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 models 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, mcopy) 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 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 ∼ 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 mode 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.](image)

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 semantics 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 dependencies). During the replay phase, the logged inputs are fed into 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 SPECComp 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 com-
1. Record with full execution

```
[8x100]1. Record with full execution
[25x84]Original UPC
[33x74]Program
[35x16]Binary
by the execution differs from value stored in the shadow memory.

Figure 2: Overview of SReplay.
```

2. Offline log processing

```
T3: SN(34)
wait: [12,0,0,0]
wake: [1,0,0,0]
.....
T1: SN(14)
check T0 SN(30)
check T1 SN(28)
.....
??
??
```

3. Partial Replay

```
3. Partial Replay

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, \( V_{sm} \).

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: \( ValLog \): read value log of \( T_i \). |
| Value log entry format: \( (V_i[i], len, val) \). |

```
Algorithm 1: Value Logging by thread \( T_i \) in sub-group.

```
1. switch type of an access \( e_i \) do
2. case \( e_i \) is a read of range \( (a, len) \) do
3. if \( V(a, len) \neq V_{sm}(a, len) \) then
4. new \( ValLog \) entry: \( (V_i[i], a, len, V(a, len)) \)
5. end
6. case \( e_i \) is a write of range \( (a, len) \) do
7. \( V_{sm}(a, len) \leftarrow V(a, len) \)
8. \( V_i[i] \leftarrow V_i[i] + 1 \)
endsw
```

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 conventional 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 \( V^w \) and \( V^w_{sm} \) be two additional \( n \)-dimensional vectors for each shared address, we call \( V^w \) and \( V^w_{sm} \) 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 \( V \) and \( V' \), we define \( V \leq V' \) 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 \( e_i \)
Algorithm 2: Vector Clock for Shared Memory

Procedure OnMemAcc (ei, in Ti, AccRange)

Data: V_i: vector clock of thread Ti
V^m_a: write vector clock of address x
V^w_a: access vector clock of address x

All vector clocks have r entries, r is the size of sub-group.

Output: O_T: Event orders need to obey in replay

1. \( V_i[i] \leftarrow V_i[i] + 1 \)
2. switch type of \( e_i \) do
3. case \( e_i \) is a read
4. \( \text{foreach } x \in \text{AccRange} \) do
5. \( O_i \leftarrow O_i \cup \text{GO}(V^m_a, V^w_a, i) \)
6. \( V_i \leftarrow \max\{V_i, V^m_a\} \)
7. \( V^w_a \leftarrow \max\{V^w_a, V_i\} \)
8. end
9. case \( e_i \) is a write
10. \( \text{foreach } x \in \text{AccRange} \) do
11. \( O_i \leftarrow O_i \cup \text{GO}(V^w_a, V^m_a, i) \)
12. \( V^m_a \leftarrow V^m_a \leftarrow i \leftarrow \max\{V^m_a, V_i\} \)
13. end
14. endsw

Procedure GO

Input: \( V_{my}, V_m, my, \text{pid} \)

Output: \( O_n \): New event orders

12. \( \text{foreach } 1 \leq i \leq r, i \neq \text{pid} \) do
13. if \( V_m[i] > V_{my}[i] \) then
14. \( O_n \leftarrow O_n \cup (T_i : V_m[i] \rightarrow T_{my} : V_{my}[my]) \)
15. end
16. return \( O_n \)

\[\text{Figure 3 shows a concrete example. We consider three threads}\]
\[\text{and two shared memory addresses (x and y).} V_i (i=1,2,3) \text{after each}\]
\[\text{memory access is indicated below the memory accesses. On the left, we}\]
\[\text{see the trace of} V^m_{i[x,y]} \text{and} V^w_{i[x,y]} \text{updates. Consider}\]
\[\text{the second access in} T_1 \text{.} (\text{i.e.} r(x), V_1(r(x))) = [2,2,1], V^m_1 \text{is} [1,2,1].\]
\[\text{T}_1 \text{can infer that the current operation} r(x) \text{is ordered after the}\]
\[\text{most recent write to address} x. \text{However, from [1,2,1], it does not know}\]
\[\text{which remote access previously wrote to} x. \text{The issue is similar to}\]
\[\text{the case in one-sided communication in that, a read does not know}\]
\[\text{the most recent writer of a memory location.}\]

\[\text{We propose a simplified mechanism to generate conservative}\]
\[\text{causal relationships of events. Consider} V_i(e_0), \text{it captures the set}\]
\[\text{of all accesses from all threads that causally happened before} e_0.\]
\[\text{We could consider it as a global layer, denoted as} GL[e_0]. \text{It}\]
\[\text{captures the boundary of most recent previous accesses in all threads}\]
\[\text{that are causally executed before} e_0. \text{When} T_i \text{performs the}\]
\[\text{next memory access} e_{c+1}, \text{similarly,} V_i(e_{c+1}) \text{represents a different global}\]
\[\text{layer} GL[e_{c+1}]. \text{To reproduce the event orders in an execution, it}\]
\[\text{is sufficient to execute} e_{c+1} \text{after the accesses in each remote thread}\]
\[\text{on} GL[e_{c+1}]. \text{These accesses are denoted as} V_i(e_{c+1})[j], j \neq i. \text{It}\]
\[\text{is possible that} V_i(e_{c+1})[j] = V_i(e_0)[j] \text{for some} j, \text{it means that}\]
\[\text{T}_j \text{did not perform any access after} e_0 \text{that is causally happened}\]
\[\text{before} e_{c+1}. \text{In this case, no new causal relationship needs to be}\]
\[\text{generated. Therefore, condition for generating causal relationship is,}\]
\[\text{V_i(e_{c+1})[j] \rightarrow e_j \text{ if} j \neq i \text{and} V_i(e_{c+1})[j] \neq V_i(e_0)[j].}\]
\[\text{This approach generates a set of causal relationships between individu-}\]
\[\text{al accesses to ensure that the replay and record enforce the same}\]
\[\text{orders for all events.}\]

\[\text{Figure 4 shows the insight. From the vector clocks,} T_2 \text{can identify}\]
\[\text{the difference between} GL_0 \text{and} GL_1. \text{According to our rule, the}\]
\[\text{second} r(x) \text{in} T_2 \text{is causally ordered after} w(x) \text{in} T_0. \text{In} T_3,\]
\[\text{there is no memory access performed between the two global layers, so}\]
\[\text{there is no order generated.} \text{T}_4 \text{performs a memory access}\]
\[w(x), \text{but it is not conflicting with} r(x) \text{in} T_2, \text{so there is no causal}\]
\[\text{relationship between the two and also no order generated. For the}\]
\[\text{example in Figure 3, before} r(x) \text{in} T_1 \text{is performed, the current}\]
\[\text{vector clock in the thread is} [1,0,0]. \text{After the operation, the vector}\]
\[\text{clock becomes} [2,2,1]. \text{According to the rule,} r(x) \text{needs to be or-}\]
\[\text{dered after} w(x) \text{in} T_2 \text{and} w(y) \text{in} T_3. \text{Note that} w(y) \text{in} T_3\]
\[\text{does not conflict with} r(x) \text{in} T_1, \text{but it is causally ordered before}\]
\[r(x) \text{in} T_1. \text{Specifically, it is because the vector clock obtained in} T_1\]
\[\text{at} r(x) \text{(most recently updated by} w(x) \text{in} T_2) \text{include} w(y) \text{in} T_3\]
\[\text{due to} T_2 \text{'s} y), \text{— they are indeed conflicting accesses.}\]

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

\[\text{The order generation rule is implemented by} \text{GO in Algorithm 2.}\]
\[\text{It takes two vector clocks (} V_{my} \text{and} V_m \text{) and thread ID of the}\]
\[\text{calling thread as inputs.} V_{my} \text{is the vector clock for} T_i \text{before}\]
\[\text{executing the current memory access.} V_m \text{is the vector clock ob-}\]
\[\text{tained from shared memory, it is either} V^m_a \text{for writes} \text{or} V^w_a \text{for}\]
\[\text{reads. This function is called before the vector clock updates in local}\]
\[\text{threads and shared memory (line 6-7 and 11). An event order is}\]
\[\text{in the format of} (T_i : S_{N_i} \rightarrow T_j : S_{N_j}), \text{which enforces that an}\]
\[\text{access in} T_j \text{with} S_{N_j} \text{executed after an access in} T_i \text{with} S_{N_i} \text{in}\]

4.3 Scalability Enhancements

The overhead of Algorithm 2 is high for following reasons.

Storage Overhead. Two vectors (\( V^m_a \) and \( V^w_a \)) 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 (\( V^m_a \) and \( V^w_a \) (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
Time

GL

1

0

T

w(x)

[1,0,0]

T1

T2

T3

w(y)

[0,0,1]

[0,1,1]

[1,2,1]

[1,0,0]

[0,0,1]

[0,1,1]

[1,2,1]

[2,2,1]

---

---

---

---

Figure 3: Running Example of Algorithm 2.

Figure 4: Tracking Event Order.

Algorithm 3: Value Check Log Generation

Procedure ValCheckGen (ValLog, i ∈ 1,...,r)

Output: VCL[i]: A map from local SN to remote SN.

foreach i ∈ 1,...,r do

foreach val ∈ ValLog do

foreach j ∈ 1,...,r do

if j ≠ i then

VCL[i][Val[j]] ← Val[i]

else

VCL[i][Val[j]] ← Val[j]

end

end

end

generated could be imprecise. It does not affect the replay correctness because it is based on data-replay. It does occasionally 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 ei in Tj with SNi, we introduce two maps: wake[SNi][j] and wait[SNi][j]. Each of them maps an SN to a vector that is of the same size as sub-group. wake[SNi][j] requires that after a memory access with SNi, in Tj is executed, Tj should notify Tj with its sequence number SNj. wait[SNj][j][i] indicates a sequence number SNj from Tj, that before a memory access with SNj in Tj can be executed, it needs to wait for a notification from Tj containing SNj. With this notion, each order (Ti : SNi → Tj : SNj) generated in the record phase incurs the following updates to the two maps: wake[SNi][j][i]=1, wait[SNj][j][i]=SNj. 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 T2 that incur new values logged (e21,e22,e23). 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 e21, both T2 and T3 could compare their last write value to x with the value in T2’s value log. The communication is inferred when the two values match. In the example, T2 will conclude that its write value is consumed by T2. 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) for each thread. VCL is a map from local SN to remote SN. For Tj, VCL[j][SNi]=SNj indicates that after Tj finished the access with SNi, it needs to match all its locally written values up to SNj (inclusive) with the logged values in Tj from the next value after the previous match (by Tj) to the value with SNj. 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 T2. We see that V(e21)[3] and V(e22)[3] are the same, according to the algorithm, we will eventually have VCL[3]=V(e22)[3]=V(e22)[2]. It ensures that after T2 finishes x = 1 operation, it will try to match its previous write values with the value of both e21 and e22. Since V(e23)[3] is larger than V(e22)[3], a new map is generated, which ensures all writes in T2 up to the boundary specified by V(e23) 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 $V(e_{23})$, 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 case for $e_{23}$.

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 communication. 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 $\sim$ 24).

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.

Algorithm 4: Partial Replay

Procedure OnMemAcc ($e_i$ in $T_i$, AccRange, ValLog)

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 $ValLog$
$R_{val}$: 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 $j \in 1, ..., r$ do
5.         block ← block ∨ (wait[V_i[i][j] ≤ notify[i][j]])
6.     end
7. until block == false
8. switch type of $e_i$ do
9.     case $e_i$ is a read
10.        if $V_i[i] == SN_{next, val}$ then
11.            Fill value from $ValLog[V_i[i]]$
12.            $ShMem[AccRange] ← ValLog[V_i[i]]$
13.            $R_{sm}[AccRange] ← ValLog[V_i[i]]$
14.        else
15.            if $ShMem[AccRange] == R_{sm}[AccRange]$ then
16.                $R_{val} ← ShMem[AccRange]$
17.            else
18.                $R_{val} ← R_{sm}[AccRange]$
19.            end
20.        end
21. case $e_i$ is a write
22.        $W_{sm}[AccRange] ← (W_{val}, V_i[i])$
23.        foreach $j \in 1, ..., r$ do
24.            if $VCLj[V_i[i]] \neq 0$ then
25.                CheckComm
26.                ($W_{sm}[AccRange], VC[j], VCLj[V_i[i]])$
27.                $VC[j] ← V_i[i]$
28.            end
29.        end
30. endsw
31. foreach $j \in 1, ..., r$ do
32.    if $wake[V_i[i][j]] \neq 0$ then
33.        notify[j][i] ← $V_i[i]$
34.    end
35. end

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
Listing 1: Communication in UTS Algorithm

```c
int ss_steal(StealStack *ss, int victim, int k) {
    long stealIndex;
    long stealAmt;

    stealIndex = WAITING_FOR_WORK;
    while (stealIndex == WAITING_FOR_WORK) {
        stealIndex = s->stolen_work_addr;
    }
    if (stealIndex >= 0) {
        upc_fence;
        // ensure correctness in relaxed memory consistency
        stealAmt = s->stolen_work_amt;
        SMEMPY(&(s->stack)[s->top]),
        &(stealStack[victim]->stack_g)[stealIndex],
        stealAmt = sizeof(Node));
        s->nSteal += stealAmt;
    }
}
void checkSteal(StealStack *ss){
    long d, position;
    int stealAmt;
    int requester;

    if (doSteal) {
        int d = ss_localDepth(ss);
        if (d > 2 * chunkSize) {
            // enough work to share
            requester = ss->req_thread;
            if (requester >= 0){
                stealAmt = (d/2)*chunkSize;
                // make chunk(s) available
                position = ss->local;
                ss->local += stealAmt;
                ss->nRelease++;
                // advertise correct amount of work left locally
                ss->workAvail = d - stealAmt;
            }
            ss->req_thread = REQ_AVAILABLE;
            stealStack[requester]->stolen_work_amt = stealAmt;
            upc_fence;
            // ensure correctness in relaxed memory consistency
            stealStack[requester]->stolen_work_addr = position;
            return;
        }
    }
}
```

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 ~ 19). In CheckComm function, the relevant values in $W_{sm}$ are checked against value entries in remote threads’ value log.

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 ~ 35). Finally, it publishes the work using one-sided communication and writes directly (Put) to the work stack of the remote thread which requested the work (line 38 ~ 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, $T_2$ 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 interleave-
ing 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 inserted 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 constraint 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.

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:

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

will be translated to:

```
tmpl = upc_alloc(); (1)
p_addr = tmpl (2)
p_addr = 5 (3)
```

At (2), the value at address `tmpl` (denoted as `@tmpl`) is logged for `p_addr` (because `@tmpl` 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, `@tmpl` holds the value returned by `upc_alloc()`. At (2), we find the value `@tmpl 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 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 in four aspects: (1) replay overhead in different sub-group sizes; (2) log size; (3) memory consumption and (4) quantify the effects 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 28). 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 (∼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 × to 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 (m cop and fib). It is because of the nondeterministic behavior in the algorithms. For example, m cop’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 separated 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 (T₀ and T₀−1) 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={T₀,T₀−1}. It will generate value and order logs for these two threads. For each entry (vi) in value log of T₀, we check whether the remote write from T₀−1 that produced the value vi is ordered before the read in T₀ that gets vi 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.

<table>
<thead>
<tr>
<th>App</th>
<th>Native Exec.</th>
<th>sub-group=2</th>
<th>sub-group=4</th>
<th>sub-group=8</th>
<th>sub-group=16</th>
<th>Shadow Memory</th>
<th>Log Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>36s</td>
<td>8.38x</td>
<td>8.48x</td>
<td>8.35x</td>
<td>8.41x</td>
<td>9.73 MB</td>
<td>1.6 GB</td>
</tr>
<tr>
<td>CG</td>
<td>50s</td>
<td>5.79x</td>
<td>5.84x</td>
<td>5.93x</td>
<td>6.16x</td>
<td>7.51 MB</td>
<td>16.9 GB</td>
</tr>
<tr>
<td>EP</td>
<td>4s</td>
<td>5.79x</td>
<td>3.98x</td>
<td>3.97x</td>
<td>4.03x</td>
<td>0.13 MB</td>
<td>0.12 MB</td>
</tr>
<tr>
<td>FT</td>
<td>35s</td>
<td>27.5x</td>
<td>28.1x</td>
<td>28.5x</td>
<td>29.4x</td>
<td>703.12 MB</td>
<td>15 GB</td>
</tr>
<tr>
<td>IS</td>
<td>26s</td>
<td>1.39x</td>
<td>1.44x</td>
<td>1.51x</td>
<td>1.57x</td>
<td>13.08 MB</td>
<td>13 MB</td>
</tr>
<tr>
<td>LU</td>
<td>56s</td>
<td>13.03x</td>
<td>13.89x</td>
<td>14.32x</td>
<td>15.04x</td>
<td>1.75 MB</td>
<td>770 MB</td>
</tr>
<tr>
<td>MG</td>
<td>176s</td>
<td>11.20x</td>
<td>11.38x</td>
<td>11.64x</td>
<td>12.18x</td>
<td>58.20 MB</td>
<td>759 MB</td>
</tr>
<tr>
<td>SP</td>
<td>1229s</td>
<td>1.82x</td>
<td>1.83x</td>
<td>1.83x</td>
<td>1.82x</td>
<td>9.65 MB</td>
<td>2.8 GB</td>
</tr>
<tr>
<td>guppie</td>
<td>160s</td>
<td>4.49x</td>
<td>4.67x</td>
<td>4.74x</td>
<td>4.89x</td>
<td>64 MB</td>
<td>519 MB</td>
</tr>
<tr>
<td>laplace</td>
<td>154s</td>
<td>8.55x</td>
<td>12.84x</td>
<td>14.76x</td>
<td>13.14x</td>
<td>0.52 MB</td>
<td>0.15 MB</td>
</tr>
<tr>
<td>mcop</td>
<td>247s</td>
<td>0.24x</td>
<td>0.52x</td>
<td>0.31x</td>
<td>0.29x</td>
<td>86.05 MB</td>
<td>121 MB</td>
</tr>
<tr>
<td>fb</td>
<td>13s</td>
<td>0.98x</td>
<td>0.99x</td>
<td>0.98x</td>
<td>1.14x</td>
<td>0.26 MB</td>
<td>1.31 MB</td>
</tr>
<tr>
<td>nqueens</td>
<td>123s</td>
<td>12.2x</td>
<td>12.8x</td>
<td>12.9x</td>
<td>13.4x</td>
<td>0.28 MB</td>
<td>85 MB</td>
</tr>
<tr>
<td>uts-upc</td>
<td>5s</td>
<td>25.4x</td>
<td>25.3x</td>
<td>26.0x</td>
<td>26.4x</td>
<td>40 MB</td>
<td>204 MB</td>
</tr>
<tr>
<td>Meraculous</td>
<td>216s</td>
<td>5.18x</td>
<td>5.44x</td>
<td>5.17x</td>
<td>5.79x</td>
<td>5.3 GB</td>
<td>2.1 GB</td>
</tr>
</tbody>
</table>

Table 3: Quantifying Imprecise Event Order

Table 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× to 29× 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


