SReplay: Deterministic Sub-Group Replay for One-Sided Communication

Xuehai Qian
University of Southern California
xuehai.qian@usc.edu

Koushik Sen
University of California Berkeley
ksen@cs.berkeley.edu

Paul Hargrove
Costin Iancu
Lawrence Berkeley National Laboratory
{phhargrove,cciancu}@lbl.gov

ABSTRACT

Replay of parallel execution is required by HPC debuggers and resilience mechanisms. Up-to-date, there is no existing deterministic replay solution for one-sided communication. The essential problem is that the readers of updated data do not have any information on which remote threads produced the updates, the conventional happens-before based ordering tracking techniques are challenging to work at scale. This paper presents SReplay*, the first software tool for sub-group deterministic record and replay for one-sided communication. SReplay allows the user to specify and record the execution of a set of threads of interest (sub-group), and then deterministically replays the execution of the sub-group on a local machine without starting the remaining threads. SReplay ensures sub-group determinism using a hybrid data- and order-replay technique. SReplay maintains scalability by a combination of local logging and approximative event order tracking within sub-group. Our evaluation on deterministic and nondeterministic UPC programs shows that SReplay introduces an overhead ranging from* $1.3\times$ *to* $29\times$*, when running on 1,024 cores and tracking up to 16 threads.*

CCS Concepts

•**Software and its engineering** → **Runtime environments;**

Keywords

PGAS, One-Sided Communication, Debugging Tools

1. INTRODUCTION

The ability to reproduce a parallel execution is desirable for debugging and reliability purposes. In debugging [56], a programmer needs to travel back in time and deterministically examine the same execution, while for resilience [19, 20, 22, 25, 36, 57] this is automatically performed by the the application upon failure. To be useful, deterministic record and replay (R&R) is required (i.e. replay faithfully reproduces the original execution). For parallel programs the main challenge of R&R is inferring and recording the order of conflicting operations (data races). This problem has been investigated intensively in the context of shared memory [14, 18, 33, 47] and distributed memory programs [65]. Our main interest is to enable R&R for programming models based on one-sided communication [6, 10] that are increasingly used in large-scale scientific applications.

Shared memory R&R techniques either monitor thread scheduling [14, 18, 33] by tracking synchronization APIs, or log [47] the memory accessed within each thread. In distributed memory, R&R techniques for MPI [65] have been developed with emphasis on scalability. They track two-sided `MPI_Send/MPI_Recv` operations and ignore local memory accesses. Unfortunately, none of existing solutions are sufficient to enable deterministic R&R for distributed shared memory with *one-sided communication*. This mode is the base for Partitioned Global Address Space (PGAS) languages such as UPC [6], Co-Array Fortran [21, 37], Chapel [4], X10 [9, 20, 59], OpenSHMEM [42, 62] and an important feature of the new MPI-3 RMA [10, 26, 61].

Existing deterministic R&R tools for shared memory [44, 47] supports two "end points" in the design space. On one end, a R&R tool could log the inputs (values) to loads to *one* thread, with these values injected into replay execution at the right points, this thread could be replayed in an isolated manner. It is called *data-replay* [34, 47]. On the other end, a R&R tool could detect and record the order of events from *all* threads, a deterministic replay could be achieved by scheduling events in the same order. It is called *order-replay*. However, the two designs could *not simultaneously* achieve the *usability* and *scalability*. Data-replay incurs only local instrumentation overhead but provide little insights on communications between threads. Order-replay incurs high overhead (increases with system size) to track event orders in large-scale distributed memory. Therefore, the question is **how to design a useful and scalable R&R tool for one-sided communication in distributed memory**?

This paper attempts to answer this question by proposing the first scalable partial R&R tool, ***SReplay***, combining the best of data- and order-replay. SReplay is a *hybrid* design in that it performs *coordinated* deterministic replay of a sub-group (i.e. a set of threads of interest) (instead of an individual isolated thread) and *reconstructs event orders* based on information logged in record phase. Similar to data-replay, each thread in the sub-group generates value logs for loads, in addition, we also track event orders among threads in the sub-group. The value logs are not only used to ensure isolated thread replay, but also used to *infer* communications based on value matching assuming the logged event order. Threads not in sub-group are *not* executed in replay.

Practically, SReplay makes it possible to debug a large-scale execution on a smaller (or even local) machine, relieving users from monitoring a large number of concurrent events from thousands of threads. At the same time, it provides the insights on communications between threads in the sub-group for debugging purpose. The scalability is ensured by several simplifications so that SReplay it could be used in large executions involving thousands of threads. Moreover, partial replay is intrinsic to the scalability of resilience techniques [19, 22, 36] using uncoordinated or quasi-synchronous checkpointing and recovery. In this paper, we focus on the usage of R&R in debugging.

The ideas in SReplay can be applied to any programming mod-

els based on one-sided communication with memory access instrumentation. We built a prototype based on Unified Parallel C [1] programming language. UPC is a typical PGAS language which is defined with a relaxed memory consistency model that allow memory access reordering for high performance. Nondeterministic execution is common in UPC applications with fine-grained parallelism. In UPC, global shared memory could be accessed with either local load/store instructions or one-sided remote operations (e.g. `Put`/`Get`). We modified the compiler and runtime system to instrument all memory accesses to shared memory.

The evaluation is conducted on Edison, a Cray XC30 supercomputer at NERSC. We evaluate SReplay using eight NAS Parallel Benchmarks [5] (BT, CG, EP, FT, IS, LU, MG, SP), two applications using work stealing from the UPC Task Library [38] (fib, nqueens), three applications in the UPC test suite (guppie, laplace, mcop) and Unbalanced Tree Search (UTS) [45]. In addition we evaluate a large-scale production application performing Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly (Meraculous) [23]. We measure the record overhead and confirmed that the sub-group replay could produce correct results. The sub-group replay requires similar instrumentation as in record phase and we found that its overhead is very similar to record overhead. Most applications are first executed on about 40 nodes (1,024 cores or threads) of Edison and we replay the threads in sub-group on a single node monitor and replay threads that can be contained on single node (up to 24 cores or threads per node). We see that SReplay incurs an overhead from 1.3x $\sim$ 29x among all applications and different sub-group sizes (2,4,8,16 threads), when running most of the original programs on 1,024 cores. Compared with start-of-the-art instrumentation-based software R&R tools (e.g. [8]), such overhead is moderate and acceptable for a software-only R&R scheme used for debugging in large-scale.

The main contributions of this paper are:

- We introduce a novel deterministic R&R scheme for one-sided communication. It allows users to deterministically replay a subgroup of threads in a full execution without executing the rest of threads. To the best of our knowledge, SReplay is the first software tool to support deterministic partial replay for one-sided communication with good usability and scalability.
- We implement a prototype with the proposed SReplay techniques in UPC and demonstrate its usage model and overhead on 15 applications.

The rest of the paper is organized as follows. Section 2 presents background of deterministic R&R and UPC. Section 3 explains the essence of one-sided communication and each step of SReplay by a concrete example. Section 4 shows the value logging and simplified vector clock algorithm in record phase. Section 5 describes the offline mechanisms to generate logs for replay phase. Section 6 describes the communication inference mechanisms and the sub-group replay algorithm. Section 7 presents several usage models of SReplay, it is followed with the implementation details in Section 8 and the evaluation in Section 9. The paper concludes in Section 10.

2. BACKGROUND

2.1 One-Sided Communication

Traditional large-scale HPC applications are based on message passing and they typically use Message Passing Interface (MPI) as communication mechanism. Several modern programming models for distributed shared memory use *one-sided communication abstractions* that offers better performance with less synchronization. This model is particularly suitable for irregular applications [24]. Several Partitioned Global Address Space (PGAS) languages, including Unified Parallel C (UPC) [6], Co-Array Fortran [21, 37], Chapel [4], X10 [9, 20, 59] and OpenSHMEM [42, 62], are based on one-sided communication. In addition, the new MPI-3 [10] introduced efficient support for one-sided communication with remote memory access (RMA) [26].

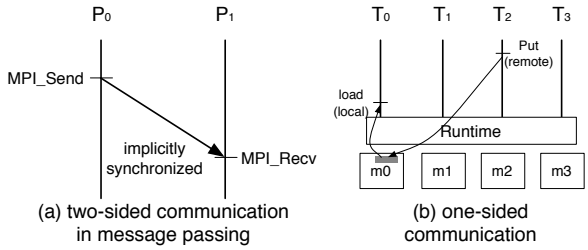

Figure 1: Two-sided and One-sided Communication.

The essence of one-sided communication is its **implicit** nature. In MPI, a typical communication involves `MPI_Send/MPI_Recv` pairs, which carries *both data transfer and synchronization semantic* and the initiating task can be determined in the receive operation (shown in Figure 1 (a)). Also, the memory location involved in the communication is visible only to one rank. These features made MPI communication easy to intercept at runtime. In one-sided communication (shown in Figure 1 (b)), a thread could write to any remote memory location in shared address space by a `Put` *without* notifying others. Later, a thread could read the new value produced by that earlier writer using a local access (since the thread is affiliated with the shared memory module), the reader is *not* aware of the thread which previously produced the value. Such implicit one-sided communication removes the implicit synchronization between sender and receiver in message passing and potentially offers better performance. However, this advantage comes at the price of nondeterminism and complex debugging because previous techniques based on dependence tracking could not apply.

2.2 Deterministic Record and Replay

Deterministic Record and Replay (R&R) consists of monitoring the execution of a multithreaded application on a parallel machine, and exactly reproducing this execution later. R&R requires recording in a log all nondeterministic events that occurred during the initial execution. They include the inputs to the execution (e.g., return values from system calls) and the order of the inter-thread communications (e.g., the interleaving of the inter-thread data dependences). During the replay phase, the logged inputs are fed into to the execution at the correct times, and the memory accesses are forced to interleave according to the log.

Deterministic replay is a powerful technique for debugging HPC applications. In principle, replay tools for HPC applications typically fall into two categories [34]. *Data-replay* tools record all incoming messages to each thread during program execution, and provide the recorded messages to threads during replay and debugging at the correct execution points. This approach allows individual threads to be replayed in isolation. In contrast, *Order-replay* tools only record the order of nondeterministic events in inter-thread communication during program execution. Since order-replay does not record actual inputs to threads, it typically generate

smaller logs than data-replay. However, detecting event orders at large-scale poses scalability challenge.

PinPlay [47, 3] is a modern Pin-based [2] R&R tool that leverages the principles of both data- and order-replay. For all programs, it could log the inputs to each thread and generates a log (i.e. pinball) during a record execution and then execute the same thread in isolation by injecting logged values to loads at right points. For multithreaded programs, it could detect and log orders of conflicting accesses to shared addresses by implementing a directory-based coherence protocol in software. However, even on a single machine with shared memory, this incurs high overhead (up to 197× slowdown for SPECOMP 2001) [8].

Because event orders could be easily tracked in message passing, existing research has been mainly focusing on MPI R&R debugging [15, 65]. Sub-group reproducible replay (SRR) [65] tries to find a good balance between data-replay and order-replay by considering a hybrid approach. SRR divides all MPI ranks into disjoint replay groups, based on the insight that ranks communicate mostly with few other ranks in the same group. During the record phase, SRR records the contents of messages across group boundaries using data-replay but records communication orderings within a group. Each group could then be replayed independently. The idea of sub-group in SRR is similar to our ideas in SReplay. However, due to the fundamental difference between two-sided and one-sided communication, the techniques in SRR could not be applied to our context. For example, during the runtime of record phase, we could not determine whether a communication is within or between different sub-groups.

Extensive efforts were made in recent years to use hardware support to reduce overhead in tracking orders of conflicting accesses [13, 17, 27, 28, 29, 39, 43, 48, 49, 50, 51, 60, 63, 64], they are all based on cache-coherent shared memory. In distributed memory, MPReplay [58] proposes architectural supports for deterministic R&R for MPI programs. The hardware tracks nondeterministic synchronization events such as wildcard receives (e.g. MPI_ANY_SOURCE, MPI_ANY_TAG, etc.). They are MPI two-sided specific mechanisms and not applicable in our context.

Due to the fundamental feature of one-sided communication, tracking event order is inherently much more challenging than in two-sided communication. The record and replay of individual thread in Pinplay [3] could be potentially directly applied in this context. However, as we discussed in Section 1, it does not provide sufficient insight for debugging purpose. There is no previous work addressing this problem.

2.3 Unified Parallel C

Unified Parallel C (UPC) [6] is an extension to ISO C 99 that provides a Partitioned Global Address Space (PGAS) abstraction using Single Program Multiple Data (SPMD) parallelism. The memory is partitioned in a task (unit of execution in UPC) local heap and a global heap. All tasks can access memory residing in the global heap, while access to the local heap is allowed only for the owner. The global heap is logically partitioned between tasks and each task is said to have local affinity with its sub-partition. Global memory can be accessed either using pointer dereferences (load and store) or using bulk communication primitives (memget(), memput()). The language provides synchronization primitives, namely locks, barriers and split phase barriers. Most of the existing UPC implementations also provide non-blocking communication primitives, e.g. upc_memget_nb(). The language provides a memory consistency model which imposes constraints on message ordering.

We implemented a prototype of SReplay based on UPC. The underlying principles are directly applicable to other one-sided communication paradigms, most notably MPI-3 RMA.

3. OVERVIEW OF SReplay

We present a overview of SReplay based on UPC. The details of each components are discussed in the following sections. As shown in Figure 2, it involves the three steps.

Record at full concurrency. The user first specifies sub-group, a subset of threads that need to be replayed. A modified compiler is used to build a binary with recording instrumentation, tracking both `load/store` instructions, as well as communication operations (e.g. `Put/Get`). The instrumented binary is then executed at full scale on a modified UPC runtime system that records the execution. For any tasks within sub-group, we record `load` values of each thread in its value log, we also track the order of `Put/Get` operations from threads in sub-group in distributed event order logs. The event order log indicates the order of conflicting operations accessing the global memory at coarse-grain. The execution of threads outside sub-group and their communication with sub-group are not tracked.

In Figure 2, the shaded region indicates sub-group. White dots indicate read accesses that do not have value log entries; black dots indicate read accesses that generate value log entries; brey dots indicate write accesses. We will discuss how we avoid logging the value for every read in Section 4.1. The arrows indicate detected event orders, which is a superset of orders between conflicting accesses. A read could be ordered after multiple writes (such as the second read in the second thread) but it could only get value from one write ordered before. We infer the precise order between writes and reads in replay phase. Some read could get values written by threads outside sub-group, such as the second black dot in the fourth thread in sub-group. In this case, there may be no write event ordered before the read in sub-group.

Log processing. The value and order logs generated in full execution are processed to obtain the required event order in replay. Based on the distributed event order log, this pass generates a replay order log for each thread in sub-group. The event orders are translated into *wait* and *wake* operations so that threads in sub-group could collaboratively enforce the order present in the original execution. In addition, a write check log is generated for each thread so that it could try to match its own written values with remote read values in certain ranges at correct points in replay phase. We use this value-based approach to infer communications between threads in sub-group because there is no explicit matching between senders and receivers in one-sided communication.

Sub-group Replay SReplay only executes the threads in sub-group in replay phase. The effects (e.g. remote writes) of any other tasks can be reconstructed from the logs. Each thread in sub-group reproduces the same execution by injecting the values in its value log at correct points. The operations from different threads are scheduled to execute in an order according to the replay order log. In addition, after a thread performs certain writes, it needs to check whether all the local writes so far could contribute to some read value log entries of remote threads. On a value match, a communication is assumed to happen between the two threads. This process is driven by the write check log. For each read log entry of a thread in sub-group, SReplay could infer one of two possibilities: (a) the value is produced by a thread inside sub-group, if so, the specific thread is given; (b) the value is not produced by any thread inside sub-group. In Figure 2, the question marks indicate the value matching operation.

SReplay uses the principle of data-replay to ensure the correct replay of each thread in sub-group based on value logs. We use a combination of order-replay and value matching to infer the com-

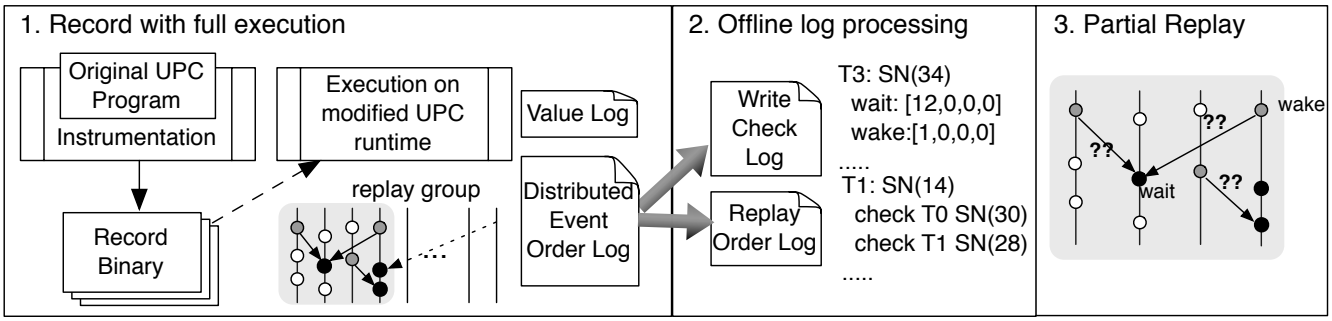

Figure 2: Overview of SReplay.

Algorithm 1: Value Logging by thread T_i in sub-group.

Data: $V(a, len)$: values of (a, len) in T_i
$V_{sm}(a, len)$: values of (a, len) in shadow memory of T_i
V_i[i] is the sequence number (SN) of T_i.
Output: $ValLog_i$: read value log of T_i.
Value log entry format: $(V_i[i], len, val)$.

```
1  switch type of an access e_i do
2      case e_i is a read of range (a,len)
3          if V(a,len) ≠ V_sm(a,len) then
4              new ValLog_i entry: (V_i[i], a, len, V(a,len))
5              V_sm(a,len) ← V(a,len)
           end
6      case e_i is a write of range (a,len)
7          V_sm(a,len) ← V(a,len)
8      V_i[i] ← V_i[i] + 1
   endsw
```

Algorithm 2: Vector Clock for Shared Memory

```
   Procedure OnMemAcc (e_i in T_i, AccRange)
       Data: V_i: vector clock of thread T_i
       V_x^w: write vector clock of address x
       V_x^a: access vector clock of address x
       All vector clocks have r entries, r is the size of sub-group.
       Output: O_i: Event orders need to obey in replay
1      V_i[i] ← V_i[i] + 1
2      switch type of e_i do
3          case e_i is a read
4              foreach x ∈ AccRange do
5                  O_i ← O_i ∪ GO (V_i, V_x^w, i)
6                  V_i ← max{V_i, V_x^w}
7                  V_x^a ← max{V_x^a, V_i}
               end
8          case e_i is a write
9              foreach x ∈ AccRange do
10                 O_i ← O_i ∪ GO (V_i, V_x^a, i)
11                 V_x^w ← V_x^a ← V_i ← max{V_x^a, V_i}
               end
       endsw

   Procedure GO
       Input  : V_my, V_m, my_pid
       Output: O_n: New event orders
12     foreach 1 ≤ i ≤ r, i ≠ my_pid do
13         if V_m[i] > V_my[i] then
14             O_n ← O_n ∪ (T_i : V_m[i] → T_my : V_my[my])
           end
       end
15     return O_n
```

munications between threads in sub-group. This idea is novel and has not been exploited in previous work. This design principle is critical for improving usability since purely relying on order-replay requires replaying all threads (not satisfying requirement of partial replay). Due to non-atomic instrumentation, it is very challenging to generate precise event orders. Our current approach does not rely on precise event order among threads in sub-group.

4. RECORDING THE EXECUTION

4.1 Value Logging

SReplay maintains a *shadow memory* in each thread in sub-group. The shadow memory indicates the current local view of shared memory of a thread. Each address in the shadow memory is associated with a sequence number (SN). The contents of a memory address are logged either at its first read or when the value read by the execution differs from value stored in the shadow memory. Similar schemes [44, 47] are described for R&R of shared memory programs.

Algorithm 1 shows the detail of the value logging mechanism in SReplay. Each thread maintains its local shadow memory, V_{sm}. On a read, $V(a, len)$ is the value obtained from the current shared memory. If the value is the same as the current value in V_{sm}, no log is generated. If not, a new value log entry is generated and V_{sm} is updated, so that next time T_i will not log the same value again. On a write, $V(a, len)$ is the written value and it also updates the shadow memory. This could avoid logging the values generated by the local thread and avoid logging addresses of dynamically allocated objects (see Section 8). The SN (V_i[i]) is updated on both reads and writes. V_i[i] in an value log entry indicates that this value should be consumed by T_i in replay phase when its SN is increased to the same number.

4.2 Event Order Logging

For tasks within sub-group, we use vector clock to obtain event orders of conflicting accesses during execution. This information is used to schedule the conflicting accesses in the replay phase and infer communications. Vector clock [53] is a powerful tool to track causal relationship of events in concurrent systems. The conven-

tional vector clock algorithms assume explicit sender and receiver and they are matched when a communication happens. We present a vector clock algorithm based on the one described in [55] and propose mechanisms to generate event orders of conflicting accesses in one-sided communication. The algorithm is shown in Algorithm 2 as a function `OnMemAcc`.

Let V_i be an n-dimensional vector of natural numbers for thread T_i, $1 \leq i \leq n$. Let V_x^a and V_x^w be two additional n-dimensional vectors for each shared address, we call V_x^a and V_x^w *access vector clock* and *write vector clock*, respectively. All the vector clocks are initialized to 0 at the beginning of computation. For two n-dimensional vectors we say that $V \leq V'$ if and only if $V[j] \leq V'[j]$ for all $1 \leq j \leq n$; $max\{V, V'\}$ is defined as the vector with $max\{V, V'\}[j] = max\{V[j], V'[j]\}$ for each $1 \leq j \leq n$. $V_i[i]$ also represents the SN of the event in T_i which caused $V_i[i]$ increased to the current value. In SReplay, we only run the vector clock algorithm within sub-group, therefore $n = r$, r is the size of sub-group.

It is proved in [54] that `OnMemAcc` ensures $e_i \rightarrow e_j$ ($\rightarrow$ indicates causal relationship), if and only if $V(e_i) < V(e_j)$. Using this property, by keeping and comparing the vector clock of all memory accesses, an external observer can obtain the complete causal relationship of events.

In reality, the lack of "external observer" limits the information available in tracking complete event order. After each access e_i in T_i, two vector clocks are available to T_i, one is the updated V_i after the access (denoted as $V_i(e_i)$), the other is V_x^a (if e_i is a write) or V_x^w (if e_i is a read) from shared memory, assuming e_i accesses x. T_i can infer whether there is a causal relationship between e_i and the *most recent* access to x (and by transitivity, the accesses causally ordered before it). From the vector clock of the most recent access, V_x^a or V_x^w, T_i cannot tell the specific remote access and cannot generate orders between two specific accesses.

Figure 3 shows a concrete example. We consider three threads and two shared memory addresses (x and y). V_i (i=1,2,3) after each memory access is indicated below the memory accesses. On the right, we show the trace of $V_{\{x,y\}}^a$ and $V_{\{x,y\}}^w$ updates. Consider the second access in T_1 (i.e. r(x)), V_1(r(x)) is [2,2,1], V_x^w is [1,2,1]. T_1 can infer that the current operation r(x) is ordered after the most recent write to address x. However, from [1,2,1], it does not know which remote access previously wrote to x. The issue is similar to the case in one-sided communication in that, a read does not know the most recent writer of a memory location.

We propose a simplified mechanism to generate conservative causal relationship of events. Consider $V_i(e_{i0})$, it captures the set of all accesses from all threads that causally happened before e_{i0}. We could consider it as a global layer, denoted as $GL[e_{i0}]$. It captures the boundary of most recent previous accesses in all threads that are causally executed before e_{i0}. When T_i performs the next memory access e_{i1}, similarly, $V_i(e_{i1})$ represents a different global layer $GL[e_{i1}]$. To reproduce the event orders in an execution, it is sufficient to execute e_{i1} after the accesses in each remote thread on $GL[e_{i1}]$. These accesses are denoted as $V_i(e_{i1})[j]$, $j \neq i$. It is possible that $V_i(e_{i1})[j] = V_i(e_{i0})[j]$ for some j, it means that T_j did not perform any access after e_{i0} that is causally happened before e_{i1}. In this case, no new causal relationship needs to be generated. Therefore, condition for generating causal relationship is, $V_i(e_{i1})[j] \rightarrow e_{i1}$ if $j \neq i$ and $V_i(e_{i1})[j] \neq V_i(e_{i0})[j]$. This approach generates a set of causal relationships between individual accesses to ensure that the replay and record enforce the same orders for all events.

Figure 4 shows the insight. From the vector clocks, T_2 can identify the difference between GL_0 and GL_1. According to our rule, the second r(x) in T_2 is causally ordered after w(x) in T_0. In T_3, there is no memory access performed between the two global layers, so there is no order generated. T_4 performs a memory access w(z), but it is not conflicting with r(x) in T_2, so there is no causal relationship between the two and also no order generated. For the example in Figure 3, before r(x) in T_1 is performed, the current vector clock in the thread is [1,0,0], after the operation, the vector clock becomes [2,2,1]. According to the rule, r(x) needs to be ordered after w(x) in T_2 and w(y) in T_3. Note that w(y) in T_3 does not conflict with r(x) in T_1, but it is causally ordered before r(x) in T_1. Specifically, it is because the vector clock obtained in T_1 at r(x) (most recently updated by w(x) in T_2) include w(y) in T_3 due to T_2's r(y), — they are indeed conflicting accesses.

Because program order contributes to causal relationship, the event orders detected are conservative. It is why in Figure 3 r(x) in T_1 is causally ordered after w(y) T_3: w(y) in T_3 conflicts with r(y) in T_2, r(y) and w(x) in T_2 are ordered by program order, w(x) in T_2 conflicts with r(x) in T_1, so transitively, r(x) in T_1 is also causally ordered after w(y) in T_3.

The order generation rule is implemented by `GO` in Algorithm 2. It takes two vector clocks (V_{my} and V_m) and thread ID of the calling thread as inputs. V_{my} is the vector clock for T_i before executing the current memory access. V_m is the vector clock obtained from shared memory, it is either V_x^a (for writes) or V_x^w (for reads). This function is called before the vector clock updates in local threads and shared memory (line 6-7 and 11). An event order is in the format of $(T_i : SN_i \rightarrow T_j : SN_j)$, which enforces that an access in T_j with SN_j executed after an access in T_i with SN_i in replay.

4.3 Scalability Enhancements

The overhead of Algorithm 2 is high for following reasons.

Storage Overhead. Two vectors (V_x^a and V_x^w) are associated with each shared memory location.

Atomic vector clock updates. It implicitly requires that the updates to vector clocks happen atomically with the actual memory accesses. In a large-scale distributed memory system, to satisfy this each memory access will be associated with a lock operation when modifying the vector clock.

Update order requirement. The updates of vector clocks associated with memory addresses (V_x^w and V_x^a) (line 7 and 11) should be consistent with program order. Because updates to vector clocks are ordinary memory accesses to shared memory, UPC runtime may reorder them. Strictly enforcing the order requires using fences, which is expensive.

We relax Algorithm 2 to make it practical. To reduce storage overhead, we associate a range of addresses with a single vector clock. For UPC, we naturally partition the shared address space according to the affinity (owner) of shared address and assign one vector clock for each partition. We choose to not maintain atomicity of instrumentation and not use fences to ensure vector clock updates order.

The consequence of those relaxations is that the event orders generated could be imprecise. It does not affect the replay correctness because it is based on data-replay. It does occasionally incur mis-reported communication but this is acceptable for a best-effort debugging tool.

5. LOG PROCESSING

5.1 Replay Order Log Generation

The order log is used to reproduce the orders generated in the record phase. For each memory access e_i in T_i with SN_i, we in-

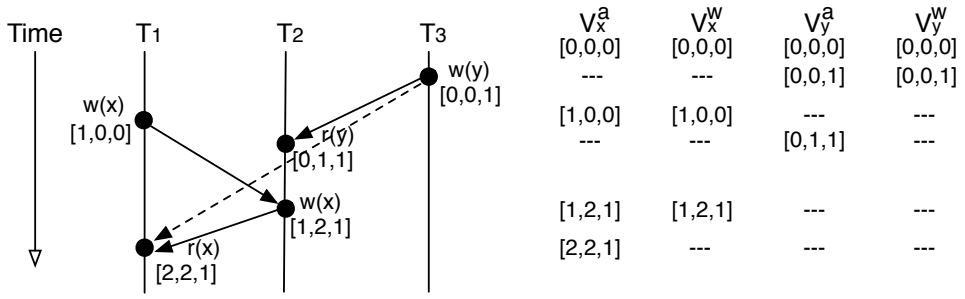

Figure 3: Running Example of Algorithm 2.

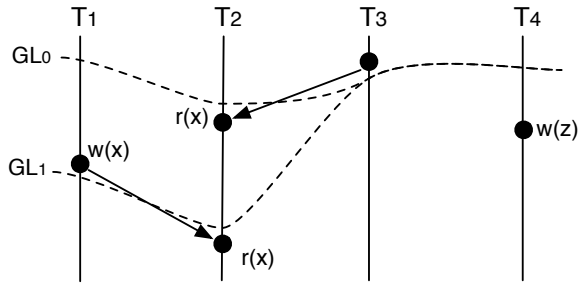

Figure 4: Tracking Event Order.

Algorithm 3: Value Check Log Generation

```
   Procedure ValCheckGen (ValLog_i, i ∈ 1, ..., r)
   |  Output: VCL_i: A map from local SN to remote SN.
   |          i ∈ 1, ..., r
1  |  foreach i ∈ 1, ..., r do
2  |  |  foreach val ∈ ValLog_i do
3  |  |  |  foreach j ∈ 1, ..., r do
4  |  |  |  |  if j ≠ i then
5  |  |  |  |  |  VCL_j[V_val[j]] ← V_val[i]
   |  |  |  |  end
   |  |  |  end
   |  |  end
   |  end
```

troduce two maps: wake_up map ($wake$) and wait_for map ($wait$). Each of them maps an SN to a vector that is of the same size as sub-group. $wake$[SN_i][j] requires that after a memory access with SN_i in T_i is executed, T_i should notify T_j with its sequence number SN_i. $wait$[SN_j][i] indicates a sequence number SN_i from T_i, that before a memory access with SN_j in T_j can be executed, it needs to wait for a notification from T_i containing SN_i. With this notion, each order ($T_i : SN_i \rightarrow T_j : SN_j$) generated in the record phase incurs the following updates to the two maps: $wake$[SN_i][j]=1, $wait$[SN_j][i]=SN_i. After processing all distributed event order logs, a map is generated for each thread in sub-group, it is then written to an order log used during replay.

5.2 Write Check Log Generation

In SReplay, communication is inferred by matching values written by a potential producer with the values in remote threads' value log. Consider the scenario in Figure 5. In record phase, there are three read accesses from T_2 that incur new values logged (e_{21},e_{22},e_{23}). The number indicates the return value of each read. When each one is performed, its vector clock represents a global layer that indicates the set of remote accesses that ordered before it. Such global layers are denoted by dashed lines. The arrows indicate the remote accesses that produced the new values logged. The goal of value matching is to infer the solid arrows in replay phase.

In replay phase, by following order log, we can order the three read accesses after the accesses before the global layers specified by their vector clocks. The value matching could be done at producer side as follows. Consider e_{21}, both T_1 and T_3 could compare their last write value to x with the value in T_2's value log. The communication is inferred when the two values match. In the example, T_3 will conclude that its write value is consumed by T_2. Therefore, the purpose of the value check log is to give the potential producer threads information about, at which point, the thread should match its written values with which remote new read values in remote threads' value log.

Algorithm 3 shows the value check log generation algorithm. The input is the value logs of all threads in sub-group. The output is a value check log (VCL_i) for each thread. VCL_i is a map from local SN to remote SN. For T_i, VCL_j[SN_i]=SN_j indicates that after T_i finished the access with SN_i, it needs to match all its locally written values up to SN_i (inclusive) with the logged values in T_j from the next value after the previous match (by T_i) to the value with SN_j. This algorithm processes all entries in the value log of all threads in sub-group, and continuously updates VCL of remote threads. To simplify notation, we assume that for each value in value log, its full vector is available. Such information could be obtained with some extra information in record phase and offline processing.

Let us consider Algorithm 3 in the scenario in Figure 5. We consider the value check log (VCL) for T_2. We see that V(e_{21})[3] and V(e_{22})[3] are the same, according to the algorithm, we will eventually have VCL_3[V(e_{22})]=V(e_{22})[2]. It ensures that after T_3 finishes $x = 1$ operation, it will try to match its previous write values with the value of both e_{21} and e_{22}. Since V(e_{23})[3] is larger than V(e_{22})[3], a new map is generated, which ensures all writes in T_3 up to the boundary specified by V(e_{23}) are matched with the new value logs in T_2 from the one after e_{22} to e_{23}. Each thread keeps the most recent locally written value to shared addresses and the value matching is always against them. For example T_1 performs two writes to z, but only the second one is matched with e_{23}. It is important to ensure that value matching needs to consider all previous writes performed by a thread, not only the accesses on a global layer or between two global layers. For example, T_4 performed a write $y = 2$ before V(e_{21}), but it is only matched with e_{22} after V(e_{22}). When a value cannot be matched by writes in sub-group, it is deemed to be produced by threads outside sub-group. It is the

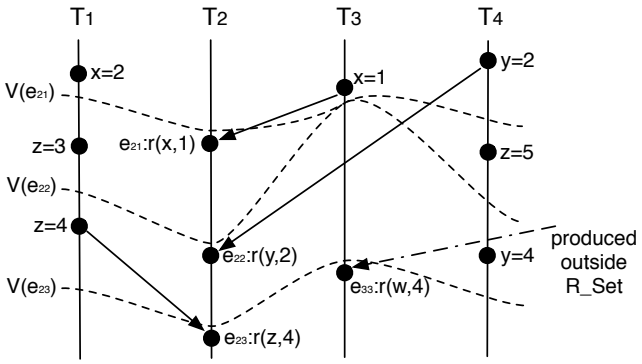

Figure 5: Inferring Communication in Replay.

case for e_{33}.

In summary, the value matching procedure could provide the producer of a new value in value log if it is produced by some thread in sub-group. Otherwise, SReplay will conclude that the values are performed outside sub-group.

6. PARTIAL REPLAY

Using the value log, order log and the value check log, SReplay can replay the threads in sub-group without executing any other threads. The partial replay algorithm is shown in Algorithm 4. In the replay phase, SReplay executes the memory accesses according to the order log. The correctness is always ensured by the value log.

The order of memory accesses in different threads is enforced by a logically shared data structure $notify$. It has $r \times r$ entries, each entry is an SN that will be set by remote threads by one-sided update. The i-th row of $notify$ is used by T_i to check whether its next access needs to wait due to event order. Physically, the i-th row is associated with the local shared memory of T_i.

If T_i needs to wait at V_i[i], then for some j, $wait[V_i[i]][j]$ is non-zero and it indicates the SN of remote access from T_j it needs to wait. Before an access can be executed, T_i needs to make sure that all $wait[V_i[i]][j]$ entries are less than or equal to $notify[i][j]$ (less is because $wait[V_i[i]][j]$ is zero if T_i's current access does not need to wait for T_j) (line 4 $\sim$ 5). If the condition is not true, then $block$ is $true$ and the thread blocks at this point. Similarly, after an access from T_i is executed, if $wake[V_i[i][j]]$ is set, T_i will update i-th entry in T_j's row in $notify$ using one-sided communication (line 20 $\sim$ 21).

For a read access, if there is a value log entry for it, then the value from value log is used (line 8 $\sim$ 9). The value is written to shared memory (line 10). Such value may or may not be the same as the current values in shared memory. If the value is produced by a thread not in sub-group, then shared memory does not contain it because that thread does not execute in replay. In this case, value log is used to construct the partial states in shared memory.

Each thread still maintains a shadow memory for values read from value log (line 11). The purpose is to tolerate the incorrect event orders generated in record phase. When there is no value log entry for a read access, the thread accesses corresponding values in both shared memory and read shadow memory (R_{sm}) (line 12). If they disagree, then the value in read shadow memory is used (line 13 $\sim$ 14). The reason is that in record phase, there could be a conflicting remote write happened after the read, and changes the value in shared memory. It is due to the occasionally mis-reported event orders in record phase. With this support, the replay correctness is not affected because the read will get the (correct) old value from shadow memory.

Finally, for write accesses, each thread updates a write shadow memory (W_{sm}) (line 16). It keeps the most recent local write values produced by the local thread and is used in communication inference. After a write access, value check is performed when its next VCL indicates that there is a need to check the current local writes with a set of remote read value log entries (line 17 $\sim$ 19). In `CheckComm` function, the relevant values in W_{sm} are checked against value entries in remote threads' value log.

Algorithm 4: Partial Replay

```
   Procedure OnMemAcc (e_i in T_i, AccRange, ValLog_i)
       Data: V_i: vector clock of thread T_i
       ShMem: actual shared memory in execution
       W_sm: shadow memory for local written values
       R_sm: shadow memory for values read from log
       SN_next_val: SN of the next new value from ValLog_i
       R_val: return value of a read
       W_val: written value of a write
       VC: a vector indicating the most recent SN of remote new
       value checked
       notify: data structure in shared memory to enforce order.
1      V_i[i] ← V_i[i] + 1
2      block ← false
3      repeat
4          foreach j ∈ 1, ..., r do
5              block ← block|(wait[V_i[i]][j] ≤ notify[i][j])
           end
       until block == false
6      switch type of e_i do
7          case e_i is a read
8              if V_i[i] == SN_next_val then
9                  Fill value from ValLog_i[V_i[i]]
10                 ShMem[AccRange] ← ValLog_i[V_i[i]]
11                 R_sm[AccRange] ← ValLog_i[V_i[i]]
               else
12                 if
                   ShMem[AccRange] == R_sm[AccRange]
                   then
13                     R_val ← ShMem[AccRange]
                   else
14                     R_val ← R_sm[AccRange]
                   end
               end
15         case e_i is a write
16             W_sm[AccRange] ← (W_val, V_i[i])
               foreach j ∈ 1, ..., r do
17                 if VCL_j[V_i[i]] ≠ 0 then
18                     CheckComm
                       (W_sm[AccRange], VC[j], VCL_j[V_i[i]])
19                     VC[j] ← V_i[i]
                   end
               end
           endsw
           foreach j ∈ 1, ..., r do
20             if wake[V_i[i]][j] ≠ 0 then
21                 notify[j][i] ← V_i[i]
               end
           end
```

```
1  int ss_steal(StealStack *s, int victim, int k) {
2    long stealIndex;
3    long stealAmt;
4
5    stealIndex = WAITING_FOR_WORK;
6    while (stealIndex == WAITING_FOR_WORK) {
7      stealIndex = s->stolen_work_addr;
8    }
9    if (stealIndex>=0) {
10     upc_fence;
11     // ensure correctness in relaxed memory consistency
12     stealAmt=s->stolen_work_amt;
13     SMEMCPY(&((s->stack)[s->top]),
14               &(stealStack[victim]->stack_g)[stealIndex],
15               stealAmt * sizeof(Node));
16     s->nSteal += stealAmt;
17   }
18 }
19 void checkSteal(StealStack *ss){
20   long d, position;
21   int stealAmt;
22   int requestor;
23
24   if (doSteal) {
25     int d = ss_localDepth(ss);
26     if (d > 2 * chunkSize) {
27       //enough work to share
28       requestor = ss->req_thread;
29       if (requestor >= 0){
30         stealAmt = (d/2/chunkSize)*chunkSize;
31         //make chunk(s) available
32         position = ss->local;
33         ss->local += stealAmt;
34         ss->nRelease++;
35         //advertise correct amount of work left locally
36         ss->workAvail = d - stealAmt;
37       }
38       ss->req_thread = REQ_AVAILABLE;
39       stealStack[requestor]->stolen_work_amt = stealAmt;
40       upc_fence;
41       // ensure correctness in relaxed memory consistency
42       stealStack[requestor]->stolen_work_addr = position;
43       return;
44     }
45   }
46 }
```

Listing 1: Communication in UTS Algorithm

7. CASE STUDIES

7.1 Unbalanced Tree Search

We discuss usage cases of SReplay based on a real-world application. We consider Unbalanced Tree Search (UTS) benchmark [45], a typical application that could leverage the advantages of one-sided communication. UTS exploits a synthetic tree-structured search space that is highly imbalanced. An efficient parallel implementation of the search relies on asynchronous work-stealing to keep processors engaged in the search.

Listing 1 shows two functions related to work stealing. The first function, `checkSteal`, is called by a thread which will potentially share certain amount of its own work to another thread. The thread first checks whether it has enough work to share (line 25). If so, it updates local stack information (line 29 $\sim$ 35). Finally, it publicizes the work using one-sided communication and writes directly (`Put`) to the work stack of the remote thread which requested the work (line 38 $\sim$ 40). The first write (line 38) indicates the stolen work amount. The second write (line 40) indicates the stolen work address.

`ss_steal` is called by a thread that has posted the stealing request and is waiting for stolen work that will be granted by a remote thread. The `stealIndex` is initially `WAITING_FOR_WORK`, indicating that it is waiting, then the thread busy waits on a while-loop, until the local variable `stealIndex` is updated by a remote thread using one-sided communication. After this, the local thread will observe the update by a local read (line 7) and then leaves the loop. If some work is successfully stolen, the local thread will then read the second write performed by remote thread, `stolen_work_amt`, to find out the amount of stolen work. Finally, it completes the work stealing by copying data from the stack of remote thread to its local stack.

In this example, a thread that receives the stolen data could only implicitly find the thread which provided stolen work by the owner of address (`s->stolen_work_addr`), but there is no explicit send and receive operation posted for this communication. In different executions, a thread may receive the stolen work from different remote threads at different execution points. Debugging such application is difficult due to nondeterministic behavior, especially in a large-scale system. Next, we show how SReplay could help in debugging.

7.2 Tracking Nondeterminism

During a period of execution, assumes that T_0 steals from T_2 and T_3 consecutively and sub-group is $\{T_0, T_2\}$. In the record phase, in both steals, SReplay logs the values of `s->stolen_work_addr` and `s->stolen_work_amt` written by T_2 and T_3 in sequence. This captures the read order of remote written values. In replay phase, these values will be fed into T_0 at the same execution points. This ensures that T_0 can be replayed repeatedly and correctly in isolation. Based on the logs generated by the offline processing step, SReplay ensures that the write operations in T_2 are executed before the read operations in T_0 that caused the exit of the while-loop. After writes in T_2 are performed, T_2 will check whether its writes performed so far could match the next value in read log in T_0. In assumed scenario, T_0 first steals work from T_2, so there there will be matches for the first pair of values of `s->stolen_work_addr` and `s->stolen_work_amt`. Based on this fact, SReplay infers that the communication happened from T_2 to T_0 at T_0's first local reads. As a result, a user could infer T_0 steals from T_2. Suppose there is some data access error on these data, a user could track the source in T_2 and we indeed log sufficient information in T_2 as well.

In this case, T_2 happens to be in sub-group, so that SReplay could provide the insights. When T_0 steals from T_3, SReplay could only replay T_0's execution but cannot gives information on which thread performed the writes, except that it is not from any thread in sub-group.

7.3 Memory Consistency Analysis

Memory consistency models [11] specify the order in which memory accesses performed by one processor become visible to other processors. It is a central concept in shared memory parallel architecture [11] and programming models based on shared memory (i.e. UPC [32, 66]). Sequential Consistency (SC) is a strong memory model mandates that the global memory order is an interleaving of memory accesses of each thread with each thread's memory accesses appearing in program order. Sequential consistency violations (SCVs) happen when non-SC behaviors are allowed by architecture or runtime system due to the lack of synchronizations (e.g. fences). It is critical to monitor and detect SCVs as they likely indicate concurrency bugs. Recent works [31, 41, 52] show the techniques to dynamically detect SCVs. Unfortunately, these proposals rely on the ability to detect conflicting accesses (i.e. data races) based on cache coherence, which does not exist in implicit one-sided communication.

In UTS, the `upc_fence` operations in line 10 and line 39 are in-

serted for this purpose. If we delete those fences, we indeed found incorrect behavior in a machine with PowerPC processors, which supports a more relaxed memory consistency model than Intel processors and allows the reordering of write operations.

It was shown that ensuring the correctness in relaxed memory consistency with synchronization operations is challenging [16, 41, 31]. It is important to provide programmers of distributed memory with with one-sided communication with the ability to analyze these bugs. SReplay readily made it possible to analyze memory consistency based on values returned by load operations. For each thread in sub-group, SReplay provides the returned value for each load. The values in record phase could be affected by the semantics in relaxed memory model: a value returned and logged may not be possible in an SC execution. In replay phase, with much fewer number of threads, we could easily ensure that memory operations from different threads in sub-group are executed according to SC. The value returned in replay with an SC execution may be different from the value logged in record, and a user could consider this as a potential SCV. It is possible that the value is produced by some threads not in sub-group, therefore, such analysis is not precise. Nevertheless, it provides the users with good hints to pinpoint the root cause.

7.4 Sub-Group Selection

So far, we assumed that sub-group is pre-determined. A natural question is how to decide sub-group in reality. We outline a new debugging iterative concurrency reduction approach that could be built based on SReplay. This analysis is based on two recent SMT-based [40] techniques. CLAP [30] is a technique for reproducing concurrency bugs, via symbolic constraints. It generates a full, buggy, multithreaded schedule via thread path profiling and symbolic constraint solving. Symbiosis [35] is based on CLAP but further localize the cause of the bug. These two techniques only require collecting *local* execution information, which is particularly suitable for large scale system. However, these approach could not scale to large systems yet due to the large number of constrains for the solver.

With SReplay, we could provide SMT solvers with only constrain formulations for threads in sub-group. If a solution is found, it means that a schedule of threads within sub-group will reproduce the bug. In another word, we could conclude that such a bug is only caused by threads in sub-group. In this case, the users could inspect the schedule and replay the buggy schedule repeatedly. If a solution is not found, it means that some read values are not produced by any threads in sub-group, an SMT solver, like Z3 [40], will produce an unsatisfiable (UNSAT) core which is a subset of constraint clauses that conflict, leaving the formula unsatisfiable. UNSAT core could localize the read values that are produced by threads outside sub-group. Then, we could search the external threads that ever wrote the required value to these read addresses by a new execution, and add all those threads in a new sub-group. Then we could let SMT solver try to find a solution again. Such iterative search continues until the SMT solver finds a solution to the constraints of the extended sub-group.

Debugging tools such as data race detectors [46] or stack inspectors [12] could help identify the initial sub-group. We leave the systematic exploration of this topic as future work.

8. IMPLEMENTATION

The instrumentation of memory accesses is conducted in both UPC runtime and UPC compiler. For each local memory accesses that are casted from shared pointers, we add "before" and "after" instrumentation by compiler. For `Put/Get` operations, we modify the UPC runtime to intercept them. Both instrumentations increase the SN of the thread.

Set	Apps	Description
NAS	BT	class=D, NP=1024
	CG	class=D, NP=256
	EP	class=D, NP=1024
	FT	class=D,NP=512,-shared-heap=512
	IS	class=C, NP=256
	LU	class=D, NP=1024
	MG	class=D, NP=1024
	SP	class=D, NP=1024
Tests	guppie	NP=1024
	laplace	NP=1024
	mcop	NP=1024, problem size: 4000
Task	fib	NP=1024, fib(60)
	nqueens	NP=1024, 8×8
	uts-upc	NP=1024, $T3XXL
	meraculous	NP=480, human genomes

Table 1: Applications Parameters. NP denotes the number of cores used for the record execution.

Shadow memory is implemented as a hash map. Each entry maps a key to a block of consecutive bytes. The size of the block is configurable, we choose 64-byte block. Depending on the size of accessed address range, multiple blocks may be accessed for value comparison.

Some applications have dynamically allocated objects in shared memory. Their addresses could be different in record and replay phase. We cannot log any shared address of those objects as values to avoid bad pointer. Consider the following code:

```
shared int *p=upc_alloc(..);
*p=5;
```

will be translated to:

```
tmp1=upc_alloc();    (1)
p_addr=tmp1          (2)
*p_addr=5            (3)
```

At (2), the value at address `tmp1` (denoted as `@tmp1`) is logged for "p_addr" (because `@tmp1` in shadow memory is uninitialized). In replay phase, the value in the log (which is an object address) will be assigned to `p_addr`. Then, 5 will be written to an bad address that has never been allocated in replay phase.

We solve this problem by updating shadow memory for thread local stores. When later a thread reads some addresses written by itself, no value log is generated because the values from shared memory and shadow memory is unchanged. In our example, after (1), in shadow memory, `@tmp1` holds the value returned by `upc_alloc()`. At (2), we find the value `@tmp1` *unchanged*, as if the thread previous already observed it. No value for `p_addr` is logged. So replay phase will correctly use the address of actually allocated object. Essentially we write the dynamically allocated addresses into shadow memory, so it will not be logged later.

Finally, we also instrument the shared memory allocation function and always set the content of newly allocated object to zero. This is to avoid the occasional missed log values because of the same values in shadow memory.

9. EVALUATION

We use fifteen UPC benchmarks to evaluate SReplay. Eight NAS Parallel Benchmarks [5] (BT, CG, EP, FT, IS, LU, MG, SP) and three applications in the UPC test suite (guppie, laplace, mcop) are

App	Native Exec.	sub-group=2	sub-group=4	sub-group=8	sub-group=16	Shadow Memory	Log Size
BT	363s	8.38x	8.48x	8.35x	8.41x	9.73 MB	1.6 GB
CG	508s	5.79x	5.84x	5.93x	6.16x	7.51 MB	16.9 GB
EP	4s	5.79x	3.98x	3.97x	4.03x	0.13 MB	0.12 MB
FT	35s	27.5x	28.1x	28.5x	29.4x	703.12 MB	15 GB
IS	26s	1.39x	1.44x	1.51x	1.57x	13.08 MB	13 MB
LU	56s	13.03x	13.89x	14.32x	15.04x	1.75 MB	770 MB
MG	176s	11.20x	11.38x	11.64x	12.18x	58.20 MB	759 MB
SP	1229s	1.82x	1.83x	1.83x	1.82x	9.65 MB	2.8 GB
guppie	160s	4.49x	4.67x	4.74x	4.89x	64 MB	519 MB
laplace	154s	8.55x	12.84x	14.76x	13.14x	0.52 MB	0.15 MB
mcop	247s	0.24x	0.52x	0.31x	0.29x	86.05 MB	121 MB
fib	13s	0.98x	0.99x	0.98x	1.14x	0.26 MB	1.31 MB
nqueens	123s	12.2x	12.8x	12.9x	13.4x	0.28 MB	85 MB
uts-upc	5s	25.4x	25.3x	26.0x	26.4x	40 MB	204 MB
Meraculous	216s	5.18x	5.44x	5.17x	5.79x	5.3 GB	2.1 GB

Table 2: SReplay Overhead

deterministic. The rest are nondeterministic by design: two applications in the UPC Task Library [7, 38] (fib, nqueens), Unbalance Tree Search (UTS) [45] and Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly (Meraculous) [23]. Table 1 shows the parameters and data sets used in experiments.

De novo whole genome assembly reconstructs genomic sequence from short, overlapping, and potentially erroneous fragments called reads. We use optimized parallelized program of the most time-consuming phases of Meraculous, a state-of-the-art production assembler [23]. It is a novel algorithm that leverages one-sided communication capabilities of UPC to facilitate the requisite fine-grained parallelism and avoidance of data hazards. A lightweight synchronization scheme is the heart of the parallel de Bruijn graph traversal.

9.1 Experiment Setup

Experiments are conducted on Edison, a Cray XC30 supercomputer at NERSC. Edison has a peak performance of 2.57 petaflops/sec, with 5576 compute nodes, each equipped with 64 GB RAM and two 12-core 2.4GHz Intel Ivy Bridge processors for a total of 133,824 compute cores, and interconnected with the Cray Aries network using a Dragonfly topology.

We are interested four aspects: (1) replay overhead in different sub-group size; (2) log size; (3) memory consumption and (4) quantify the affects of imprecise event order detection. For each experiment, we choose four different sub-group sizes: 2,4,8 and 16. sub-group size is expected to be small for partial replay. Since each node in Edison contains 24 cores, we make sure that threads in sub-group execute on different nodes (e.g. when sub-group is 2, the threads are T_{24} and T_{48}). In total, we conduct 60 executions (4 for each application). The concurrency during the initial program run and the recording phase is given by the parameter NP in Table 1. The replay correctness is verified manually by comparing the results and outputs. We use only one node of Edison (24 cores) for the replay phase, down from the original 1,024 cores ($\sim$ 40 nodes) in most cases.

9.2 Experimental Results

Table 2 shows our results. For each application, we show the native execution time without any instrumentation, the overhead for different sub-group sizes, size of shadow memory allocated and the largest log size among all logs generated by threads in sub-group. In addition, we wrote a micro-benchmark program to quantify the inaccuracy in event order detection.

9.2.1 Record Overhead

We see that SReplay introduce overhead from 1.39× $\sim$ 27.5× for small sub-group size (2). For FT, the high overhead (27.5×) is due to the large ratio between log size and shadow memory size. For uts-upc, the high overhead (25.4×) is due to the large number of shared memory accesses. They appear in when polling (busy-waiting) on remote variables when waiting for the stolen work from remote threads (e.g. line 7 in Listing 1). The overhead for the other applications are mostly under 10x. Note that the replay phase runs faster with instrumentation for two applications (mcop and fib). It is because of the nondeterministic behavior in the algorithms. For example, mcop's data distribution depends on random numbers generated. Therefore, we observed different execution characteristic in record and replay executions. We do not expect the native execution to have the same behavior as the recorded executions. Among all sub-group sizes, SReplay introduces 29.4x overhead at most in FT with 16 replayed threads, which is significantly lower than PinPlay [8] (up to 197×).

9.2.2 Overhead vs. sub-group Size

With different replay group sizes (2,4,8,16), we see that the record overhead only increases slightly or almost the same. The reason is two-fold. First, the main overhead is introduced by instrumentation of read and write accesses. They are local overhead and do not increase when the number of threads in replay group increases. Second, the overhead due to vector clock does increase when replay group size increases. But SReplay size is not expected to be large.

9.2.3 Log and Shadow Memory Size

For each application, we show the size of shadow memory allocated. It includes both read and write shadow memory. We see that different applications show drastically different characteristics. We found that shadow memory size increases after the executions start and then become stable after certain points. The largest shadow memory size appears in Meraculous. It is due to large input data size (150 GB). SReplay also uses a separate shadow memory to keep written values. The final column shows the largest log size generated by a thread in sub-group for each application.

Overall, we found that the runtime overhead is mainly decided by: (1) instrumentation of local load/store or remote put/get; (2)

shadow memory size. Applications typically show a large difference on the two aspects, therefore, we see variations in record overhead. In particular, uts-upc has a large overhead, it is partially due to the instrumented shared memory accesses in busy wait. For applications with large shadow memory size, we see that the overhead could be large (as for FT). Because shadow memory needs to be accessed on all instrumented reads, large shadow memory tends to have poor cache locality. This could explain why CG has lower overhead than FT, because the shadow memory size is much smaller. For Meraculous, although the size of shadow memory is much larger than FT, the log size is in fact smaller than shadow memory size. This suggests that the data in shadow memory are mostly allocated and written once. In another word, when deciding whether some values need to be logged, we mostly find that chunk of data not appear in shadow memory. Therefore, there are no byte level comparisons in those cases. Since the overhead is depending on multiple factors, we cannot draw conclusion based on a single factor, for example, for both CG and LU, the ratio between log size and shadow memory is large and shadow memory size is small. However, the overheads are different. In this case, the different overhead is due to (1), — the instrumentation.

9.2.4 Quantifying Imprecise Event Order Detection

The event order detection algorithm in SReplay is imprecise due to the simplification of vector clock algorithm. It is challenging to quantify this effect in applications evaluated. Most applications use bulk synchronous model, where conflicting accesses are saperated by barriers. In this case, all orders detected by our vector clock algorithm are deemed to be precise. Direct inspecting order logs of applications using data race in synchronization (e.g. UTS) is not feasible, because it is impossible to tell at replay time whether a mismatched value is due to mis-reported event order or due to the lack of value producer in sub-group.

We wrote a small test program to quantify the imprecision. The program is shown in on the left of Table 3. We let two threads (T_0 and T_{n-1}) read and write a shared variable (`sh_v`) concurrently for a several times (5000 in our experiment) and create numerous data dependences. `sh_v` is affiliate with a remote memory module for both threads and is initially zero. The other threads are idle. We run this program on SReplay and with sub-group=$\{T_0, T_{n-1}\}$. It will generate value and order logs for these two threads. For each entry (v_i) in value log of T_0, we check whether the remote write from T_{n-1} that produced the value v_i is ordered before the read in T_0 that gets v_i according to order log. If this is the case, the order is correctly detected, otherwise, our algorithm mis-reported an order. We conduct this experiment on different system size (4,16,64,256,512,1024) and the two threads are running on the first and last node (they are the same for system size 4 and 16). We show the percentage of correctly detected order for each system size.

We see that if two threads running on the same node, our algorithm practically does not produce any incorrect event order, while in theory, it is possible. When the threads run on different nodes, we do see a small percentage of mis-reported orders and it increases with system size. It is reasonable since the larger system produce more variances in memory access latency and the effects of non-atomic instrumentation become more significant. However, even with 1024 threads, we only have 15% of mis-reported orders. The consequence of such mis-reported orders is the potential imprecise information provided in the debugging tool, but the replay correctness is *never* affected. Moreover, this result is from the test program that artificially generates a large number of data dependences together with each other, which is unlikely to be the case for real applications. Therefore, we believe that our simplified vector clock algorithm does a good job in detecting event orders in large-scale executions.

Test Code	System Size (n)	4	16	64	256	512	1024
T_0 / n=0; / while(n<5000){ / x=sh_v; / n++; / } — T_{n-1} / n=0; / while(n<5000) { / sh_v=n; / n++; / }	Percentage of Correctly Detected Dependence Order	100%	100%	99.6%	95.6%	87.8%	85%

Table 3: Quantifying Imprecise Event Order

10. CONCLUSION

One-sided communication is widely used in Partitioned Global Address Space (PGAS) programming models and recently integrated in to MPI-3 standard. Despite performance advantages, its inherent nondeterminism makes debugging even more difficult. We present *SReplay*, a general mechanism to support R&R for one-sided communication. SReplay allows users focus on events within a sub-group of threads. The key idea is to use a hybrid data- and order-replay technique to enable local thread determinism and inferring inter-thread communication based on values at replay. We implemented a prototype of SReplay based on Berkeley UPC which scales to more than a thousand cores. To the best of our knowledge, SReplay is the first tool that supports deterministic R&R for one-sided communication. We demonstrate practicality of our approach by evaluating the tool using 15 applications. SReplay introduced overheads ranging from $1.3\times$ to $29\times$ with 1,024 threads and tracking up to 16 threads.

11. ACKNOWLEDGMENTS

We thank our anonymous reviewers for their useful feedback. Support for this work was majorly provided through the X-Stack program funded by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under collaborative agreement numbers DE-SC0008699.

12. REFERENCES

[1] *Berkeley UPC. http://upc.lbl.gov*.

[2] *Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool*.

[3] *Program Record/Replay Toolkit. https://software.intel.com/en-us/articles/program-recordreplay-toolkit*.

[4] *The Chapel Parallel Programming Language. http://chapel.cray.com/index.html*.

[5] *The NAS Parallel Benchmarks. Available at http://www.nas.nasa.gov/Software/NPB*.

[6] *UPC Home Page. http://upc-lang.org*.

[7] *UPC Task Library. http://upc.lbl.gov/task.shtml*.

[8] *Using PinPlay for Reproducible Analysis and Replay Debugging*.

[9] *X10: Performance and Productivity at Scale. http://x10-lang.org*.

[10] *MPI: A Message-Passing Interface Standard. Version 3.0*. Message Passing Interface Forum, 2012.

[11] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Western Reseach Laboratory-Compaq. Research Report 95/7*, September 1995.

[12] D. Arnold, D. Ahn, B. de Supinski, G. Lee, B. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging.

In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007.

[13] A. Basu, J. Bobba, and M. D. Hill. Karma: Scalable Deterministic Record-Replay. In *International Conference on Supercomputing*, 2011.

[14] M. D. Bond, M. Kulkarni, M. Cao, M. F. Salmi, and J. Huang. Efficient Deterministic Replay of Multithreaded Programs Based on Efficient Tracking of Cross-Thread Dependences. In *Ohio Sate CSE Tech Report OSU-CISRC-12/14-TR20*. Ohio State University, 2014.

[15] A. Bouteiller, G. Bosilca, and J. Dongarra. Retrospect: Deterministic Replay of MPI Applications for Interactive Distributed Debugging. In *EuroPVM/MPI*, pages 297–306. LNCS, 2007.

[16] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: Checking consistency of concurrent data types on relaxed memory models. In *Prog. Lang. Des. and Impl.*, Jun 2007.

[17] Y. Chen, W. Hu, T. Chen, and R. Wu. LReplay: A Pending Period based Deterministic Replay Scheme. In *International Symposium on Computer Architecture*, 2010.

[18] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '98, 1998.

[19] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[20] D. Cunningham, D. Grove, B. Herta, A. Iyengar, K. Kawachiya, H. Murata, V. A. Saraswat, M. Takeuchi, and O. Tardieu. Resilient X10: Efficient Failure-aware Programming. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, February 2014.

[21] D. Eachempati, A. Richardson, S. Jana, T. Liao, H. Calandra, and B. M. Chapman. A Coarray Fortran Implementation to Support Data-intensive Application Development. *Cluster Computing*, 17(2):569–583, 2014.

[22] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[23] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick. Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly. In *Proceedings of the 26th ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, November 2014.

[24] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick. Parallel de bruijn graph construction and traversal for de novo genome assembly. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, 2014.

[25] P. Hao, P. Shamis, M. G. Venkata, S. Pophale, A. Welch, S. W. Poole, and B. M. Chapman. Fault Tolerance for OpenSHMEM. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models (PGAS)*, Oct 2014.

[26] T. Hoefler, J. Dinan, R. Thakur, B. B. P. Balaji, W. Gropp, and K. D. Underwood. Remote Memory Access Programming in MPI-3. *IEEE Transactions on Parallel Computing*, 2(2), Sept 12015.

[27] N. Honarmand, N. Dautenhahn, J. Torrellas, S. King, G. Pokam, and C. Pereira. Cyrus: Unintrusive Application-Level Record-Replay for Replay Parallelism . In *International conference on Architectural Support for Programming Language s and Operating Systems*, 2013.

[28] N. Honarmand and J. Torrellas. RelaxReplay: Record and Replay for Relaxed-Consistency Multiprocessors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[29] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *International Symposium on Computer Architecture*, 2008.

[30] J. Huang, C. Zhang, and J. Dolby. CLAP: Recording Local Executions to Reproduce Concurrency Failures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[31] M. M. Islam and A. Muzahid. Characterizing Real World Bugs Causing Sequential Consistency Violations. In *Proceedings of Hot Topics in Parallelism (HotPar)*, June 2013.

[32] W. Kuchera and C. Wallace. The UPC Memory Model: Problems and Prospects. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Apr 2004.

[33] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *ACM SIGMETRICS'10*, 2010.

[34] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471–482, April 1987.

[35] N. Machado, L. Rodrigues, and B. Lucia. Concurrency Debugging with Differential Schedule Projections. In *Proceedings of 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, June 2015.

[36] D. Manivannan and M. Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. *IEEE Transactions on Parallel Distributed Systems*, 10(7):703–713, July 1999.

[37] J. Mellor-Crummey, L. Adhianto, G. Jin, and W. N. S. III. A New Vision for Coarray Fortran. In *The Third Conference on Partitioned Global Address Space Programming Models (PGAS)*, October 2009.

[38] S.-J. Min, C. Iancu, and K. Yelick. Hierarchical Work Stealing on Manycore Clusters. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models (PGAS)*, Oct 2011.

[39] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *International Symposium on Computer Architecture*, 2008.

[40] L. D. Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08/ETAPS'08 Proceedings of the Theory and practice of software, 14th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[41] A. Muzahid, S. Qi, and J. Torrellas. Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically. In *International Symposium on*

Microarchitecture, December 2012.

[42] N. Namashivayam, S. Ghosh, D. Khaldi, D. Eachempati, and B. M. Chapman. Native Mode-Based Optimizations of Remote Memory Accesses in OpenSHMEM for Intel Xeon Phi. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models (PGAS)*, Oct 2014.

[43] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[44] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic Logging of Operating System Effects to Guide Application-level Architecture Simulation. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, 2006.

[45] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An Unbalanced Tree Search Benchmark. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06, 2007.

[46] C.-S. Park, K. Sen, and C. Iancu. Scalable Data Race Detection for Partitioned Global Address Space Programs. *SIGPLAN Not.*

[47] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, 2010.

[48] G. Pokam, C. Pereira, K. Danne, R. Kassa, and A.-R. Adl-Tabatabai. Architecting a Chunk-based Memory Race Recorder in Modern CMPs. In *International Symposium on Microarchitecture*, 2009.

[49] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, J. Ha, and Y. Wu. CoreRacer: A Practical Memory Race Recorder for Multicore X86 TSO Processors. In *International Symposium on Microarchitecture*, 2011.

[50] X. Qian, H. Huang, B. Sahelices, and D. Qian. Rainbow: Efficient Memory Race Recording with High Replay Parallelism for Relaxed Memory Model. In *International Symposium on High Performance Computer Architecture*, 2013.

[51] X. Qian, B. Sahelices, and D. Qian. Pacifier: Record and Replay for Relaxed-Consistency Multiprocessors with Distributed Directory Protocol. In *International Symposium on Computer Architecture*, 2014.

[52] X. Qian, B. Sahelices, J. Torrellas, and D. Qian. Volition: Scalable and Precise Sequential Cons istency Violation Detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[53] R. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, March 1994.

[54] K. Sen. Scalable Automated Methods for Dynamic Program Analysis. In *Ph.D Thesis*. University of Illinois, Urbana-Champaign, 2006.

[55] K. Sen, G. Rosu, and G. Agha. Runtime safety analysis of multithreaded programs. In *ESEC/SIGSOFT FSE*, pages 337–346, 2003.

[56] J. Sloan, R. Kumar, and G. Bronevetsky. Large Scale Debugging of Parallel Tasks with AutomaDeD. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[57] J. Sloan, R. Kumar, and G. Bronevetsky. An Algorithmic Approach to Error Localization and Partial Recomputation for Low-overhead Fault Tolerance. In *The 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2013.

[58] C. Svensson, D. Kesler, R. Kumar, and G. Pokam. MPreplay: Architecture Support for Deterministic Replay of Message Passing Programs on Message Passing Many-core Processors. In *UIUC Technical Report UILU-09-2209*, Apr 2009.

[59] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. A. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri. X10 and APGAS at Petascale. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, February 2014.

[60] G. Voskuilen, F. Ahmad, and T. N. Vijaykumar. Timetraveler: Exploiting Acyclic Races for Optimizing Memory Race Recording. In *International Symposium on Computer Architecture*, 2010.

[61] P. Wang, H. Jiang, X. Liu, and J. Han. Towards Hybrid Programming in Big Data. In *Proceedings of 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, June 2015.

[62] A. Welch, S. Pophale, P. Shamis, O. R. Hernandez, S. W. Poole, and B. M. Chapman. Extending the OpenSHMEM Memory Model to Support User-Defined Spaces. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models (PGAS)*, Oct 2014.

[63] M. Xu, R. Bodik, and M. D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *International Symposium on Computer Architecture*, 2003.

[64] M. Xu, R. Bodik, and M. D. Hill. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[65] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, 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*, pages 251–260. ACM, February 2009.

[66] K. Yelick, D. Bonachea, and C. Wallace. A Proposal for a UPC Memory Consistency Model, v1.0. In *Lawrence Berkeley National Lab Tech Report LBNL-54983*, May 2004.