8] to test dMPI. 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 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.

To evaluate the performance, we setup a baseline execution (the performance of the standard MPICH2). We compared the performance of dMPI with the baseline execution. For each benchmark, we ran it 10 times to collect the mean value.

3.2 Execution Time

The performance of dMPI is shown in Figure 2. 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; (3) the *determinism overhead*, which is caused by the deterministic mechanisms. Overall, the average overhead of dMPI is small (below 8%).

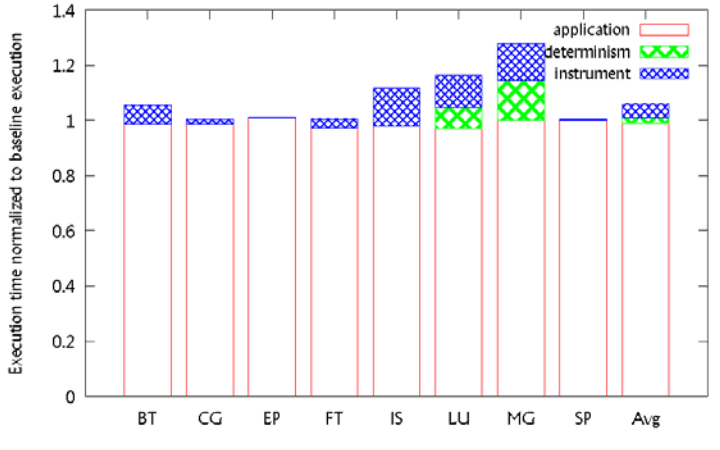

Fig. 2. The execution time of dMPI compared with that of the standard MPI library.

Table 1. The Detailed Information of Programs Running with dMPI

	dMPI profiling			Normal messages		Control messages		
benchmark	DTP	wildcard	deadlock	count	size (Byte)	count	size (byte)	count rate (%)
bt	2418	0	0	2436	283918712	0	0	0
cg	1680	0	0	1683	46658367	0	0	0
ep	0	0	0	9	780	0	0	0
ft	0	0	0	40	201330833	0	0	0
is	0.75	0	0	128	92358592	0	0	0
lu	510	510	13.25	31661	122780595	789	3194	2.5
mg	714	714	9	867	32057558	622	2490	71.7
sp	4818	0	0	4836	493724232	0	0	0
avg	1267.5	153	2.78	2436	283918712	176.4	710.5	29

In Table 1, we provide the detailed profiling data of programs running under dMPI. As seen from this table, the impact of our deterministic message passing mechanism is trivial: For each program on average, dMPI additionally handles about 1200 DTPs (Column 2), 150 wildcard receiving operations (Colum 3), 2 or 3 deadlocks (Column 5). The introduced control messages are also trivial compared with the normal messages in size and number (Column 5-9).

4 Related Work

Record & replay is a typical method to eliminate the nondeterminism of MPI [1-5]. Some of them record the nondeterministic message order in the recording phase, and force the messages to follow the order of the recorded logs in the replay phase [1]. Some others record the contents of messages instead of their orders [4]. A hybrid method—MPIWiz [3] record both message order and contents. Distinct from these works, we provide a solution for inherent determinism without any external supports (e.g., logs), which is efficient in performance and is convenient to deploy.

Inherent determinism has been introduced in the shared-memory multi-processing field to facilitate debugging of multi-threaded programs [6, 9]. Although we share the same concept of inherent determinism with these works, our work is to address the nondeterministic problem of the message passing mechanism in the distributed environment, which is different from the problem of the shared-memory environment.

5 Conclusion and Future Work

This paper designs dMPI to facilitate the debugging of MPI programs. dMPI ensures inherent determinism by using logical time to control the finishing points of asynchronous transmissions and the accepted messages of wildcard receiving operations. We implemented and evaluated dMPI to demonstrate its practicality. The evaluation results show that the overhead of dMPI is practical and low (8%). In the future work,

we will use buffering and speculation mechanism to further mitigate the performance impact of dMPI.

Acknowledgment

This work is partially supported by National High-tech R&D Program of China (863 Program) under Grants 2012AA01A301, and by National Science Foundation (NSF) China 61103082, 61003075, 61170261 and 61103193.

References

1. 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.
2. 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.
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," in PPoPP, 2009, pp. 251-260.
4. M. Maruyama, T. Tsumura, and H. Nakashima, "Parallel program debugging based on data-replay," in PDCS, 2005, pp. 151C156.
5. D. Kranzlmller, C. Schaubschl ger, 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.
6. D. Joseph, L. Brandon, C. Luis, and O. Mark, "DMP: deterministic shared memory multiprocessing," in Proceeding of the 14th international conference on Architectural support for programming languages and operating systems Washington, DC, USA: ACM, 2009.
7. "MPICH2, http://www.mcs.anl.gov/research/projects/mpich2/."
8. D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Mail Stop T 27 A-1, Moffett Field, CA 94035- 1000, USA, Dec. 05 1995.
9. R. L. Bocchino Jr, V. S. Adve, S. V. Adve, and M. Snir, "Parallel programming must be deterministic by default," in Proceedings of the First USENIX conference on Hot topics in parallelism, 2009, pp. 4-4.
10. 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.
11. C. J. Fidge., "Partial orders for parallel debugging," in ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, January 1989, pp. 24(1): 183-194.
12. L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Communications of the ACM, vol. 21, pp. 558-565, 1978.