Leveraging partial determinism in MPI applications for efficient fault tolerance

Thomas Ropars

thomas.ropars@imag.fr

Université Grenoble Alpes

Context

Fault tolerance at extreme scale is a challenge

- Increase in the number of components
  - Millions of computing cores
- Increased failure rate
  - Failures can also be due to software

Different kinds of failures

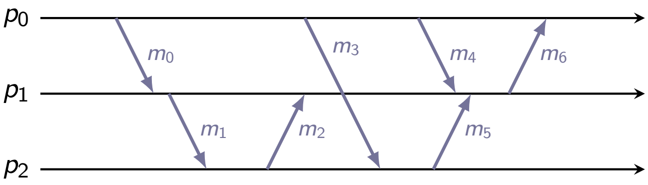
- Crash failures
- Data corruption
  - Soft errors

- This talk is about crash failures

FT for tightly-coupled distributed applications

Message-passing applications

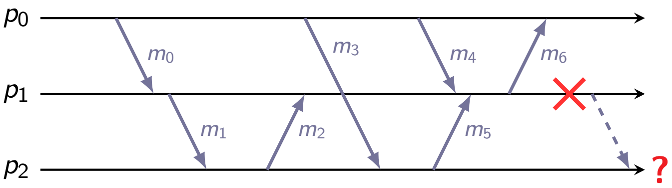
- A set of processes
- Communicate using messages
  - MPI



FT for tightly-coupled distributed applications

Message-passing applications

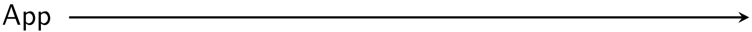
- A set of processes
- Communicate using messages
  - MPI



One process crash prevents the application from progressing

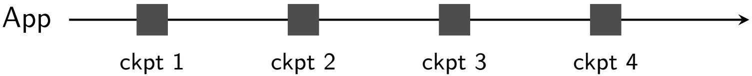
Checkpointing

- Periodically save the state of the application



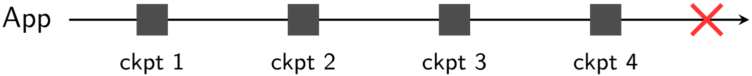
Checkpointing

- Periodically save the state of the application



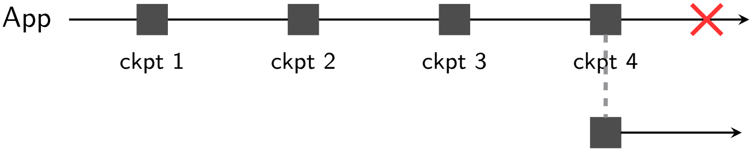
Checkpointing

- Periodically save the state of the application



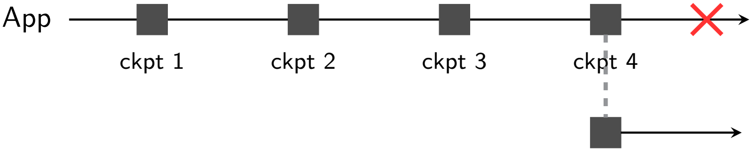
Checkpointing

- Periodically save the state of the application
- Restart from last checkpoint in the event of a failure



Checkpointing

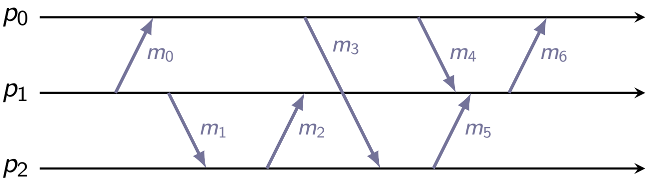
- Periodically save the state of the application
- Restart from last checkpoint in the event of a failure



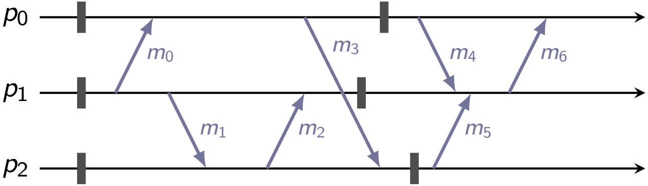
Efficiency depends on

- The time to checkpoint
- The time to restart the application from a checkpoint after a failure
- The time to replay lost computation

Coordinated checkpointing



Coordinated checkpointing



Standard solution in HPC systems

- Checkpoints form a consistent global state
- When a process fail, all processes restart from the last checkpoint

The FT challenge

Status in 2010

- Coordinated checkpoints saved on a PFS
  - Extreme scale application footprint
- Failure rate increase
  - MTBF of a few hours

The FT challenge

Status in 2010

- Coordinated checkpoints saved on a PFS
  - Extreme scale application footprint
- Failure rate increase
  - MTBF of a few hours

More than 50% of
the computing resources could be wasted

Questions related to checkpointing

- Where to save the checkpoints?
- What data to save?
- How to ensure that the execution is correct?

Questions related to checkpointing

- Where to save the checkpoints?
  - Multi-level checkpointing [Moody et al, 2011; Bautista et al, 2010]
- What data to save?
  - Application-level checkpointing
- How to ensure that the execution is correct?
  - **Purpose of the checkpointing protocol**

Questions related to checkpointing

- Where to save the checkpoints?
  - Multi-level checkpointing [Moody et al, 2011; Bautista et al, 2010]
- What data to save?
  - Application-level checkpointing
- How to ensure that the execution is correct?
  - **Purpose of the checkpointing protocol**

Can we do better than coordinated checkpointing?

Questions related to checkpointing

- Where to save the checkpoints?
  - Multi-level checkpointing [Moody et al, 2011; Bautista et al, 2010]
- What data to save?
  - Application-level checkpointing
- How to ensure that the execution is correct?
  - **Purpose of the checkpointing protocol**

Can we do better than coordinated checkpointing?

- Maybe if we take into account the characteristics of MPI HPC apps

Towards a scalable checkpointing protocol

Goals

- Partial restart (failure containment)
- Good performance
  - Low failure-free execution overhead
  - Fast recovery
- Low resource usage
  - Computation
  - Data storage

Research direction

- Revisit checkpointing theory taking into account the characteristics of MPI applications

Contributions

SPBC: A scalable hierarchical protocol

- Perfect failure containment
- No events logged
- Negligible overhead during failure free execution
- Speedup for the rework time

Execution models

- Channel-deterministic algorithms
  - Most SPMD MPI applications are channel deterministic.
- The always-happens-before relation
  - Partial-order relation on the events of a channel-deterministic algorithm

Background

Problem statement

- Asynchronous distributed system
  - FIFO channels
- A message-passing application
  - Fix set of processes
  - MPI application
- Crash-stop failures
  - Multiple concurrent failures

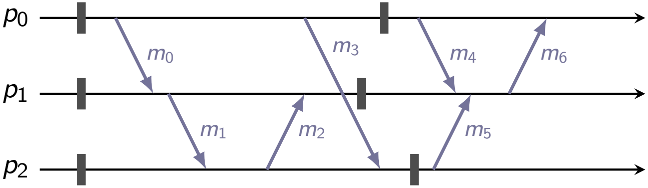
Consistent global state

Causal dependencies between messages

- Message exchanges create dependencies between the state of the processes
  - Events are partially ordered by Lamport's *Happened-Before* relation ($\rightarrow$)
    - $send(m_0) \rightarrow recv(m_0)$
    - $recv(m_0) \rightarrow recv(m_2)$

Restart from a consistent global state

Problem of restarting from a random state



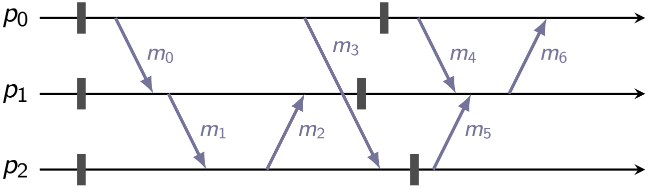
Consistent global state

Causal dependencies between messages

Restart from a consistent global state

- A state that could have existed in a failure free execution
- $e' \in C \textit{ and } e \rightarrow e' \implies e \in C$

Problem of restarting from a random state



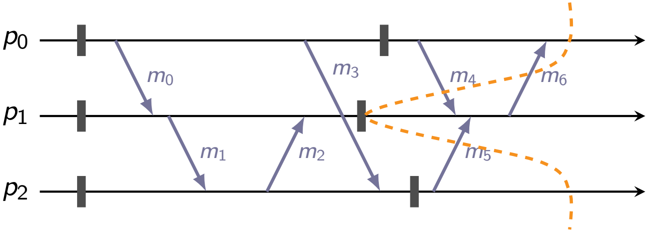
Consistent global state

Causal dependencies between messages

Restart from a consistent global state

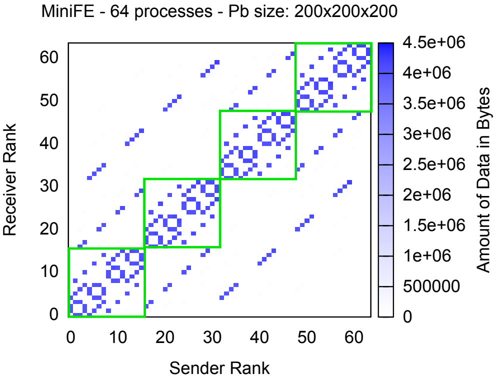
Problem of restarting from a random state

- Message m_6 is orphan
- What if we cannot replay m_4 and m_5?
- What if they are not received in the same order?



Hierarchical protocols

Meneses et al, 2010; Bouteiller et al, 2011

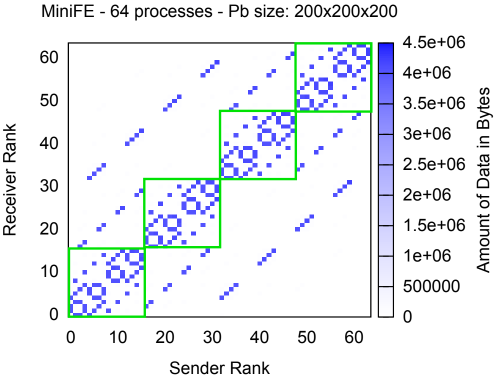


Clustering of the processes

- Coordinated checkpointing inside clusters
- Log inter-cluster messages

Hierarchical protocols

Meneses et al, 2010; Bouteiller et al, 2011



Clustering of the processes

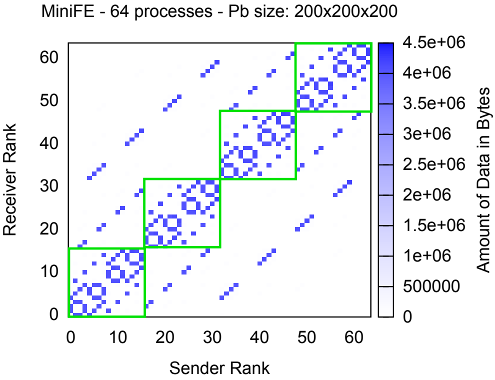
- Coordinated checkpointing inside clusters
- Log inter-cluster messages

Advantages

- Perfect failure containment
- Low number of messages to log

Hierarchical protocols

Meneses et al, 2010; Bouteiller et al, 2011



Clustering of the processes

- Coordinated checkpointing inside clusters
- Log inter-cluster messages

Advantages

- Perfect failure containment
- Low number of messages to log

Problem

- **All non-deterministic events need to be logged**
- Overhead on failure free performance

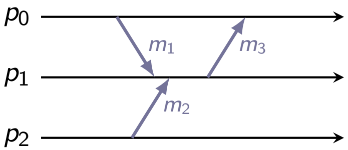
The SPBC protocol

Channel-deterministic algorithm

MPI channel

- One-way channels
- A channel is defined in the context of a communicator

Definition



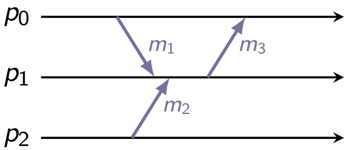
An algorithm A is channel-deterministic, if for an initial state Σ, and for any channel c, the sequence of send events on c is the same in any valid execution of A.

Channel-deterministic algorithm

MPI channel

- One-way channels
- A channel is defined in the context of a communicator

Definition



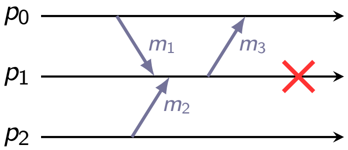
The relative order of the messages received by a process has no impact on the content and the order of the messages sent by this process on each channel.

Validity of the model

Study of the determinism in MPI applications [Cappello et al, 2010]

- 27 applications
- **26 over 27 are channel-deterministic**
  - One master/worker application is not

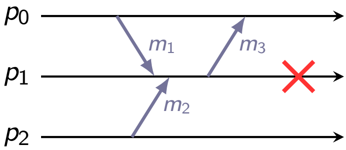
Impact of channel-determinism on event logging



What "causality" says?

- Events recv(m_1) and recv(m_2) have to be logged to ensure that m_3 remains valid after a failure

Impact of channel-determinism on event logging



What "causality" says?

- Events recv(m_1) and recv(m_2) have to be logged to ensure that m_3 remains valid after a failure

With channel-determinism

- Message m_3 does not depend on the relative order of m_1 and m_2
- Events recv(m_1) and recv(m_2) do not need to be logged

Impact of the MPI interface on event logging

The other role of event logging

- Choosing which logged message to deliver

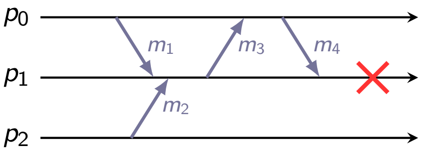


Figure: First execution

Impact of the MPI interface on event logging

The other role of event logging

- Choosing which logged message to deliver

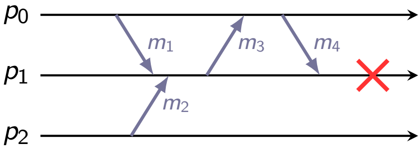


Figure: First execution

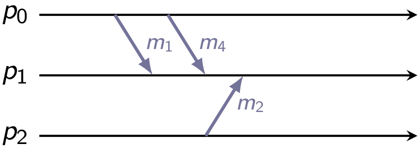


Figure: Replay

Impact of the MPI interface on event logging

The other role of event logging

- Choosing which logged message to deliver

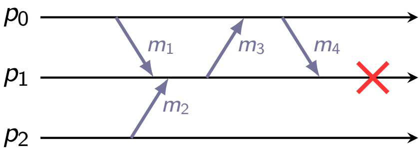


Figure: First execution

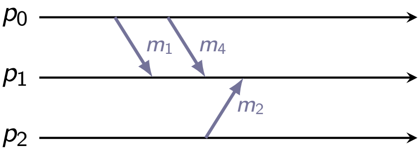


Figure: Replay

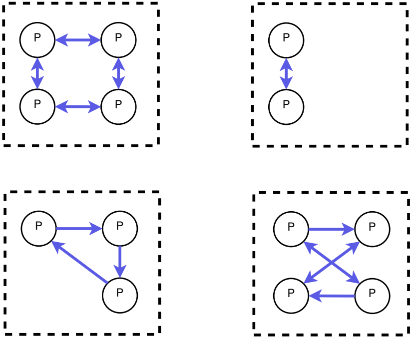
Most MPI messages are received using *named* requests

- m_4 cannot be received instead of m_2
- What if `MPI_ANY_SOURCE` is used?

The protocol

Failure-free execution

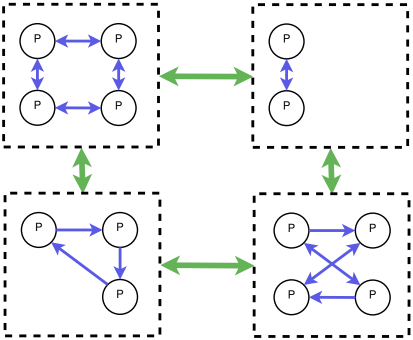
- Take coordinated checkpoints inside clusters periodically



The protocol

Failure-free execution

- Take coordinated checkpoints inside clusters periodically
- Log inter-cluster messages
  - **No event logging**

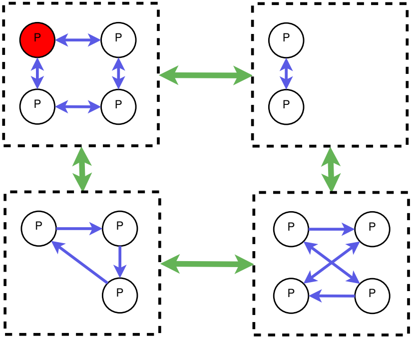


The protocol

Failure-free execution

- Take coordinated checkpoints inside clusters periodically
- Log inter-cluster messages
  - **No event logging**

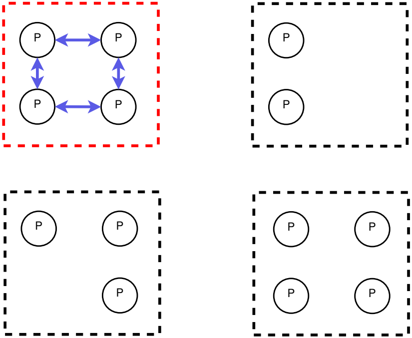
Recovery



The protocol

Failure-free execution

- Take coordinated checkpoints inside clusters periodically
- Log inter-cluster messages
  - **No event logging**



Recovery

- Restart the failed cluster from the last checkpoint

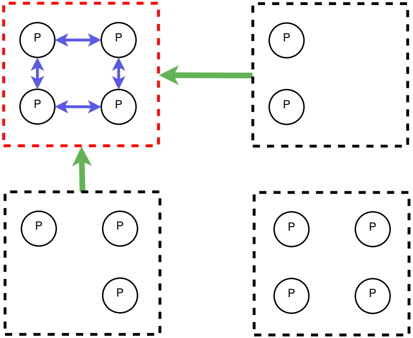
The protocol

Failure-free execution

- Take coordinated checkpoints inside clusters periodically
- Log inter-cluster messages
  - **No event logging**

Recovery

- Restart the failed cluster from the last checkpoint
- Replay missing inter-cluster messages from the logs
  - Same order as before the failure



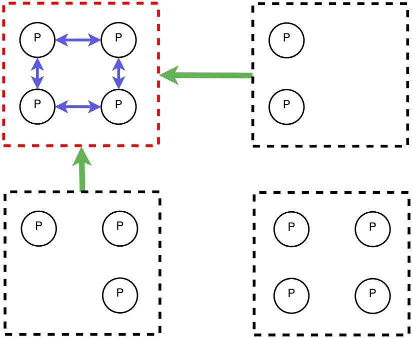
The protocol

Failure-free execution

- Take coordinated checkpoints inside clusters periodically
- Log inter-cluster messages
  - **No event logging**

Recovery

- Restart the failed cluster from the last checkpoint
- Replay missing inter-cluster messages from the logs
  - Same order as before the failure



Correct for channel-deterministic applications not including MPI_ANY_SOURCE

Always-happens-before relation

Comparing events from different executions

In a channel-deterministic algorithm A, the same messages are exchanged in all valid executions of A (for a given initial state).

- The relative order of send and recv events can be compared in different executions of A.

Definition

Event e_1 always-happens-before event e_2 if there is a happened-before relation between e_1 and e_2 in all valid executions of A

- Notation: $e_1 \xrightarrow{A} e_2$

Non-valid execution and always-happens-before relation

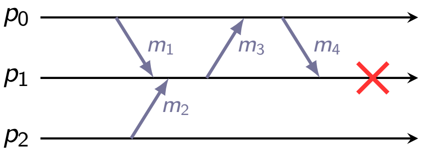


Figure: First execution

Always-happens-before relations:

- $recv(m_1) \xrightarrow{A} send(m_4)$
- $recv(m_2) \xrightarrow{A} send(m_4)$

Non-valid execution and always-happens-before relation

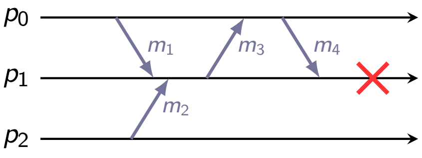


Figure: First execution

Always-happens-before relations:

- $recv(m_1) \xrightarrow{A} send(m_4)$
- $recv(m_2) \xrightarrow{A} send(m_4)$

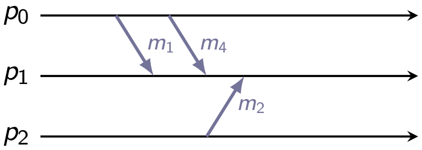


Figure: Replay

We have shown that:

- If a reception request r and a message m can be mismatched during recovery, then $r \xrightarrow{A} m$.

Transformation of the algorithm

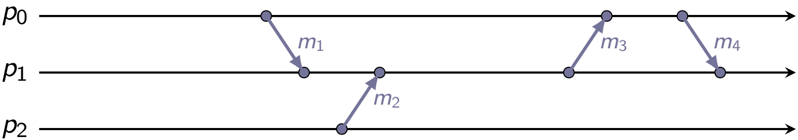
Meaning of AHB

- Mismatches have to be avoided by the programmer in failure free execution
  - She builds in the required synchronization between processes
  - She defines communication patterns

Our solution

- During recovery, a logged messages should be replayed in the pattern it belongs to.
- **We propose to add extra ids on messages and reception requests**
  - Tuple {`pattern_id`, `iteration_id`}

The API



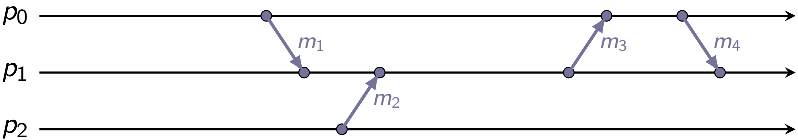
Code of p_0:

```
pat1=Declare_pattern();
...
Begin_iteration(pat1);
MPI_Send(dest: p1); /*m1*/
...
End_iteration(pat1);
MPI_Recv(source: p1); /*m3*/
MPI_Send(dest: p1); /*m4*/
```

Code of p_1:

```
pat1=Declare_pattern();
...
Begin_iteration(pat1);
MPI_Recv(source: ANY); /*m1*/
MPI_Recv(source: ANY); /*m2*/
...
End_iteration(pat1);
MPI_Send(dest: p0); /*m3*/
MPI_Recv(source: p0); /*m4*/
```

The API



Code of p_0:

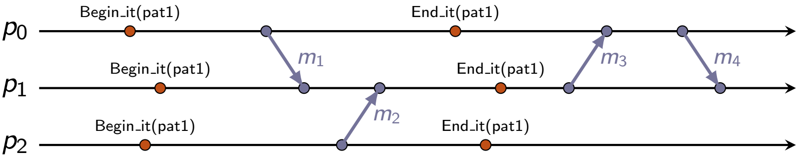
```
pat1=Declare_pattern();
...
Begin_iteration(pat1);
MPI_Send(dest: p1); /*m1*/
...
End_iteration(pat1);
MPI_Recv(source: p1); /*m3*/
MPI_Send(dest: p1); /*m4*/
```

Code of p_1:

```
pat1=Declare_pattern();
...
Begin_iteration(pat1);
MPI_Recv(source: ANY); /*m1*/
MPI_Recv(source: ANY); /*m2*/
...
End_iteration(pat1);
MPI_Send(dest: p0); /*m3*/
MPI_Recv(source: p0); /*m4*/
```

- All communication calls that are not inside a programmer-defined pattern are associated with a default pattern

The API



Code of p_0:

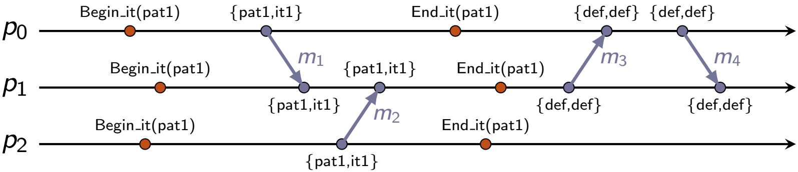
```
pat1=Declare_pattern();
...
Begin_iteration(pat1);
MPI_Send(dest: p1); /*m1*/
...
End_iteration(pat1);
MPI_Recv(source: p1); /*m3*/
MPI_Send(dest: p1); /*m4*/
```

Code of p_1:

```
pat1=Declare_pattern();
...
Begin_iteration(pat1);
MPI_Recv(source: ANY); /*m1*/
MPI_Recv(source: ANY); /*m2*/
...
End_iteration(pat1);
MPI_Send(dest: p0); /*m3*/
MPI_Recv(source: p0); /*m4*/
```

- All communication calls that are not inside a programmer-defined pattern are associated with a default pattern

The API



Code of p_0:

```
pat1=Declare_pattern();
...
Begin_iteration(pat1);
MPI_Send(dest: p1); /*m1*/
...
End_iteration(pat1);
MPI_Recv(source: p1); /*m3*/
MPI_Send(dest: p1); /*m4*/
```

Code of p_1:

```
pat1=Declare_pattern();
...
Begin_iteration(pat1);
MPI_Recv(source: ANY); /*m1*/
MPI_Recv(source: ANY); /*m2*/
...
End_iteration(pat1);
MPI_Send(dest: p0); /*m3*/
MPI_Recv(source: p0); /*m4*/
```

- All communication calls that are not inside a programmer-defined pattern are associated with a default pattern

Experiments

Implementation

- Integration in MPICH v3.0.2
- Matching messages and requests:
  - Modified message header to include `pattern_id` and `iteration_id`
  - Modification of the matching function

Setup

64-node cluster (grid'5000)

- 2.5 GHz Intel Xeon CPUs (2x4 cores per node)
- 16 GB of memory
- Infiniband 20G
- MPICH-3.0.2 with IPoIB

6 applications

- MiniFe (modified to work with SPBC)
- MiniGhost
- Boomer-AMG (modified/SPBC)
- GTC (modified/SPBC)
- MILC (modified/SPBC)
- CM1

- Modifications are very simple

Failure-free performance (16 clusters)

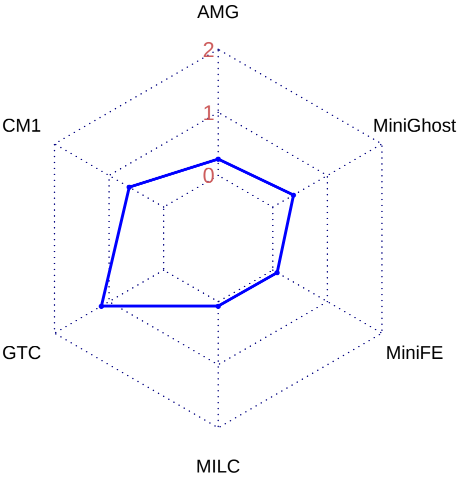
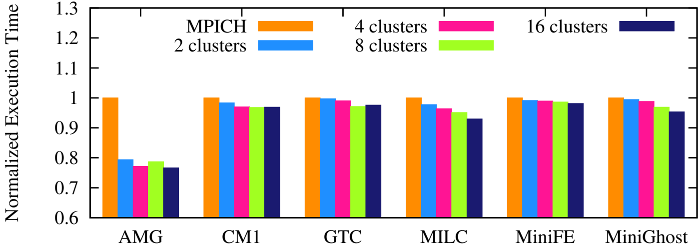


Figure: Performance overhead in %

The overhead is at most 1%

- Overhead of message logging
- Less with larger clusters

Performance during recovery



Always faster during recovery:

- Recovering processes can skip sending inter-cluster messages
- Logged messages can be available in advance

Conclusion

A new approach

- Design a fault tolerant solution that works efficiently with many MPI applications

New concepts

- Channel-deterministic algorithms
- The always-happens-before relation

The SPBC checkpointing solution

- A hierarchical checkpointing protocol
- No events logged during failure free execution
- Minor modifications of the applications (if any)
- Efficient in failure free execution and in recovery

Research directions

Managing logs in hierarchical protocols

- Dedicated logger nodes [Martsinkevich et al, 2015]

Replication of MPI processes

- Replication for channel-deterministic applications [Lefray et al, 2013]
- Highly efficient replication [Ropars et al, 2015]

Thanks

My co-workers

- Elisabeth Brunet, Franck Cappello, Amina Guermouche, Laxmikant Kale, Tatiana Martsinkevitch, Esteban Meneses, André Schiper, Marc Snir, Bora Ucar.

[1] Thomas Ropars et al. "SPBC: Leveraging the Characteristics of MPI HPC Applications for Scalable Checkpointing". *SuperComputing*. 2013.

[2] Amina Guermouche et al. "HydEE: Failure Containment without Event Logging for Large Scale Send-Deterministic MPI Applications". *IPDPS*. 2012.

[3] Amina Guermouche et al. "Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic Message Passing Applications". *IPDPS*. 2011.

[4] Thomas Ropars et al. "On the Use of Cluster-Based Partial Message Logging to Improve Fault Tolerance for MPI HPC Applications". *Euro-Par*. 2011.