Clock Delta Compression for Scalable Order-Replay of Non-Deterministic Parallel Applications

Kento Sato, Dong H. Ahn, Ignacio Laguna, Gregory L. Lee, Martin Schulz
Lawrence Livermore National Laboratory
7000 East Ave. Livermore, CA
{kento, ahn1, ilaguna, lee218, schulzm}@llnl.gov

ABSTRACT

The ability to record and replay program execution helps significantly in debugging non-deterministic MPI applications by reproducing message-receive orders. However, the large amount of data that traditional record-and-reply techniques record precludes its practical applicability to massively parallel applications. In this paper, we propose a new compression algorithm, Clock Delta Compression (CDC), for scalable record and replay of non-deterministic MPI applications. CDC defines a reference order of message receives based on a totally ordered relation using Lamport clocks, and only records the differences between this reference logical-clock order and an observed order. Our evaluation shows that CDC significantly reduces the record data size. For example, when we apply CDC to Monte Carlo particle transport Benchmark (MCB), which represents common non-deterministic communication patterns, CDC reduces the record size by approximately two orders of magnitude compared to traditional techniques and incurs between 13.1% and 25.5% of runtime overhead.

CCS Concepts

•**Software and its engineering** → **Software testing and debugging;** *Ultra-large-scale systems;* •**Information systems** → *Data compression;* •**Theory of computation** → *Parallel computing models;*

Keywords

Debugging tools, Non-determinism, Compression

1. INTRODUCTION

Debugging massively parallel applications remains a highly challenging task. With trends towards larger and more complex supercomputers [17, 23, 18], remarkably increasing degrees of parallelism, more parallelism options (e.g., heterogeneity), and emerging programming models, applications gain higher performance and scalability by using more asynchronous algorithms. However, they come at a productivity cost: they introduce non-determinism in parallel program execution—i.e., the applications do not produce the same output in different runs—and this makes debugging even a greater challenge.

A particularly well-known source of non-determinism at large scale is the message-passing interface (MPI) [10]. As network and system noise can affect the order of received messages [12], applications can take different computation paths depending on the order of the received messages. This complicates debugging since computation paths and associated computational results may vary between the original run (where a bug manifested itself) and the debugged runs.

Record-and-replay tools are promising solutions to eliminate non-determinism in MPI applications. When an application executes, this approach records the execution of each MPI process as trace data, which may include payloads of exchanged messages as well as the order of the message receives. Then, during debugging, a replay mechanism uses these recorded traces to ensure that every MPI process observes the same message exchanges as the recorded run.

In record-and-replay, there exist three common approaches: (1) all the received messages, i.e., both message payloads and the order, are recorded in each process, also known as *data-replay* [21, 6, 2]; (2) only the order of the received messages is recorded, also known as *order-replay* [13, 15, 14]; and (3) hybrid approaches which combine data- and order-replay [28]. While these record-and-replay approaches significantly help in debugging non-deterministic parallel applications, they record a large amount of data, which limits their practical use at extreme scale.

Most high-end computing systems make use of a parallel file system for storage, thus the performance of any record-and-replay technique is ultimately limited by how efficiently recorded data can be placed in this storage. One can conceive a more scalable approach by storing the data in node-local storage (e.g., ramdisk), but this is a highly limited space (e.g., a typical application uses over 90% of memory) which can quickly run out. Therefore, a recording approach that requires a minimal storage footprint is highly valuable in scaling a record-and-replay tool.

In this paper, we propose a storage-efficient, low-overhead recording scheme called *Clock Delta Compression* (CDC) that dramatically reduces the size of the recorded data for scalable order-replay. CDC defines a reference logical-clock order of message receives based on a totally ordered relation using Lamport clocks[16] and only records the differences

SC '15, November 15-20, 2015, Austin, TX, USA
 ISBN 978-1-4503-3723-6/15/11…$15.00
DOI: http://dx.doi.org/10.1145/2807591.2807642

between this reference logical-clock order and an observed order. This builds on the observation that the reference logical-clock orders in each MPI rank are very similar to actual observed orders. Recording only the order differences is also highly compressible, further reducing the storage use. In addition, CDC reduces recording performance overhead by using asynchronous recording via a dedicated thread. We use the MPI profiling interface (PMPI) to intercept MPI calls and piggyback Lamport timestamps. In contrast to previous approaches that use dynamic instrumentation to intercept MPI calls [28], our scheme incurs substantially less application slowdown and provides higher portability. More specifically, this paper makes the following contributions:

- A highly scalable, low-overhead recording scheme for order-replay to mitigate the harmful effects of non-determinism in MPI applications;
- A detailed explanation of the CDC mechanisms and algorithm that significantly reduce the size of the recorded data;
- A correctness proof of the CDC algorithm as well as the subsequent replay phase that uses the CDC-based recording scheme;
- A quantitative performance evaluation of our order-replay approach compared to other approaches.

Our performance evaluation on MCB, a Monte Carlo particle transport benchmark (from the CORAL benchmark suite [1]) featuring a common asynchronous, non-deterministic communication pattern, shows that CDC minimally slows down the application at large scale and uses a compact storage footprint. For example, to record a 24-hour MCB simulation at 3,072 MPI processes, our prototype slows down the execution as low as 14.2% and limits the trace data as little as 500 MB per compute node. The storage efficiency is largely attributed to the compression algorithm. Especially, the CDC compression rate is 44.4× higher than one without compression, and 5.7× higher than gzip [8], which is a commonly used general-purpose file compression technique.

2. MOTIVATIONAL CASES

Reproducibility, the ability to repeat program executions with the same numerical result or code behavior, is highly desirable for debugging applications, including (or particularly) MPI applications running at large scale. But there exist many MPI features that can prevent them from achieving this ability. For instance, wild-card parameters such as `MPI_ANY_SOURCE` allow a receive to be matched with a message sent by any MPI rank, and waiting and testing receive requests (e.g., `MPI_Waitany`, `MPI_Waitsome`, `MPI_Testany` and `MPI_Testsome`) can also allow receives to be matched by any subset of MPI ranks out of all waiting and testing receive requests. While these MPI functions enable asynchronous algorithms, they can vary the application's code behavior and numerical results. More applications are adopting asynchronous algorithms to achieve high scalability, and this presents significant challenges to debugging.

2.1 Non-deterministic MPI applications

A classical trade-off between performance and reproducibility can be found in a domain-decomposed particle Monte Carlo algorithm [3], which is in use by many codes developed at the Lawrence Livermore National Laboratory (LLNL). In any such code, the MPI processes must exchange two types of messages: a message for the particles that cross domain boundaries and need to be sent to neighbor processes; and a message to coordinate the exit of the particle-processing loop at the end of the time step. The scientists found that they must make *all* communications needed for these exchanges asynchronous, in order to scale to today's massively parallel systems with millions of hardware threads [17].

Unfortunately, this asynchrony caused the algorithm to produce different numerical solutions from one run to another, even under the exact same environment. In the beginning of each time step, each MPI process posts non-blocking receives for all possible incoming messages and detects incoming message receives by periodically calling `MPI_Testsome` after processing a certain number of local tasks, e.g., processing local particles. Upon detecting a newly received particle, the MPI process simply appends it to its local particle list and immediately posts another non-blocking receive for the next message from the same MPI process.

While highly efficient, the *first-come, first-served* algorithm allows each MPI rank to process its particles in a different order from run to run. If two neighbors send particles at the same time, the receiver may process the particle sent by either sender in the opposite order in two different runs. As double precision arithmetic is not associative, i.e., $a + (b + c) \neq (a + b) + c$, summing up certain global tallies over the particles may produce deviations. These varying solutions are all statistically equivalent in production simulations. However, the programmers have had difficult times in debugging the codes, in particular because the variation can hide or confuse the effects of a bug.

The particle Monte Carlo algorithm is just one example that illustrates how massively parallel MPI applications are becoming increasingly non-deterministic. In order to run simulations efficiently on massively parallel systems with hundreds of thousands or millions of compute elements [17], asynchronous parallel algorithms are becoming increasingly commonplace. Further, the MPI standard has been responding to their needs with more advanced features (e.g., non-blocking collectives) and network vendors have been introducing various optimization for collectives; both of which can add an additional dimension to non-determinism.

2.2 Record-and-replay as General Solution

A highly scalable record-and-replay technique is an attractive solution in combating non-determinism. With this technique, programmers can simply turn on recording of communication behavior in one run and deterministically replay the same communication behavior for subsequent runs. This is a far more general approach than *ad hoc* algorithmic solutions that programmers often employ to mitigate their *numerical reproducibility* issues [7, 19]. In particular, this technique not only helps programmers debug their code through numerical variations, but also through other variations in code behavior (e.g., intermittent crashes) without having to require any change to the code.

Sadly, the existing approaches [21, 6, 2, 13, 15, 14, 28] fall far short of meeting the needs of the high-end HPC environments. Most require very large storage and use excessive file

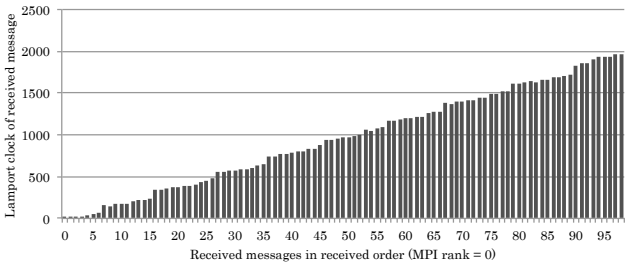

Figure 1: Lamport clock values of received messages in MCB (MPI rank = 0)

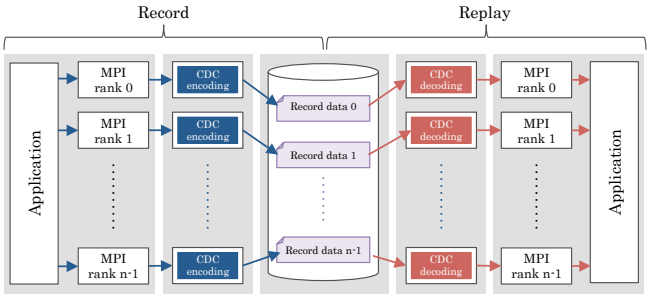

Figure 2: Record and Replay with CDC

I/O to record the traces. To make this technique practical at large scale, we must significantly lower the size of recorded data, ideally, small enough to fit into node-local storage (e.g., ramdisk), even for very long simulations. Avoiding global file system accesses not only improves the performance of the recording phase but also that of the replay: a subsequent replay accesses the data only from local storage where the application was initially executed.

3. CLOCK DELTA COMPRESSION (CDC)

Providing highly-scalable order-replay as a general solution for non-determinism control requires at least one order of magnitude more efficient storage usage. We achieve this by exploiting our empirical observation on the common non-deterministic communication patterns described in Section 2: *the global order of messages exchanged among MPI processes are very similar to a logical-clock order (e.g., Lamport clock)*. As an indication of this observation, Figure 1 shows the Lamport clock values of the particle-exchange messages that MPI rank 0 of MCB has received in sequence when running at 48 processes. It is remarkable that the received Lamport-clock values almost always monotonically increase. In other words, most messages are received in the order of Lamport clocks.

Exploiting this observation for the first time, we can significantly decrease the storage usage by recording only those receives that deviate from this reference logical-clock order. One can cheaply establish the total-order reference system of messages by piggybacking each message with a Lamport clock and using the MPI rank as the arbitrary mechanism to break ties. For a long simulation, however, even a small deviation can still grow the recorded data large. Thus, we use

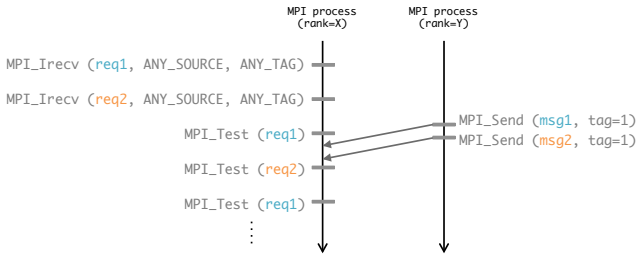

Figure 3: Application-level out-of-order receives

a series of compression techniques that are highly tailored to this order differential data.

This novel encoding, compression and decoding scheme is called Clock Delta Compression (CDC). More specifically, Figure 2 illustrates the end-to-end tool flow of the record-and-replay scheme with CDC. Each application process intercepts MPI calls via PMPI to keep track of the information on receives. During the record phase, when an MPI process handles communication events, MPI passes these events to CDC, which then encodes and compresses these events before placing them to storage. During the replay phase, CDC reads and decodes the recorded data, and then the MPI processes replay the corresponding communication events.

Message sends become deterministic if message receives are replayed in most of non-deterministic applications (as formulated later in Definition 7) [5]. Even if message sends are non-deterministic due to return values of non-deterministic function calls for acquiring current times or random numbers, we can still make these functions deterministic by simply recording the return values of the functions [28]. Thus, this work only targets message receives, similarly to other order-replay techniques [13, 15, 14].

In the remaining of this section, we first describe the minimum set of information required by general record-and-replay techniques for correct replays (Section 3.1) and then detail how CDC compresses the record data (Section 3.2, 3.3 and 3.4).

3.1 Base Record Needed for Order-Replay

In general, to replay a specific message-receive order in message matching functions (MF), such as the `MPI_Wait` family (i.e., `MPI_Wait`, `UPI_Waitall`, `MPI_Waitany` and `MPI_Waitsome`) and the `MPI_Test` family (i.e., `MPI_Test`, `MPI_Testall`, `MPI_Testany` and `MPI_Testsome`), we need to record a trio: *matching status*, *matched message set* and *message identifier* on each MF call for every MPI process. The *matching status* is to record if the test is matched or not in the `MPI_Test` family. By recording the `flag` value returned via an `MPI_Test` call, we record matching statuses. In `MPI_Waitall`, `MPI_Waitsome`, `MPI_Testall` and `MPI_Testsome`, multiple messages can be matched in a single MF call. The *matched message set* is to record which messages are matched in a single MF call. The *message identifier* is to identify a message uniquely, distinguishing it from the other messages.

Traditional order-replay techniques record the `source` and `tag` obtained via `MPI_Status` as the message identifier. However, through our analysis using non-deterministic applications, we found that a pair of `source` and `tag` cannot

	matching status	*matched message set*	*message identifier*	
count	flag	with_next	rank	clock
1	1	0	0	2
2	0	--	--	--
1	1	1	0	13
1	1	0	2	8
1	1	0	1	8
1	1	0	0	15
1	1	0	1	19
3	0	--	--	--
1	1	0	0	17
1	0	--	--	--
1	1	0	0	18

Figure 4: Original Record

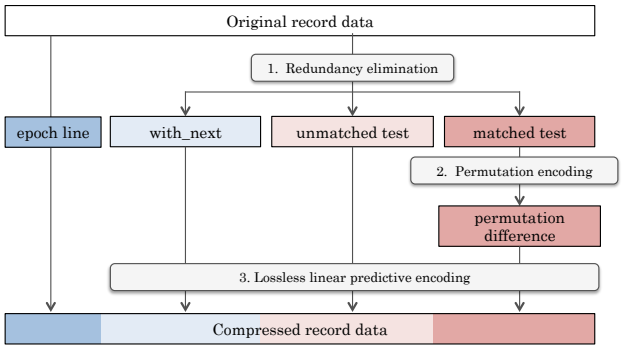

Figure 5: Overview of CDC

uniquely identify a message. While MPI guarantees that an MPI rank receives messages in the order of sends from a same sender, applications can still receive these messages out of order because of subtle timing issues with MPI calls. Figure 3 shows such an example. Since messages are received in the order of sends from a same sender at the MPI level, `req1` matches `msg1`, and `req2` matches `msg2`. At the application-level, however, the application is notified of the message receive of `msg2` first, and then `msg1`. Both `msg1` and `msg2` have the same message identifier, i.e., source=Y, and tag=1, but the order between `msg1` and `msg2` can change across different runs.

If an order-replay system records `source` and `tag` for the message identifier, the system cannot correctly replay when `msg1` is notified first during the record phase, but `msg2` is notified first during a replay. Unless the application attaches genuine message identifiers to each message in order to distinguish messages (e.g. `msg1` and `msg2`), the application can experience incorrect replays. To solve the application-level out-of-order problem, we use logical clocks, *Lamport cock* (`clock`). In our scheme, a Lamport clock contributes to uniquely identifying messages. We piggyback each message with a clock, and we use `source` as well as the `clock` as our message identifier. The detailed definition of our Lamport clock rule is in Definition 4.

In summary, recording correct replays requires five values: `count`, `flag`, `with_next`, `rank`, and `clock`. `count` is the recurrences of the same event captured in the row, `flag` records the *matching status*, and `with_next` is to record if a message is received with the next message, retaining the *matched message set* for `MPI_Waitall`, `MPI_Waitsome`, `MPI_Testall`, and `MPI_Testsome`. Figure 4 shows an example of a recording table that each process must manage independently. Because order-replay needs to record a quintuple for correct replays, this process needs to write 55 values (the five values $\times$ 11 events) to storage in this example. In the rest of the sections, we demonstrate how to reduce the 55 values down to 19 values through CDC; *redundancy elimination*, permutation encoding and *linear predictive encoding* (Figure 5).

3.2 Redundancy Elimination

We first perform redundancy elimination since the recording table has redundant information. CDC divides the record table into a *matched-test* table, a *with_next* table, and an *unmatched-test* table (Figure 6) so that we can remove the redundant information.

For `with_next`, we only record which messages are received with the next message. Therefore, if an application does not call `MPI_Waitall`, `MPI_Waitsome`, `MPI_Testall` and `MPI_Testsome`, the size of the with_next table becomes zero because only one message can match in a single MF call. For the unmatched-test table, we only record how many times unmatched-test events happen before receiving messages. If no unmatched-test event happens before receiving messages, CDC does not include the event in the table. For example, if an application does not call the `MPI_Test` family, unmatched-test events also will not occur, and the size of the unmatched-test table becomes zero. After this redundancy elimination, CDC can reduce the number of recording values to 23 values in the example.

3.3 Permutation Encoding

Next, CDC applies permutation encoding to the matched-test table. In the permutation encoding, CDC defines a reference logical-clock order based on Lamport clocks (*reference order*), and computes the permutation difference to an actual receive order (*observed order*) by using an edit distance algorithm (Section 4.1). Then, CDC records in the permutation-difference table how many messages are delayed (`delay`) compared to the reference order as in Figure 7. If the observed order is completely identical to the reference order, CDC records *nothing* for the matched-test table. For example, if a rank receives messages from senders with monotonically increasing clock values, the recording size for the matched-test table becomes zero.

If a rank receives multiple messages with same clocks (e.g. clock 8 in Figure 4), CDC defines a message from a smaller `rank` to be earlier than the ones from bigger `rank`s (Definition 6). Because the permutation difference between an observed order and a reference order is based on receiving clocks, CDC cannot correctly replay if the receiving clocks change from run to run for replays. Since the message receives are non-deterministic, the piggybacked clocks may vary from run to run. However, as validated in Section 5, CDC can send consistent piggybacked clock values, and therefore, can correctly replay.

3.4 Linear Predictive Encoding

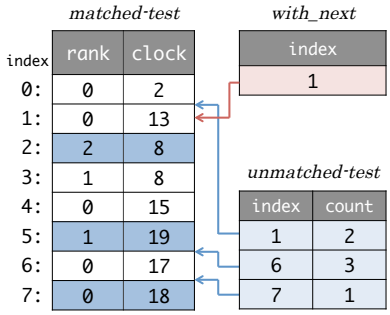

Figure 6: Redundancy elimination

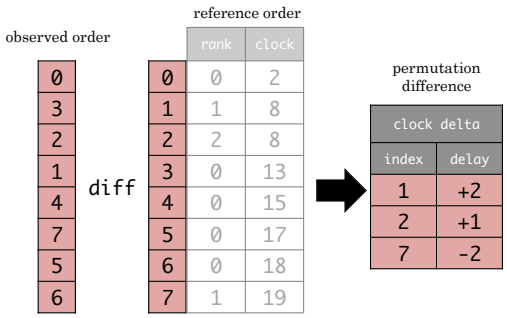

Figure 7: Record only permutation difference by using edit distance

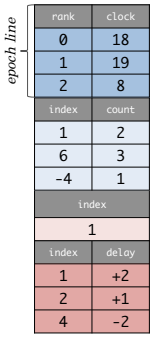

Figure 8: CDC encoding format

If the permutation difference is small, each `delay` value in the permutation-difference table is also expected to be close to zero. Meanwhile, `index` values in the with_next, unmatched-, and matched-test tables monotonically increase as the length of the table increases. To achieve a higher compression rate to `index` values in gzip, CDC applies Linear Predictive (LP) encoding to the `index` values.

LP encoding is a compression technique used in audio data [4] and predicts $x_{n(>0)}$ from the past p number of values as:

$$\hat{x}_n = \sum_{i=1}^{p} a_i \times x_{n-i} \quad (x_{i\leq 0} = 0) \tag{1}$$

$$e_n = x_n - \hat{x}_n \tag{2}$$

where $\hat{x}_n$ is a predicted value of x_n, e_n is an error. With LP encoding, if we store $\{e_1, \ldots, e_n\}$, we can recursively restore $\{x_1, \ldots, x_n\}$, because e_1 is always same as x_1 ($e_1 = x_1 - \hat{x}_1 = x_1 - 0 = x_1$). In terms of the length of storing values, $\{x_1, \ldots, x_n\}$ and $\{e_1, \ldots, e_n\}$ are the same size. However, if we can give an accurate prediction to x_i by a choice of p, and $a_{i=1\ldots p}$, e_i becomes close to zero, thereby we can achieve a high compression rate to the sequence using gzip.

For the prediction of integer values for `index`, we predict x_n assuming x_n is on an extension of a line created by x_{n-1} and x_{n-2}, i.e., $x_n - x_{n-1} = x_{n-1} - x_{n-2}$, which gives $p = 2, (a_1, a_2) = (2, -1)$. Thus, e_n can be represented as:

$$e_n = x_n - 2x_{n-1} + x_{n-2} \quad (x_{n\leq 0} = 0) \tag{3}$$

For example, if we encode sequence of $\{1, 2, 4, 6, 8, 12, 17\}$, the encoded sequence becomes $\{1, 0, 1, 0, 0, 2, 1\}$. If we apply the linear predictive encoding to `index` values in the tables of Figure 6, 7, and combine the tables, the CDC encoding format is represented as in Figure 8.

3.5 Epoch Enforcement

In practice, debugging tools need to minimize memory usage, therefore, record-and-replay needs to periodically flush out records from memory to local storage as a chunk. However, if applications periodically flush out the chunks, CDC may not correctly replay the order of message receives in some cases. For example, if CDC encodes and flushes out the record shown in Figure 4 as a chunk in record mode, incorrect replay can occur. In replay mode, for example, if the MPI rank receives a message with (`rank`:2, `clock`:17) earlier than a message with (`rank`:0, `clock`:18), incorrect replay occurs because the message (`rank`:2, `clock`:17) needs to be replayed using one of the subsequent chunks.

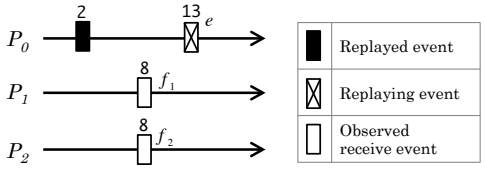

Figure 9: Example: Condition for correct replay

To guarantee that each message is replayed using an appropriate chunk, CDC also adds an *epoch line* table. With the epoch line, CDC can know which chunk of record should be used for each receive message. In this example, because the message (`rank`:2, `clock`:17) runs off the epoch line, CDC can know this message should be replayed by one of the subsequent chunks in the record. The complete CDC format is represented in Figure 8. By applying the redundancy exclusion, the permutation encoding, and the LP encoding to the record in Figure 4, we can reduce the number of storing values from 55 to 19. Finally, CDC applies gzip to the CDC encoding format. The CDC format is highly compressible by gzip because most of the values are expected to be close to zero.

3.6 Condition for Correct Replay

When decoding for a replay, CDC retains multiple messages, permutates the messages, and returns them in the permutated order to the application. Therefore, CDC can correctly replay only when CDC receives the certain number of messages, and can create the same reference order as the one in the record. To formulate the condition, we give an example in Figure 9. Figure 9 shows a snapshot when CDC replays receive event e where the first message receive is already replayed, and other messages (f_1, f_2), are already received but not replayed. First, to correctly replay e, clocks of the receive events, f_1, f_2 and e, need to be replayed (Axiom 1 (i)). Then, given the receive events, CDC computes reference order based on the received clocks ($\{f_1 = 8, f_2 = 8, e = 13\}$). Thus, CDC must wait until f_1, f_2 are received (Axiom 1 (ii)). Then, CDC permutates e to become the next replay event, i.e., $\{f_1 = 8, f_2 = 8, e = 13\} \rightarrow \{e = 13, f_1 = 8, f_2 = 8\}$ based on the permutation difference table. In addition, the reference order of message receive order must be consistent between a record and a replay. Therefore, CDC needs to make sure that the

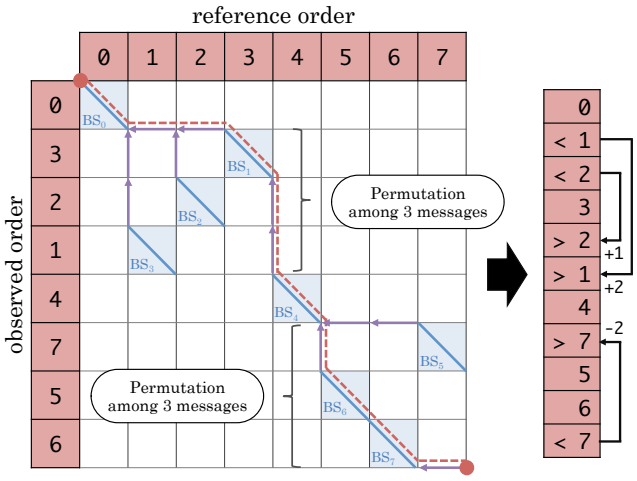

Figure 10: Edit Distance Matrix in CDC

minimum clock of the next receive message is more that 13 (Axiom 1 (iii)). We denote the minimum clock of the next receive message as Local Minimum Clock (LMC). In general, these three conditions for correct replay of CDC can be formulated as Axiom 1 in Section 5. We will prove the correctness in Section 5.

4. IMPLEMENTATION OF CDC

4.1 Fast Edit Distance Algorithm for CDC

To compute the permutation difference between an observed order, B ($= \{b_0, \ldots, b_{N-1}\}$), and a reference order, P ($= \{p_0, \ldots, p_{N-1}\}$), we use an edit distance algorithm (EDA) [20]. Edit distance algorithms are used in natural language processing and computational biology, e.g., to compute the sequence alignment of two DNA strings, the similarity of two documents, or to perform spell checking. For our purposes, we apply the edit distance algorithm to compute differences between observed and reference order of message receives. Figure 10 illustrates an edit distance matrix (EDM) created to compute the permutation distance between $B = \{0, 3, 2, 1, 4, 7, 5, 6\}$ and $P = \{0, 1, 2, 3, 4, 5, 6, 7\}$ in the example of Figure 7. When an EDA creates an EDM, the EDA finds all of indices i and j where $b_i = p_j$, which are illustrated as a *blue backslash* (BS_k) in Figure 10. Thus, time complexity of the EDA is generally $\mathcal{O}(N^2)$ to find a minimum edit distance.

Thankfully, if we apply an EDA to CDC, we can reduce the time complexity to $\mathcal{O}(N + D)$ where D is the edit distance because we can make two assumptions to input arrays, B and P. First, B is a permutation of P, i.e., $x \in B \Leftrightarrow x \in P$. Second, P consists of sequential numbers. Thus, CDC can easily find indices i and j where $b_i = p_j$, such that $j = b_i$ by $\mathcal{O}(1)$. The time complexity of this operation becomes $\mathcal{O}(N)$ for B and P of length N. In addition, to find the minimal edit distance, we can dynamically find the shortest path from the BS_k based on *Manhattan distance* when BS_k is created. This operation is illustrated as a *purple arrow* in Figure 10. For example, when BS_4 is created, the EDA finds the shortest path from the BS_4 to the $BS_{k=1,2,3}$. Thus, time complexity of this operation becomes $\mathcal{O}(D)$, and overall time complexity of EDA for CDC becomes $\mathcal{O}(N + D)$. If differences between the reference order and an observed order are small, D becomes small, thereby the encoding time becomes short as well.

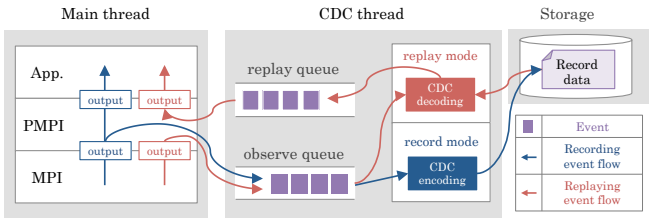

Figure 11: Overview of CDC implementation

In EDA, an edit distance is generally described by *insertion*(>), *deletion*(<) and *substitution*(|). In this example, the difference is described in the array on the right-side of Figure 10. Since the condition, $x \in B \Leftrightarrow x \in P$, is held in CDC, substitution does not exist, and a pair of "$< x$" and "$> x$" exists in any permutated x. Thus, the pair of the $<$ and $>$ can be replaced into *permutation* as shown in Figure 7.

4.2 Asynchronous Recording

Both encoding and file I/O are costly operations. If CDC blocks application processes whenever encoding and writing record chunks, the overall performance of the application would significantly degrade. To avoid this overhead, we asynchronously record message receives. Figure 11 shows the CDC implementation overview for record and replay. Using a CDC-dedicated thread (*CDC thread*), CDC overlaps the application's computation (*main thread*).

In record mode, CDC tracks all of message-receive events through the MPI profiling interface (PMPI) [10]. The main thread enqueues events to the *observe queue*, which is managed by the CDC thread to receive the events. One event contains `count`, `flag`, `with_next`, `rank` and `clock` as described in Figure 4. Then, the CDC thread dequeues, and applies CDC to the events in the observe queue, then writes the encoded record to storage. CDC creates a single observe queue for a pair of main and CDC thread. Since this queue model is a single-producer (main thread), single-consumer (CDC thread) model, i.e., *SPSC model*, both main and CDC thread can concurrently enqueue and dequeue events race free without needing explicit mutual exclusion.

In replay mode, CDC also tracks all message-receive events through PMPI. Then, the CDC thread permutates the observed message-receive events according to the record written during the record phase and enqueues the replayed events to the *replay queue*. Then, the PMPI layer dequeues the replayed events and returns to the main thread. Since the replay queue is also an SPSC model, the main and CDC threads can exchange events without needing explicit mutual exclusion. Because CDC permutates observed message-receive events for the replay, the CDC thread needs to retain a certain number of messages in the observe queue and wait until enough messages are received for the permutation. In the example of Figure 10, CDC waits until all the three messages, b_1, b_2 and b_3, are received to replay them, and similarly in b_5, b_6 and b_7.

4.3 Replayable Clock

Clock piggybacking is indispensable for CDC to create the

reference order. Some systems have the wall-clock time generated from a highly accurate physical global clock, and on such systems one may think this would create a reference order, which is more close to the corresponding observed order. However, wall clock is neither deterministic (run to run) nor replayable because it changes the reference order in subsequent replays, and thus cannot be used for reliable replay in CDC. As mentioned before, our approach is to use a Lamport clock. Although *Lamport clocks* received by an MPI process can vary slightly from run to run due to non-determinism in message receives, Lamport clocks are replayable, which we validate in Theorem 2 in Section 5. Another approach would be to use a *Vector clock*. Unfortunately, Vector clocks are not scalable [26]. Thus, we employ a Lamport clock following rules defined in Definition 4 for creating the reference order of message receives. For future work, we will consider other replayable clock definitions to further increase similarity between the reference and observed orders.

To send a piggyback clock, we use MPI datatypes to attach piggyback data [24] to a message payload. Because we use several PMPI layers for CDC, we integrate the PMPI layers using the P^{N}MPI infrastructure [25]. Piggybacking in MPI is known to degrade communication performance [24]. However, as shown in Figure 16, with improved datatype support in modern MPI implementations the overhead is small, in particular for the domain of debugging tools. What is more important is for the application to maintain scalability under record and replay, which CDC enables.

4.4 Matching Function (MF) Identification

Non-deterministic applications usually use several MF calls at different locations in the program. Different MF instances are used for different purposes, therefore there are different dependencies among messages exchanged via different MFs. If we separately create reference orders for different MFs, we can create a reference order that more closely follows the corresponding observed order. To achieve this, when MFs are called, we analyze the call stacks of the function calls, and separately manage the record tables (Table in Figure 4) for the different MF call instances.

5. REPLAY CORRECTNESS

As mentioned in Section 4.3, CDC can correctly replay a message-receive order only if the Lamport clock is correctly replayed. To validate that the clock is replayable and that CDC can correctly replay program executions, we describe the proof in this section.

Definition 1 (Ordered set)**.** If $X = \{x_1, x_2, \dots\}$ is an ordered set, then "$X = \acute{X}$" $\Leftrightarrow$ "$x_i = \acute{x}_i$" where $x_i \in X, \acute{x}_i \in \acute{X}$

Definition 2 (Events)**.** Let e be a send or receive event. Let E be an ordered set of e and contain only send events or only receive events. Let E_i^x be i-th E of process P_x. Let $\boldsymbol{E}$ be an ordered set of E. Let $\boldsymbol{E}^x$ be $\boldsymbol{E}$ of process P_x. Under the definition, if E_i^x is an ordered set of send events, E_{i+1}^x is an ordered set of receive events. Likewise, if E_i^x is an ordered set of receive events, E_{i+1}^x is an ordered set of send events. With this definition, we can describe a process as a series of the events, e. In the example of Figure 12, the process (P_1) can be described as $\{e_0, \dots, e_6\} = \{E_1^1,\ E_2^1,\ E_3^1\} = \boldsymbol{E}^1$.

Definition 3 (Event dependency)**.** If E depends on $\boldsymbol{E}$, we denote the dependency as $\boldsymbol{E} \rightarrow E$. In Figure 12, E_2^0 has a dependency, $\{E_1^1, E_1^0\} \rightarrow E_2^0$.

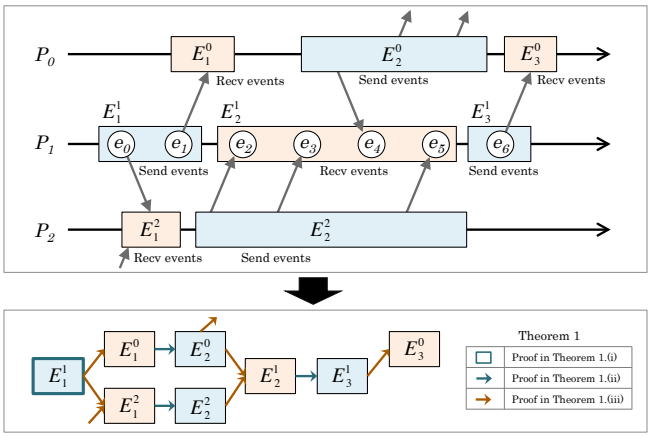

Figure 12: Example: Communication dependency graph with three processes

Definition 4 (Lamport clock)**.** Let a Lamport clock be updated following two rules: (i) When a process sends a message, the process attaches its current clock to the message, then increments the clock by 1; (ii) When a process receives a message, the process sets its clock to be the maximum of the received clock and its own clock, then increments the clock by 1.

Definition 5 (Event clock function)**.** Let f_c be $f_c : E \mapsto \mathbb{N}$, $f_c(e)$ is a clock value of event e. Therefore, "$e \rightarrow f$" $\Rightarrow$ "$f_c(e) < f_c(f)$", or "$f_c(e) \geq f_c(f)$" $\Rightarrow$ "$e \not\rightarrow f$".

Definition 6 (Totally ordered relation for creating the reference order)**.** Let f_m be $f_m : E \mapsto \mathbb{N}$, $f_m(e)$ is an ordering number for message-receive events in CDC where "$f_m(e) < f_m(f)$" $\Leftrightarrow$ "(i) $f_c(e) < f_c(f)$ or (ii) rank of sender $e <$ rank of sender f if $f_c(e) = f_c(f)$". Based on this totally ordered relation, CDC creates reference logical-clock orders.

Definition 7 (Determinism in message send)**.** In non-deterministic applications, we can make two assumptions. (i) The first send events are deterministic, i.e., $^\forall x$ s.t. "E_1^x is send events" $\Rightarrow$ "E_1^x is deterministic", or "$\phi \rightarrow E_1^x$" $\Rightarrow$ "E_1^x is deterministic". In Figure 12, E_1^1 is send events, and has no dependency, i.e., $\phi \rightarrow E_1^1$. Therefore, $E_1^1 = \{e_0, e_1\}$ are deterministic. (ii) Send events are deterministic if the all previous message events are replayed, i.e., "$^\forall \boldsymbol{E} \rightarrow E$ s.t. $\boldsymbol{E}$ is replayed, E is a send event set" $\Rightarrow$ "E is deterministic". In Figure 12, if E_1^1 and E_1^0 are replayed, the next series of send events (E_2^0) becomes deterministic because of $\{E_1^1, E_1^0\} \rightarrow E_2^0$.

Definition 8 (CDC observed receive-event set: B)**.** Let B be a set of observed receive events. In Figure 11, when the main thread enqueues a receive event (e), e is included in B, i.e., $e \in B$.

Axiom 1 (Condition for correct replay of e)**.** "CDC can correctly replay e" $\Leftrightarrow$ "$\{^\forall f \in E \mid f_m(f) < f_m(e)\}$ s.t. (i) clocks of f, e is replayed, (ii) $f \in B$ and (iii) $f_c(e) < LMC$". LMC is the local minimum clock. (Qualitative explanation is in Section 3.6).

Proposition 1. Receive events are replayable if the all previous message events are replayed, i.e., "$^{\forall}\boldsymbol{E} \rightarrow E$ s.t. $\boldsymbol{E}$ is replayed, E is a receive event set" $\Rightarrow$ "E is replayable".

PROOF. Show $^{\forall}e \in E$ s.t. e is replayed if $\boldsymbol{E}$ is replayed by proving e holds conditions in Axiom 1 (i)(ii)(iii). (i) $\boldsymbol{E}$ is replayed. Therefore, clock of f, e is replayed. In Figure 9, clocks of f_1, f_2 and e are replayed because $\boldsymbol{E}$ is replayed. (ii) $^{\forall}f \in E$ s.t. $f_m(f) < f_m(e)$. Therefore, it holds $e \not\rightarrow f$ ($\because$ Definition 5, 6). Because f has no dependency to e, receive event f eventually happens, and is enqueued to B, i.e., $f \in B$. In Figure 9, f_1 and f_2 are eventually received because $e \not\rightarrow f_1, f_2$ is met. (iii) Suppose CDC cannot replay event e due to $f_c(e) \geq$ LMC, i.e., $^{\exists}\hat{g} \in E$ s.t. $f_m(e) > f_m(\hat{g})$. (iii-a) If $e \not\rightarrow \hat{g}$, receive event $\hat{g}$ is eventually received, and LMC is incremented. This LMC incrementation continues until $f_c(e) < LMC$ or $e \rightarrow \hat{g}$ is met. (iii-b) If $e \rightarrow \hat{g}$, $f_c(e) < f_c(\hat{g})$ is met. Therefore, $f_m(e) < f_m(\hat{g})$. This is contrary to $f_m(e) > f_m(\hat{g})$. According to (i)(ii)(iii), "$^{\forall}\boldsymbol{E} \rightarrow E$ s.t. $\boldsymbol{E}$ is replayed, E is a receive event set" $\Rightarrow$ "E is replayable" ■

Theorem 1. CDC can correctly replay message events, that is, $\boldsymbol{E} = \hat{\boldsymbol{E}}$ where $\boldsymbol{E}$ and $\hat{\boldsymbol{E}}$ are ordered sets of events for a record and a replay mode.

PROOF (MATHEMATICAL INDUCTION). **(i) Basis**: Show the first send events are replayable, i.e., $^{\forall}x$ s.t. "E_1^x is send events" $\Rightarrow$ "E_1^x is replayable". As defined in Definition 7.(i) E_1^x is deterministic, that is , E_1^x is always replayed. In Figure 12, E_1^1 is deterministic, that is, is always replayed. **(ii) Inductive step for send events**: Show send events are replayable if the all previous message events are replayed, i.e., "$^{\forall}\boldsymbol{E} \rightarrow E$ s.t. $\boldsymbol{E}$ is replayed, E is send event set" $\Rightarrow$ "E is replayable". As defined in Definition 7.(ii), E is deterministic, that is, E is always replayed. **(iii) Inductive step for receive events**: Show receive events are replayable if the all previous message events are replayed, i.e., "$^{\forall}\boldsymbol{E} \rightarrow E$ s.t. $\boldsymbol{E}$ is replayed, E is receive event set" $\Rightarrow$ "E is replayable". As proofed in Proposition 1, all message receives in E can be replayed by CDC. Therefore, all of the events can be replayed, i.e., $\boldsymbol{E} = \hat{\boldsymbol{E}}$. (Mathematical induction processes are graphically shown in Figure 12.) ■

Theorem 2. CDC can replay piggyback clocks.

PROOF. As proved in Theorem 1, since CDC can replay all message events, send events and clock ticking are replayed. Thus, CDC can replay piggyback clock sends. ■

6. EVALUATION

Two of our main goals are to reduce the size of the recorded data and to minimize an impact to the application performance. In this section, we show how much CDC can reduce the record size for order-reply compared to the traditional method. We also show the performance impact to the applications for recording. We conduct our evaluations on the Catalyst cluster at LLNL. The details of Catalyst are described in Table 1. We use local storage, Intel SSD 910 Series, for recording data.

6.1 Compression Efficiency

First, we evaluate the compression efficiencies of various methods. We use MCB as a representative benchmark for non-deterministic applications [1]. MCB, one of the CORAL benchmark codes, simulates the behavior of particles based on the heuristic transport equation model as described in Section 2.1. Figure 13 shows the total compressed record sizes produced by different compression methods on MCB at 3,072 processes. *gzip* is a method that applies gzip to the baseline format as shown in Figure 4. We denote a method which only applies redundancy elimination (Section 3.2) as *CDC(RE)*. *CDC(RE + PE + LPE)* is a method which applies the permutation encoding (Section 3.3) and the linear predictive encoding (Section 3.4), and *CDC* is the complete method of CDC which applies all of the presented techniques including the MF identification (Section 4.4). We use zlib [9] for the gzip method.

Table 1: Catalyst Specification

Nodes	304 batch nodes
CPU	2.4 GHz Intel Xeon E5-2695 v2 (24 cores in total)
Memory	128 GB
Interconnect	InfiniBand QDR (QLogic)
Local Storage	Intel SSD 910 Series (PCIe 2.0, MLC)

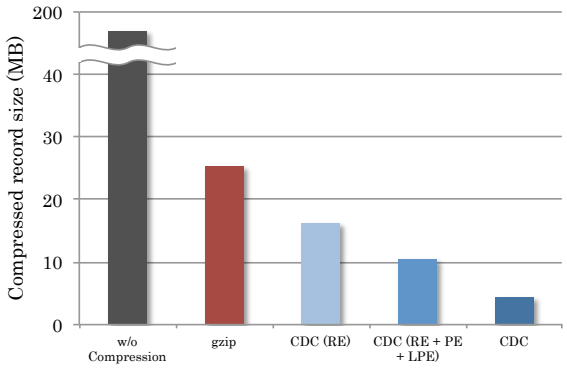

Figure 13: Total compressed record sizes on MCB at 3,072 processes (Execution time: 12.3 seconds)

As shown in the Figure 13, CDC exhibits a higher compression rate than gzip, and the compression rate of CDC is 5.7 times higher than gzip. Especially, the average number of bytes required to record a single receive event (bytes/event) is 0.51 bytes, which is approximately only 4 bits to record a single event, i.e., `count`, `flag`, `rank`, `with_next` and `clock`. In MCB, particles are exchanged as a message, and an MPI process does not send a particle x until the MPI process receives the particle x from another process, and the particle x hits the boundary. Such communication dependency constrains the variety of message receives, which effectively decreases the difference between a reference and an observed order.

For our evaluation at 3,072 MPI processes, about 9.7 million of message-receive events are observed in total. If we recorded these receives without compression as in the traditional order-replay technique, i.e., the format in Figure 4, the record size becomes significantly large. For example, if we use `count` (64 bits), `flag` (1 bit), `rank` (32 bits), `with_next` (1 bit), `clock` (64 bits) for an event (162 bits in total), the size becomes 197.0 MB. Therefore, the effec-

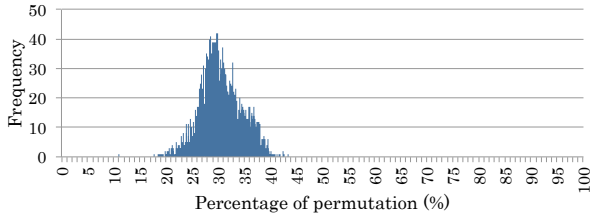

Figure 14: Percentage of permutation on MCB at 3,072 processes (Execution time: 12.3 seconds)

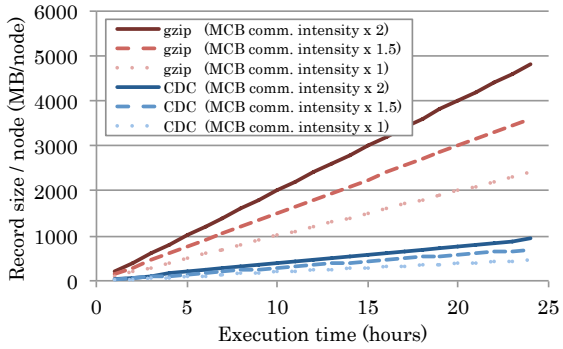

Figure 15: Per-node record-size estimates as simulation time increases (24 processes/node case)

tive compression rate of CDC is higher by almost two orders of magnitude. Since the compression rate of CDC is better than the other CDC methods, i.e., CDC(RE), CDC(RE + PE + LPE), we only evaluate CDC in the rest of the evaluations.

We also evaluate how similar the observed order is to the reference order on MCB at 3,072 processes. Figure 14 shows the similarity histogram for each and every MPI rank. To quantify the similarity, we use the percentage of permutated messages, which is computed by the number of permutated messages (N_p) divided by the total number of received messages (N), i.e., $\frac{N_p}{N}$. Thus, the percentage becomes 37.5% (= 3/8) in the example of Figure 7. As shown in the figure, the similarity is approximately 30% on average, which means 70% of the observed messages followed the reference order. The more similar, the more efficiently CDC can compress the recorded data relative to gzip as well.

Reducing the record size is important so that all of the recorded data can fit into node-local storage. We estimate the required storage size per node based on the results in Figure 13. Figure 15 shows the per-node record size estimate as simulation time increases. As shown in the figure, the record size grows as simulation time gets longer. However, because the CDC compression rate is far higher than gzip, the slopes of CDC are flatter than the ones of gzip. In this evaluation, we record on local SSDs. But, for example, if a system can provide only 500 MB of memory space (e.g., ramdisk), the gzip approach only allows the tool to record data for 5 hours for MCB. Meanwhile, with CDC, applications can run for 24 hours while keeping all of the recorded data in local memory.

MCB is just one non-deterministic application and other applications may have greater communication intensity. To

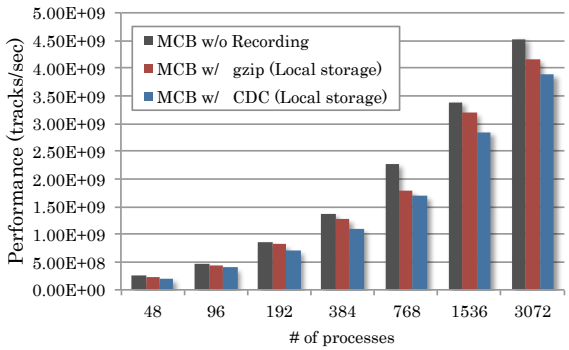

Figure 16: Recording overhead to MCB

estimate the record size of an application whose communication intensity is higher than that of MCB, we estimate the size when dealing with higher communication intensity. As shown in Figure 15, gzip requires much more storage space than CDC as communication intensity increases. Meanwhile, CDC record data requires 1 GB of local space to fit a 24-hour simulation with communication intensity two times higher than that of MCB.

6.2 Scalability of Recording

We evaluate the runtime overhead of CDC as measured by how much CDC degrades the performance of the application. Again, we run MCB with and without CDC at increasing numbers of processes. We use weak-scaling where the number of simulated particles per process is always constant: 4,000. In this evaluation, we configure CDC to write the record data to the local storage of Catalyst. Figure 16 shows the performance of MCB with and without gzip and CDC. The performance metric, tracks/sec, means how many particles are tracked per second during the simulation. As shown in this figure, MCB is still scalable with CDC, and its performance only marginally decreases with CDC: 13.1% to 25.5%. Because CDC asynchronously encodes and writes the record data to the storage without blocking application threads, the overhead is reduced accordingly.

However, we find that CDC incurs higher runtime overhead than gzip. Because CDC trades off storage efficiency with increased computation for compression, this observation is expected. Compared to gzip, the performance difference is 4.6%-13.9%. The overhead is attributed to the edit distance algorithm and is constant regardless of the number of processes because CDC does not require communication across processes. However, reducing the record size is much more critical than reducing the runtime overhead in scaling order-replay. For example, if an application only provides 500 MB memory for record-and-replay, the memory space is filled up after 5 hours in gzip as shown in Figure 15, and the application is forced to flush it out to the next level in the storage hierarchy such as a parallel file system, which may not be scalable. Meanwhile, with CDC, an application runs longer than 50 hours, running scalably for a much longer period of time. Thus, reducing the record size is more critical than reducing the performance overhead for scalability unless the overhead is unacceptably huge.

In CDC, the *observe queue*, which is used for transferring events from the main thread to the CDC thread, is

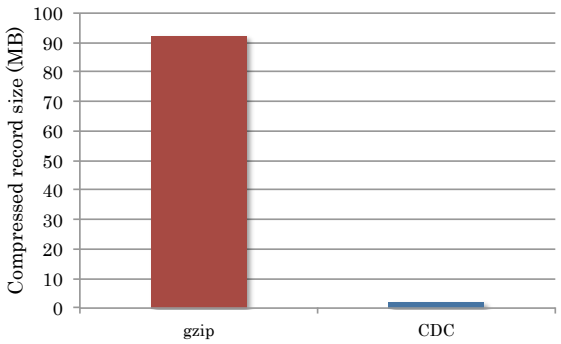

Figure 17: Compression size in hidden deterministic communications with 6,114 processes (1K iterations)

bounded and will block the main thread when the queue is filled up. In practice, CDC is not expected to block the MPI thread because the evaluation shows the recording speed (dequeuing speed), 331K [events/sec/process], is much faster than the event-production speed (enqueuing speed), 258 [events/sec/process], due to the high compression rate of CDC. Another expected source of overhead is clock piggybacking. However, the overhead is negligibly small because of the following reasons. First, the piggybacking clock value takes up only 8 bytes, a size much smaller than a typical message payload. Second, because of asynchronous algorithms used in non-deterministic MPI applications, communications are often overlapped with computations. Our evaluation shows that 8-byte piggybacking increases the execution time of MCB only by 1.18%.

6.3 Hidden Deterministic Communication

An application can use `MPI_ANY_SOURCE` in MF calls even though the actual message-receive order is deterministic (*hidden determinism*). No existing record-and-replay technique can detect whether the MF calls are deterministic or non-deterministic without having observed the runtime behavior. We need to record all of the message-receive orders that have the potential to be non-deterministic, unless users explicitly annotate these calls to be excluded from being recorded. In the case of such hidden determinism, record-and-replay tools can consume storage with unnecessary recording. Although CDC also cannot detect the hidden determinism, CDC can become more powerful in coping with this problem because LP encoding can predict numerical sequences more accurately when the communication has regular patterns, which deterministic communications usually exhibit.

To evaluate the compression rate for hidden-deterministic applications, we record the message receive order of an application [11], which solves the Poisson's equation solution using the Jacobi iteration method. Figure 17 shows the compression sizes on this application. As shown in the figure, the record size of gzip is 91MB while that of CDC is only 2MB (2.2%). Because the current CDC format is designed to record all MPI MF calls, CDC always records the `with_next` table. As shown in the result, CDC exhibits significantly high compression rate for deterministic communications. Therefore, even if users have no idea whether a communication pattern of a running application is non-deterministic, deterministic or both, CDC can efficiently record them as if deterministic communications are automatically excluded for recording.

7. RELATED WORK

Debugging non-deterministic parallel applications is an arduous task. Tools, which ease debugging non-deterministic parallel applications, are becoming more important as applications are becoming increasingly complex and non-deterministic. Deterministic replay is one of the approaches that can ease the debugging of non-deterministic applications and can be further divided into three categories: data-replay, order-replay and hybrid.

Data-replay [21, 6, 2] records message payloads in addition to the message receive order. With data-replay, programmers can choose to replay only one target process as it can use the recorded message payloads. However, data-replay approaches need to record all communication events including both deterministic and non-deterministic events, and produce vastly large amounts of recorded data in order to save the message payloads. Because the recorded data cannot fit in memory nor local storage, it is not feasible to scale this approach to an extreme scale.

Another approach is order-replay [13, 15, 14]. Although order-replay requires running all processes in a replay mode, the approach significantly reduces the record size compared to data-replay. However, even with order-replay, the record size can grow large for long running simulations, and this approach needs to flush out the recorded data to a parallel file system, which can hamper scalability. Further reduction of the recorded data like ours, therefore, significantly advances this approach. In addition, existing order-replay techniques cannot correctly replay executions in a case like the one described in Section 3.1. Xue et al. [28] proposed a hybrid approach of data- and order-replay. However, this approach suffers from the storage issue for high-end computing.

Compression is extensively studied in the context of communication tracing tools [29, 22, 27]. Noeth et al. [22] proposed a compression technique that takes advantage of similarities in communication patterns. By only recording the differences, this approach reduces the trace data. But non-determinism can hamper this approach by removing the similarities. CYPRESS [29] extracts the communication patterns by applying static and dynamic analysis to the code. In deterministic applications, each process exchanges messages as statically written in the code, and thus the static analysis can improve the compression rate. In non-deterministic applications, however, each process randomly exchanges messages, which can reduce the effectiveness of this approach. On the other hand, CDC takes advantage of the similarities in communication orders between an observed order and a logical-clock order and achieves a high compression rate even in non-deterministic applications whose communication patterns are random. Wu et al. [27] proposed lossy compression techniques on communication trace data for replay. While lossy techniques can reduce the record size, it can be largely ineffective for debugging, which generally requires *exact* replay.

To the best of our knowledge, this work is the first compression technique that can scale order-replay to extreme concurrency.

8. CONCLUSION

We have proposed a new and highly scalable approach, CDC, to record and deterministically replay MPI-based massively parallel executions. CDC creates a reference logical-clock order based on Lamport clocks and then records only the differences in message-receive orders between this reference and the observed order. Our evaluation showed that CDC can reduce the recording size by a factor of 5.7 over gzip. As dedicated threads record the traces asynchronously to the application, CDC also keeps the performance impact on the application at bay. When recording hidden deterministic communications, our results show that CDC can practically avoid any recording. With highly compact storage footprints, minimal performance overheads, and effective corner-case handling, CDC exhibits all of the characteristics needed to combat the side effects of non-determinism as a general solution.

Acknowledgment

The authors acknowledge the contributions of Christopher M. Chambreau of Lawrence Livermore National Laboratory (LLNL), and Ganesh Gopalakrishnan and Zvonimir Rakamarić of the University of Utah who reviewed our work and provided insightful feedback on it. This work was performed under the auspices of the U.S. Department of Energy by LLNL under contract DE-AC52-07NA27344 (LLNL-CONF-669878).

9. REFERENCES

[1] CORAL Benchmarks. https://codesign.llnl.gov/mcb.php.

[2] A. Bouteiller, G. Bosilca, and J. Dongarra. Retrospect: Deterministic Replay of MPI Applications for Interactive Distributed Debugging. In F. Cappello, T. Herault, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4757 of *Lecture Notes in Computer Science*, pages 297–306. Springer Berlin Heidelberg, 2007.

[3] T. A. Brunner and P. S. Brantley. An efficient, robust, domain-decomposition algorithm for particle Monte Carlo. *Journal of Computational Physics*, 228(10):3882–3890, 2009.

[4] A. Bundy and L. Wallen. Linear Predictive Coding. In *Catalogue of Artificial Intelligence Tools*, Symbolic Computation, pages 61–61. Springer Berlin Heidelberg, 1984.

[5] F. Cappello, A. Guermouche, and M. Snir. On Communication Determinism in Parallel HPC Applications. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1–8, Aug 2010.

[6] C. Clemencon, J. Fritscher, M. Meehan, and R. Ruhl. An Implementation of Race Detection and Deterministic Replay with MPI. In *EURO-PAR '95 Parallel Processing*, volume 966 of *Lecture Notes in Computer Science*, pages 155–166. Springer Berlin Heidelberg, 1995.

[7] M. Cleveland, T. Brunner, and N. Gentile. *Numerical reproducibility for implicit Monte Carlo simulations.* American Nuclear Society - ANS; La Grange Park (United States), Jul 2013.

[8] P. Deutsch. GZIP File Format Specification Version 4.3, 1996.

[9] P. Deutsch and J.-L. Gailly. ZLIB Compressed Data Format Specification Version 3.3, 1996.

[10] M. P. Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

[11] R. Himeno. Himeno Benchmark. http://accc.riken.jp/2444.htm.

[12] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010.

[13] J. C. d. Kergommeaux, M. Ronsse, and K. D. Bosschere. MPL*: Efficient Record/Play of Nondeterministic Features of Message Passing Libraries. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 141–148, London, UK, UK, 1999. Springer-Verlag.

[14] D. Kranzlmüller, C. Schaubschläger, and J. Volkert. An Integrated Record & Replay Mechanism for Nondeterministic Message Passing Programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131 of *Lecture Notes in Computer Science*, pages 192–200. Springer Berlin Heidelberg, 2001.

[15] D. Kranzlmüller and J. Volkert. NOPE: A Nondeterministic Program Evaluator. In P. Zinterhof, M. Vajteršic, and A. Uhl, editors, *Parallel Computation*, volume 1557 of *Lecture Notes in Computer Science*, pages 490–499. Springer Berlin Heidelberg, 1999.

[16] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[17] Lawrence Livermore National Laboratory. Advanced Simulation and Computing Sequoia. https://asc.llnl.gov/computing_resources/sequoia. Retrieved 16 May, 2015.

[18] Lawrence Livermore National Laboratory. CORAL/Sierra. https://asc.llnl.gov/coral-info. Retrieved 16 May, 2015.

[19] A. A. Mirin and P. H. Worley. Improving the performance scalability of the community atmosphere model. *International Journal of High Performance Computing Applications (IJHPCA)*, 26(1):17–30, 2012.

[20] S. B. NEEDLEMAN and C. D. WIJN. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. 1970.

[21] R. H. B. Netzer and B. P. Miller. Optimal Tracing and Replay for Debugging Message-passing Parallel Programs. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, pages 502–511, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[22] M. Noeth, F. Mueller, M. Schulz, and B. de Supinski. Scalable Compression and Replay of Communication Traces in Massively Parallel Environments. In *Parallel and Distributed Processing Symposium, 2007. IPDPS*

2007. IEEE International, pages 1–11, March 2007.

[23] Oak Ridge National Laboratory. Introducing Titan - Advancing the Area of Accelerated Computing. https://www.olcf.ornl.gov/titan/. Retrieved 16 May, 2015.

[24] M. Schulz, G. Bronevetsky, and B. R. Supinski. On the Performance of Transparent MPI Piggyback Messages. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 194–201, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] M. Schulz and B. de Supinski. PNMPI tools: a whole lot greater than the sum of their parts. In *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–10, Nov 2007.

[26] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. Supinski, M. Schulz, and G. Bronevetsky. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[27] X. Wu and F. Mueller. Elastic and Scalable Tracing and Accurate Replay of Non-deterministic Events. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 59–68, New York, NY, USA, 2013. ACM.

[28] 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 *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 251–260, New York, NY, USA, 2009. ACM.

[29] J. Zhai, J. Hu, X. Tang, X. Ma, and W. Chen. Cypress: Combining Static and Dynamic Analysis for Top-down Communication Trace Compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 143–153, Piscataway, NJ, USA, 2014. IEEE Press.