Scalable Replay with Partial-Order Dependencies for Message-Logging Fault Tolerance

Jonathan Lifflander
Department of Computer Science
University of Illinois Urbana-Champaign
Email: jliffl2@illinois.edu

Esteban Meneses
Center for Simulation and Modeling
University of Pittsburgh
Email: emeneses@pitt.edu

Harshitha Menon, Phil Miller
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: {gplkrsh2,mille121}@illinois.edu

Sriram Krishnamoorthy
Advanced Computing, Mathematics, and Data Division
Pacific Northwest National Laboratory
Email: sriram@pnnl.gov

Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: kale@illinois.edu

***Abstract*—Deterministic replay of a parallel application is commonly used for discovering bugs or to recover from a hard fault with message-logging fault tolerance. For message passing programs, a major source of overhead during forward execution is recording the order in which messages are sent and received. During replay, this ordering must be used to deterministically reproduce the execution. Previous work in replay algorithms often makes minimal assumptions about the programming model and application to maintain generality. However, in many applications, only a partial order must be recorded due to determinism intrinsic in the program, ordering constraints imposed by the execution model, and events that are commutative (their relative execution order during replay does not need to be reproduced exactly). In this paper, we present a novel algebraic framework for reasoning about the minimum dependencies required to represent the partial order for different orderings and interleavings. By exploiting this framework, we improve on an existing scalable message-logging fault tolerance scheme that uses a total order. The improved scheme scales to 131,072 cores on an IBM BlueGene/P with up to $2\times$ lower overhead.**

***Keywords*—replay, partial-order dependencies, fault tolerance, message logging, determinism, execution model**

I. Introduction

As we approach the exascale era, distributed algorithms must evolve to meet the increasing demands of large-scale application development. With continuing increases in core counts, researchers have posited that failures may become the limiting factor to increasing performance [1]–[3]. Thus, applications will have to adopt advances in fault tolerance practice beyond checkpointing to external storage and whole-job restart after failures. At such large scales, debugging parallel applications will also require new techniques.

Both online fault tolerance and parallel debugging are popular uses of *deterministic replay* techniques. Replay can be used during debugging to repeatedly execute the same sequence of operations, so that developers can make diagnostic observations and test potential bug fixes. In fault tolerance protocols, replay techniques can optimize recovery and enable consistent uncoordinated checkpoints. Thus, improvements in replay techniques are broadly beneficial.

Message-logging fault tolerance protocols use a hybrid of: (a) online checkpoints to limit recovery efforts only to work executed since the checkpoint [4], (b) *data-driven* replay to bound recovery just to work executed by the failed unit of the system [5], and (c) *control-driven* replay (recording the order of events) to ensure that the recovered results are consistent with the parts of the system that did not experience a fault [6]. The presence of checkpoints also limits the volume of stored messages and control information, since records from before a successful checkpoint can safely be discarded. All of these approaches have benefits as described, but they also come with costs in the form of time and storage overhead during forward execution. Ongoing research attempts to address each of these sources of overhead in various ways. In this paper, we focus on new techniques for reducing the control-associated overhead.

The control-associated overhead in replay protocols typically stems from creating *determinants* and reliably storing them in multiple locations so they will persist past a failure. For a given event (a message arriving, for instance), a traditional message-logging determinant will record the sending and receiving processors along with local sequence numbers (processor-local message counters) for the sender and receiver. However, in general, control-driven replay schemes must only record the minimum amount of information to reliably reproduce the application's previous execution behavior. Any additional control information that a given scheme records can be considered "excess" and can be removed. Prior work has considered only limited sets of assumptions about what control information must be recorded. Generic schemes record all control, and attempt to minimize the overhead this imposes. Implementations that consider *causality* among events only can record details of a definitive subset of events [7]–[9]. Other work has shown that where a program can be shown to be perfectly *deterministic*, no control needs to be recorded [10].

However, applications often have additional characteristics that enable further optimizations. Application codes, along with their underlying programming and execution models, embody *intrinsic order* that implicitly determines much of the overall control, and thus communication order, even if the application is not fully deterministic. One example of intrinsic order is MPI's "non-overtaking" receive semantics [11]. Applications can also contain sets of events that are *commutative*, in that reordering them relative to one another will produce the same resulting state. Based on these observations, we develop a scheme that distinguishes events whose order does not matter

to avoid recording their determinants, while still providing replay in an equivalent order.

Based on these ideas, this paper makes the following contributions:

- A broader conception of optimization opportunities in control-driven replay schemes
- An algebra that models how intrinsic order and commutativity affect the dependency information necessary for replay (§ IV)
- A map of a partial-order dependency from the algebraic description to a determinant along with its proof of correctness (§ VI)
- A fault tolerance implementation using message-logging called PARTIALDETFT that uses partial-order determinants to store the control ordering, as reduced by causality, intrinsic order, and commutativity (§ VII-C)
- A demonstration that our approach leads to practical speedups by empirically measuring the forward-execution overhead and recovery time on up to 131,072 cores of Intrepid for three benchmarks (§ VIII).

II. RELATED WORK

The seminal works on distributed control-oriented replay assumed that all the control operations were stored, i.e., a total order on the control sequence was saved [5], [12]–[14]. The first work to diverge noted that causality tracking can be used to aid in parallel debugging by defining a partial order based on causality [9]. The work following this made the observation that in message-passing programs, only the order of messages that race must be saved [7]. The authors conclude that this can be detected at runtime by comparing the previous reception for a given process to an incoming message and concluding that if both messages are on the same channel, then there was a potential race. They also track causality to distinguish between causally separated messages. However, the major assumption made is that every message is received exactly once. For the sake of fault tolerance or, in some cases, debugging, this assumption may not hold. If an endpoint fails or crashes unexpectedly, there may be messages in flight that could race, which were not considered. Also, this work is very specific to a sequenced message-passing model. Other work has identified theoretically how messages that race can be detected [15]. Later work improved on this practically by defining *block races*, or sets of messages that can possibly race, and concluded that constructs, such as MPI_Iprobe in MPI, may need to be traced [8]. Further work elucidates the patterns of MPI primitives that introduce non-determinism [16], and replay schemes have been devised for those constructs [17]. Our work also seeks to reduce the amount of control data required, but it solves a more general problem not specific to a certain programming paradigm. Specific schemes for reducing control-related data have been studied extensively for a wide variety of algorithms [18]–[22].

Recent work has revived interest in message-logging for tolerating hard failures by making the observation that some MPI program are *send deterministic* [10]. A send deterministic MPI program is defined as a program where every process sends the same sequence of messages in every valid execution regardless of the reception order of non-causally related messages. It is essentially the subset of MPI programs that are deterministic given the non-overtaking (FIFO) assumptions of the MPI model. Ff a program is send deterministic, the authors conclude that no determinants are needed. In comparison, our work is not specific to MPI and can elide determinants selectively rather than making an all-or-nothing decision based on whether or not the whole program is deterministic.

Other work has theoretically discussed the granularity of events when modeling program executions [23], which is related to how we model commutative regions (by collapsing transition states/events from the environment's perspective). In the conclusion of [23], the authors mention that this may be useful for optimizing replay algorithms. KAAPI is a DAG-based system in which the primary control is both fully intrinsic and programmatically accessible to the runtime system. The authors have shown how this can be used to reduce the effort required during recovery from a pure checkpoint to the minimal set that the failed process depends upon [24].

III. MOTIVATION

We have observed that there is generally some amount of order in parallel programs, which typically is intrinsic and results from how the program is written. In general, writing a parallel program is difficult, so encoding ordering constraints in the code often makes reasoning about correctness easier. Algorithms and applications may also have commutative regions, where a subset of the messages may have their order transposed and the resultant state is identical.

All previous work in this area has made the assumption that even if some control data is omitted due to intrinsic determinism, during replay each process will always execute exactly the same sequence of events. Our work diverges from that notion in two ways: (1) by defining an endpoint more generally (logical endpoints on the same physical process may interleave differently) and (2) allowing for events to change order for commutative regions within an endpoint. As far as we know, this is the first work to allow replay to vary in order, which significantly changes how the replay protocol must be formulated.

For the sake of replay, exploiting commutative regions to reduce storage may introduce pernicious behavior if the programmer manually annotates a region of code as commutative and is actually wrong. However, by informing the replay system of this presumption, it may actually aid in debugging. For the sake of fault tolerance, exploiting commutativity leads only to advantages, assuming it is correct.

IV. THEORY

We define an *endpoint* in a distributed system to be a control unit that is scheduled by the system. Previous work in this area considered processes to be endpoints, but our definition broadens it to a task, object, virtual process, etc.

To analyze the number of dependencies required to produce a partial order for replay, we model the execution of a concurrent program as a sequence of *events*. An event e is a concurrent operation. In some programming models, this is sending or receiving a message, but it may include participating in any non-deterministic choice. For the sake of reducing description complexity, we will consider incoming and outgoing messages to be the only types of environmental events. We make minimal assumptions about how messages are sent or received or regarding how the execution model schedules work. We intend to provide a theory that is applicable to many communication and runtime models.

We use event equality to define whether events are considered identical by the encapsulating system. Hence, we will not define a universal equivalence relation because it is very dependent on the programming and execution model. For example, in MPI an event may be distinguished by the communicator, possibly the tag, and/or the source processor. Also, for events sent from the same processor, they may be distinguishable by their sending order, when MPI non-overtaking rules apply.

The following is a simple example of a message-passing equivalence relation that differentiates between events based on the tag (where a tag could be "ANY") and source:

$$\begin{aligned}(e_1 = e_2) \quad \equiv \quad & (e_1.source = e_2.source \wedge \\ & (e_1.tag = e_2.tag \vee \\ & e_1.tag = \text{ANY} \vee e_2.tag = \text{ANY}))\end{aligned}$$

A concurrent program transitions through a sequence of events—some of which are deterministic, and others that are non-deterministic (for example, the result of a network race). When we have events that need to be ordered due to some non-determinism, we use the term *dependency* to mean the specification of an order between them.

A. Ordering Algebra

We define $\mathcal{O}(n, d)$ as a set of n events and d dependencies that can be accurately replayed from a given starting point. The dependencies d can be among the events in the set, or on preceding events in the program's execution. One can intuitively think of these structures as ordered sets of events. In this section, we will describe how events with various relationships to one another can reach this form. Our primary purpose is to count and characterize the dependencies necessitated by each such relationship.

To start, we define a sequencing operation, $\boxplus$, that places the execution of one event after another:

$$\mathcal{O}(1, d_1) \boxplus \mathcal{O}(1, d_2) = \mathcal{O}(2, d_1 + d_2 + 1)$$

Intuitively, this tells us that between two atomic events, we need a single dependency to tell us which one comes first. We can then observe that constructing a set of n events by sequencing adds $n - 1$ dependencies to them in total:

$$\boxplus_i^n \mathcal{O}(1, d_i) = \mathcal{O}(n, \sum_i^n d_i + n - 1)$$

Generalizing this notion to a larger set of events:

$$\mathcal{O}(n_1, d_1) \boxplus \mathcal{O}(n_2, d_2) = \mathcal{O}(n_1 + n_2, d_1 + d_2 + 1)$$

Concretely, we can view this as adding a dependency from the final event of the first set to the first event of the second set.

We now consider an "unordered" set of events, $\mathcal{U}(n, d)$, which occurs in any order during initial execution but must replay in the same order. An example of an unordered set is the common communication pattern found in codes where several messages are sent to an endpoint, which will all be received eventually. However, depending on which message arrives first, the eventual state will be different.

We can decompose such a set $\mathcal{U}(n, d)$ into atomic events with an additional dependency between each successive pair:

$$\begin{aligned}\mathcal{U}(n, d) &= \mathcal{O}(1, d_1) \boxplus \mathcal{O}(1, d_2) \boxplus \cdots \boxplus \mathcal{O}(1, d_n) \\ &= \mathcal{O}(n, d + n - 1)\end{aligned}$$

where $d = \sum d_i$. Thus, we add an additional $n - 1$ dependencies to fully order the n events. From this, we can derive the fact that sequencing two unordered sets of events is equivalent to ordering the combination of the two:

$$\begin{aligned}\mathcal{U}(n_1, d_1) \boxplus \mathcal{U}(n_2, d_2) &= \mathcal{O}(n_1, d_1 + n_1 - 1) \boxplus \\ &\quad \mathcal{O}(n_2, d_2 + n_2 - 1) \\ &= \mathcal{O}(n_1 + n_2, n_1 + n_2 + d_1 + d_2 - 1) \\ &= \mathcal{U}(n_1 + n_2, d_1 + d_2)\end{aligned}$$

B. Interleaving

To address cases where multiple independent sequences of ordered events interleave at a single end point, we must generate dependencies sufficient to record their relative ordering. An example of this would be several parallel modules interacting, which by themselves are ordered, but may interleave non-deterministically.

We define an interleaving operator, $\boxtimes$, that interleaves events. We first consider the interleaving of two ordered sets of inputs:

Lemma IV.1. Any possible interleaving of two ordered sets of events $A = \mathcal{O}(m, d)$ and $B = \mathcal{O}(n, e)$, where $A \cap B = \emptyset$, is given by:

$$\mathcal{O}(m, d) \boxtimes \mathcal{O}(n, e) = \mathcal{O}(m + n, d + e + \min(m, n)) \quad (1)$$

Proof: Without loss of generality let us assume that $m \geq n$. In that case, there are $m+1$ locations of A in which n events of B can be executed. Therefore, to represent the order, we require an additional n dependencies. This operation results in $m+n$ total events with $d+e+\min(m, n)$ ordering constrains. ■

Corollary IV.2. In the two ordered set of events discussed in Lemma IV.1, for every dependency of the form $a_i \rightarrow b_j$, we replace it with a logical event a_{ij}. Note that there can be only one $\rightarrow$ to b_j. This reduces the interleaving of the two ordered set of events A and B described in the earlier lemma into an ordered set of events of A' with $\max(m, n)$ logical events and additional $\min(m, n)$ dependencies.

Lemma IV.3. Any possible ordering of n ordered set of events $\mathcal{O}(m_1, d_1), \mathcal{O}(m_2, d_2), \ldots, \mathcal{O}(m_n, d_n)$, when $\bigcap_i \mathcal{O}(m_i, d_i) = \emptyset$, can be represented as:

$$\boxtimes_{i=1}^n \mathcal{O}(m_i, d_i) = \mathcal{O}(m, d + m - \max_i m_i)$$

where

$$m = \sum_{i=1}^n m_i \wedge d = \sum_{i=1}^n d_i$$

Proof: Base case 1: The number of dependencies required to represent any interleaving of two ordered sets $\mathcal{O}(m_1, d_1)$ and $\mathcal{O}(m_2, d_2)$ of events is given by:

$$\mathcal{O}(m_1, d_1) \boxtimes \mathcal{O}(m_2, d_2) = \mathcal{O}(m_1 + m_2, d_1 + d_2 + \min(m_1, m_2))$$

(from Lemma IV.1). $\min(m_1, m_2)$ is equivalent to $(m_1 + m_2 - \max(m_1, m_2))$.

From Corollary IV.2, this results in an ordered set with $\max(m_1, m_2)$ logical events.

We now assume it holds for $n = k$, and we show that it holds for $n = k + 1$. Because it holds for $n = k$, this means that:

$$\bigboxtimes_{i=1}^{k} \mathcal{O}(m_i, d_i) = \mathcal{O}(\sum_{i=1}^{k} m_i, \sum_{i=1}^{k} d_i + \sum_{i=1}^{k} m_i - \max_{i}^{k} m_i)$$

This results in an ordered set of events with $\max_i^k m_i$ logical events. For this, there are two possible cases when we consider the $\mathcal{O}(m_{k+1}, d_{k+1})$ ordered set.

Case 1: When the number of events in $\mathcal{O}(m_{k+1}, d_{k+1})$ is fewer than the logical events in the kth ordered set, i.e., $m_{k+1} < \max_i^k m_i$. The number of dependencies required to interleave the $\mathcal{O}(m_{k+1}, d_{k+1})$ ordered set with the ordered set formed after interleaving the first k sets is $\min(m_{k+1}, \max_i^k m_i)$. The additional number of dependencies required is:

$$\begin{aligned} D_{additional} &= \sum_{i=1}^{k} m_i - \max_{i}^{k} m_i + m_{k+1} \\ D_{total} &= \sum_{i=1}^{k+1} d_i + \sum_{i=1}^{k+1} m_i - \max_{i}^{k+1} m_i \\ \bigboxtimes_{i=1}^{k+1} \mathcal{O}(m_i, d_i) &= \mathcal{O}(\sum_{i=1}^{k+1} m_i, \sum_{i=1}^{k+1} d_i + \sum_{i=1}^{k+1} m_i - \max_{i}^{k+1} m_i) \end{aligned}$$

Case 2: When the number of events in $\mathcal{O}(m_{k+1}, d_{k+1})$ is greater than the number of logical events formed by the interleaving of the previous k sets, i.e., $m_{k+1} \geq \max_i^k m_i$. The number of dependencies required is $\min(m_{k+1}, \max_i^k m_i)$. The additional number of dependencies required is:

$$\begin{aligned} D_{additional} &= \sum_{i=1}^{k} m_i - \max_{i}^{k} m_i + \max_{i}^{k} m_i \\ D_{total} &= \sum_{i=1}^{k+1} d_i + \sum_{i=1}^{k+1} m_i - \max_{i}^{k+1} m_i \\ \bigboxtimes_{i=1}^{k+1} \mathcal{O}(m_i, d_i) &= \mathcal{O}(\sum_{i=1}^{k+1} m_i, \sum_{i=1}^{k+1} d_i + \sum_{i=1}^{k+1} m_i - \max_{i}^{k+1} m_i) \end{aligned}$$

■

When a single end point will interleave independent sequences of unordered events, this produces a result akin to a combined set of events with the additional specification of their interleaving:

$$\begin{aligned} \mathcal{U}(m, d) \boxtimes \mathcal{U}(n, e) &= \mathcal{O}(m, m + d - 1) \boxtimes \mathcal{O}(n, n + e - 1) \\ &= \mathcal{O}(m + n, m + n + d + e - 1 - 1 \\ &\qquad + \min(m, n) + 1) \\ &= \mathcal{U}(m + n, d + e + \min(m, n)) \end{aligned}$$

C. Determinism

As noted earlier, many programs exhibit some degree of determinism in the order in which they will process available events. We can view a sequence of n deterministically ordered events as structurally equivalent to an ordered set of n events with no associated explicit dependencies, such that $\mathcal{D}(n) = \mathcal{O}(n, 0)$. For the purposes of sequencing, this is sufficient. However, if such a set of events is to be non-deterministically interleaved with another set of events, then its inherent determinism is partially lost. In this type of sequence, there are some particular points where execution can be "interrupted" to interleave events of another set. If we take the set $\mathcal{D}(n, k)$ to have k interruption points between chains of events that occur in a given order, we instead have $\mathcal{O}(k, k-1)$. Note that in the limit of an uninterrupted sequence, such as calling into a piece of library code that will execute n events fully deterministically, and then return, (k equals 1) this reduces to $\mathcal{O}(1, 0)$—effectively a single event.

D. Commutative Events

One particularly common form of non-deterministic behavior in parallel programs consists of events that can be treated as *commutative*—those for which their execution order has no influence on program behavior. Despite the high-level safety of executing these events arbitrarily, existing theories of message logging and record-replay rely on strictly repeating a recorded order to produce their promised results. We aim to lift that restriction.

For purposes of counting the number of dependencies that must be recorded to reproduce an execution, we can model a set of commutative events as $\mathcal{O}(2, 1)$—a beginning and end event sequenced together. Sequencing other sets of events with a commutative region simply places them before or after the region. Interleaving other events with the commutative region (on the assumption that the associated execution behaviors are isolated between the commutative events and their interleaved brethren) places those events in three buckets: (1) before the begin event, (2) during the commutative region, and (3) after the end event. This exactly corresponds to an ordered set of two events.

E. Summary

Taken together, the considerations described in this section enable meaningful analysis of a large proportion of distributed parallel software. In cases where in-depth analysis in terms of this theory is limited, the formulation still produces meaningful results, in which most events are directly sequenced. Where a subset of a program's events have their execution accounted for by the theory, the associated dependencies can be proportionately optimized. The information necessary to make these determinations can come from manual inspection, annotation and semi-automatic elaboration, or full static compiler analysis. In the next section, we examine several codes that do not necessarily satisfy previous definitions of determinism, but can benefit from our analysis.

V. Example Codes

A. Sorting: Sequoia IOR Benchmark

An example of commutativity in code is when a processor performs a sequential stable sort using data that it has received from other processors in any order. Because the outcome will be identical regardless of the input order, the incoming messages are commutative. This pattern arises in the Sequoia IOR benchmark:

```
if (rank == 0) {
  for (i = 1; i < numTasks; i++)
    MPI_Recv(arr[i], ..., MPI_ANY_SOURCE, MPI_ANY_TAG)
  sort(arr);
} else {
  MPI_Send(data, 0, ...);
}
```

On non-zero ranks, this is clearly deterministic. On rank 0, there are `numTasks` messages received in arbitrary order, followed by a single sorting call. These receives form a commutative region whose start and end events are deterministically linked to their predecessor and successor events.

B. Halo exchange

This code shown here is an example of halo exchange, where data is exchanged between pairs of "neighbor" ranks. This is a common pattern found in many codes operating on structured and unstructured meshes. The `MPI_Irecv` has a distinct tag for every sender followed by `MPI_Waitall`. Therefore, irrespective of the input order of messages, they would be received in the appropriate buffers.

```
for(i = 0; i < num_nbors; i++) {
  MPI_Irecv(rbuf[i], ..., ANY_SOURCE, tag[i], ...);
  MPI_Isend(sbuf[i], ..., nbor[i], tag[i], ...);
}
MPI_Waitall(...);
```

This, too, illustrates a commutative region whose entry and exit are deterministic.

LULESH [25] (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) follows a similar communication pattern as a halo exchange, where each node and element exchanges data multiple times with varying number of neighbors, depending on the input. We use this code in our experimental validation, in § VIII.

C. AMG2006

AMG2006 is an implementation of an algebraic multigrid solver for linear systems resulting from unstructured meshes. It makes a compelling example because it contains a communication pattern that starkly rules out the logic of send-determinism in eliminating the need for determinants [10]. This pattern is illustrated in Figure 1.

During this segment of execution, each rank sends a set of requests for data to a number of other ranks. Each rank then loops around a set of tests for both new incoming requests and responses to its own requests. Once all of a rank's own requests have been satisfied, it then shifts to polling a tree-based asynchronous barrier to determine when all other ranks' requests have been satisfied, so that execution can proceed. This implements a form of dynamic sparse data exchange [26].

This pattern includes three distinct commutative regions: (1) receiving requests and responding to them, (2) receiving and recording responses to posted requests, and (3) receiving fan-in message from children in the upward propagation of the barrier tree. The work done in response to events in each region is fully independent of that in the other regions. This independence allows us to conclude that no order needs to be enforced among these events to produce a repeatable execution.

D. Master-Slave

A more intriguing example for our theory can be seen in codes structured according to a master-slave paradigm:

```
if (rank == 0 {
 while ((work = nextUnit()) != NULL) {
   MPI_Recv(&request_rank, 1, MPI_INT, MPI_ANY_SOURCE...);
   MPI_Send(work, . . ., request_rank); }
} else {
 while(1) {
   MPI_Send(rank, 1, MPI_INT, 0, . . .);
   MPI_Recv(work, . . ., 0);
   doWork(work); }
}
```

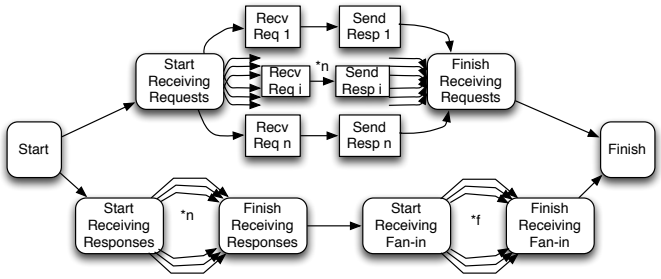


Fig. 1. The communication flow of the non-deterministic segment of AMG2006. The top branch shows a set of commutative events where requests for data are received from other processors, and responses are sent to them. The bottom branch shows the receipt of those responses, and a subsequent asynchronous barrier to ensure that no further requests will be received before proceeding.

We must consider the master rank and slaves separately. First, we note that the messages sent and received by the slave ranks are fully ordered by the master's sends. Any messages they may send are directly causally dependent on those receives. Thus, the slaves require no dependencies to reproduce their top-level execution flow.

Because of the wildcard matching on the master and the non-deterministic order in which other ranks may make requests, those events are naturally unordered, and must be explicitly sequenced. Therefore, this code represents a hybrid, in which one rank must record nearly all of its operations, while other ranks record none.

E. Molecular Dynamics: LeanMD Benchmark

The Charm++ [27] molecular dynamics benchmark, LeanMD, has several communication patterns that are commutative. In Charm++, work is decomposed into collections of objects. In our theory, we model each object as a distinct endpoint. In LeanMD, the physical space is spatially decomposed into a grid of "box" objects, each of which holds a set of particles that interact. Every interaction between a box and one of its spatial neighbors is assigned to an additional "compute" object.

In every step, each box sends its set of particles to its particular associated interaction object. Hence, each interaction object receives two sets of particles, which may arrive in any order. However, these receives are commutative because the computation waits for both sets to arrive. Once each interaction object finishes calculating the forces, it contributes the forces to a sum reduction to each connected box. Due to how reductions are implemented in Charm++, the sum-reduction is deterministic and will produce the same result, regardless of the order.

Thus, according to our theory, we can conclude that LeanMD does not require any dependencies.

VI. PO-Replay: Partial-Order Message Identification Scheme

The theoretical bounds on dependencies for different types of orders in a distributed application, expressed in § IV, allow us to reduce the amount of control data required for deterministic replay depending on the application. We now define a replay algorithm that maintains correctness with partial-order dependencies (including commutative regions of code that may

be reordered during replay). Our algorithm has the following properties:

- It tracks causality using Lamport clocks and replays messages in causal order, except for commutative regions.
- It uniquely identifies a sent message, whether or not the order of message receptions is transposed.
- It requires exactly the number of *determinants* as dependencies as those specified in § IV.

A determinant is a piece of information stored to maintain the order in a given execution of the application. For replay, it may be written to disk or memory. For fault tolerance, it will be sent to other processes and stored *stably*, or propagated to as many processes as needed for it to persist past a failure scenario, depending on the reliability requirements of the system.

We assume an interface that allows us to determine if a message being received is *order-dependent* or *order-independent* based on the theory. An order-dependent message requires a dependency to be stored for it to execute properly during replay. An order-independent message is either ordered by the code or is part of a commutative region. The interface specifically returns for a given incoming message m, any previous messages P that this message depends on to execute correctly (that is, P must execute before m). Note that, in general, we do not assume how the theory is implemented. User annotations, static compiler analysis, etc., could be possible implementations depending on the particular parallel paradigm/language.

Replay or fault tolerance protocols typically store a *sender sequence number* and *receiver sequence number* along with the sending and receiving process as a determinant. The sequence numbers are endpoint-local scalars that increase linearly as messages are sent and received, respectively. However, in our theoretical formulation, we require more than process sequence numbers because we intend to allow receptions to transpose during replay for commutative regions, which means that the sequences may be different.

For the replay to be correct for fault tolerance, messages that are sent must be uniquely identifiable, so duplicate messages can be ignored. Duplicates may arise because during forward execution message logging makes no assumptions about whether or not messages have actually arrived at their destination. Hence, during replay on a subset of the endpoints, all messages are re-sent, while the ones that also arrived during forward execution are simply ignored. Thus, our algorithm must ensure that every sent message in the system can be uniquely identified globally.

To identify messages regardless of the reordering of commutative messages, we mark messages based on their path to their causal ancestor within a transitive commutative region. In this way, the identification system is not based unilaterally on the endpoint's current state, but on the chain of commutative messages that caused the message to execute. We show this method is sufficient in Theorem VI.1 by proving that inside a commutative region, other messages that do not commute are not allowed.

The extra assumptions we make for our unique identification scheme are that the size of a commutative region is known *a priori* (when the first message for it arrives), and for each reception inside the commutative region, there is a known upper bound on the number of messages that will be sent (the number of messages is causally dependent on the message received).

Given these assumptions, we can assign a number to a commutative message that represents its path from its causal ancestor in the transitive group. A message m_{ij} on level i and jth child among c children is assigned a path number p_{ij}, which is $p_{parent}bin_j$, where p_{parent} is the path of the parent and bin_j is the binary representation of the jth child. Hence, bin_j will consist of $\log_2 c$ bits.

Theorem VI.1. Every message identified by the tuple (SRN, SPE, CPI) forms a unique global identifier, regardless of the execution order, where:

SRN is the ***sender region number***, which is an endpoint-local sequence number incremented for every send outside a commutative region and incremented once when a commutative region starts

SPE is the ***sender process endpoint***, which is a unique identifier for every endpoint in the system

CPI is the ***commutative path identifier***, which is zero outside a commutative region and equal to a sequence of bits that represents the path to the root of the commutative region.

Proof: Recall from the definition of a commutative region in § IV-D that events occurring within a commutative region must be isolated, both with respect to each other and any non-commutative interleaved events.

We can now consider two cases:

1) *Non-commutative region*: Because non-commutative regions are completely ordered, the SRN and SPE uniquely identify every message sent. Because the start of each commutative region on an endpoint increments the SRN, it is ordered with respect to the non-commutative regions.
2) *Commutative region*: We already have shown that commutative regions are ordered with respect to non-commutative regions and cannot be overlapped with a non-commutative region. Given this, we now show that each message inside the commutative region can be uniquely identified. Each commutative region is uniquely identified by the SRN and SPE, and the size of the region is known. Hence, messages in different commutative regions can be distinguished. The CPI includes a set of bits that represents its path to its ancestor. Because every branch has a different bit pattern according to our scheme, the path will be unique for any message received.

■

VII. PartialDetFT: Fault Tolerance with PO-Replay

A. System Model

We assume a system with P processes that communicate via message passing. A process in this model is the unit at which a failure may occur. Processes communicate along non-FIFO channels using messages that are sent asynchronously and possibly out of order, but they are guaranteed to arrive sometime in the future if the recipient process has not failed. Before a message is processed, it may be preprocessed (or buffered) by the system or user to delay the reception based on its content.

Each process may have a set of *tasks* mapped to it. A task in our model is an entity that the runtime system maps to

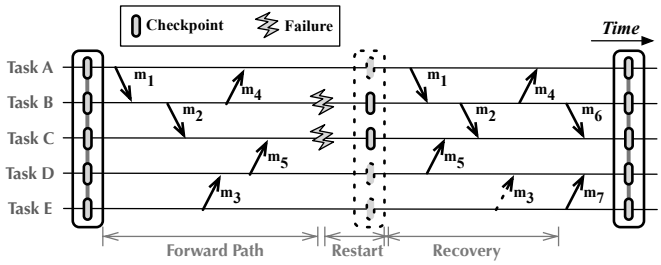


Fig. 2. Execution through failures with rollback-recovery techniques. Dotted elements appear only in checkpoint/restart, but not in message logging.

a process, and is possibly relocated during execution. A task has associated data with it (that migrates with it) and some functions that execute when a message arrives. A message is always directed toward a task and a particular function that executes on that task.

We assume a *fail-stop* model for all failures: failed processes do not recover from failures, nor do they behave maliciously (non-Byzantine failures). Process replacement is not required after a failure, but excess processes/nodes can be used in post-failure execution. Failures are detected by a system component that notifies the runtime of a failure. Failure detection is not the focus of this paper, so we assume an adequate mechanism.

B. Background

1) Checkpoint/Restart: A well-established method for providing fault tolerance in a large-scale system is by saving a snapshot of the system state and rolling back to a previous snapshot in case of failure, commonly known as *checkpoint/restart*. With checkpoint/restart, the system takes periodic checkpoints that are used to recover the state if the system crashes. There are several libraries for HPC applications that provide checkpoint/restart functionality. Checkpoint/restart has many variants that depend on the amount of data included in the checkpoint.

2) Message Logging: Message logging is an extension to the checkpoint/restart mechanism that reduces the amount of work that must be re-executed when a failure occurs. Instead of rolling back all the processes at the point of failure, only the tasks on failed processes must be re-executed. There are many variants of message logging [28], but we focus on the causal pessimistic protocols, which have been shown to perform well [29], [30].

Sender-based pessimistic message logging works by each process logging all of the messages that it sends to another process past the checkpoint (messages before the checkpoint can be discarded). When a failure occurs, the checkpoint is restored for the failed processes, and the messages logged for the failed processes are used to rebuild the state of the failed processes. All other processes can continue execution while the recovery occurs but may be limited if they depend on the failed processes until the failed processes catch up.

To maintain a correct recovery (equivalent to replay), determinants must be stored for each message reception, so the order of received messages is preserved during re-execution. Message-logging protocols typically make the assumption of piecewise determinism: the property that the order of message receptions matched with a given sent message is the only non-deterministic event that can affect the state. A determinant is composed of the sender sequence number and process along with the receiver sequence number and process. Determinants must be propagated as soon as a process sends a message because, at this point, it may affect the state of the system as a whole. Typically, determinants are either synchronously sent before an application message is allowed to be sent, or they are appended to any sent messages until an acknowledgment is received that they are saved in enough locations, given the system's reliability requirements.

Figure 2 shows an example of an execution with five tasks using a rollback-recovery mechanism. The *forward path* is the portion of the execution with no failures. As soon as a failure affects the system (in this example, affecting Tasks B and C), the system rolls back to a previous checkpoint and resumes execution. During recovery, different message receptions are possible, depending on the determinism of communication of the application. For instance, if checkpoint/restart is used, message m_5 may be received in a different order after the failure. By using determinants, traditional message logging guarantees that message delivery occurs exactly as it did before the crash. So, with message logging, message m_5 will be delivered after m_2 during recovery.

C. PARTIALDETFT Algorithm

We now describe the PARTIALDETFT algorithm for tolerating hard failures. The replay algorithm based on our theory is a subset of the algorithm presented in this section. We use the following data structure to store the local state for each process and determinant:

```
struct Determinant { int SRN, SPE; bitvector CPI; }
struct ProcessLocalData {
    bitvector parCPI;
    int RSN, SRN, LCSN;
    list<Determinant> unackedDets;
    list<Message> msgLog;
    int lClock;
}
```

The local state of each process is initialized as follows:

```
void initProcessLocalData() {
  parCPI = RSN = SRN = LCSN = lClock = 0;
}
```

When a message is received from an endpoint (not including self sends), we execute the following:

```
when recvMessage(Message msg) {
  // ignore duplicate msg
  if (msgLog.contains(msg)) return;

  lClock = max(msg.lClock, lClock) + 1;
  RSN++;
  RPE = myEndpointID;

  if (isOrderDependent(msg)) {
    unackedDets.add(Determinant(msg.SRN, msg.SPE, RSN,
        RPE));
    // ask for ack on the new determinant, and remove from
        unackedDets when it is received
  }
  parCPI = msg.CPI;
}
```

When a message is sent from a process, the following is performed, and several pieces of information may be augmented to the sent message:

```
when sendMessage(Message msg, int toPe) {
  // add on all the unacked determinants
  msg.augmentDets(unackedDets);
  SPE = myEndpointID();
  CPI = parCPI;
  lClock++;

  if (isCommutative(msg)) {
    if (isNewCommutative(msg)) {
      SRN++; LCSN = 0;
    } else
      LCSN++;
    CPI = parCPI.append(LCSN);
  } else SRN++;

  msg.augmentSeq(SRN, SPE, CPI);
  msg.augmentClock(lClock);
  msgLog.add(msg, toPe);
}
```

Now, we provide a summary of the algorithm for replay, after a failure occurs:

```
when failureNotify(list<int> failedEndpoints) {
  // find all determinants for messages sent to the failedEndpoints
  // send determinants to the endpoint that logged the message
  // wait for all sends to complete (or termination detection)
  foreach (msg,toPe) in msgLog {
    if (toPe in failedEndpoints) {
      // add on determinants for msg
      // resend msg to toPe
    }
  }
  // wait for all messages to be arrive on failed endpoints
  // sort messages for each endpoint based on msg.lClock, unless
       the msg has an non−zero CPI
  // call recvReplayMsg on each replay message
}

when recvReplayMsg(Message msg) {
  foreach d in (msg.determinants) {
    // check if dependency is fulfilled
    // if not add to buffer
    // else if not duplicate then execute msg
  }
  // check buffer for messages that were dependent on 'msg'
  // call recvReplayMsg on them
}
```

VIII. Experimental Results

As described in § V, we have taken several benchmarks written in Charm++ [27] and analyzed them with regard to our theory. Charm++ is an interesting target because it maps multiple objects to a processor, which we can treat as different endpoints. We found that all three of these benchmarks do not need any determinants, given their implementation (either the endpoints totally order the receptions or have commutative regions). We use these benchmarks to compare our protocol with the full causal protocol [29] implemented in Charm++, referred to as FullDetFT, and the in-memory checkpoint/restart scheme [31], which has been a topic of research and has been well optimized.

For all of the experiments (in both the PartialDetFT and FullDetFT protocols), we parallelized restart by redistributing the objects that were on the failed processor to 16 other processes in a round-robin fashion.

We performed all of our experiments on IBM Blue Gene/P "Intrepid", a 40960-node system, each node consisting of one quad-core 850MHz PowerPC 450 processor and 2GB DDR2 memory. Our codes were compiled with IBM XL C/C++ Advanced Edition for Blue Gene/P, V9.0.

Table I. Benchmarks and corresponding configurations used for experimental evaluation.

Benchmark	Configuration
LeanMD	600K atoms, 2-away XY, 75 atoms/cell
Stencil3D	matrix: 4096^3, chunk: 64^3
LULESH	matrix: $1024\text{x}512^2$, chunk: $16\text{x}8^2$

The benchmarks we evaluated are: LeanMD, a molecular dynamics simulation of the behavior of atoms using the Lennard-Jones potential similar to the miniMD application in the Mantevo benchmark suite; Stencil3D, a three-dimensional 7-point stencil that uses the Jacobi method; and LULESH, a shock hydrodynamics challenge problem defined by LLNL. The benchmark configurations are shown in Table I.

A. Forward Path

Figure 3 shows the percent overhead incurred during the forward path for PartialDetFT and FullDetFT compared to using no fault tolerance protocol. For the FullDetFT protocol, overhead on the forward path will be due to logging messages and creating and storing determinants stably. For all the benchmarks, PartialDetFT incurs under 5% overhead compared to FullDetFT, which incurs over 15% in some cases. The overhead of PartialDetFT is less because these applications do not require determinants, thereby eliminating the overhead due to determinants. The Stencil3D benchmark shows very little overhead for both protocols because the grain sizes are large and the number of messages are low compared to the amount of computation.

Figure 5 shows the expense of a coordinated checkpointing, which is incurred regardless of the fault tolerance protocol used. LeanMD has very low checkpointing cost because the amount of data is quite small to checkpoint and is bound by latency.

B. Recovery Times

Using Daly's formula and an MTBF per socket of 10 years, we estimated checkpoint periods for machines at exascale (shown in Table II). We used this estimation to derive an ostensible checkpoint period to test the recovery time for the various protocols. We injected a fault in the middle of the period and compared the execution time required for recovery of PartialDetFT to the checkpoint/restart mechanism in Figure 4(a). In Figure 4(b), we compare the recovery times between PartialDetFT and the FullDetFT protocol that uses full determinants. We obtain notable speedups compared to checkpoint/restart, and LeanMD shows the most benefit due to many objects per processor, enabling a faster parallel recovery. Compared to FullDetFT, our protocol scales much better during recovery and obtains speedups over checkpoint/restart at large scales.

IX. Concluding Remarks

We have presented a comprehensive approach for reasoning about execution orders and interleavings with the goal of minimizing the amount of information required to reproduce a distributed execution deterministically. Our work contains a novel observation about distributed programs, showing that the amount of information stored can be reduced in proportion to the knowledge of order flexibility in the programming paradigm, which often is present in modern languages and

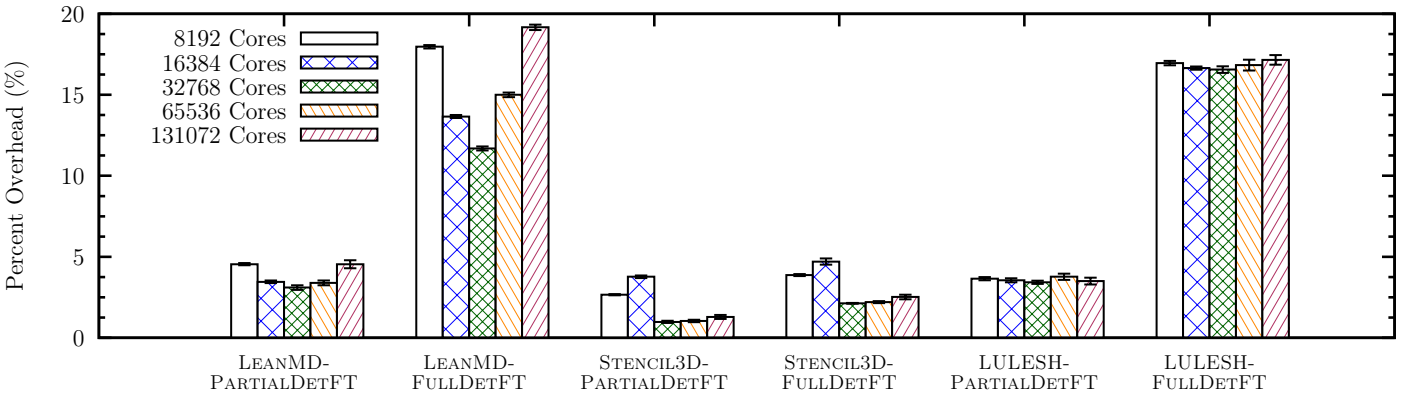
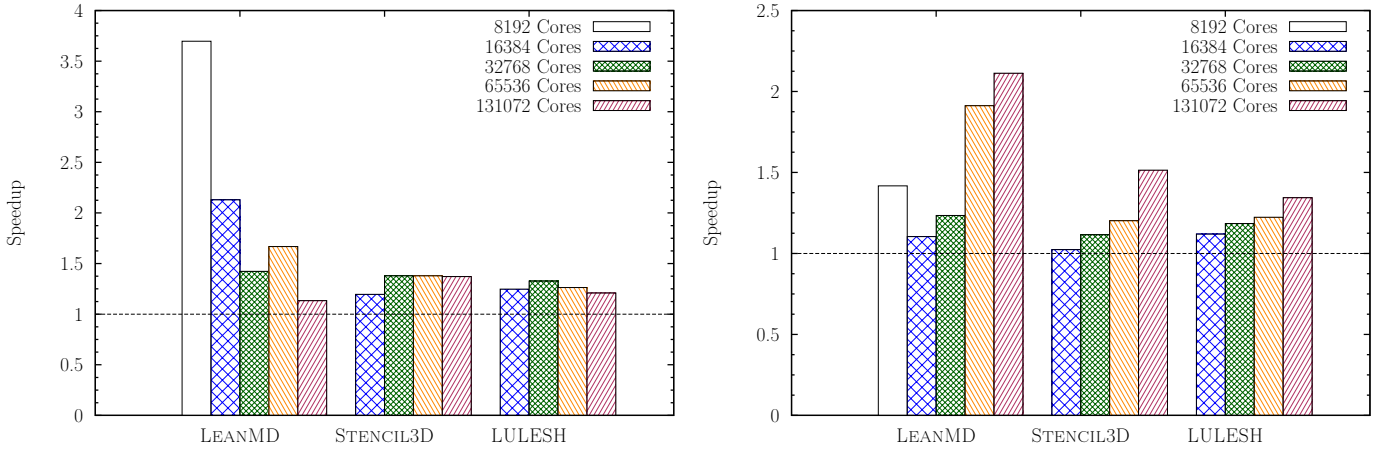


Fig. 3. Mean percent overhead during forward execution compared to execution without message logging, comparing the two protocols PARTIALDETFT and FULLDETFT. Each bar represents the mean of five runs. Error bars are standard error with a Student's T-test and a confidence of 90%.



(a) PARTIALDETFT recovery speedup relative to checkpoint/restart.

(b) PARTIALDETFT recovery speedup relative to FULLDETFT.

Fig. 4. The speedup during recovery that PARTIALDETFT achieves compared to checkpoint/restart and the full determinant scheme. We obtain this by injecting a failure in the middle of the checkpoint period, which is calculated according to Daly's formula.

Table II. Using Daly's formula for the optimal checkpoint period, we projected on socket counts from 64K to 1M the optimal checkpoint period for the three benchmarks. This checkpoint period was used in Figure 4 to determine a checkpointing period for injecting a failure.

Projected Sockets	Ckpt Time (ms)	Optimal τ (s)	Ckpt Period (steps)
LEANMD			
65536	59.9	23.9	90
131072	58.2	16.7	123
262144	57.4	11.7	168
524288	57.3	8.3	206
1048576	56.2	5.8	220
STENCIL3D			
65536	455	65.8	112
131072	223	32.6	109
262144	112	16.3	109
524288	57	8.2	110
1048576	30	4.3	115
LULESH			
65536	370	59.3	195
131072	187	29.8	194
262144	96	15.1	196
524288	49	7.6	197
1048576	26	3.9	198

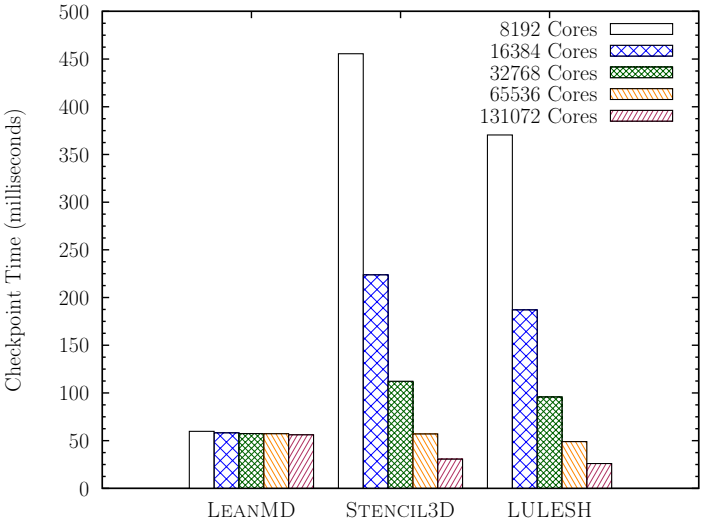


Fig. 5. Coordinated checkpoint time per checkpoint in milliseconds for each benchmark.

models. Using this observation combined with intrinsic ordering in codes, we can define an optimized replay protocol and show substantial advantages in the context of fault tolerance.

We believe that new languages and models should grow to encompass more semantic knowledge of order flexibility so it can be exploited by distributed algorithms whose performance relies on this characteristic. We also believe that making ordering and interleaving transparent and explicit in the paradigm leads to a clearer framework for reasoning about correctness and performance, along with reducing the complexity of distributed algorithms that rely on this key attribute.

ACKNOWLEDGMENTS

This paper is based on work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under contract number 63823. An award of computer time was provided by the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This research was supported in part by the US Government under grants DE-SC0001845, DE-SC0006706, NSF ITR 0205611, NSF OCI-0725070, and by a machine allocation on the Teragrid under award ASC050039N.

REFERENCES

[1] M. Snir, W. Gropp, and P. Kogge, "Exascale Research: Preparing for the Post Moore Era," https://www.ideals.illinois.edu/bitstream/handle/2142/25468/Exascale%20Research.pdf, 2011.

[2] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," www.cse.nd.edu/Reports/2008/TR-2008-13.pdf, 2008.

[3] E. N. Elnozahy, R. Bianchini, T. El-Ghazawi, A. Fox, F. Godfrey, A. Hoisie, K. McKinley, R. Melhem, J. S. Plank, P. Ranganathan and J. Simons, "System resilience at extreme scale," Defense Advanced Research Project Agency (DARPA), Tech. Rep., 2008.

[4] G. Zheng, L. Shi, and L. V. Kalé, "FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI," in *2004 IEEE Cluster*, San Diego, CA, September 2004, pp. 93–103.

[5] L. Wittie, "The bugnet distributed debugging system," in *Proceedings of the 2nd workshop on Making distributed systems work*, ser. EW 2. New York, NY, USA: ACM, 1986, doi: 10.1145/503956.504005.

[6] E. Leu, A. Schiper, and A. Zramdini, "Execution replay on distributed memory architectures," in *Parallel and Distributed Processing, Proceedings of the Second IEEE Symposium on*, 1990, pp. 106–112.

[7] R. Netzer and B. Miller, "Optimal tracing and replay for debugging message-passing parallel programs," in *Supercomputing '92., Proceedings*, 1992, pp. 502–511.

[8] C. Clémençon, J. Fritscher, M. J. Meehan, and R. Rühl, "An implementation of race detection and deterministic replay with MPI," 1995.

[9] C. J. Fidge, "Partial orders for parallel debugging," in *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed Debugging*. ACM, 1988, doi: 10.1145/68210.69233.

[10] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic MPI applications," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE Computer Society, 2011, doi: 10.1109/IPDPS.2011.95.

[11] W. Gropp and E. Lusk, "The MPI communication library: its design and a portable implementation," in *Proceedings of the Scalable Parallel Libraries Conference, 1993*. IEEE Computer Society Press, 1994.

[12] T. LeBlanc and J. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Transactions on Computers*, vol. C36, no. 4, 1987.

[13] B. P. Miller and J.-D. Choi, "A mechanism for efficient debugging of parallel programs," *SIGPLAN Not.*, vol. 23, no. 7, Jun. 1988, doi: 10.1145/960116.54004.

[14] E. Leu, A. Schiper, and A. W. Zramdini, "Efficient execution replay technique for distributed memory architectures," in *EDMCC*, ser. Lecture Notes in Computer Science, A. Bode, Ed., vol. 487. Springer, 1991.

[15] S. K. Damodaran-Kamal and J. M. Francioni, "Nondeterminancy: testing and debugging in message passing parallel programs," *SIGPLAN Not.*, vol. 28, no. 12, Dec. 1993, doi: 10.1145/174267.166789.

[16] D. Kranzlmüller, "Event graph analysis for debugging massively parallel programs," 2000.

[17] J. C. de Kergommeaux, M. Ronsse, and K. De Bosschere, "MPL: Efficient record/replay of nondeterministic features of message passing libraries," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 1999, pp. 141–148.

[18] J. Lifflander, P. Miller, and L. Kale, "Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection," in *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*, February 2013.

[19] W. Ma and S. Krishnamoorthy, "Data-driven fault tolerance for work stealing computations," in *Proceedings of the 26th ACM International Conference on Supercomputing*. ACM, 2012, doi: 10.1145/2304576.2304589.

[20] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2012, doi: 10.1145/2145816.2145845.

[21] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Steal tree: Low-overhead tracing of work stealing schedulers," ser. PLDI '13. New York, NY, USA: ACM, 2013, doi: 10.1145/2491956.2462193.

[22] J. Lifflander, S. Krishnamoorthy, and L. Kale, "Optimizing data locality for fork/join programs using constrained work stealing," ser. SC '14 (to appear), 2014.

[23] Leu and Schiper, "On the granularity of events when modeling program executions," *IEEE Symposium on Parallel and Distributed Processing*, 1993, doi: 10.1109/SPDP.1993.395502.

[24] X. Besseron and T. Gautier, "Optimised recovery with a coordinated checkpoint/rollback protocol for domain decomposition applications," in *Modelling, Computation and Optimization in Information Systems and Management Sciences*, ser. Communications in Computer and Information Science, H. Le Thi, P. Bouvry, and T. Pham Dinh, Eds. Springer Berlin Heidelberg, 2008, vol. 14, pp. 497–506.

[25] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[26] T. Hoefler, C. Siebert, and A. Lumsdaine, "Scalable communication protocols for dynamic sparse data exchange," *SIGPLAN Not.*, vol. 45, no. 5, Jan. 2010, doi: 10.1145/1837853.1693476.

[27] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[28] L. Alvisi and K. Marzullo, "Message logging: pessimistic, optimistic, and causal," *International Conference on Distributed Computing Systems*, pp. 229–236, 1995.

[29] E. Meneses, G. Bronevetsky, and L. V. Kale, "Evaluation of simple causal message logging for large-scale fault tolerant HPC systems," in *16th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems in 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011).*, May 2011.

[30] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello, "Improved message logging versus improved coordinated checkpointing for fault tolerant MPI," *IEEE International Conference on Cluster Computing*, 2004.

[31] G. Zheng, X. Ni, and L. V. Kale, "A Scalable Double In-memory Checkpoint and Restart Scheme towards Exascale," in *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.