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

Jonathan Lifflander*, Esteban Meneses†, Harshitha Menon*,
Phil Miller*, Sriram Krishnamoorthy‡, Laxmikant V. Kale*

jliffl2@illinois.edu, emeneses@pitt.edu, {gplkrsh2,mille121}@illinois.edu,
sriram@pnnl.gov, kale@illinois.edu

*University of Illinois Urbana-Champaign (UIUC)
†University of Pittsburgh
‡Pacific Northwest National Laboratory (PNNL)

September 23, 2014

Deterministic Replay & Fault Tolerance

- Fault tolerance often crosses over into replay territory!

Deterministic Replay & Fault Tolerance

- Fault tolerance often crosses over into replay territory!
- Popular uses
  - Online fault tolerance
  - Parallel debugging
  - Reproducing results

Deterministic Replay & Fault Tolerance

- Fault tolerance often crosses over into replay territory!
- Popular uses
  - Online fault tolerance
  - Parallel debugging
  - Reproducing results
- Types of replay
  - Data-driven replay
    - Application/system data is recorded
    - Content of messages sent/received, etc.
  - Control-driven replay
    - The ordering of events is recorded

Deterministic Replay & Fault Tolerance

→ Our Focus

- Fault tolerance often crosses over into replay territory!
- Popular uses
  - **Online fault tolerance**
  - Parallel debugging
  - Reproducing results
- Types of replay
  - Data-driven replay
    - Application/system data is recorded
    - Content of messages sent/received, etc.
  - **Control-driven replay**
    - The ordering of events is recorded

Online Fault Tolerance

→ Hard failures

- Researchers have predicted that hard faults will increase

Online Fault Tolerance

→ Hard failures

- Researchers have predicted that hard faults will increase
  - Exascale!

Online Fault Tolerance

$\rightarrow$ **Hard failures**

- Researchers have predicted that hard faults will increase
  - Exascale!
  - Machines are getting larger

Online Fault Tolerance

→ Hard failures

- Researchers have predicted that hard faults will increase
  - Exascale!
  - Machines are getting larger
  - Projected to house more than 200,000 sockets

Online Fault Tolerance

$\rightarrow$ Hard failures

- Researchers have predicted that hard faults will increase
  - Exascale!
  - Machines are getting larger
  - Projected to house more than 200,000 sockets
  - Hard failures may be frequent and only affect a small percentage of nodes

Online Fault Tolerance

→ Approaches

- Checkpoint/restart (C/R)
  - Well-established method
  - Save snapshot of system state
  - Roll back to previous snapshot in case of failure

Online Fault Tolerance

→ Approaches

- Checkpoint/restart (C/R)
  - Well-established method
  - Save snapshot of system state
  - Roll back to previous snapshot in case of failure
- Motivation beyond C/R
  - If a single node experiences a hard fault, why must all the nodes roll back?
  - Recovering from C/R is expensive at large machine scales
    - ★ Complicated because it depends on many factors (e.g checkpointing frequency)

Online Fault Tolerance

→ Approaches

- Checkpoint/restart (C/R)
  - Well-established method
  - Save snapshot of system state
  - Roll back to previous snapshot in case of failure
- Motivation beyond C/R
  - If a single node experiences a hard fault, why must all the nodes roll back?
  - Recovering from C/R is expensive at large machine scales
    - ★ Complicated because it depends on many factors (e.g checkpointing frequency)
- Solutions
  - Application-specific fault tolerance
  - Other system-level approaches
  - Message-logging!

Hard Failure System Model

- P processes that communicate via message passing

Hard Failure System Model

- P processes that communicate via message passing
- Communication is across non-FIFO channels
  - Sent asynchronously
  - Possibly out of order

Hard Failure System Model

- P processes that communicate via message passing
- Communication is across non-FIFO channels
  - Sent asynchronously
  - Possibly out of order
- Guaranteed to arrive sometime in the future if the recipient process has not failed

Hard Failure System Model

- P processes that communicate via message passing
- Communication is across non-FIFO channels
  - Sent asynchronously
  - Possibly out of order
- Guaranteed to arrive sometime in the future if the recipient process has not failed
- *Fail-stop* model for all failures
  - Failed processes do not recover from failures
  - They do not behave maliciously (non-Byzantine failures)

Sender-Based Causal Message Logging (SB-ML)

- Combination of data-driven and control-driven replay
  - Data-driven
    - Messages sent are recorded
  - Control-driven
    - *Determinants* are recorded to store the order of events

Sender-Based Causal Message Logging (SB-ML)

- Combination of data-driven and control-driven replay
  - Data-driven
    - Messages sent are recorded
  - Control-driven
    - *Determinants* are recorded to store the order of events
- Incurs costs in the form of time and storage overhead during forward execution

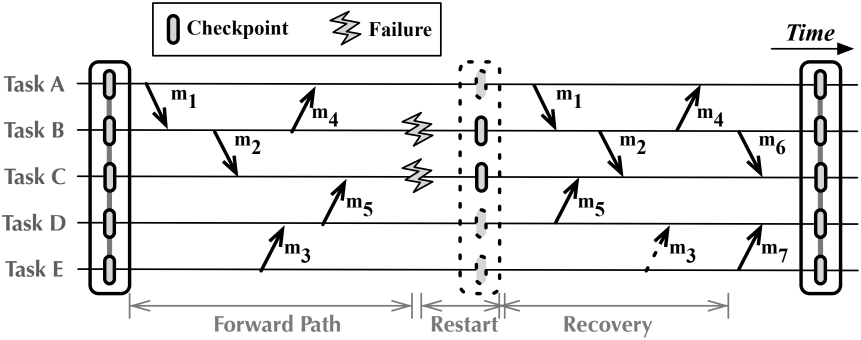
Sender-Based Causal Message Logging (SB-ML)

- Combination of data-driven and control-driven replay
  - Data-driven
    - Messages sent are recorded
  - Control-driven
    - *Determinants* are recorded to store the order of events
- Incurs costs in the form of time and storage overhead during forward execution
- Periodic checkpoints reduce storage overhead
  - Recovery effort is limited to work executed after the latest checkpoint
  - Data stored before the checkpoint can be discarded

Sender-Based Causal Message Logging (SB-ML)

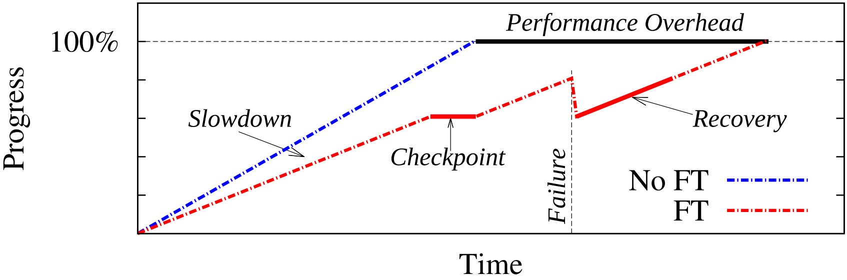
- Combination of data-driven and control-driven replay
  - Data-driven
    - Messages sent are recorded
  - Control-driven
    - *Determinants* are recorded to store the order of events
- Incurs costs in the form of time and storage overhead during forward execution
- Periodic checkpoints reduce storage overhead
  - Recovery effort is limited to work executed after the latest checkpoint
  - Data stored before the checkpoint can be discarded
- Scalable implementation in Charm++

Example Execution with SB-ML



Motivation

→ Overheads with SB-ML



Forward Execution Overhead with SB-ML

- Logging the messages
  - Just requires a pointer to be saved and message is not deallocated!
  - Increases memory pressure

Forward Execution Overhead with SB-ML

- Logging the messages
  - Just requires a pointer to be saved and message is not deallocated!
  - Increases memory pressure
- Determinants, 4-tuple of the form: $<$SPE,SSN,RPE,RSN$>$
  - Components:
    - ⋆ Sender processor (SPE)
    - ⋆ Sender sequence number (SSN)
    - ⋆ Receiver processor (RPE)
    - ⋆ Receiver sequence number (RSN)

Forward Execution Overhead with SB-ML

- Logging the messages
  - Just requires a pointer to be saved and message is not deallocated!
  - Increases memory pressure
- Determinants, 4-tuple of the form: $<$SPE,SSN,RPE,RSN$>$
  - Components:
    - Sender processor (SPE)
    - Sender sequence number (SSN)
    - Receiver processor (RPE)
    - Receiver sequence number (RSN)
  - Must be stored *stably* based on the reliability requirements
    - Propagated to n processors
    - Unacknowledged determinants are augmented onto new messages (to avoid frequent synchronizations)

Forward Execution Overhead with SB-ML

- Logging the messages
  - Just requires a pointer to be saved and message is not deallocated!
  - Increases memory pressure
- Determinants, 4-tuple of the form: $<$SPE,SSN,RPE,RSN$>$
  - Components:
    - Sender processor (SPE)
    - Sender sequence number (SSN)
    - Receiver processor (RPE)
    - Receiver sequence number (RSN)
  - Must be stored *stably* based on the reliability requirements
    - Propagated to n processors
    - Unacknowledged determinants are augmented onto new messages (to avoid frequent synchronizations)
  - Recovery
    - Messages must be replayed in a total order

Forward Execution Microbenchmark (SB-ML)

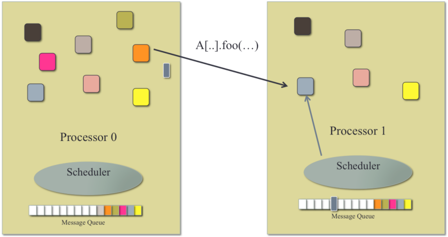
Component	Overhead (%)
Determinants	84.75%
Bookkeeping	11.65%
Message-envelope size increase	3.10%
Message storage	0.50%

- Using the LeanMD (molecular dynamics) benchmark
- Measured on 256 cores of Ranger
- Largest source of overhead is determinants
  - Creating, storing, sending, etc.

Benchmarks

→ Runtime System—Charm++

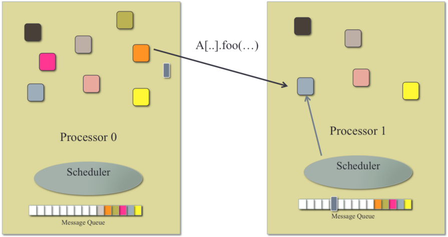
- Decompose parallel computation into objects that communicate
  - More objects than number of processors
  - Objects communicate by sending messages
  - Computation is oblivious to the processors



Benchmarks

→ Runtime System—Charm++

- Decompose parallel computation into objects that communicate
  - More objects than number of processors
  - Objects communicate by sending messages
  - Computation is oblivious to the processors
- Benefits
  - Load balancing, message-driven execution, fault tolerance, etc.



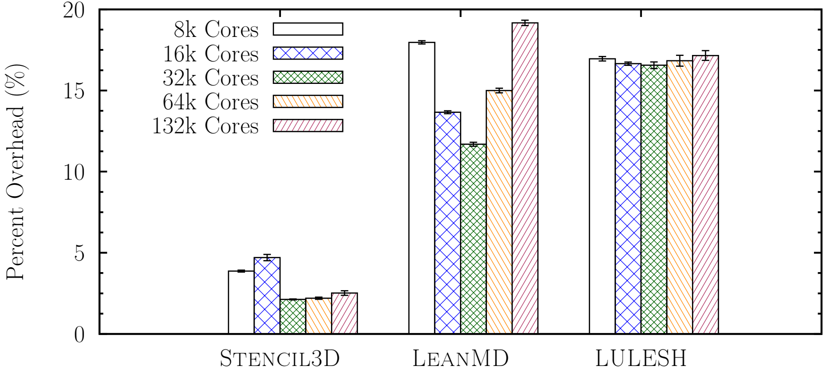
Benchmarks

→ Configuration & Experimental Setup

Benchmark	Configuration
STENCIL3D	matrix: 4096^3, chunk: 64^3
LEANMD (mini-app for NAMD)	600K atoms, 2-away XY, 75 atoms/cell
LULESH (shock hydrodynamics)	matrix: $1024 \text{x} 512^2$, chunk: $16 \text{x} 8^2$

- All experiments on IBM Blue Gene/P (BG/P), 'Intrepid'
- 40960-node system
  - Each node consists of one quad-core 850MHz PowerPC 450
  - 2GB DDR2 memory
- Compiler: IBM XL C/C++ Advanced Edition for Blue Gene/P, V9.0
- Runtime: Charm++ 6.5.1

Forward Execution Overhead with SB-ML



- The finer-grained benchmarks, LeanMD and LULESH, suffer from significant overhead

Reducing the Overhead of Determinants

- Design Criteria
  - We must maintain full determinism

Reducing the Overhead of Determinants

- Design Criteria
  - We must maintain full determinism
  - We must devolve well for all cases (even very non-deterministic programs)

Reducing the Overhead of Determinants

- Design Criteria
  - We must maintain full determinism
  - We must devolve well for all cases (even very non-deterministic programs)
  - Need to consider tasks or lightweight objects

Reducing the Overhead of Determinants

- 'Intrinsic' determinism
  - Many researchers have noticed that programs have internal determinism

Reducing the Overhead of Determinants

- ‘Intrinsic’ determinism
  - Many researchers have noticed that programs have internal determinism
    - Causality tracking (1988: Fidge, *Partial orders for parallel debugging*)

Reducing the Overhead of Determinants

- 'Intrinsic' determinism
  - Many researchers have noticed that programs have internal determinism
    - Causality tracking (1988: Fidge, *Partial orders for parallel debugging*)
    - Racing messages (1992: Netzer, et al., *Optimal tracing and replay for debugging message-passing parallel programs*)

Reducing the Overhead of Determinants

- 'Intrinsic' determinism
  - Many researchers have noticed that programs have internal determinism
    - ⋆ Causality tracking (1988: Fidge, *Partial orders for parallel debugging*)
    - ⋆ Racing messages (1992: Netzer, et al., *Optimal tracing and replay for debugging message-passing parallel programs*)
    - ⋆ Theoretical races (1993: Damodaran-Kamal, *Nondeterminancy: testing and debugging in message passing parallel programs*)

Reducing the Overhead of Determinants

- ‘Intrinsic’ determinism
  - Many researchers have noticed that programs have internal determinism
    - ★ Causality tracking (1988: Fidge, *Partial orders for parallel debugging*)
    - ★ Racing messages (1992: Netzer, et al., *Optimal tracing and replay for debugging message-passing parallel programs*)
    - ★ Theoretical races (1993: Damodaran-Kamal, *Nondeterminancy: testing and debugging in message passing parallel programs*)
    - ★ Block races (1995: Clemencon, *An implementation of race detection and deterministic replay with MPI*

Reducing the Overhead of Determinants

- ‘Intrinsic’ determinism
  - Many researchers have noticed that programs have internal determinism
    - Causality tracking (1988: Fidge, *Partial orders for parallel debugging*)
    - Racing messages (1992: Netzer, et al., *Optimal tracing and replay for debugging message-passing parallel programs*)
    - Theoretical races (1993: Damodaran-Kamal, *Nondeterminancy: testing and debugging in message passing parallel programs*)
    - Block races (1995: Clemencon, *An implementation of race detection and deterministic replay with MPI*
    - MPI and Non-determinism (2000: Kranzlmuller, *Event graph analysis for debugging massively parallel programs*)

Reducing the Overhead of Determinants

- ‘Intrinsic’ determinism
  - Many researchers have noticed that programs have internal determinism
    - ★ Causality tracking (1988: Fidge, *Partial orders for parallel debugging*)
    - ★ Racing messages (1992: Netzer, et al., *Optimal tracing and replay for debugging message-passing parallel programs*)
    - ★ Theoretical races (1993: Damodaran-Kamal, *Nondeterminancy: testing and debugging in message passing parallel programs*)
    - ★ Block races (1995: Clemencon, *An implementation of race detection and deterministic replay with MPI*
    - ★ MPI and Non-determinism (2000: Kranzlmuller, *Event graph analysis for debugging massively parallel programs*)
    - ★ . . .

Reducing the Overhead of Determinants

- 'Intrinsic' determinism
  - Many researchers have noticed that programs have internal determinism
    - ★ Causality tracking (1988: Fidge, *Partial orders for parallel debugging*)
    - ★ Racing messages (1992: Netzer, et al., *Optimal tracing and replay for debugging message-passing parallel programs*)
    - ★ Theoretical races (1993: Damodaran-Kamal, *Nondeterminancy: testing and debugging in message passing parallel programs*)
    - ★ Block races (1995: Clemencon, *An implementation of race detection and deterministic replay with MPI*
    - ★ MPI and Non-determinism (2000: Kranzlmuller, *Event graph analysis for debugging massively parallel programs*)
    - ★ . . .
    - ★ Send-determinism (2011: Guermouche, et al., *Uncoordinated checkpointing without domino effect for send-deterministic MPI applications*)

Our Approach

- In many cases, only a *partial order* must be stored for full determinism
  - Program = *internal determinism* + *non-determinism* + *commutative*

Our Approach

- In many cases, only a *partial order* must be stored for full determinism
  - Program = *internal determinism* + *non-determinism* + *commutative*
  - Internal determinism requires no determinants!

Our Approach

- In many cases, only a *partial order* must be stored for full determinism
  - Program = *internal determinism* + *non-determinism* + *commutative*
  - Internal determinism requires no determinants!
  - Commutative events require no determinants!

Our Approach

- In many cases, only a *partial order* must be stored for full determinism
  - Program = *internal determinism* + *non-determinism* + *commutative*
  - Internal determinism requires no determinants!
  - Commutative events require no determinants!
  - Approach: use determinants to store a partial order for the non-deterministic events that are not commutative

Ordering Algebra

$\rightarrow$ Ordered Sets, $\mathcal{O}$

- $\mathcal{O}(n, d)$
  - Set of n events and d dependencies

Ordering Algebra

→ Ordered Sets, $\mathcal{O}$

- $\mathcal{O}(n, d)$
  - Set of n events and d dependencies
  - Can be accurately replayed from a given starting point

Ordering Algebra

→ Ordered Sets, $\mathcal{O}$

- $\mathcal{O}(n, d)$
  - Set of n events and d dependencies
  - Can be accurately replayed from a given starting point
  - Dependencies d can be among the events in the set, or on preceding events

Ordering Algebra

$\rightarrow$ **Ordered Sets, $\mathcal{O}$**

- $\mathcal{O}(n, d)$
  - Set of n events and d dependencies
  - Can be accurately replayed from a given starting point
  - Dependencies d can be among the events in the set, or on preceding events
  - Intuitively, they are ordered sets of events

Ordering Algebra

$\rightarrow$ Ordered Sets, $\mathcal{O}$

- $\mathcal{O}(n, d)$
  - Set of n events and d dependencies
  - Can be accurately replayed from a given starting point
  - Dependencies d can be among the events in the set, or on preceding events
  - Intuitively, they are ordered sets of events
- Define sequencing operation, $\boxplus$:
  $\mathcal{O}(1, d_1) \boxplus \mathcal{O}(1, d_2) = \mathcal{O}(2, d_1 + d_2 + 1)$
  - Intuitively, if we have two atomic events, we need a single dependency to tell us which one comes first
- Generalization: $\mathcal{O}(n_1, d_1) \boxplus \mathcal{O}(n_2, d_2) = \mathcal{O}(n_1 + n_2, d_1 + d_2 + 1)$

Ordering Algebra

$\rightarrow$ Unordered Sets, $\mathcal{U}$

- $\mathcal{U}(n, d)$
  - Unordered set of n events and d dependencies

Ordering Algebra

→ Unordered Sets, $\mathcal{U}$

- $\mathcal{U}(n, d)$
  - Unordered set of n events and d dependencies
  - Example is where several messages are sent to a single endpoint

Ordering Algebra

$\rightarrow$ Unordered Sets, $\mathcal{U}$

- $\mathcal{U}(n, d)$
  - Unordered set of n events and d dependencies
  - Example is where several messages are sent to a single endpoint
  - Depending the order of arrival, the eventual state will be different
- We decompose this 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 $d = \sum d_i$

Ordering Algebra

$\rightarrow$ Unordered Sets, $\mathcal{U}$

- $\mathcal{U}(n, d)$
  - Unordered set of n events and d dependencies
  - Example is where several messages are sent to a single endpoint
  - Depending the order of arrival, the eventual state will be different
- We decompose this 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 $d = \sum d_i$

  - Result: additional $n - 1$ dependencies required to fully order n events

Ordering Algebra

→ Interleaving Multiple Independent Sets, ⊠ operator

Lemma

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

Lemma

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:* $\displaystyle\mathop{\boxtimes}_{i=1}^{n} \mathcal{O}(m_i, d_i) = \mathcal{O}(m, d + m - \max_i m_i)$ *where* $\displaystyle m = \sum_{i=1}^{n} m_i \wedge d = \sum_{i=1}^{n} d_i$

Internal Determinism

$\rightarrow \mathcal{D}$

- $\mathcal{D}(n) = \mathcal{O}(n, 0)$
- n deterministically ordered events are structurally equivalent to an ordered set of n events with no associated explicit dependencies!

Internal Determinism

$\rightarrow \mathcal{D}$

- $\mathcal{D}(n) = \mathcal{O}(n, 0)$
- n deterministically ordered events are structurally equivalent to an ordered set of n events with no associated explicit dependencies!
- What happens if we interleave internal determinism with something else?

Internal Determinism

$\rightarrow \mathcal{D}$

- $\mathcal{D}(n) = \mathcal{O}(n, 0)$
- n deterministically ordered events are structurally equivalent to an ordered set of n events with no associated explicit dependencies!
- What happens if we interleave internal determinism with something else?
- k interruption points $=> \mathcal{O}(k, k-1)$

Communtative Events

$\rightarrow \mathcal{C}$

- Some events in programs are communtative
  - Regardless of the execution order the state will be identical

Communtative Events

$\rightarrow \mathcal{C}$

- Some events in programs are communtative
  - Regardless of the execution order the state will be identical
- All existing theories of message logging execute record a total order on them

Communtative Events

$\rightarrow \mathcal{C}$

- Some events in programs are communtative
  - Regardless of the execution order the state will be identical
- All existing theories of message logging execute record a total order on them
- However we can reduce a commutative set to:
  - $\mathcal{C}(n) = \mathcal{O}(2, 1)$
  - A beginning and end event sequenced together
  - Sequencing other sets of event around the region just puts them before and after

Communtative Events

$\rightarrow \mathcal{C}$

- Some events in programs are communtative
  - Regardless of the execution order the state will be identical
- All existing theories of message logging execute record a total order on them
- However we can reduce a commutative set to:
  - $\mathcal{C}(n) = \mathcal{O}(2, 1)$
  - A beginning and end event sequenced together
  - Sequencing other sets of event around the region just puts them before and after
  - Interleaving other events puts them in three buckets:
    - ★ (1) before the begin event
    - ★ (2) during the commutative region
    - ★ (3) after the end event

Communtative Events

$\rightarrow \mathcal{C}$

- Some events in programs are communtative
  - Regardless of the execution order the state will be identical
- All existing theories of message logging execute record a total order on them
- However we can reduce a commutative set to:
  - $\mathcal{C}(n) = \mathcal{O}(2, 1)$
  - A beginning and end event sequenced together
  - Sequencing other sets of event around the region just puts them before and after
  - Interleaving other events puts them in three buckets:
    - (1) before the begin event
    - (2) during the commutative region
    - (3) after the end event
  - This corresponds exactly to an ordered set of two events!

Applying the Theory

→ PO-REPLAY: Partial-Order Message Identification Scheme

- Properties
  - It tracks causality with Lamport clocks
  - It uniquely identifies a sent message, whether or not its order is transposed
  - It requires exactly the number of *determinants* and dependencies produced by the ordering algebra

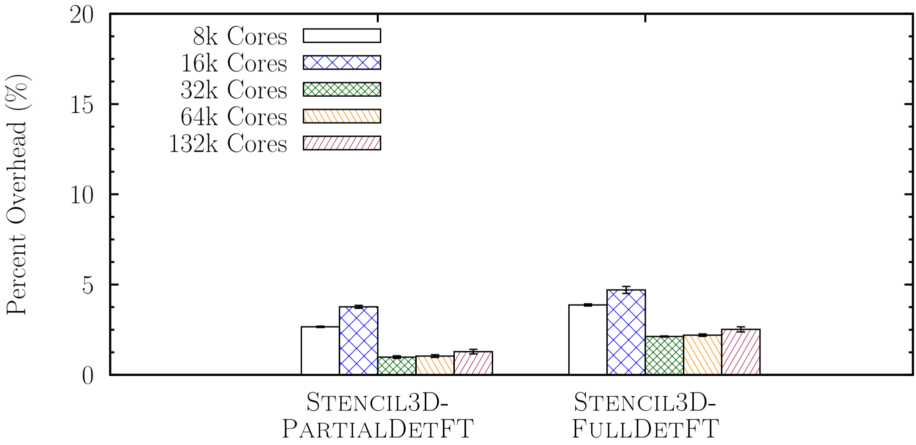
Applying the Theory

→ PO-REPLAY: **Partial-Order Message Identification Scheme**

- Properties
  - It tracks causality with Lamport clocks
  - It uniquely identifies a sent message, whether or not its order is transposed
  - It requires exactly the number of *determinants* and dependencies produced by the ordering algebra
- Determinant Composition (3-tuple): $<$SRN,SPE,CPI$>$
  - SRN: sender region number, incremented for every send outside a commutative region and incremented once when a commutative region starts
  - SPE: sender processor endpoint
  - CPI: commutative path identifier, sequence of bits that represents the path to the root of the commutative region

Experimental Results

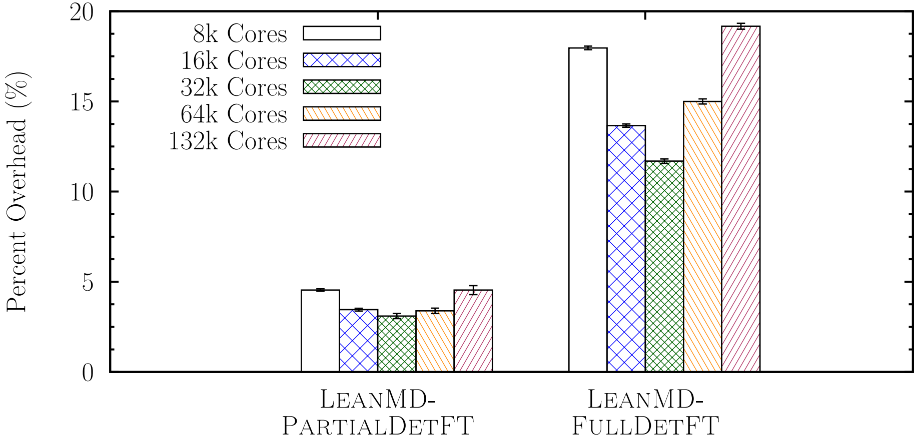
→ Forward Execution Overhead: Stencil3D



- Course-grained, shows small improvement over SB-ML

Experimental Results

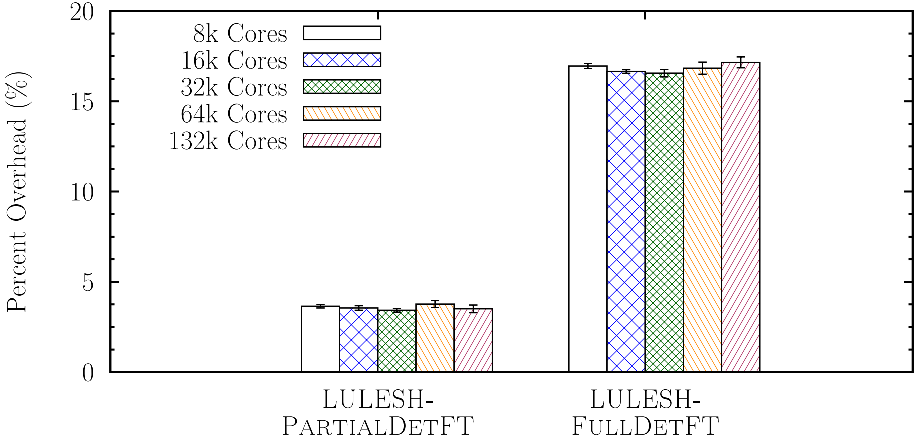
→ Forward Execution Overhead: LeanMD



- Fine-grained, reduction from 11-19% overhead to <5%

Experimental Results

→ Forward Execution Overhead: LULESH



- Medium-grained, many messages, 17% overhead to <4%

Experimental Results

→ Fault Injection

- Measure the recovery time for the different protocols

Experimental Results

→ Fault Injection

- Measure the recovery time for the different protocols
  - We inject a simulated fault on a random node

Experimental Results

→ Fault Injection

- Measure the recovery time for the different protocols
  - We inject a simulated fault on a random node
  - During approximately the middle of the period

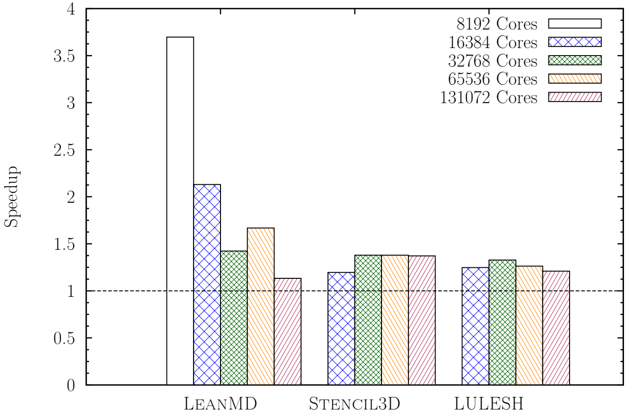
Experimental Results

→ Fault Injection

- Measure the recovery time for the different protocols
  - We inject a simulated fault on a random node
  - During approximately the middle of the period
  - We calculate the optimal checkpoint period duration using Daly's formula
    - Assuming 64K–1M socket count
    - Assuming MTBF of 10 years

Experimental Results

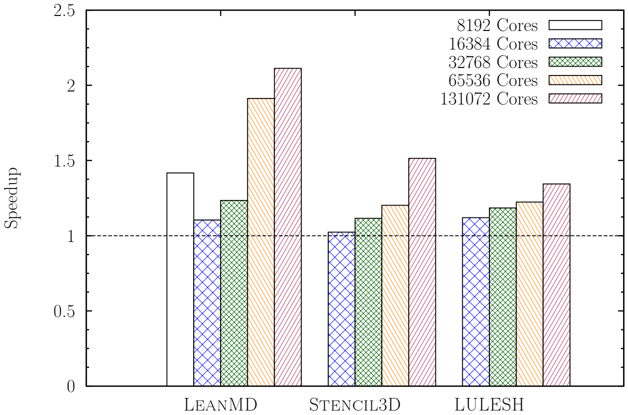
→ Recovery Time Speedup C/R



- LeanMD has the most speedup due to its fine-grained, overdecomposed nature
- We achieve speedup in all cases in recovery time

Experimental Results

→ Recovery Time Speedup SB-ML



- Increased speedup with scale, due to expense of coordinating determinants and ordering

Experimental Results

→ Summary

- Our new message logging protocol has about $<$5% overhead for the benchmarks tested

Experimental Results

→ Summary

- Our new message logging protocol has about <5% overhead for the benchmarks tested
- Recover is significantly faster than C/R or causal

Experimental Results

→ Summary

- Our new message logging protocol has about <5% overhead for the benchmarks tested
- Recover is significantly faster than C/R or causal
- Depending on the frequency of faults, it may perform better than C/R

Future Work

- More benchmarks

Future Work

- More benchmarks
- Study for broader range of programming models

Future Work

- More benchmarks
- Study for broader range of programming models
- Memory overhead of message logging makes it infeasible for some applications

Future Work

- More benchmarks
- Study for broader range of programming models
- Memory overhead of message logging makes it infeasible for some applications
- Automated extraction of ordering and interleaving properties

Future Work

- More benchmarks
- Study for broader range of programming models
- Memory overhead of message logging makes it infeasible for some applications
- Automated extraction of ordering and interleaving properties
- Programming language support?

Conclusion

- Comprehensive approach for reasoning about execution orderings and interleavings

Conclusion

- Comprehensive approach for reasoning about execution orderings and interleavings
- We observe that the information stored can be reduced in proportion to the knowledge of order flexibility

Conclusion

- Comprehensive approach for reasoning about execution orderings and interleavings
- We observe that the information stored can be reduced in proportion to the knowledge of order flexibility
- Programming paradigms should make this cost model clearer!

Questions?