An Uncoordinated Checkpointing Protocol for Send-deterministic HPC Application

Amina Guermouche[1,2], Thomas Ropars[1], Elisabeth Brunet[1], Marc Snir[3], Franck Cappello[1,3]

(1): INRIA Saclay-Île de France, F-91893 Orsay, France
(3): University of Illinois at Urbana-Champaign - College of Engineering, Urbana, IL, USA
(2): Université Paris Sud, LRI, F-91405 Orsay, France

Introduction

- Parallel computers become larger and faster
- Failure frequency increases
- Applications need fault tolerance protocol that:
  - minimizes the impact of failures
  - provides good performances on failure free execution
- MPI Applications

Existing fault tolerance protocols

- Communications between processes create dependences between them (Lamport's happened-before relation)
- Consistent global state : state that could have been seen during failure free execution
- Fault tolerance protocol keeps a consistant global state:
  - Global restart
  - Save all communication events

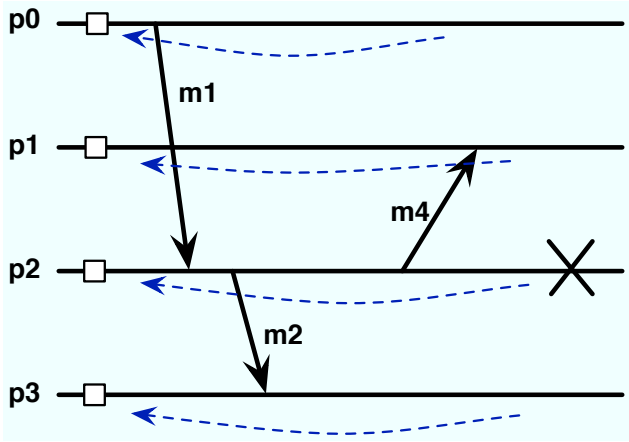
3 main families of fault tolerance protocols:

1. Coordinated checkpoint
2. Uncoordinated checkpoint
3. Message logging

Existing Fault Tolerance Protocols

Coordinated Checkpointing

- Assumption: Non-deterministic applications
- Checkpoints are coordinated on every process
- Failure $\rightarrow$ all processes rollback to the same checkpoint and then restart



Advantages

- Easy recovery
- Easy garbage collect

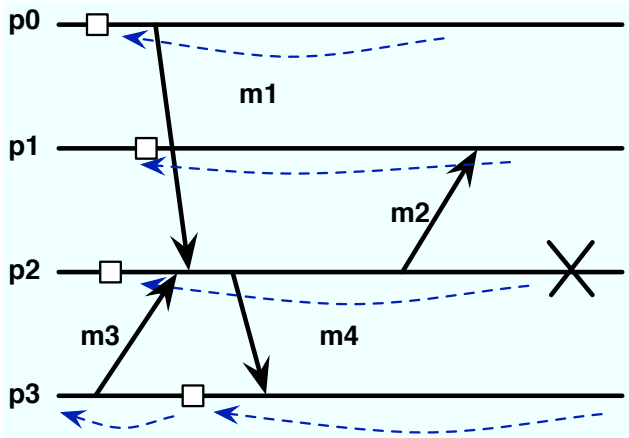
Drawbacks

- Global restart
- Congestion on I/O resources (*Oldfield 2007*)

Existing Fault Tolerance Protocols

Uncoordinated Checkpointing

- Assumption: Non-deterministic applications
- Checkpoints are taken independently among processes



Advantages

- No forced global restart
- Checkpoint scheduling

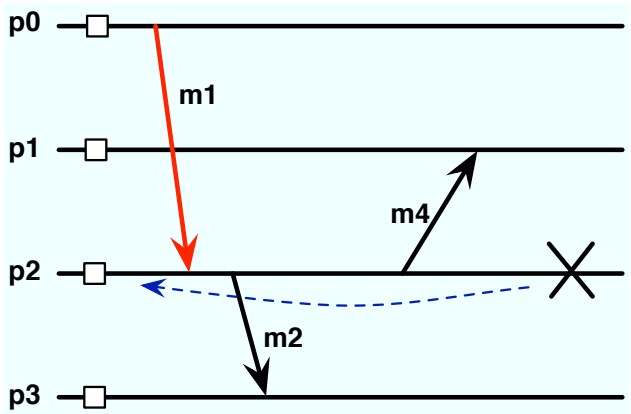
Drawbacks

- Risk of domino effect
- Complex garbage collect

Existing Fault Tolerance Protocols

Message logging

- Assumption: Piecewise deterministic applications
- Processes log the messages they send
- Checkpoints are used to reduce the extent of rollback during recovery
- Failure → Failed processes roll back to their most recent checkpoint and recover thanks to logged messages



Advantages

- Limit the number of processes to roll back
- Simple garbage collect
- No domino effect

Drawbacks

- All messages are logged:
  - Impact on performance
  - Memory consumption

Contributions

Existing fault tolerance protocol assumes that applications are:

- Non-deterministic: After restart, execution may be different
- Piecewise deterministic: If the sequence of receptions is the same on every process, the behavior is the same

1. Application analysis to show that HPC applications are *send-deterministic*:
   - Sequence of sends is always the same
   - Even if receptions order is not the same, the behavior is the same
2. Design a protocol that avoids:
   - Domino effect
   - Logging all messages
   - Global restart

Applications Analysis

Methodology: Static analysis of applications consisting of looking at each communication pattern and studying its deterministic nature (*Cappello 2010*)

Analyzed applications:
27 MPI HPC applications

- Nas Benchmarks
- 6 NERSC Benchmarks
- 2 USQCD Benchmarks
- 6 Sequoia Benchmarks
- SpecFEM3D, Nbody, Ray2mesh, ScaLAPACK SUMMA

Applications Analysis

Deterministic communication pattern

Sequences of sends and receives are the same from one execution to another

```
 MPI_Irecv(T[i],..., i, ...);
/*  where i is the source */
 MPI_Isend(x, ...,  i, ...);
/* where i is the destination */
MPI_WaitAll(2, ...);
```

Applications Analysis

Non-deterministic communication pattern

Sequences of send and receives may be different from one execution to another

```
while()
 {
  MPI_IProbe(ANY_SOURCE, tag1)
  while(flag ==true)
    {
     proc = &status.mpi_SOURCE;
     MPI_Recv(x, ..., proc,
        tag1, ...);
     //modify x
     MPI_Send(x, ..., proc,
        tag2, ...);
    }
```

Application analysis

Send deterministic communication pattern

Sequence of sends is the same from one execution to another

Send-deterministic communication pattern

- No forced reception order
- Same result on different reception order
- Same sequence of sends on every process with the same content

```
 for(i = 0 ; i < nb ; i++)
{
  MPI_Irecv(T[i],..., i, ...);
/*  where i is the source */
  MPI_Send(x, ...,  i, ...);
/* where i is the destination *
 }
for(i = 0 ; i < nb ; i++)
{
   MPI_WaitAny(...);
   ...
}
```

Application	Collective communications	SD patterns	CD patterns	ND patterns	Type
ScaLAPACK SUMMA	x	0	x	0	D
SP	13	1	6	0	SD
BT	12	1	4	0	SD
LU	x	0	x	0	D
CG	x	0	x	0	D
MG	x	0	x	0	D
FT	x	0	x	0	D
EP	x	0	x	0	D
DT	x	0	x	0	D
Nbody	x	0	x	0	D
USQCD-CPS	2	31	0	0	SD
USQCD-MILC	1099	517	111	0	SD
Sequoia-UMT	52	1	1	0	SD
Sequoia-lammp	867	4	33	0	SD
Sequoi-IOR	18	2	0	0	SD
Sequoia-AMG	41	76	4	1	ND
Sequoia-Sphot	7	7	1	0	SD
Sequoia-IRS	x	x	x	0	SD
NERSC-CAM	700	61	4	0	SD
NERSC-IMPACT	0	12	97	0	SD
NERSC-MAESTRO	21	9	9	0	SD
NERSC-GTC	x	0	x	0	D
NERSC-PARATEC	x	0	x	0	D
SpecFEM3D	x	0	x	0	D
Jacoby	x	0	x	0	D
Ray2mesh	7	2	0	0	SD
Ray2mesh-MS	4	2	1	3	ND

Applications Analysis

Results

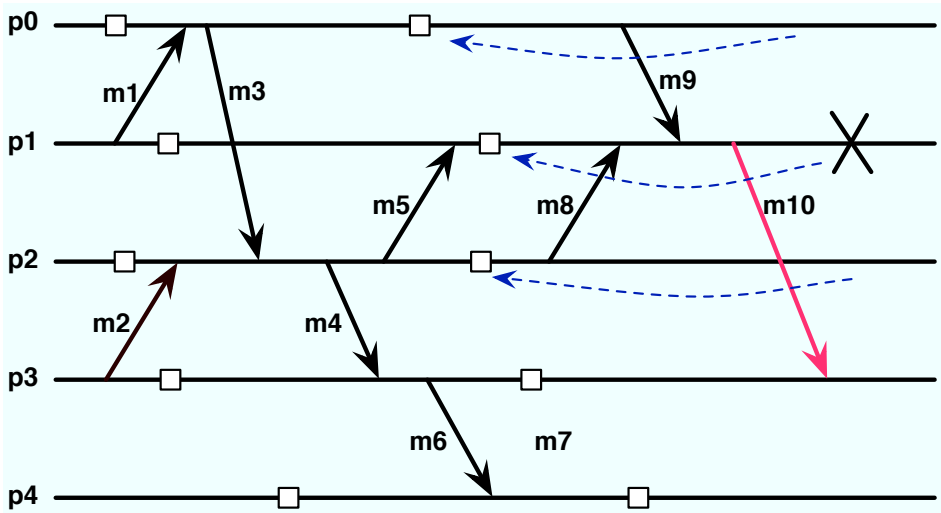
- 44,5% are deterministic
- 48,1% are send-deterministic
- 7,4% are non-deterministic

Most analyzed applications are send-deterministic

Impact of Send-determinism

Orphan message: a message that is received but not sent

- send-determinism $\rightarrow$ all message are re-sent: Orphan messages are always re-sent
- Message order for non-dependent messages has no impact on message send: No need to keep sequences numbers

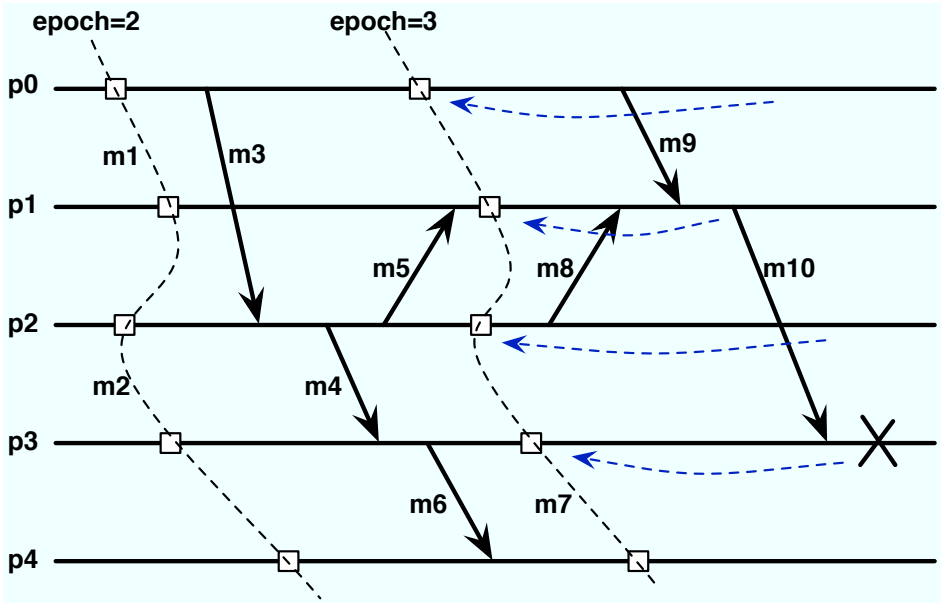


Uncoordinated checkpointing for send-deterministic MPI applications

- Uncoordinated checkpointing
- No message logging
- Message delivery order not forced during recovery

Upon Crash

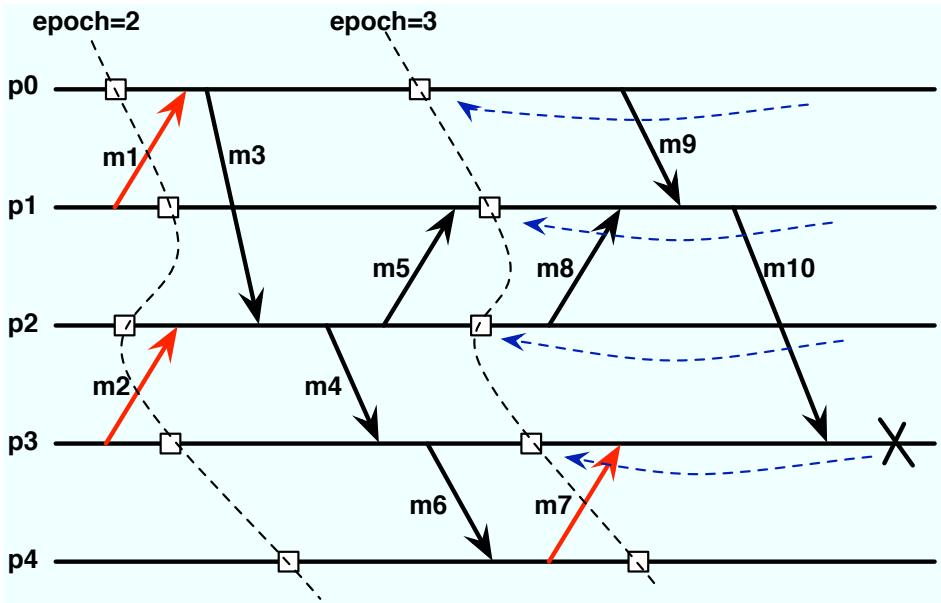
- Failed processes roll back to their most recent checkpoint
- Every process that sent a message to a failed process roll back



Uncoordinated checkpointing for send-deterministic MPI applications

No logged messages $\rightarrow$ Risk of domino effect

- Epoch: checkpoint sequence number
- Log messages from the past to the future:
  - Send ACK containing the epoch on the reception of a message
  - Compare ACK epoch and sender epoch: log if greater

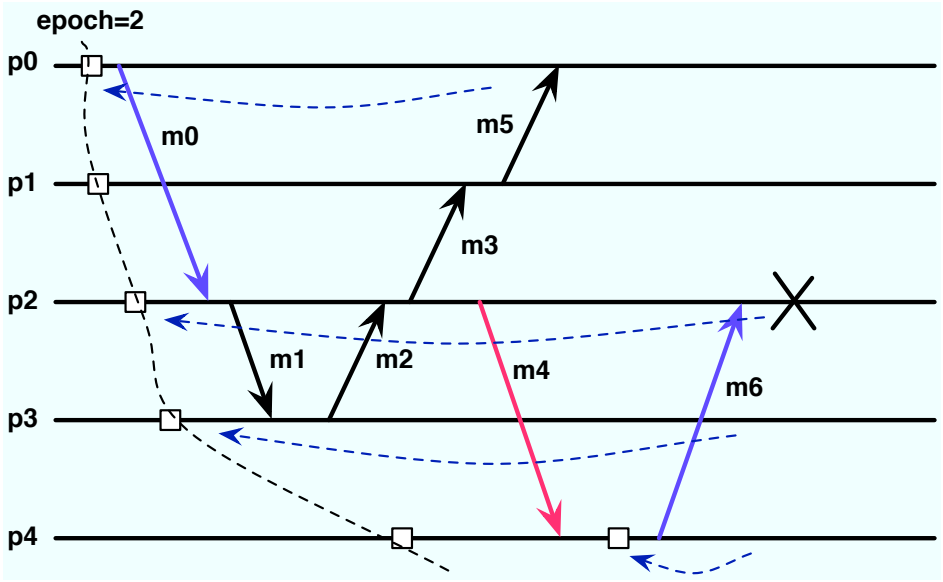


Uncoordinated checkpoint for send-deterministic MPI applications

Causality Problem No constraints on replayed messages order after a failure
But Causality order has to be ensured

Problem

1. Rollbacks follow causal dependency paths
2. Logged messages and checkpoints break these paths
3. Processes do not roll back to receive orphan messages
4. Processes do not have enough information to order causally dependent messages



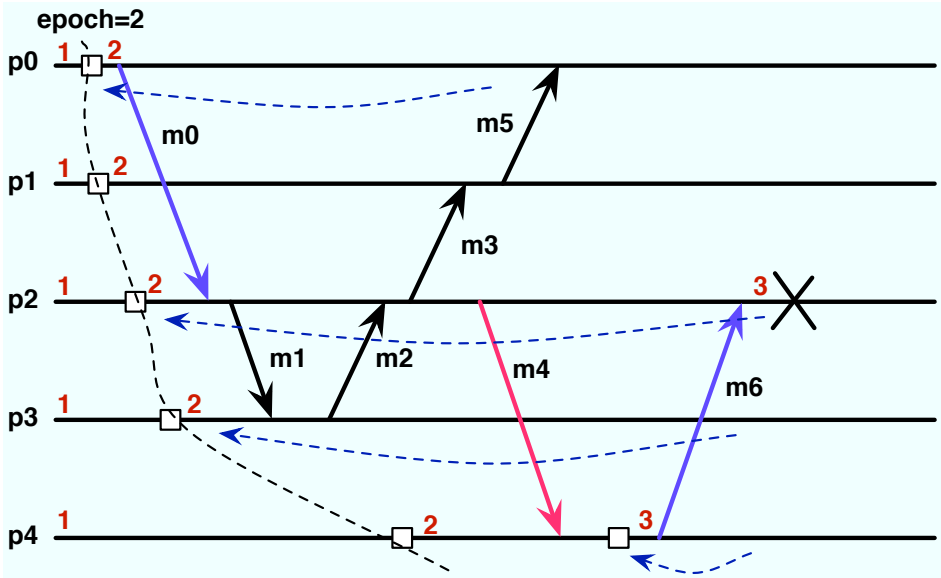
Uncoordinated checkpoint for send-deterministic MPI applications

Recovery: Causality Problem

Phase number: Messages in a causal path that is not interrupted by a checkpoint or a logged messages are in the same phase

- Piggyback phase number on each message
- checkpoint: increment phase number
- logged messages: update and increment receiver's phase number if smaller
- non-logged message: update receiver's phase number

Upon Recovery: Messages sent according to phase numbers



Uncoordinated checkpointing for send-deterministic applications

Advantages

- No global rollback
- No domino effect
- No need to log all messages
- Easy garbage collect

Drawbacks

- Global coordination for recovery

Implementation

Implementation of our prototype in MPICH2:

- Communication management in Nemesis (driver communication device):
  - ACK management
  - Message logging
  - Implementation on TCP and Myrinet 10G
- Rollback/Recovery management in Hydra (MPICH process manager)
  - Uncoordinated process checkpointing
  - Failure detection (processes failure only)
  - Computation of the set of processes to rollback
  - Processes restart (ongoing work)

Performance evaluation

- NetPipe on 2 nodes: evaluation of bandwidth and latency on Myrinet 10G
- CLASS D NAS Benchmark 3.3 Performance on Myrinet 10G
- Number of rolled back processes and logged messages on CLASS D NAS Benchmark on 128 processes

Experimental setup

- 45 nodes
- 2 Intel Xeon E5440 QC (4 cores) processors
- 8 GB of memory
- 10G-PCIE-8A-C Myri-10G NIC
- Linux with kernel 2.6.26

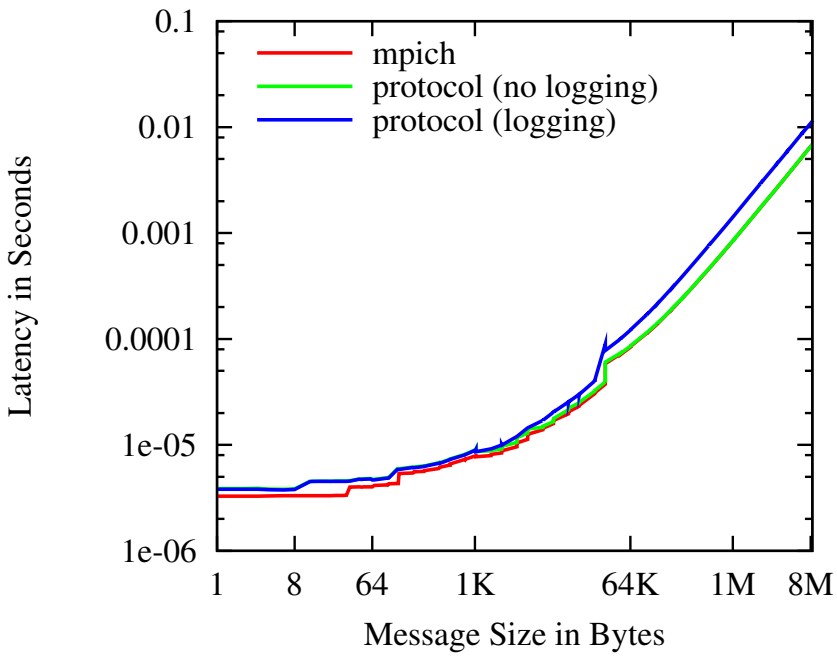
Performance evaluation

Implementation details

1. ACK implementation:
   - Small messages ($<$ 1KB):
     1. ACK sent only when the message should be logged
     2. Messages are copied until the ACK is received
   - Big messages: ACK sent for each message
2. Rolled back processes: Offline computation of set of processes to roll back according to epoch number

Latency and bandwidth

- Small overhead (15%) for small messages → Management of piggybacked data
- Impact of 39% on the bandwidth for big messages → Extra copy of message



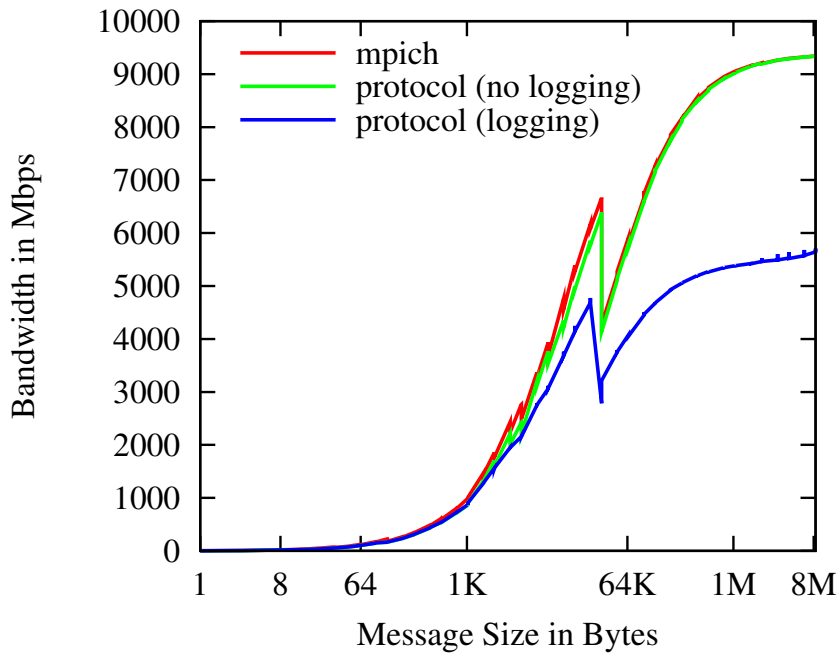


Figure: Myrinet 10G Ping-Pong Performance

Overhead on failure free execution

- Almost no impact without logging
- At most 5% overhead when all messages are logged

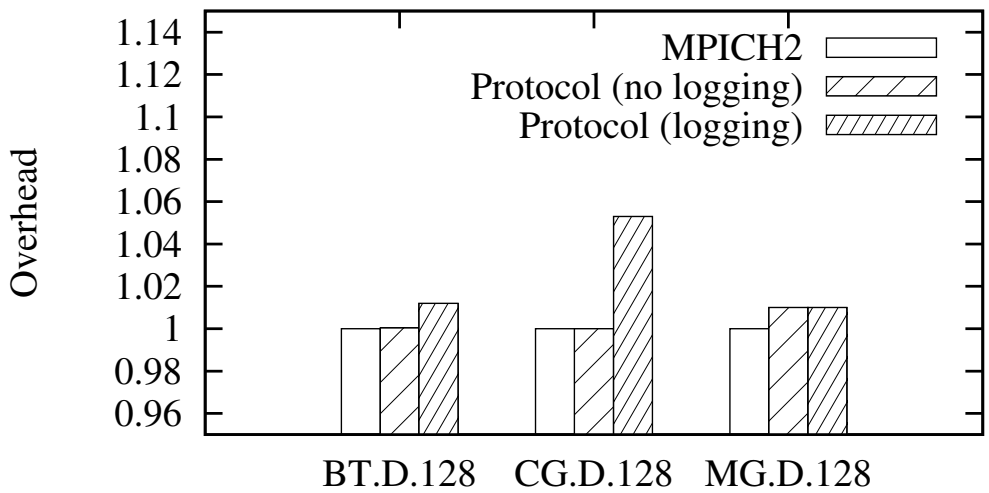


Figure: NAS Performance on Myrinet 10G

Number of rolled back processes

Evaluation of the number of processes to rollback: almost all processes rollback

- Random uncoordinated checkpoint is not always the right solution
- Take communication patterns into account
  $\rightarrow$ processes often communicate with their neighbors

Idea

- Create cluster of processes to limit the numbers of rollback
  $\rightarrow$ Force message logging by using different epochs for different clusters
- Carefully choose clusters to limit the number of logged messages

Process clustering example

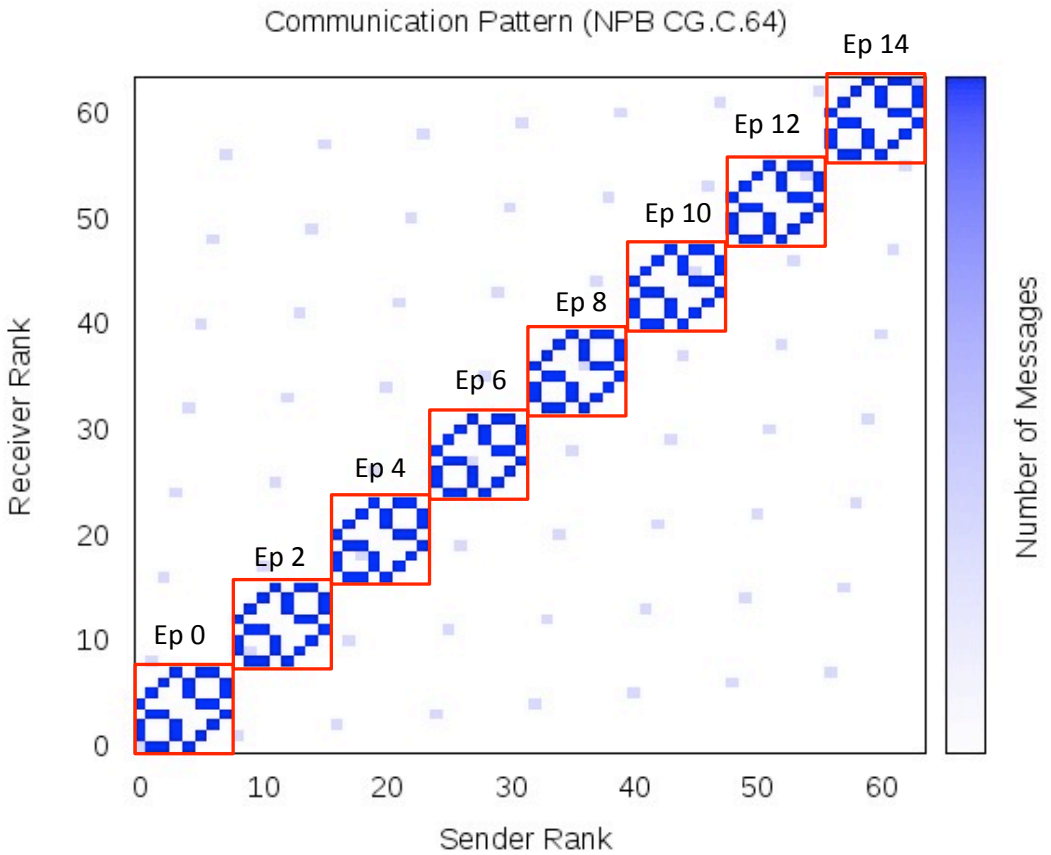


Figure: Communication density graph and clustering in CG

Number of logged messages and rolled back processes

- Only a subset of messages are logged
- Only a subset of processes rollback
- Small cluster size $\rightarrow$ few rolled back processes
- Big cluster size $\rightarrow$ few logged messages

Tradeoff between number of logged messages and number of rolled back processes

Cluster	4		8		16	
	%log	%rl	%log	%rl	%log	%rl
MG	9.5	62.5	17.1	56.3	25.4	42.1
LU	10.35	62.5	24.14	56.3	25.9	42.1
FT	37.3	62.5	43.6	56	46.8	53
CG	2.9	62.5	3.4	56.3	15	43.8
BT	13	62.6	25.2	56.4	36.7	53.3

Table: Number of logged messages and rolled back processes according to cluster size on class D NAS Benchmark for 128 processes

Conclusion and future work

Conclusion

- Presentation of Send-determinism
- Presentation of an uncoordinated checkpointing protocol using send-determinism
  - No global restart
  - Only a subset of logged messages
- Protocol performance:
  - Almost no overhead on latency
  - 39% overhead on bandwidth for big logged messages
  - Small impact on application performance
- Clustering solution based on communication pattern:
  - Number of rolled back processes approaching 50%
  - Small percent of logged messages

Conclusion and future work

Future work

- Evaluation of recovery time
- Further study on the association of send-determinism and clustering
- Impact of send-determinism on other rollback/recovery protocols