Scalable Replay with Partial-Order Dependencies

Abstract

Replay of a parallel application is important for debugging and has recently gained traction for tolerating hard, fail-stop failures by logging messages and re-executing them. Previous work in replay algorithms often makes minimal assumptions about the programming model and application. However, parallel applications often have determinism intrinsic in the code and may have ordering constraints imposed by the execution model. In this paper, we elucidate the spectrum of determinism found in systems and thereby redefine what a determinant constitutes and when it must be created. In general, an execution of a code will partially specify a set of orders that are always followed, leading to only a *partial-order* that must be recorded to ensure deterministic replay. By exploiting this methodology in fault tolerance for several benchmarks, we present a scalable message-logging implementation with lower overhead compared to using full determinants that scales up to 131,072 cores on Intrepid BG/P.

Categories and Subject Descriptors D.4.5 [*Reliability*]: Fault-tolerance; C.4 [*Performance of Systems*]: Fault tolerance

Keywords replay, partial-order determinants, fault tolerance, message logging

1. Introduction

As we approach the exascale era, distributed algorithms must evolve to meet the increasing demands of large-scale application developement. Debugging parallel applications on large distributed-memory machines—now reaching to millions of cores—requires new methodologies that are highly scalable. In addition, machines at this scale may require fault tolerance; as the number of cores continues to increase, researchers have posited that failures may become the limiting factor [7, 11, 23].

For distributed-memory applications, replay is a technique often used to reproduce a particular execution schedule. In this context, replay can be used for a variety of applications, of which debugging and fault tolerance are the most prominent areas in use today. For debugging, an application's execution will be recorded in some manner (and saved to memory or disk), and then the execution will be replayed using the information recorded. In the seminal work in this area, two major methodologies emerged. The first was called *data-driven* replay (also called *content-driven*); in this formulation all the inputs to a process are recorded, and the re-execution uses these inputs in the same sequence to provide deterministic re-execution. The disadvantage of this approach is that the inputs may require a massive amount of storage. The second major methodology is *control-driven* replay: instead of recording the actual data, the control flow is recorded, leading to much faster initial execution [16]. However, the disadvantage of this approach is that all the processes that interacted must participate in the replay so the data is regenerated.

Replay is applied to the area of fault tolerance to speed up recovery from a checkpoint or to form a consistent uncoordinated checkpoint. In this context, messages are recorded by all processes (in the data-driven style) along with the control. The control is then saved in a *stable* manner; i.e. it is copied to k other processes, so that it is not lost due to a failure. Hence during replay (equivalent to recovery), the control can be recovered and because the data was also recorded by the non-failed processes, it can be used during replay so all the processes do not have to roll back to the checkpoint. Therefore, this approach, called *message logging*, combines the two approaches effectively.

The advantage of combining these two approaches is that all the processes do not have to re-execute during replay. Hence, the amount of resources required during replay may be much lower, making this an alternative that is also applicable for debugging on a very large parallel supercomputer. However, the protocol to handle this is more complex, because the data is being partially regenerated, and hence must be coherent with the stored control sequence. Recent work [25] has proposed combining these two approaches using a hybrid deterministic replay with subgroups in MPI. For today's large-scale systems, we posit that a combination of these two approaches is desirable.

This paper describes a new replay protocol that allows both data and control to be recorded, but primarily focuses on how to optimize the amount of control that must be stored, while keeping it cohesive with the data. The previous work in this area assumes that either all the control is recorded, or makes very specific assumptions about what control must be recorded based on the programming language. This paper formalizes various types of orderings and interleavings to provide a framework for minimizing the amount of control that must be recorded. We then present a novel algorithm for replaying code regions with are flexible in terms of execution order. Our protocol for optimizing the control that must be recorded considers the following:

- parallel applications that often have intrinsic order in the code that dictates that some communication operations will always happen in a deterministic order given a particular input;
- codes that often have *commutative* regions in the control flow: if a subset of nondeterministic communication operations are transposed the resultant states will be identical;
- execution models that may impose an order on the external inputs to a process, leading to a reduction in the number of possible input orderings and further reducing nondeterminism that must be recorded; and

- causality in the execution that may dictate that some operations always occur after others.

The synergy of these orders that are naturally established in codes can greatly reduce the amount of information that is required for a deterministic replay. In this paper, we combine these observations to develop a novel methodology for recording and replaying codes which are only partially non-deterministic, making the following contributions:

- we define in the CSP (communicating sequential processes) concurrency model how intrinsic order and two variants of commutativity can be modeled (§ 3);
- we define how a partial-order dependency from the CSP model maps to a determinant and prove its correctness (§ 4);
- we describe a fault-tolerance implementation using message-logging called PARTIALDETFT that uses partial-order determinants to store the control ordering (§ 5.3);
- we demonstrate the 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 (§ 6); and

2. Motivation and Background

We have observed that there is generally order in parallel programs; this order is often intrinsic and due to how the program is written. In general, writing a parallel program is difficult, so encoding ordering constrains in the code often makes reasoning about correctness easier. Algorithms and applications also may have commutative regions where a subset of the messages may have their order transposed and the resultant state is identical. However, as far as we know, there are no algorithms that exploit this execution order flexibility for replay or fault tolerance.

For the sake of replay, exploiting commutative regions to reduce storage may introduce pernicious behavior if the programmer (for example) 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 commutatively leads only to advantages, assuming it is correct.

2.1 Assumptions and Definitions

We define an *endpoint* in a distributed system to be a control unit that is scheduled by the system. Previous work in the realm considered processes to be endpoints, but our definition broadens it to a task, object, virtual process, etc. We define *control* as the environmental events that occur on the endpoint along with any nondeterministic choices it autonomously makes. In this paper, for the sake of reducing complexity in the descriptions, we will consider messages to be the only type of event.

We try to make minimal assumptions about how messages are sent or received or about how the execution model schedules work. We intend to provide a theory that is applicable to many communication and runtime models.

2.2 Related Work

The seminal research on distributed control-oriented replay assumed that all the control operations were stored; i.e. a total order on the control sequence was saved [13, 17, 19, 24]. The first work to diverge from this approach made the observation that in message-passing programs only the order of messages that race must be saved [21]. This paper concludes 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 a 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 [5]. Later work improved on this practically by defining *block races*, or sets of messages that can possibly race, and concluded that constructs such as `IProbe` in MPI need to be traced in some cases [4]. Further work elucidates the patterns of MPI primitives that introduce nondeterminism [12], and replay schemes have been devised for those constructs [6].

Recent work has revived this in the context of message-logging for tolerating hard failures making the observation that some MPI program are *send deterministic* [8]. 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. In this work, if a program is send deterministic, the paper concludes that no determinants are needed. Compared to this paper, it is specific to MPI, and does not provide any facility when the program is not send deterministic.

Other work has theoretically discussed the granularity of events when modeling program executions [15], which is related to how we model commutative regions (by collapsing transition states/events from the environment's perspective). In the paper's conclusion, they mention that this may be useful for optimizing replay algorithms.

All previous work in this area has made the assumption that even if some determinants are omitted due to intrinsic determinism, during replay each process will always execute the same sequence of events. However, this paper diverges from this notion by defining an endpoint more generally (logical endpoints on the same physical process may interleave differently) and 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.

3. Theory: Defining Order in CSP

We begin with a gentle introduction to the CSP (communicating sequential processes) model for the sake of accessibility [22]. Firstly, the "sequential process" that we are modeling is the sequence of communications an *endpoint* participates in (not a physical process). The fundamental assumptions in CSP are that communication are instantaneous: although communications may be asynchronous in practice the model assumes that they occur instantly when the endpoint and environment agree on them. Once agreement is reached, the communication is obligated to occur according to the model. Intuitively, the agreement in message passing can be viewed as an endpoint being ready to receive and another endpoint sending a matching message.

Because we are using CSP to model the communication from the viewpoint of a single endpoint, the lack of asynchrony in the model does not inhibit us from modeling communications that may be overlapped. For each possible state of an endpoint, we can model the various possibilities given the control flow logic.

We define the environment as any input that can possibly change the state of the endpoint. In an asynchronous model, the potential non-determinacy on the network acts as one of the possible inputs. An endpoint transitions to a new state when some input becomes available, or when the endpoint makes a unilateral *non-deterministic decision*. We distinguish between a *non-deterministic input* (the result of a network race, for instance) and a decision the process makes that is non-deterministic: for example, using a random number generator to determine the next state. We use the term *event* synonymously with the term *input*.

The fundamental operator in CSP is the prefix operator, which denotes that an endpoint is willing to communicate the event a and will wait indefinitely for a to occur. We use lowercase letters to denote events, and uppercase to denote states. After the event a is communicated, the endpoint behaves as P:

$$a \rightarrow P$$

We can also define sequences of communications as follows:

$$a \rightarrow b \rightarrow P$$

We define a *non-deterministic input* to an endpoint using the following notation:

$$a \rightarrow P \,\Box\, b \rightarrow P'$$

In this case, the endpoint will make a deterministic decision between accepting a or b depending on the non-deterministic environmental input and subsequently will become P or P' depending on the input. An example of this would be the following message passing program (without any assumption on order between endpoints):

```
if (endpoint == 0):
  send(a, endpoint(1));
  send(b, endpoint(1));
else if (endpoint == 1):
  Message m = recv(endpoint(0));
  if (m == a) // behave as P
  else if (m == b) // behave as P'
```

We define a *non-deterministic decision* an endpoint makes unilaterally as follows:

$$a \rightarrow P \;\sqcap\; b \rightarrow P'$$

In this case, the endpoint does not give the environment a choice, instead it selects arbitrarily between the events. Theoretically, a non-deterministic input is equivalent to a non-deterministic decision if the two choices are the identical: $a \rightarrow P \,\Box\, a \rightarrow P' \equiv a \rightarrow P \;\sqcap\; a \rightarrow P'$.

We can define an endpoint P as hiding an event x from its environment, by denoting $P \setminus \{x\}$. In this case, P has internalized x and x will not be externally visible.

With this (a very minimal introduction to CSP), we now extend the CSP model to aid us in defining our partial-order theory.

3.1 Equality

We use event equality in CSP to define whether events are considered as identical by the encapsulating system. Hence, we will not define a universal equivalence relation, since 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.

For the sake of concreteness, 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:

$$\begin{aligned}(e_1 = e_2) \;\equiv\; & (e_1.source = e_2.source \wedge \\ & (e_1.tag = e2.tag \vee \\ & e_1.tag = \text{ANY} \vee e_2.tag = \text{ANY}))\end{aligned}$$

Interleavings	**Dependencies**	**Description**
Ordered (n)	0	Section 3.3
Unordered (n)	$n-1$	Section 3.4
Ordered (n) and Ordered (m)	$\min(m,n)$	Lemma 3.1
Unordered (n) and Ordered (m)	n	Lemma 3.5
Unordered (n) and Unordered (m)	$m+n$	Section 3.4
Commutative (n)	0	Section 3.6
n Ordered events $(X_1, X_2, \ldots, X_n)$	$\sum_{i=1}^{n} \lvert X_i\rvert - max_i \lvert X_i\rvert$	Lemma 3.3

Table 1. Number of dependencies required to order an interleaving of sets of events

3.2 Dependencies

When we have events that need to be ordered due to some non-determinisim, we use the term *dependency* to mean the specification of an order between them. In this section we will abstractly consider the number of dependencies we need for various types of CSPs. In section 4, when we describe the actual replay protocol, we will specify exactly the information that the dependency is required to have, which is dependent on the type of dependency (specifically whether it is in a commutative region or not).

3.3 Ordered, Deterministic Inputs

An endpoint may allow a set of n possible inputs, which are all ordered based on the control flow of the code:

$$a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow P_n$$

We say in this case that all the event dependencies are implicit in the code, hence no dependencies are required for the endpoint to realize state P_n.

3.3.1 Interleaving n Ordered Inputs

We now consider the problem of multiple ordered inputs that may interleave arbitrarily. An example of this would be several parallel modules interacting, which by themselves are ordered, but may interleave non-deterministically.

We first consider the interleaving of two possible ordered sets of inputs:

Lemma 3.1. *Any possible ordering of two ordered sets of events A of size m and B of size n can be represented with* $\min(m,n)$ *dependencies*

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 n dependencies. □

Definition 3.2. Logical Event: *In the two ordered set of events discussed in lemma 3.1, 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 since there can only be one event directly preceding b_j. This reduces the interleaving of the two ordered set of events*

A, B described in the earlier lemma into an ordered event of A' with $\max(m, n)$ events and $\min(m, n)$ dependencies.

Lemma 3.3. *Any possible ordering of n ordered set of events $X_1, X_2, \ldots, X_n$ can be represented with $(\sum_{i=1}^{n} |X_i| - max_i|X_i|)$ dependencies.*

Proof. Base case 1: The number of dependencies required to represent any interleaving of two ordered set X_1 and X_2 of events is $\min(|X_1|, |X_2|)$ (from lemma 3.1). This is equivalent to $(|X_1| + |X_2| - max(|X_1|, |X_2|))$. This results in an ordered set with $\max(|X_1|, |X_2|)$ events including logical events.

$$\min(x, y) = x + y - \max(x, y)$$

We now assume it holds for $n = k$, show it holds for $k + 1$. Since, this holds for $n = k$, it means that the maximum number of dependencies required is $(\sum_{i=1}^{k} |X_i| - \max_{i=1}^{k} |X_i|)$. This results in an ordered event with $\max_{i=1}^{k} |X_i|$ events. For this, there are two possible cases when we consider $k + 1$ ordered set.

- Case 1: When the number of events in X_{k+1} is less than the events in the k ordered set i.e. $|X_{k+1}| < \max_{i=1}^{k} |X_i|$. The number of dependencies required to interleave the $k + 1$ ordered set with ordered set formed after interleaving k sets is $\min(|X_{k+1}|, \max_{i=1}^{k} |X_i|)$. The total number of dependencies required is

$$\begin{aligned} D &= \sum_{i=1}^{k} |X_i| - \max_{i=1}^{k} |X_i| + |X_{k+1}| \\ &= \sum_{i=1}^{k+1} |X_i| - \max_{i=1}^{k} |X_i| \\ &= \sum_{i=1}^{k+1} |X_i| - \max_{i=1}^{k+1} |X_i| \end{aligned}$$

- Case 2: When the number of events in X_{k+1} is greater than the number of events formed by the interleaving of the previous k sets i.e. $|X_{k+1}| > \max_{i=1}^{k} |X_i|$. The number of dependencies required is then $\min(|X_{k+1}|, \max_{i=1}^{k} |X_i|)$ The total number of dependencies required is

$$\begin{aligned} D &= \sum_{i=1}^{k} |X_i| - \max_{i=1}^{k} |X_i| + \max_{i=1}^{k} |X_i| \\ &= \sum_{i=1}^{k} |X_i| \\ &= \sum_{i=1}^{k} |X_i| + |X_{k+1}| - |X_{k+1}| \\ &= \sum_{i=1}^{k+1} |X_i| - \max_{i=1}^{k+1} |X_i| \end{aligned}$$

□

3.4 Unordered, Non-deterministic Inputs

We define a common relation where we have multiple non-deterministic inputs that *all* occur but in any order:

Definition 3.4. *The relation $\boxdot$ represents an unordered set of events that all occur:*

$$\begin{aligned} ((a_1 \rightarrow P(\langle a_1, \vec{v_1} \rangle)) \boxdot \ldots \boxdot (a_n \rightarrow P(\langle a_n, \vec{v_n} \rangle))) &\equiv \\ V &= \{a_1, \ldots, a_n\} \\ P_{\text{start}} &= (v_x \in V) \rightarrow P(\{v_x\}) \\ P(Y) &= (v_x \in (V \setminus Y)) \rightarrow P(V \setminus (Y \cup \{v_x\})) \end{aligned}$$

We have a set V of events that will all occur eventually in some order. We start in the state were no events have occured (P_{start}), and the set V are possible non-deterministic inputs. Each time an event is selected from the set V, we remove that as a possibility, until we reach the final state. We use the vector $\vec{v_i}$ to represent the sequence of events that occured after the first event a_i.

This relation can represent a common communication pattern found in codes where several messages are sent to an endpoint, which will all be eventually received. However, depending on which message arrives first, the eventual state will be different. The $\boxdot$ relation also represents one of the most common source of nondeterminism found in parallel applications.

We use the following shorthand to denote the same as above. Consider a finite set of n unordered events:

$$\boxdot_{i=1}^{n} a_i \rightarrow P_i(\langle a_i, \vec{v_i} \rangle)$$

Clearly, there are $n!$ possible resultant states after the endpoint transitions through all the immediate states. After n unordered events occur, we can represent one possible outcome with $n - 1$ dependencies, which establishes a total order on the n events. To store this state it will require $\log_2 n!$ bits.

We now show how an unordered set of events of size n can be interleaved with a non-intersecting arbitrary ordered set of events with only n dependencies required in the worst case.

Lemma 3.5. *Any possible ordering of a unordered set of events U of size n interleaved with an ordered set of events O of size m can be represented with n dependencies, if $U \cap O = \emptyset$.*

Proof. Base case: We have the the following sequence formed from the ordered set O: $o_1 \rightarrow \ldots \rightarrow o_m \rightarrow P_m$. Of course, this order needs no dependencies because the elements are by definition ordered. When $n = 1$, U has exactly one event u_0. In this case, u_0 can execute before or after any one of the events in O and we can clearly represent this order with one dependency.

We now assume it holds for $n = k$, show it holds for $k + 1$.

$$\boxdot_{i=1}^{k} a_i \rightarrow P(\langle a_i, \vec{v_i} \rangle) \boxdot a_{k+1} \rightarrow P(\langle a_{k+1}, \vec{v_{k+1}} \rangle)$$

If this holds for k, then we need k dependencies to order $\boxdot_{i=1}^{k} a_i \rightarrow P(\langle a_i, \vec{v_i} \rangle)$. Out of the ordered set, there are two possible cases as we observe the $k + 1$ event.

- The first case is where we have already observed all the events in O. That is, the unordered $k + 1$ event occurs after all the ordered events. $P(\vec{v_{k+m}})$ will be the state and includes all the transitions through O and all but one element of U. Hence with one dependency, we can represent that u_{k+1} happens after the state $P(\vec{v_{k+m}})$.
- Otherwise, there may be x states left to transition through O, hence the last state is $P(\vec{v_{k+m-x}})$. We can arbitrarily split the x states into two ordered subsets S_1 and S_2, where S_1 happens before S_2. Both sets will have a final state, which we will respectively call s_1 and s_2. The order we must represent is:

$$\begin{aligned} P(\vec{v_{k+m-x}}) &= s_1 \rightarrow P(\vec{v_{k+m-x+||s_1||}}) \\ P(\vec{v_{k+m-x+||s_1||}}) &= u_m \rightarrow s_2 \end{aligned}$$

This ordering can be represented with exactly one dependency by making u_m dependent on s_1. By doing this, we define the splitting point between s_1 and s_2, defining exactly where u_m happens in the sequence.

□

We could discuss how two possible orders interact if $U \cap O \neq \emptyset$. However, we deem this unnecessary because when analyzing a endpoint, we should consider all inputs at a given state. Hence, orders that are not distinct are not well-defined in our formulation.

3.5 Unordered, Non-deterministic Decisions

We now discuss the non-determinism that arises when a endpoint makes a unilateral non-deterministic decision that does not arise from the environment. If an endpoint selects between n choices, if any of those choices are non-distinct, then according to the model the endpoint selects arbitrarily between them. Any example is $a \rightarrow P \sqcap a \rightarrow P'$, which is exemplified in the following message-passing program with no ordering on messages.

```
if (endpoint == 0):
  send(a, endpoint(1));
  send(a, endpoint(1));
else if (endpoint == 1):
  Message m = recv(endpoint(0));
  // the process may now behave as P or P', due to the
       contents of 'm'
```

Clearly, if an endpoint selects between n choices non-deterministically, we must have enough information make the same choice that the endpoint made. Hence, the above sends must be distinguished and used to build a single dependency for that choice.

If an endpoint selects between n choices non-deterministically, but will always select all of them eventually, the proofs in section 3.4 apply, the only difference is a practical one, where the inputs must be differentiated.

3.6 Unordered, Non-deterministic Commutative Inputs

Often codes have the pattern where messages may race, but due to the way the code is written, any order that is executed will lead to the same resultant state. For instance, if we have a code where each message updates a distinct section of the endpoint's local data, the order the messages are executed will not matter. We can of course have more complex cases, which may access the same data, but are commutative in their operations. From the perspective of replay, there is non-determinism due to the race (and could be modeled by the $\boxdot$ relation); however, because the outcomes are symmetric the race is practically irrelevant. The following is a trivial example of commutativity:

```
int data[2];
if (endpoint == 0):
  send(a1, endpoint(1));
  send(a2, endpoint(1));
else if (endpoint == 1):
  count = 0
  fetch:
    Message m = recv(endpoint(0));
    count++;
    if (m == a1) data[0] += 1;
    else if (m == a2) data[1] += 1;
    if (count == 1): goto fetch;
    else: goto finished;
  finished:
    // resultant state is identical: (a1->a2)==(a2->a1)
```

We now define a new relation that represents a pair of non-deterministic inputs that after both are executed lead to the same resultant state. Note, that this assumes that a_1 and a_2 are non-equivalent events, although it does not seem to matter in the above example. We call this type of commutativity *weak* because it only allows non-equivalent events to commute. If there are equivalent events we assume they are not part of the region given this definition:

Definition 3.6. *The relation $\boxminus$ represents an unordered set of events that are* **weak commutative***:*

$$\begin{aligned}(a_1 \rightarrow P_1 \boxminus \ldots \boxminus a_n \rightarrow P_n) \;\; &\equiv \\ (a_1 \rightarrow P(\langle a_1, \vec{v_1}\rangle) \boxdot \ldots \boxdot a_n &\rightarrow P(\langle a_n, \vec{v_n}\rangle) \wedge \\ P(\langle a_1, \vec{v_1}\rangle) &= \ldots = P(\langle a_n, \vec{v_n}\rangle))\end{aligned}$$

If we have a set of n events that weakly commute, we have exactly one resulting state, but $(n-1)!$ transition states, which are distinct. We prove in lemma 3.8 that these transition states will be observed as exactly one state to the environment. (Any externally visible change due to a different transitional path, leads to a contradiction with the definition of commutativity.)

We now define a stronger form of commutativity allows for non-deterministic decisions to also commute:

Definition 3.7. *The relation $\boxplus$ represents an unordered set of events that are* ***strong commutative****:*

$$\begin{aligned}(a_1 \rightarrow P_1 \boxplus \ldots \boxplus a_n \rightarrow P_n) \;\; &\equiv \\ ((a_1 \rightarrow P_1 \boxminus \ldots \boxminus a_n \rightarrow P_n) \wedge a_1 &\overset{?}{=} \ldots \overset{?}{=} a_n)\end{aligned}$$

With this relation the possible outcomes are not merely an event ordering, but also how the system internally (and unilaterally) decides to "match" them. So for $a \rightarrow P_1 \boxplus a \rightarrow P_2$, there are actually four possible outcomes (two of them based on the non-deterministic input), which we are declaring to be identical ($P_1 = P_2$). Consider this example program:

```
if (endpoint == 0):
  send(a', endpoint(1));
  send(a'', endpoint(1));
else if (endpoint == 1):
  Message m = recv(endpoint(0));
  // work1: some work involving m (which may be the data
       in a' or a'')
  Message m' = recv(endpoint(0));
  // work2: some work involving m' (which is the second
       message)
```

In this example, we are assuming that $a' = a''$, but just because the equivalence relation does not distinguish between the two, the messages are distinct and may carry different data. Hence, the $\boxplus$ relation is stating that not only can be execute work1 and work2 in any order, but also that the non-deterministic decision that the endpoint makes may have a different outcome in replay. More specifically in this case, the first recv may return a different message than during the record phase.

We now prove that if a set of inputs are commutative, regardless of the order of execution, the environment can only distinguish between two different states:

Lemma 3.8. *An weak or strongly unordered commutative region,*

$$\left(\underset{i=1}{\overset{n}{\boxplus}} a_i \rightarrow P_i\right) \vee \left(\underset{i=1}{\overset{n}{\boxminus}} a_i \rightarrow P_i\right)$$

with n events has exactly two states (P_{sc}, P_{fc}) from the environment's perspective, where,

$$\begin{aligned}P_{sc} =&(a_1 \rightarrow P_{sc} \sqcap a_1 \rightarrow P_{fc})\Box\\ &(a_2 \rightarrow P_{sc} \sqcap a_2 \rightarrow P_{fc})\Box \dots \Box\\ &(a_n \rightarrow P_{sc} \sqcap a_n \rightarrow P_{fc})\end{aligned}$$

Proof. We prove this by contradiction. If this is not true, we can select some arbitrary state $P_x(\langle \vec{v_x} \rangle)$ which is not equal to final state P_{fc} and hence must be a transition state after the vector $\vec{v_x}$ is observed. If this is true, then the environment can observe $P_x(\langle \vec{v_x} \rangle)$ and communicate an arbitrary event z when it observes $P_x(\langle \vec{v_x} \rangle)$. It is possible that $P_x(\langle \vec{v_x} \rangle)$ is never reached in an alternative execution because by the definition of commutativity $\vec{v_x}$ may be ordered differently. Hence, it is possible that z is not communicated in this execution. This is a contradiction because it violates the definition of both strong and weak commutativity, which say that the final states must be identical.

Given this, if we have a strong- or weak-commutative region, we can say that the endpoint hides all transition states, since we just showed that any communication produced by the endpoint should be identical to the communication produced if that state was reached via another transition sequence. □

We have devolved a commutative region into two states from the environment's perspective, which is important because it shows that we need no information proportional to the size of the commutative region to rebuild that state from the environment's perspective.

4. PO-REPLAY: Partial-Order Replay Algorithm

The theoretical bounds on dependencies for different types of orders in a distributed application, expressed in section 3, 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 [3] and replays messages in causal order, except for commutative regions;
- it uniquely identifies a sent message regardless of whether the order of message receptions is transposed; and,
- it requires exactly the number of *determinants* as dependencies as specified in section 3.

A determinant is a piece of information stored to maintain the order in a given snapshot 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*: 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 process-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 messages have actually arrived to their destination. Hence, during replay on a subset of the endpoints, all messages are re-sent and the ones that arrived in forward execution are simply ignored. If replaying an application on a subset of the processors, in a similar fashion as message logging, this same problem may arise. Hence, our algorithm ensures that every sent message in the system is uniquely identified globally.

To identify messages regardless of replay transposition, we mark messages based on their path to a causal ancestor within a transitive commutative region. In this way, the identification system is not based unilaterally on the endpoint's current state, but the message that caused the message to execute. Due to the conclusions of section 3, we know that inside a commutative region other messages that do not commutate are not allowed.

The extra assumptions we make for our unqiue identification scheme is 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 causally dependent on the message received).

Given these assumptions, we now describe how a commutative message is marked to uniquely identified it so it can be re-executed in any order. For this purpose, a message is assigned a number that represents its path from the 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 4.1. *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 a process local sequence number that is incremented for every send outside a commutative region, and incremented once when a commutative region starts;*

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

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

Proof. We first prove that a commutative region, composed of messages C is always isolated by contradiction. If this is not true then there could be a non-commutative message arriving while a commutative region is executing. If this message m_x is non-commutative, then it depends on some message $m_y \in C$. This contradicts lemma 3.8 because now the transition states could be different depending on the interleaving of m_x and m_y.

We can now consider two cases:

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.

Commutative region we have already shown it is ordered with respect to the non-commutative region and cannot be overlapped with the non-commutative region; hence we now argue 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. The number of messages that will target the starting commutative region (the root) is known, because the size of the region is assumed to be known. Hence, the CPI includes the set of bits that identifies which message targeted it along with bits that uniquely represent each message that is sent. The number of bits needed for this is known because we assume a bound on the number of sent messages. For each message received past the root, we augment a unique bit pattern to the CPI for each sent message. Hence, the CPI for each message inside a commutative region will represent a unique path back to the root, which is idempotent of the order in which it was executed.

□

5. PARTIALDETFT: Fault Tolerance with PO-REPLAY

5.1 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 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 (i.e. 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. This could be implemented using a heartbeat mechanism built into the runtime that pings processes at a fixed interval to determine if they are responsive. Failure detection is not the focus of this paper, so we assume an adequate mechanism.

5.2 Background/Related Work

5.2.1 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 [2, 9, 20]. Checkpoint/restart has many variants which depend on the amount of data that included in the checkpoint.

5.2.2 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 [1], but we focus on the causal, pessimistic protocols, which have been shown to perform well [14, 18].

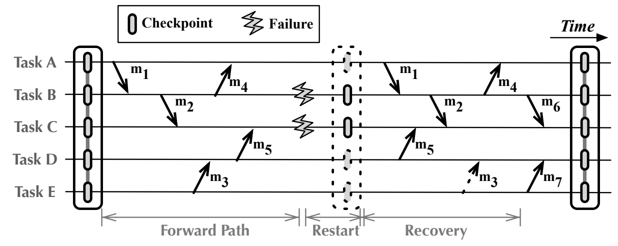


Figure 1. Execution through failures with rollback-recovery techniques. Dotted elements appear only in checkpoint/restart, but not in message-logging.

Sender-based pessimistic message logging works by each process logging all the messages that it sends to another process past the checkpoint (after this the sent messages 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. All the 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.

In order 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: essentially 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 a message is allowed to be sent, or they are augmented onto any sent messages until an acknowledgment is received that they are saved in enough places, given the reliability requirements of the system.

Figure 1 shows an example of an execution with 5 tasks with a rollback-recovery mechanism. The ***forward path*** is the portion of the execution that has 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, message m_5 may be received in a different order after the failure. Message-logging guarantees by using determinants 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.

5.3 PARTIALDETFT Algorithm

We now define the PARTIALDETFT algorithm; a general replay algorithm is essentially a subset. We use the following data structure to store the local state for each process and for each determinant.

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

```
  list<Message> msgLog;
}
```

The local state of each process is initialized as follows:

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

  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 unack'ed determinants
  msg.augmentDets(unackedDets);
  SPE = myEndpointID();
  CPI = parCPI;

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

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

Now we show the algorithm for replay, after a failure.

```
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
    }
  }
}

when recvReplayMsg(Message msg) {
  foreach d in (msg.determinants) {
    // check if dependecy 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
}
```

Benchmark	Configuration
LEANMD	600K atoms, 2-away XY, 75 atoms/cell
STENCIL3D	matrix: 4096^3, chunk: 64^3
LULESH	matrix: 1024 x 512^2, chunk: 16 by 8^2

Table 2. Benchmarks and corresponding configurations used to evaluate our protocol.

6. Empirical Results

We have taken several benchmarks written in Charm++ [10] and analyzed them with regard to our theory. We found that all three of these benchmarks do not need any determinants, given their implementation (either the benchmarks totally order the receptions or have commutative regions). We use these benchmarks to compare our protocol with the full causal protocol implemented in Charm++ and the default checkpoint/restart scheme, which has been a topic of research and has been well optimized.

For all the experiments we parallelized restart by redistributing the objects that were on the failed processor to several other processors. For all the experiments, we distributed the objects in round-robin fashion over 16 different processors.

We performed all our experiments on Argonne's 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. All our codes used the latest version of the Charm++ runtime system.

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 2.

Figure 2 shows the percent overhead incurred during the forward path for PARTIALDETFT and FULLDETFT compared to using no fault tolerance protocol. The overhead on the forward path will be due to logging messages and creating and storing determinants stably (for the FULLDETFT protocol). For all the benchmarks, PARTIALDETFT incurs under 5% overhead compared to FULLDETFT which incurs over 15% in some cases. 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 3 shows the expense of a coordinated checkpointing, which is a incurred regardless of the fault tolerance protocol that is used. LEANMD has very low checkpointing cost because the data size is very small compared to the amount of computation.

Table 3 shows the optimal checkpointing period assuming an exascale machine with socket counts ranging from 64K to 1M. We use this to derive a reasonable checkpoint period to test the recovery time. Using this checkpointing period, we injected a fault in the middle of the period and compared the execution time during recovery of PARTIALDETFT to the checkpoint/restart

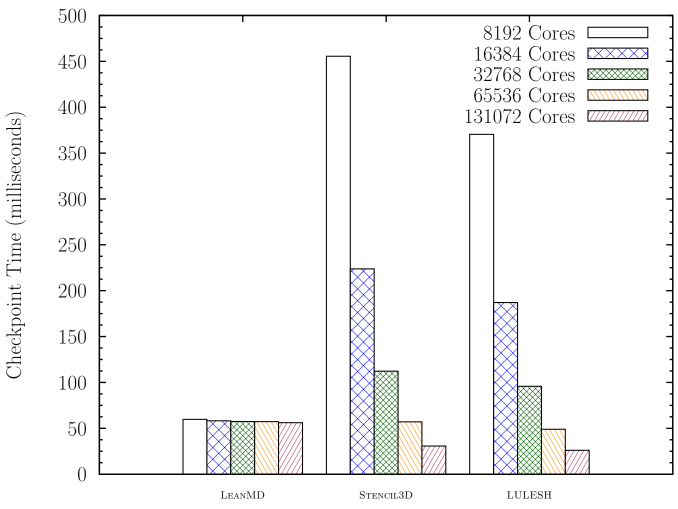


Figure 3. Coordinated checkpoint time in milliseconds for three benchmarks.

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
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
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

Table 3. 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 the below graph.

mechanism (figure 4(a)) and the PartialDetFT protocol (figure 4(b)). We observe that we obtain speedups compared to checkpoint/restart, LeanMD showing the most benefit due to many objects per processor, which enables a faster parallel recovery. Compared to FullDetFT, our protocol scales much better and obtains speedups over checkpoint/restart at large scales.

7. Concluding Remarks

References

[1] L. Alvisi and K. Marzullo. Message logging: pessimistic, optimistic, and causal. *Distributed Computing Systems, International Conference on*, 0:0229, 1995. .

[2] L. Bautista-Gomez, D. Komatitsch, N. Maruyama, S. Tsuboi, F. Cappello, and S. Matsuoka. FTI: High performance fault tolerance interface for hybrid systems. In *Supercomputing*, pages 1 –12, Nov. 2011.

[3] K. M. Chandy and L. Lamport. Distributed snapshots : Determining global states of distributed systems. *ACM Transactions on Computer Systems*, Feb. 1985.

[4] C. ClÃľmenÃğon, J. Fritscher, M. J. Meehan, and R. RÃijhl. An implementation of race detection and deterministic replay with mpi, 1995.

[5] S. K. Damodaran-Kamal and J. M. Francioni. Nondeterminancy: testing and debugging in message passing parallel programs. *SIGPLAN Not.*, 28(12):118–128, Dec. 1993. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/174267.166789`.

[6] 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*, pages 141–148. Springer, 1999.

[7] 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.

[8] 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*, IPDPS '11, pages 989–1000, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4385-7. . URL `http://dx.doi.org/10.1109/IPDPS.2011.95`.

[9] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *SciDAC*, 2006.

[10] L. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[11] 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, 2008.

[12] D. Kranzlmüller. Event graph analysis for debugging massively parallel programs. 2000.

[13] T. LeBlanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *Computers, IEEE Transactions on*, C-36(4):471–482, 1987. ISSN 0018-9340. .

[14] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. *Cluster Computing, IEEE International Conference on*, 0:115–124, 2004. .

[15] Leu and Schiper. On the granularity of events when modeling program executions. *Parallel and Distributed Processing, IEEE Symposium on*, 0:422–429, 1993. .

[16] 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*, pages 106–112, 1990. .

[17] E. Leu, A. Schiper, and A. W. Zramdini. Efficient execution replay technique for distributed memory architectures. In A. Bode, editor, *EDMCC*, volume 487 of *Lecture Notes in Computer Science*, pages 315–324. Springer, 1991. ISBN 3-540-53951-4. URL `http://dblp.uni-trier.de/db/conf/edmcc/edmcc1991.html#LeuSZ91`.

[18] 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.

[19] B. P. Miller and J.-D. Choi. A mechanism for efficient debugging of parallel programs. *SIGPLAN Not.*, 23(7):135–144, June 1988. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/960116.54004`.

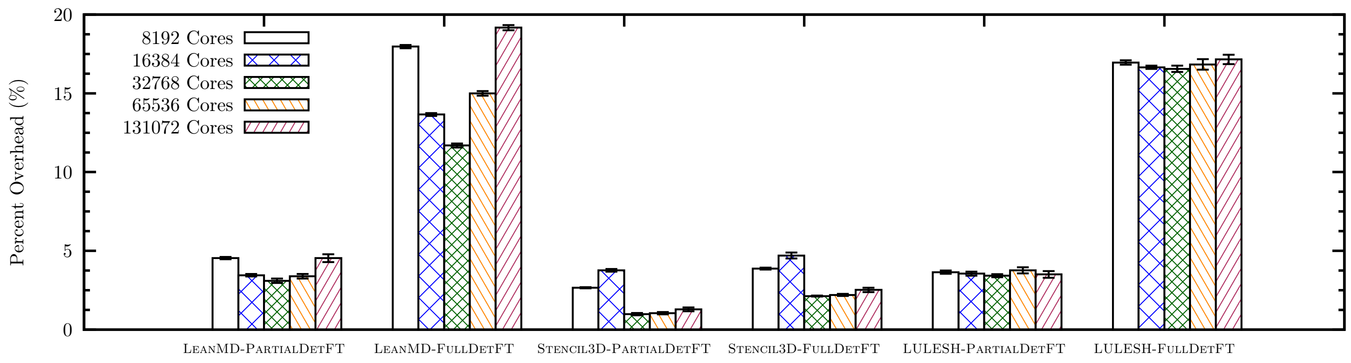
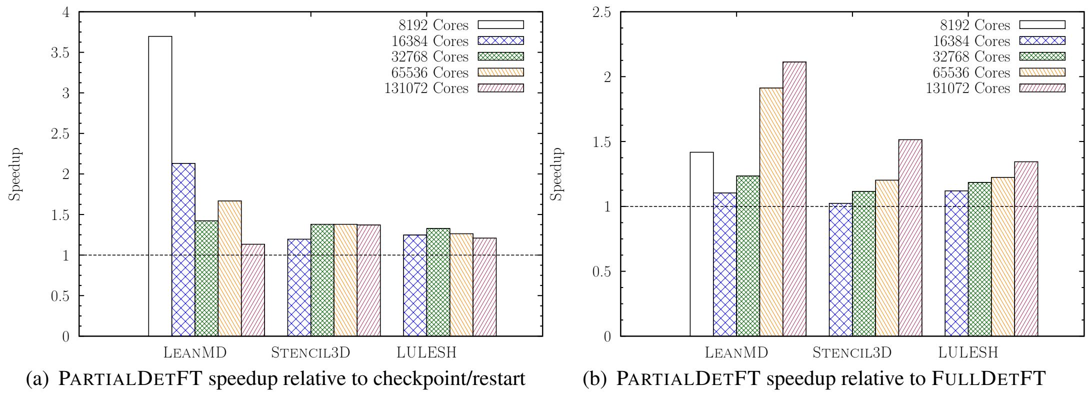


Figure 2. Mean percent overhead during forward execution compared to execution without message logging, comparing the two protocols PartialDetFT and FullDetFT. Each bar represents the mean of five runs; error bars are standard error with a Student's T-test with a confidence of 90%.



(a) PartialDetFT speedup relative to checkpoint/restart

(b) PartialDetFT speedup relative to FullDetFT

Figure 4. The speedup during recovery that PartialDetFT achieves compared to checkpoint/restart and the full determinant scheme. We obtain this by injecting a failure randomly in the middle of the checkpoint period, which is calculated according to Table 3.

[20] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC*, pages 1–11, 2010.

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

[22] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. ISBN 0136744095.

[23] 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.

[24] L. Wittie. The bugnet distributed debugging system. In *Proceedings of the 2nd workshop on Making distributed systems work*, EW 2, pages 1–3, New York, NY, USA, 1986. ACM. . URL `http://doi.acm.org/10.1145/503956.504005`.

[25] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. M. Voelker. MPIWiz: Subgroup reproducible replay of MPI applications. In *In PPoPP*, 2009.