Title
SReplay: Deterministic sub-group replay for one-sided communication

ISBN
9781450343619

Authors
Qian, X
Sen, K
Hargrove, P
et al.

Publication Date
2016-06-01

DOI
10.1145/2925426.2926264

Peer reviewed

OPR: Deterministic Group Replay for One-Sided Communication

ABSTRACT

Replay of parallel execution is required by HPC debuggers and resilience mechanisms. Up-to-date, no deterministic replay for one-sided communication has been described in literature. The essential problem is that the readers of updated data do not have any information on which remote threads produced the updates, therefore ordering of operations is challenging at scale. This paper presents OPR *(**O**ne-sided communication **P**artial **R**ecord and Replay), the first known software tool for record and deterministic replay for one-sided communication. OPR allows the user to specify a set of threads of interest and then "records" their execution, it does not maintain state for any other threads. The selected threads can be replayed on a local machine without executing the remaining threads. Determinism is provided by using a combination of data- and order-replay. Scalability is provided by optimizations: values are logged on the first read or only when changed; approximate ordering is maintained using a tailored vector clock algorithm. Our evaluation on deterministic and nondeterministic UPC programs shows that OPR introduced an overhead ranging from* $1.3\times$ *to* $29\times$*, when running on 1,024 cores and tracking up to 16 threads.*

1. INTRODUCTION

The ability to reproduce a parallel execution is desirable for debugging and program reliability purposes. In debugging [34], the programmer needs to manually step back in time, while for resilience [13, 14, 16, 18, 23, 35] this is automatically performed by the the application upon failure. To be useful, replay has to faithfully reproduce the original execution. For parallel programs the main challenge is inferring and maintaining the order of conflicting operations (data races). Deterministic record and replay (R&R) techniques have been developed for multithreaded shared memory programs [10, 12, 20, 30], as well as distributed memory programs [40]. Our main interest is techniques for large scale scientific [4, 7] programming models.

Shared memory R&R techniques use either information about thread scheduling [10, 12, 20] by tracking synchronization APIs, or log [30] the memory accessed within each thread. In distributed memory, R&R techniques for MPI [40] have been developed with emphasis on scalability. They track two-sided `MPI_Send/MPI_Recv` operations and ignore local memory accesses. None of the existing approaches can provide deterministic R&R for the new class of modern distributed programming models (MPI-3 RMA [7, 19, 38]) and Global Address Space (UPC [4], Co-Array Fortran [15, 24], Chapel [2], X10 [6, 14, 37], OpenSHMEM [26, 39]) which advocate one-sided communication abstractions.

In this paper, we present the first general tool, *OPR* (**O**ne-sided communication **P**artial **R**ecord and Replay) to support deterministic R&R for one-sided communication. The tool allows users to select a small set of threads of interest from a large scale application. It tracks their execution and upon demand it can deterministically replay the selected set of threads. As all other threads are not executed during the partial replay, the tool eases debugging experience and relieves users from monitoring all concurrent events from potentially tens of thousands of threads. OPR also makes it possible to debug a large-scale execution on a smaller (local) machine. Furthermore, partial replay is intrinsic to the scalability of resilience techniques [13, 16, 23] using uncoordinated or quasi-synchronous checkpointing and recovery.

Our OPR prototype is built for the Unified Parallel C [1] programming language. This is a typical PGAS (Partitioned Global Address Space) language whose memory consistency model allows for reordering of operations and therefore nondeterministic execution. Memory can be accessed either with load/store instructions or using one-sided communication (Put/Get). The challenge is to build a hybrid scalable mechanism able to infer the order of these disjoint multiple types of operations.

State-of-the-art deterministic R&R for shared memory programming [27, 30] handles load/store operations using value logging (referred to as data-replay [21, 30]). Determinism is attained by maintaining a shadow memory and comparing its contents against the program execution. In OPR, we use a similar approach to detect thread state changes due to remote direct loads/stores in record phase and log values at certain points.

Although the data-replay based approach enables replay in isolation, it does not provide sufficient insight on how communication happened between threads. To eliminate this drawback, we employ a *hybrid R&R scheme*. The data-replay which ensures correctness is complemented with order-replay [21] to infer inter-thread communication based on value matching. In the record phase, OPR runs a simplified and scalable vector clock algorithm only among the monitored threads to get an approximation of event orders of accesses to global memory. In the replay phase, OPR enforces the same event order and infers the communication by matching values of local writes and remote reads (Gets) (in the value log of remote threads). By combining an approximate order with matching the values in the logs, we provide scalability as well as allowing for non-atomic monitoring and recording of load/store and Put/Get operations. To the best of our knowledge, OPR is the first scheme that uses this hybrid approach.

The evaluation is conducted on Edison, a Cray XC30 supercomputer at NERSC. We evaluate OPR using eight NAS Parallel Benchmarks [3] (BT, CG, EP, FT, IS, LU, MG, SP), two applications using work stealing from the UPC Task Library [25] (fib, nqueens), three applications in the UPC test suite (guppie, laplace, mcop) and Unbalanced Tree Search (UTS) [28]. In addition we evaluate a large scale production application performing Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly (Meraculous) [17]. We focus on recording overhead and ensure that the output and the orders are right. Since a small number of threads are partially replayed, the threads can be replayed efficiently without any noticeable performance degradation. Therefore, in our experimental evaluation we only check replay fidelity and we do not focus on measurement of replay overhead. All applications are first executed on about 40 nodes (1,024 cores/threads) of Edison and we monitor and replay threads that can be contained on single node (two up to 16 cores/threads). We see that OPR incurs overhead from 1.3x $\sim$ 29x among all applications and different R_Set sizes (2,4,8,16 threads), when running the original program on 1,024 cores. Such overhead is moderate and acceptable for a software-only R&R scheme used for debugging. As discussed in Section 9, we believe that using static analysis to guide the load/-store instrumentation can lower the runtime overhead to the point that our approach is feasible for resilience techniques.

The main contributions of this paper are:

- We introduce a novel partial 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, OPR is the first software tool to support deterministic partial replay for one-sided communication.

- We implement our mechanisms on UPC in a tool called OPR and demonstrate its use on 15 applications.

The rest of the paper is organized as follows. Section 2 presents background for UPC and deterministic R&R. Section 3 explains each step in OPR 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 whole partial replay algorithm. Section 7 discusses the implementation details, it is followed by the evaluation in Section 8 and a discussion in Section 9. Section 10 summarizes the other related work. The paper concludes in Section 11.

2. BACKGROUND

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 the 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 back 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 at scale. In principle, replay tools for HPC applications typically fall into two categories [21]. *Data-replay* tools record all incoming messages to each process during program execution, and provide the recorded messages to processes during replay and debugging at the correct execution points. With this approach, developers can replay just faulty processes rather than having to replay the entire parallel application. In contrast, *order-replay tools* only record the outcome of nondeterministic events in inter-process communication during program execution. Since order-replay only records the ordering of nondeterministic events, it normally generate smaller logs than data-replay. On the other hand, the vector clocks required for ordering are known to pose scalability challenges during record execution.

MPI has been the standard programming API for scientific computing for the last decades. In MPI, the typical communication is two-sided using `MPI_Send/MPI_Recv` pairs A pair carries both data transfer and synchronization semantics and the initiating task can be determined in the Recv operations. Furthermore, in two-sided communication, any memory location modified with store instructions is visible only to one rank. Thus, MPI R&R schemes need to track only communication operations and order-replay naturally works well.

Previous research has been focusing on MPI R&R debugging [11]. The state-of-the-art is captured by subgroup reproducible replay (SRR) [40] which tries to find a good balance between data-replay and order-replay by considering a hybrid approach. SRR divides all processes into disjoint replay groups, based on the insight that ranks communicate only with few other ranks in most domain decompositions. During the record phase, SRR records the contents of messages across group boundaries using data-replay but records just message orderings for communications within a group. Each group could then be replayed independently. Scalability is determined by the total volume of communication across group boundaries during the execution, as well as the group size which affects maintaining the order within the group.

One-sided communication has been shown to provide good scalability with less synchronizations, in particular for irregular applications. It is intrinsic to the PGAS languages (UPC [4], Co-Array Fortran [24], Chapel [2], X10 [6]) and it has been adopted into MPI-3 [7]. For two-sided communication in message passing (e.g. MPI), the sender and receiver of communication are bundled with the transfer and can be easily matched at runtime. Therefore, each communication could be naturally intercepted and logged at runtime. It is the requirement of the R&R schemes for MPI, including SRR [40]. Unfortunately, this is not the case for one-sided communication.

For one-sided communication, ordering communication is more challenging. In this paradigm, a task could write (by a store or an explicit Put) to any shared memory location without notifying others. Later, when another task reads the new value produced by an earlier writer, the reader is *not* aware of who produced the value. Compared with two-sided communication, one-sided communication removes the implicit synchronization between sender and receiver and can potentially offer better performance. This performance comes at the price of nondeterminism and complex debugging.

2.1 Unified Parallel C

Unified Parallel C (UPC) [4] 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.

Although implemented for the UPC language, OPR and the underlying principles are directly applicable to other one-sided communication paradigms, most notably MPI-3 RMA.

3. OVERVIEW OF OPR

3.1 An Example of One-sided Communication

The example below illustrates the challenges to provide deterministic R&R for one-sided communication. The Unbalanced Tree Search (UTS) benchmark [28] presents a synthetic tree-structured search space that is highly imbalanced. Parallel implementation of the search requires continuous dynamic load balancing to keep all processors engaged in the search. We consider an implementation using asynchronous work-stealing. In the algorithm, a depth-first search (DFS) stack is partitioned into two regions: local and shared. Steal operations are necessary to accomplish load balancing, nodes are transferred through one-sided communication. To amortize the manipulation overheads, nodes can only be moved in chunks of size k between the local and shared regions or between the shared regions of two different threads' stacks. More detailed description of the algorithms can be found in [28].

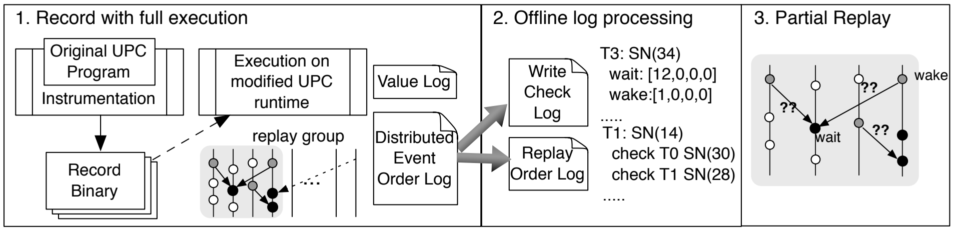

Figure 1: Overview of OPR.

```
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
10   if (stealIndex >=0) {
11     upc_fence;
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 }
20
21 void checkSteal(StealStack *ss){
22   long d, position;
23   int stealAmt;
24   int requestor;
25
26   if (doSteal) {
27     int d = ss_localDepth(ss);
28     if (d > 2 * chunkSize) {
29       //enough work to share
30       requestor = ss->req_thread;
31       if (requestor >= 0){
32         stealAmt = (d/2/chunkSize)*chunkSize;
33         //make chunk(s) available
34         position = ss->local;
35         ss->local += stealAmt;
36         ss->nRelease++;
37         //advertise correct amount of work left locally
38         ss->workAvail = d - stealAmt;
39       }
40       ss->req_thread = REQ_AVAILABLE;
41       stealStack[requestor]->stolen_work_amt = stealAmt;
42       upc_fence;
43       stealStack[requestor]->stolen_work_addr = position;
44       return;
45     }
46   }
47   ....
48 }
```

Listing 1: Communication in UTS Algorithm

Listing 1 shows two important functions related to work stealing. `checkSteal` is called by a thread which will potentially share certain amount of its own work to another thread. The thread first checks (`load`) whether it has enough work to share (line 28). If so, it updates (`store`) local stack information (line 32 $\sim$ 38). 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 40 $\sim$ 43). The first write (line 41) indicates the stolen work amount. The second write (line 43) indicates the stolen work address. These two variables are later read (`Get`) by the remote thread to complete the work stealing. The `upc_fence` between the two writes ensures that the remote thread read the updates in correct order.

`ss_steal` is called by a thread that has already posted the stealing request and is waiting for stolen work that will be granted from 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.

This example indicates a typical use case for one-sided communication. The essences are: (1) a thread could update data on remote threads directly without any of their involvement, this can happen either through `stores` or `Put` communication calls; and (2) only the initiator is aware of a communication, so there is no explicit match between sender and receiver. Specifically, 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. Deterministic R&R requires tracking both `load/store` instructions and `Put/Get` communication operations.

This example also illustrate nondeterministic behavior. In different executions, a thread may receive the stolen work from different remote threads at different execution points. Obviously, it is challenging to debug the large scale executions with nondeterminism since the developers will be overwhelmed by different thread interactions over different executions.

3.2 OPR: Deterministic Partial R&R

OPR involves the following steps (see Figure 1).

Record at full concurrency. The user first specifies the replay set, R_Set, 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 R_Set, we track `loads/stores` instructions into a value log, which contains the inputs for loads at different points. For any task within R_Set, we track `Put/Get` operations to tasks within R_Set into an distributed event order log. The event order log indicates an approximation of orders of conflicting operations accessing the global memory.

The behavior of any tasks outside R_Set, or the communication between R_Set and the outside world is not tracked.

In Figure 1, the shaded region indicates the replay group. In each thread, the white dots indicate read accesses that do not have value log entries; the black dots indicate read accesses that generate value log entries; the grey dots indicate write accesses. The arrows indicate detected event orders. We can see that some orders exist between write and read accesses, but the reads may not consume the values produced by writes, such relationship needs to be checked in replay phase. Also, some read accesses could get values produced by threads outside R_Set, such as the second black dot in the last thread in R_Set.

Log processing. The value log and order log are processed to enforce the replay order. Based on the distributed event order log, this pass generates a replay order log for each thread in R_Set. The event orders are translated into wait and wake vector clocks for the relevant operations so that threads in R_Set 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

Algorithm 1: Value Logging by thread T_i in R_Set.

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

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 R_Set because there is no explicit matching between senders and receivers in one-sided communication.

Replay only R_Set OPR only executes the threads in R_Set in the partial replay phase. The side effects of any other tasks can be reconstructed from the logs. Each thread 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 R_Set, OPR could infer one of two possibilities: (a) the value is produced by a thread inside R_Set, if so, the specific thread is given; (b) the value is not produced by any thread inside R_Set. In Figure 1, the question marks indicate the value matching operation.

Now let us consider how does OPR work for the UTS example in (Listing 1). Assume R_Set is {T_0, T_2} and in a period of execution, T_0 steals from T_2 and T_3. In the record phase, in both steals, OPR will log the values of `s->stolen_work_addr` and `s->stolen_work_amt` at the correct time. In the replay phase, these values will be fed into T_0 at the same execution points. This ensures that T_0 is replayed correctly in isolation. In addition, based on the logs generated by the offline processing step the write operations in T_2 are executed before the read operations in T_0 that caused the exit of the while-loop. Furthermore, after writes in T_2 are performed, T_2 will check whether its writes performed so far could match a read value log in T_0. In our case, since T_0 indeed steals work from T_2, there will be matches for both values of `s->stolen_work_addr` and `s->stolen_work_amt`. Based on the matched values, OPR infers that the communication happened from T_2 to T_0.

In OPR, we use the principle of data-replay to ensure the correct replay of each thread in R_Set based on value log. We use order-replay and value matching to infer the communications between threads in R_Set. This design principle is critical since purely relying on order-replay requires replaying all threads (not satisfying requirement of partial replay). More importantly, due to non-atomic instrumentation, it is very challenging to generate precise event orders. The current approach could tolerate such imprecision because replay correctness does not depend on the event order. The imprecise event order only leads to false positives or negatives in communication inference but does not affect replay correctness.

4. RECORDING THE EXECUTION

4.1 Value Logging

For value logging, OPR maintains a *shadow memory* in each

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 R_Set.
    Output      : O_i: Event orders need to obey in replay
    V_i[i] ← V_i[i] + 1
    switch type of e_i do
        case e_i is a read
            foreach x ∈ AccRange do
                O_i ← O_i ∪ GO(V_i, V_x^w, i)
                V_i ← max{V_i, V_x^w}
                V_x^a ← max{V_x^a, V_i}
            end
        case e_i is a write
            foreach x ∈ AccRange do
                O_i ← O_i ∪ GO(V_i, V_x^a, i)
                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
    foreach 1 ≤ i ≤ r, i ≠ my_pid do
        if V_m[i] > V_my[i] then
            O_n ← O_n ∪ (T_i : V_m[i] → T_my : V_my[my])
        end
    end
    return O_n
```

thread in R_Set. The shadow memory indicates the current local view of shared memory of a thread. Each address in the shadow memory has associated 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 [27, 30] are described for R&R of shared memory programs.

Algorithm 1 shows the detail of the value logging mechanism in OPR. Each thread maintains its local shadow memory, V_{sm}. It is initially empty. On each read, $V(a, len)$ is the value obtained from the current shared memory. If this 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 each 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 also avoid logging addresses of dynamically allocated objects (see Section 7 for more details). The SN (V_i[i]) is updated on both read and write accesses, this value is a part of vector clock that is used in tracking event orders.

Each value log entry includes three fields. V_i[i] indicates that this value should be consumed by T_i in replay phase when its SN is increased to the same number. We do not include the addresses in the log since they are available during replay. Another reason of not including addresses in the log is that some read addresses could be different in record and replay phase, as a thread may access dynamically allocated memory objects. It will not affect the replay correctness and will be discussed in Section 7.

4.2 Event Order Logging

For tasks within R_Set, we use a 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 [31] 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 [33] 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

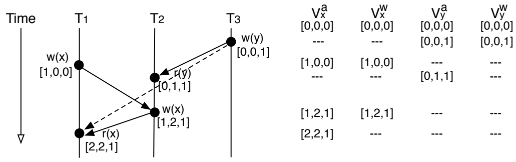

Figure 2: Running Example of Algorithm 2.

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 OPR, we only run the vector clock algorithm within R_Set, therefore $n = r$, r is the size of R_Set.

It is proved in [32] 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. However, this algorithm needs to be adapted to generate orders of conflicting accesses in our scenario.

When a thread performs a memory access to a shared address, it can only obtain the current vector clocks associated with this location but cannot observe the vector clocks of remote memory accesses. 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 $V_i(e_i)$) according to Algorithm 2, 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. Based on this information, T_i can only infer whether there is a causal relationship between e_i and the most recent access to x (and the accesses causally ordered before it). However, by 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. Unlike in [33], there is no "external observer" that keeps the vector clock of previous memory accesses in all tasks.

Figure 2 shows a running example of Algorithm 2. 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. Obviously, it is impractical to let threads keep the vector clocks of previous memory accesses and pass around such information. Therefore, the event order has to be inferred by limited information.

We propose a simplified mechanism to generate causal relationship of events conservatively. 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]. The advantage of this approach is that we can generate causal relationship between individual accesses, so that these event orders could be reproduced in replay phase.

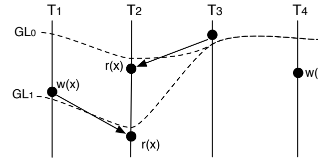

Figure 3: Event Order Detection.

Figure 3 shows the concept. 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. Now let us consider this mechanism in the example in Figure 2. 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.

The example discloses an interesting fact about causal relationship and the order between conflicting accesses: causal relationship is a *conservative approximation* of conflicting accesses. Algorithm 2 can produce causal relationship between events in different threads precisely. However, not all pairs of accesses that are causally ordered are conflicting accesses. It is because program order also contributes to causal relationship and it is exactly why in Figure 2 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. Our order generation rule will produce a *superset* of orders between conflicting accesses.

Concretely, 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). `GO` checks the exact condition that we showed (line 14). An event order in OPR is in the format of ($T_i : SN_i \rightarrow T_j : SN_j$). In replay phase, this enforces that an access in T_j with SN_j executed after an access in T_i with SN_i.

4.3 Scalability Enhancements

Algorithm 2 is able to capture all causal relationship between accesses to shared memory. However, the overhead is high for the following reasons.

Storage Overhead. Two vectors (V_x^a and V_x^w) are associated with each shared memory location. This makes the algorithm impractical to implement.

Atomic vector clock updates. It implicitly requires that the updates to vector clocks happen atomically with the actual memory accesses. On hardware without transactional memory support, to satisfy this requirement with software instrumentation, each mem-

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
    foreach i ∈ 1, ..., r do
        foreach val ∈ ValLog_i do
            foreach j ∈ 1, ..., r do
                if j ≠ i then
                    VCL_j[V_val[j]] ← V_val[i]
                end
            end
        end
    end
```

ory access will be associated with a lock operation when modifying the vector clock. This poses scalability challenges.

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. It seems to be obvious, but in reality the updates to vector clocks are ordinary memory accesses to shared memory, UPC runtime may reorder them. Strictly enforcing the order requires using fences, which also leads to extra overhead.

To make Algorithm 2 practical, we relax some of these requirements. To reduce storage overhead, we associate a range of addresses with a single vector clock. For UPC we have chose to maintain a single vector clock for all the memory that has (physical) affinity with a task. We naturally partition the shared address space according to the affinity (owner) of shared address in UPC. Essentially, this makes the accesses to addresses with same owner "conflicting", forcing a more restrictive ordering during replay. We also do not maintain atomicity of memory accesses and instrumentation, nor do we use fences to ensure vector clock updates order. To eliminate some false ordering, for a read, an order is only generated when there a new value is logged on value change.

The consequence of those relaxations is that the event orders generated could be incorrect (e.g. a read happens after a write, but according to the order generated, the write happens after the read). Note that such imprecisions do not affect the replay correctness because the right values from value logs are always injected to the threads in R_Set at right points. On the other hand, our simplified algorithm does occasionally incur mis-reported communication due to incorrect or missed event order recorded. However, 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 introduce two maps: wake_up map ($wake$) and wait_for map ($wait$). Each of them maps an SN to a vector that has size equal to R_Set. $wake$[SN$_i$][j] (the j-th element in the vector mapped from SN$_i$) requires that after a memory access with SN$_i$ in T$_i$ is executed, T$_i$ should send its sequence number SN$_i$ to T$_j$, which is supposed to wait for 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 SN$_i$, which is supposed to be sent by T_i. With this notion, each order ($T_i : SN_i \rightarrow T_j : SN_j$) generated in the record phase naturally 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 R_Set, it is then written to an order log used during replay.

5.2 Write Check Log Generation

In OPR, communication is inferred by matching values written by a potential producer with the new values logged in remote threads' value log. Consider the scenario in Figure 4. First image it is 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.

During replay, by following the orders in 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 naturally 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 R_Set. 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$_i$, if we have VCL$_j$[SN$_i$]=SN$_j$, it 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 R_Set, and continuously updates VCL of remote threads. To simplify notation, we assume that for each value in value log, its full vector is available. But as Algorithm 4.1 showed, each value only has the local SN associated with it. In the implementation, we maintain some extra information in record phase that could recover the full vector needed for value check log generation.

Let us consider Algorithm 3 in the scenario in Figure 4. 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 most recent values. 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 R_Set, it is deemed to be produced by threads outside R_Set. It is the 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 R_Set. Otherwise, OPR will conclude that the values are performed outside R_Set.

6. PARTIAL REPLAY

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

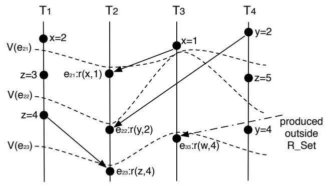

Figure 4: Inferring Communication in Replay.

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
```

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 R_Set, 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. However, this order could be incorrectly detected as the remote write happens before the read. Following this order in replay phase, when the read executes, the value in shared memory is already updated by the remote write to a new value. However, to replay correctly, the read should still get the old value. Our mechanism ensures that the read always gets correct value from read 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 so far with a set of remote read value log entries (line 17 $\sim$ 19). `CheckComm` function is straightforward: the relevant values in W_{sm} are checked against some value entries in remote threads' value log.

7. IMPLEMENTATION

The instrumentation of memory accesses is implemented 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. Shared addresses are used to generate the hash keys. Each entry maps a key to a block of consecutive bytes. The key is the start address of the byte block. The size of the block is configurable, we choose 64-byte block. On an access to the shadow memory, the key is generated based on the start address of the byte block that the access belongs to. Depending on the size of accessed address range, multiple blocks may be accessed for value comparison. The same data structure and implementation are used in both read and write shadow memory in record and replay phase.

OPR detects the value changes at instrumentation points ("before" and "after" each shared memory access). However, the instrumentation functions are not executed atomically when the memory accesses. In most cases it is not an issue, but in the case where data races are used in synchronization, it may affect execution path. Consider Listing 1, the thread waiting for stolen data busy waits in a while-loop (see `ss_steal` in Listing 1). The change of `stealIndex` will be detected at either before or after instrumentation after a remote thread writes the address. Here the problem is, the value change that is detected at the "after" instrumentation point could in fact happen before the memory access but after the "before" instrumentation point. In replay phase, if we inject the new value accordingly at the "after" instrumentation point, the effect will be only reflected at the next iteration. But in record phase, since the value change actually happens before memory access, the code will leave the while-loop in the current iteration. This extra iteration will cause the execution path diverge in the following execution, where SNs cannot be matched correctly when the value log entries. To handle this case, we also encode the source code line information in the value log and detect the diverged execution when it happens. In those cases, the diverged execution will not consume any log entries, until the execution converges again. We cannot provide a formal proof that the execution could always converge, but in practice, we found our solution worked well: only

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.

one application has this issue and it could be well-handled by our techniques.

Some applications also have the 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, otherwise bad pointers will be generated in replay phase and cause segmentation faults. This could be explained by the an example. 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 considered as unchanged. In our example, after (1), in shadow memory, `@tmp1` holds the value returned by `upc_alloc()`. At (2), we find the value `@tmp1` is *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. This technique also has the effect of reducing value log size, as it can avoid logging values produced by the local thread.

Finally, we also instrument the shared memory allocation function and always set the content of newly allocated object to zero. Otherwise, the object may contain some values that are the same as previous objects at same addresses. Those old values may be already in shadow memory. This could lead to the the side effects when we need to log the values of the new object: we may miss some values that would have been logged due to the equivalence of old values in shadow memory.

8. EVALUATION

In the evaluation, we use fifteen UPC benchmarks. Eight NAS Parallel Benchmarks [3] (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 [5, 25] (fib, nqueens), Unbalance Tree Search (UTS) [28] and Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly (Meraculous) [17]. Table 1 shows the parameters and data sets used in experiments.

App	Native Exec.	R_Set=2	R_Set=4	R_Set=8	R_Set=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: OPR Overhead

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 [17]. 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. Nondeterminism is a main feature of data-driven synchronization in de Bruijn graph traversal. To traverse the graph, all threads independently start building subcontigs and no synchronization is required unless two threads pick k-mer seeds that eventually belong in the same contig. In this case, the threads have to collaborate and resolve this conflict in order to avoid redundant work. A lightweight synchronization scheme is the heart of the parallel de Bruijn graph traversal. Essentially, the synchronization protocol maintains a distributed state machine. The readers could refer to [17] for more details.

In UTS, nondeterminism exists in dynamic work stealing, when a thread needs to steal certain amount of work from other threads, the thread that provides the stolen work depends on the current status of each thread and the order that steal requests arrive. fib and nqueens run on top of a work stealing task library.

8.1 Experiment Setup

Partial record and replay 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 record overhead and how it is affected by different replay group sizes. For each experiment, we choose four different R_Set sizes: 2,4,8 and 16. R_Set size is expected to be small for partial replay. Since each node in Edison contains 24 cores, we make sure that threads in R_Set execute on different nodes (e.g. when R_Set 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. Ideally, for replay phase, we would have modified the UPC runtime so that we can execute just threads in R_Set using smaller number of cores. We have not added this support at this point as it involves nontrivial modifications to UPC runtime system. Instead, we still start the same number of threads in replay as full execution but modify the source code to only execute the threads in R_Set after the execution starts. The replay correctness is verified manually by comparing the results and outputs. Also note that 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.

8.2 Experimental Results

Table 2 shows our results. For each application, we show the native execution time without any instrumentation, the overhead for different R_Set sizes, size of shadow memory allocated and the largest log size among all logs generated by threads in R_Set.

8.2.1 Record Overhead

We first consider the overhead of the smallest replay group size (R_Set=2). We see that OPR introduce overhead from 1.39x $\sim$ 27.5x. For FT, the high overhead (27.5x) is due to the large ratio between log size and shadow memory size. More details are explained later. For uts-upc, the high overhead (25.4x) 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. Note that we do not expect the native execution to have the same behavior as the recorded executions. Among all R_Set sizes, OPR introduces 29.4x overhead at most in FT with 16 replayed threads.

8.2.2 Overhead vs. R_Set Size

With different replay group sizes (R_Set=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. However, because replay group size is normally not large (we expect that bugs are normally localized among a small number of threads) and the scalability enhancements in our simplified vector clock algorithm, the overhead increase is almost negligible.

8.2.3 Shadow Memory

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. For all applications, we found that the shadow memory size increases when the executions start and then become stable after certain points. The largest shadow memory size appears in Meraculous. Essentially, shadow memory of each thread captures the data read and written by it. In this experiment, the input data is around 150 GB and we use 480 threads. OPR also uses a separate shadow memory to keep written values, so the total size grows to 5GB.

8.2.4 Log Size

The final column shows the largest log size generated by a thread in R_Set for each application. We also see that the log sizes vary a lot. The naive implementation performs a log file write on each access, this obviously incurs huge overhead. In our implementation, we used a 1 GB log buffer in memory and only writes logged read values into log file when the buffer is full. After this optimization, the record overhead became reasonable.

Besides the instrumentation overhead, we found that the log size and shadow memory size are also related to record overhead. In general, the larger the ratio between log size and shadow memory size, the larger record overhead tends to be. It is particularly true if the shadow memory size is large. The intuition is that, shadow memory is a "filter" to decide whether values need to be logged. Therefore, it needs to be accessed on all memory accesses. When the ratio between the two sizes are large, it indicates that for most accesses, value comparisons are needed. Such byte level comparison contributes to the record overhead. This is the case for FT, where the ratio is around 22. 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. This observation also suggests future optimizations that potentially avoids comparing values in some scenarios.

9. DISCUSSION

The overheads reported in this study are associated with the full program run and are similar to other memory tracing tools. They also capture the upper bound for values in practice as they contain program initialization stages that sweep memory and bloat the logs.

The reported overheads are acceptable for debugging, but too large for resilience purposes. This is especially true when considering that deterministic replay [40] for MPI reports less than $2\times$ slowdown. Since most of the OPR overhead comes from `load/store` instrumentation, we believe that static analysis or profiling techniques can greatly prune and reduce the instrumentation overhead. Such techniques have been exploited by Park [29], that reports data race detection at scale with less than 50% runtime overhead. The insight is that only accesses to global data need to be tracked. To disambiguate overlapping transfers (e.g. Puts), we need to capture only the load of the first word in the transfer and program slicing techniques can be employed to further reduce overhead.

We bound runtime overhead by running approximations of vector clocks and non-atomic instrumentation. For resilience purposes this has no effect on correctness - the final memory contents after replay are correct since they come from value logs. For debugging, non-atomic instrumentation may mis-report communication orderings, e.g. it may confuse the order of two `Put` operations to the same memory location. Given that we use data replay, the order can be reconstructed by reconciling the payload with the observed memory contents. Thus, the only scenario we cannot disambiguate is when two Put with identical payload occur to the same memory location, with no causality in between (i.e. separated by Gets). Hardware support may be required to this functionality when debugging. Deadlock is not possible in replay run which is based on potential imprecise event orders. Because in record phase, each access updates vector clock and generates orders in program order. It is not possible for an access in a thread to wait for an older access in the same thread. Moreover, OPR does not support broadcast yet, but the value changes due to broadcast are detected in the same way by shadow memory.

Techniques for choosing the replay sets in practice have been described by Xue et al [40]. They identify groups of threads that interact most and provide evidence that these have indeed few members only in their applications of interest. Another interesting potential approach is to use Symbiosis [22], a concurrency debugging technique based on differential schedule projections (DSPs). A DSP shows the small set of memory operations and data-flows responsible for a failure, as well as a reordering of those elements that avoids the failure. OPR could choose R_Set based on the small set of memory operations. Moreover, logs generated by OPR could also help Symboisis reproduce or search for failures. We leave this as future work. In the resilience realm, modern techniques [13] already advocate a logical decomposition into thread groups that can be independently restarted and manipulated. Other debugging tools such as data race detectors [29] or stack inspectors [9], already identify groups of threads of interest.

A separate and perhaps more interesting question when considering resilience is whether programming using one-sided communication is worth the trouble. One-sided communication is perceived as being able to provide better performance than two-sided communication. Scalable resilience requires uncoordinated recovery, aka group recovery. As our study indicates, group recovery for SPMD using one-sided communication is likely to be more expensive than group recovery for SPMD two-sided. It really remains to be seen if compiler assist can lower enough the overhead necessary to provide deterministic replay for one-sided communication.

10. OTHER RELATED WORK

Deterministic R&R has been studied for multiple programming languages and models. Early work [12] for Java infers and controls thread schedule by intercepting all calls to the synchronization API.

PinPlay [30] provides deterministic R&R for `pthreads` and MPI based programs. It uses the same technique for value logging as we do. While replaying groups of `pthreads`, PinPlay can't maintain order for process based implementations, so it can replay only a single MPI rank. OPR handles groups of tasks, independent of their instantiation (pthread or process). We have already discussed state-of-the art MPI group [40] replay and the differences between one-sided and two-sided communication.

Altekar et al [8] introduce the notion of output deterministic replay for multicore debugging. ODR infers data race outcomes from an output deterministic run. An output deterministic run inferred in polynomial time using information recorded during a test run. In a sense, our approach in OPR when using non-atomic instrumentation provides output deterministic replay.

Hardware support for replay has received attention, mostly for shared memory. In distributed memory, MPReplay [36] 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. However, architectural support for one-sided communication is likely to critical to reduce the overhead or R&R techniques. This includes atomic logging of transfers NIC/CPU to infer communication order. However, this support solves the debugging problems and it may not be worth for resilience purposes when using value logging.

11. CONCLUSION

One-sided communication is widely used in Partitioned Global Address Space (PGAS) programming models. Despite the potential performance advantages, its inherent nondeterminism makes debugging even more difficult. In this paper, we present a general tool, *OPR* (**O**ne-sided communication **P**artial **R**ecord and Replay) to support deterministic R&R for one-sided communication. Partial replay allows users focus on events within a specified small set of threads. It could ease debugging experience and relieve users from monitoring all concurrent events from potentially thousands of threads. OPR is built based on Berkeley UPC. OPR allows users to deterministically replay a subset of threads in a full execution without executing the rest of threads. The principle of data-replay is used to ensure replay correctness, inter-thread communications among threads in replay group are inferred at replay phase based on value matching. To the best of our knowledge, OPR is the first software tool that supports deterministic R&R for one-sided communication. We demonstrate practicality of our approach by evaluating the tool using 15 applications. OPR introduced an overhead ranging from $1.3\times$ to $29\times$, when running on 1,024 cores and tracking up to 16 threads. In future, we will exploit the application of our techniques on resilience mechanisms using uncoordinated or quasi-synchronous checkpointing and recovery.

12. REFERENCES

[1] *Berkeley UPC. http://upc.lbl.gov.*
[2] *The Chapel Parallel Programming Language. http://chapel.cray.com/index.html.*
[3] *The NAS Parallel Benchmarks. Available at http://www.nas.nasa.gov/Software/NPB.*
[4] *UPC Home Page. http://upc-lang.org.*
[5] *UPC Task Library. http://upc.lbl.gov/task.shtml.*
[6] *X10: Performance and Productivity at Scale. http://x10-lang.org.*
[7] *MPI: A Message-Passing Interface Standard. Version 3.0.* Message Passing Interface Forum, 2012.
[8] G. Altekar and I. Stoica. ODR: Output-deterministic Replay for Multicore Debugging. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, 2009.
[9] 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.
[10] 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.
[11] 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.
[12] 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.
[13] 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.
[14] 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.
[15] 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.
[16] 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.
[17] 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.
[18] 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.
[19] 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.
[20] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *ACM SIGMETRICS'10*, 2010.
[21] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471–482, April 1987.
[22] 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.
[23] D. Manivannan and M. Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. *IEEE Transactions on Parallel Distributed Systems*, 10(7):703–713, July 1999.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] 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.
[29] C.-S. Park, K. Sen, and C. Iancu. Scalable Data Race Detection for Partitioned Global Address Space Programs. *SIGPLAN Not.*
[30] 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.
[31] 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.
[32] K. Sen. Scalable Automated Methods for Dynamic Program Analysis. In *Ph.D Thesis*. University of Illinois, Urbana-Champaign, 2006.
[33] K. Sen, G. Rosu, and G. Agha. Runtime safety analysis of multithreaded programs. In *ESEC/SIGSOFT FSE*, pages 337–346, 2003.
[34] 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.
[35] 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.
[36] 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.
[37] 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.
[38] 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.
[39] 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.
[40] 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.