Joint Laboratory for Petascale Computation

Distributed recovery for send-deterministic HPC applications

Tatiana V. Martsinkevich, Thomas Ropars, Amina Guermouche, Franck Cappello

Fault-tolerance in HPC applications

- Number of cores on one CPU and number of CPU grows
- Can expect frequent hardware failures

- What fault tolerance protocol to use in large scale systems?

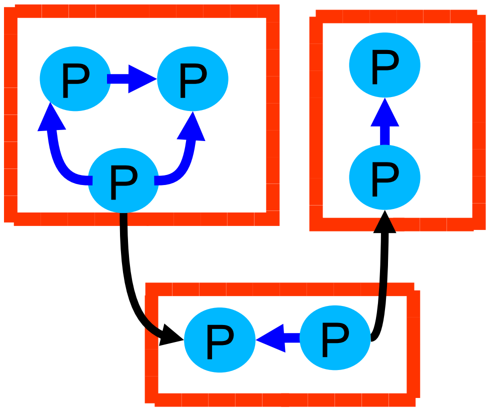
- Coordinated checkpointing, message logging, etc. protocols don't scale well as is

- For message passing applications hybrid protocols are the most promising

- Hierarchical rollback-recovery protocols

Hierarchical rollback-recovery protocols

- Goal: failure containment
- Divide ps-s in clusters
- Inside cluster: coordinated checkpointing protocol
- Between clusters: message logging protocol
  - Assume sender-based message logging
- Clustering algorithm should balance:
  - Number of ps-s in cluster (for coordinated checkpointing protocol)
  - Number of clusters (for message logging protocol)
- Upon failure:
  - When a ps fails all ps-s in the same cluster rollback and restart
  - Others re-send messages to rolled back ps-s



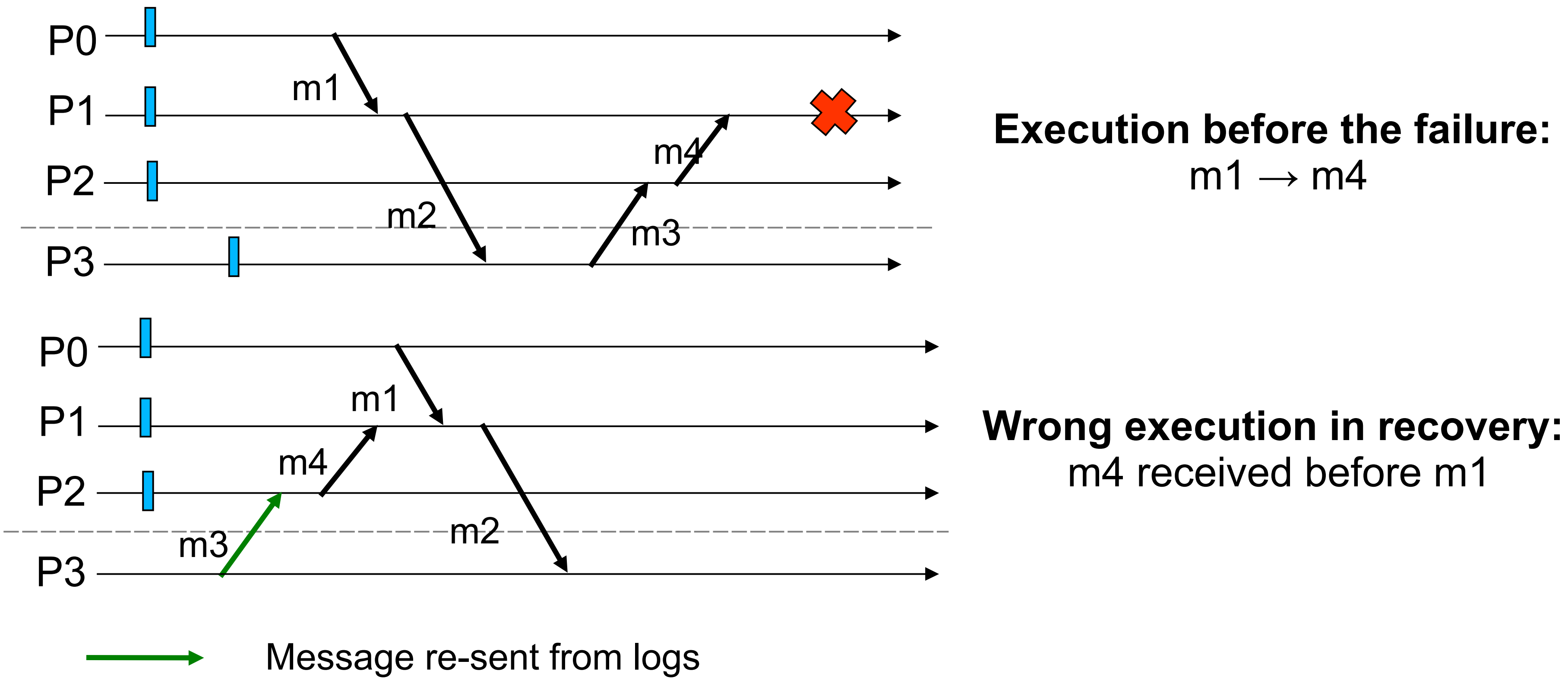
Recovery with hybrid RR protocols

- Focus is on Failure-free performance vs. Provision of enough data to be able to recover
    - ➔ What we need to log to be able to recover?

But what about optimizing performance of the recovery?

Recovery issues

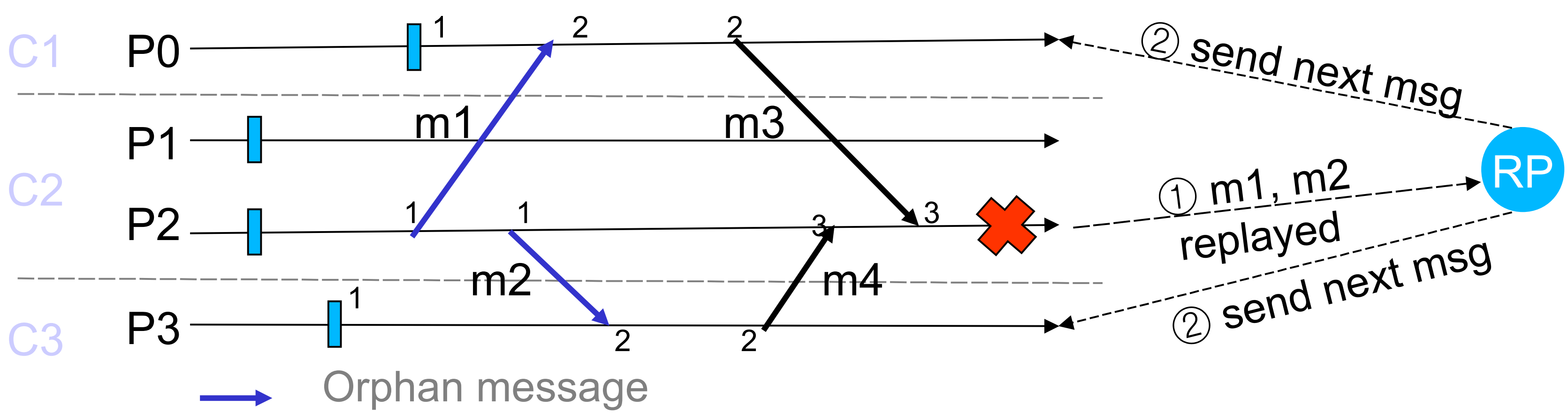
- Preserve causal dependency between messages



- Guarantee replay of *orphan* messages (m2)
  - otherwise execution path is not the same anymore

Background & Motivation: Recovery in HydEE

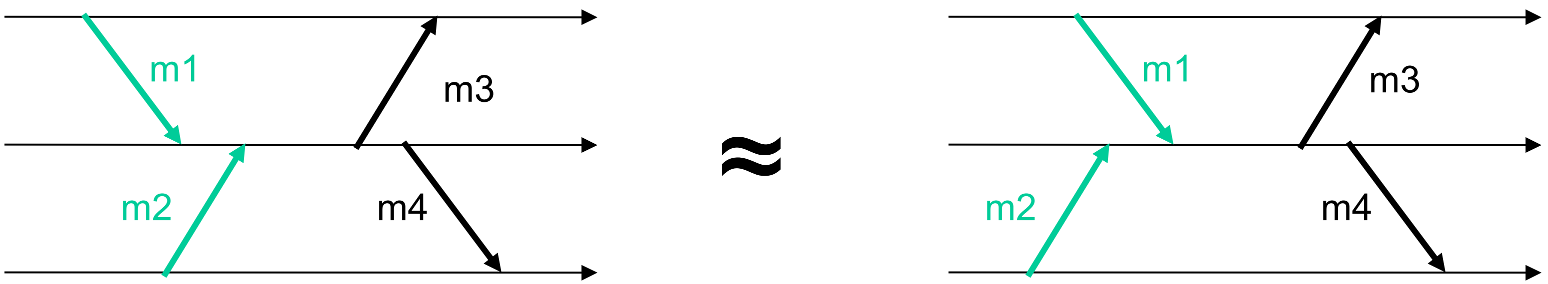
- HydEE – hierarchical rollback-recovery protocol
  - Attaches phase numbers to messages to describe causal order
- Separate recovery process controls the recovery
- It has the info about phases of logged and orphan messages
- It ensures the causal order or messages sends



- Orphan message replay is guaranteed by *send-determinism*
- Not scalable!

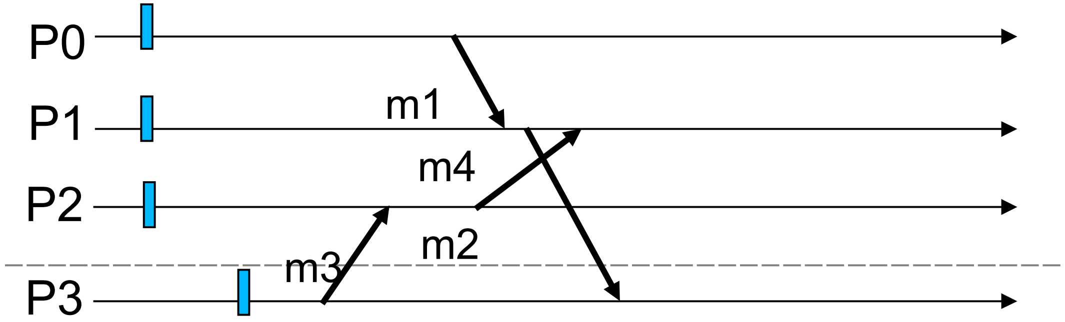
Prerequisite: Send-deterministic application

- In any correct execution:
  - Same messages are always sent in the same order
  - The reception order has no impact on the execution



Distributed recovery: principle idea

- Rolled back ps notifies everyone about the date from which it restarts
  - ps-s in the same cluster roll back too and notify everyone
  - ps-s in other clusters compute what messages to re-send and start re-sending one by one to recovering ps-s
- Replay of orphan messages: guaranteed by send-determinism
- Causal dependency? Correct order of receives?



Receive order

- Named point-to-point communication OK (assume FIFO)
- Problem arrises with anonymous reception calls

MPI_Recv(..., MPI_ANY_SOURCE, ...)

Next message selection:

? Is this a message re-sent to me from logs?

? Is this a message to be generated by another restarted PE and I need to wait for it ?

- Some additional info is necessary for message matching

match by communication pattern

Communication patterns

- Confines matching send and receive calls
- Has unique id and a counter
  - id and counter attached to every outgoing message
  - match attached value to local value upon receive

Counter++ every time we loop back here

```
for( int i = 0; i< nb_loop; i++){

  for(j=0; j < nprocs; j++) {
   MPI_Irecv( msg1, ... , MPI_ANY_SOURCE,
        tag0, ... );
   MPI_Isend( msg1, ... , j, tag0, ...);
  }
  MPI_Waitall();
  MPI_Barrier( MPI_COMM_WORLD );

}
```

Pattern “A”

Communication patterns detection

- Automatically during runtime ×
  - Too difficult to detect matching send and receive calls in the code
- Manual ○
  - Programmer adds special function calls in the code with anonymous receives

DECLARE_PATTERN(name) – declare new pattern and init its counter

BEGIN_ITERATION(name) – increment counter on every call

END_ITERATION(name) – end of comm pattern

Communication patterns detection

- Sender and receiver increment counters simultaneously
- During receive match rank, tag, pattern id and counter

```
NEW_PATTERN( ”A” ) ;
for( int i = 0; i< nb_loop; i++){
  BEGIN_ITERATION ( ”A” ) ;
  for(j=0; j < nprocs; j++) {
   MPI_Irecv( msg1, ... , MPI_ANY_SOURCE,
tag0, ... );
   MPI_Isend( msg1, ... , j, tag0, ...);
  }
  MPI_Waitall();
  MPI_Barrier( MPI_COMM_WORLD );
}
END_ITERATION( ”A” ) ;
```

Protocol: failure-free execution(1)

- Attach pattern id and counter value to each outgoing message
- Log outgoing messages and ...

What else is necessary to restore a correct execution?

Protocol: recovery

Recovering ps:

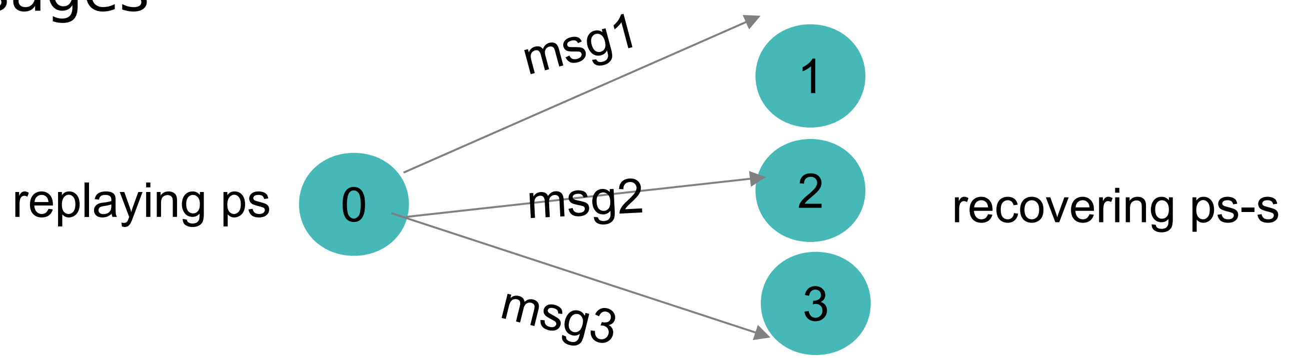
* execute normally

Match pattern id and counter for incoming messages

* don't send inter-cluster messages for real

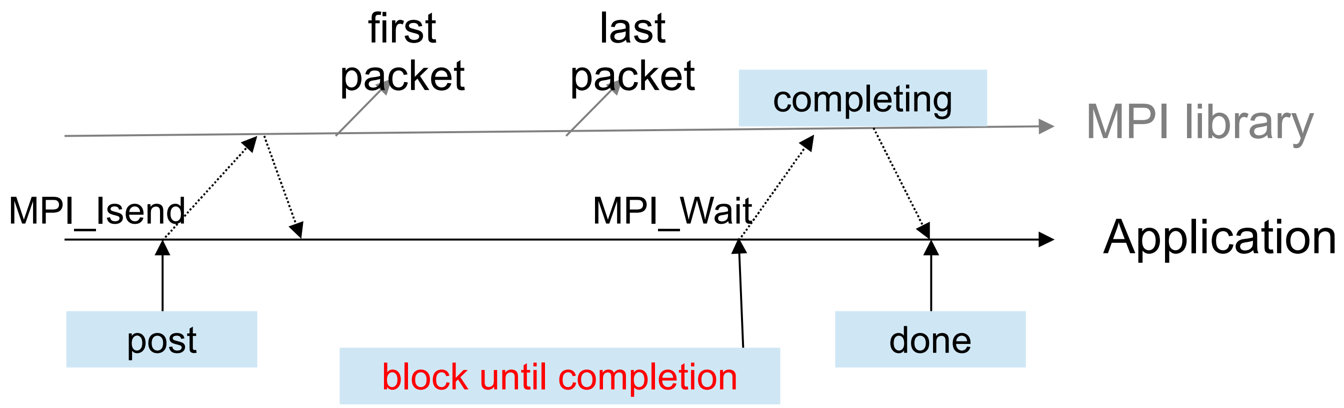
Replaying ps:

* Resend messages on each channel for which we have logged messages



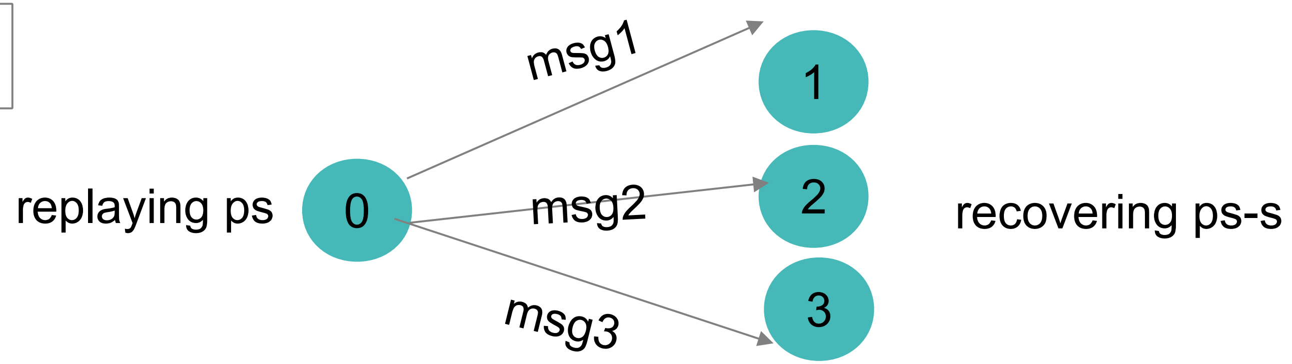
Protocol: recovery

Sending (and receiving) consists of several events



Completion order (during failure free)

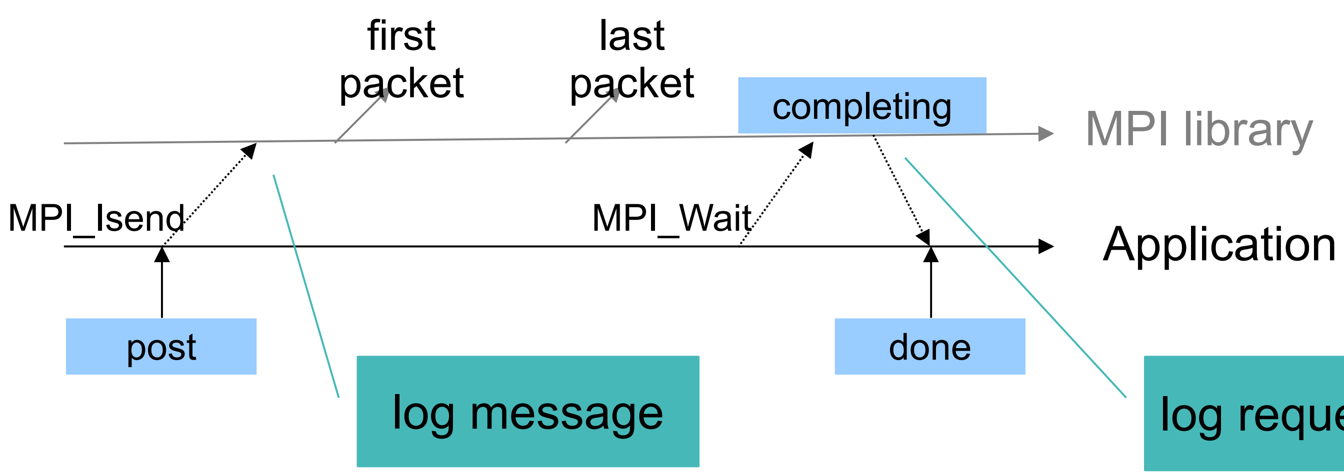
msg3
msg1
msg2



If we "wait" in wrong order in replay we can potentially block forever

Protocol: failure-free execution(2)

- Attach pattern id and counter value to each outgoing message
- Log messages and the **order of request completion of inter-cluster messages**



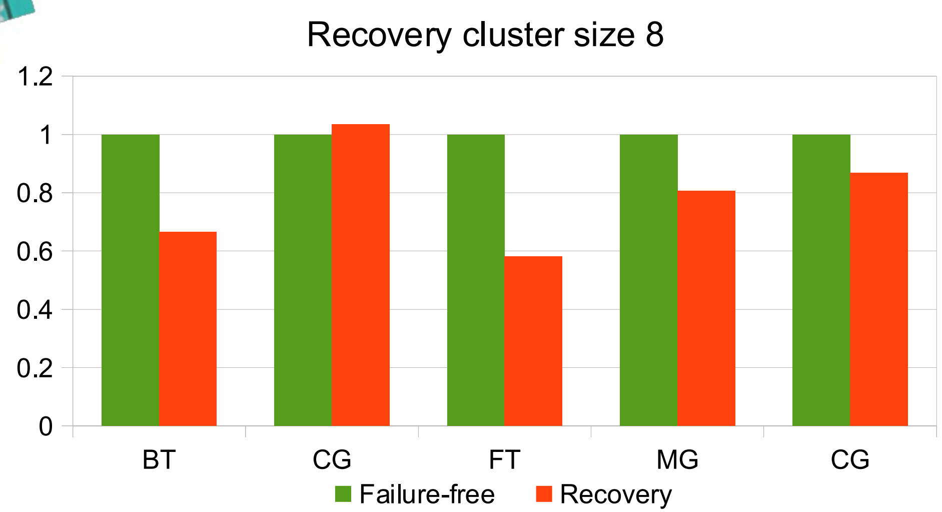
Testing

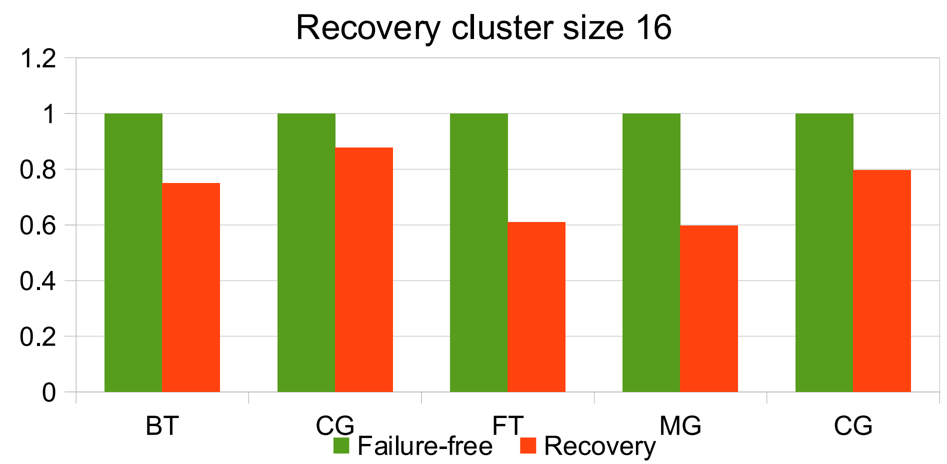
- NAS Benchmark (nprocs=64, class="B")
- Grid5000 (Nancy:graphene)
  - 1 CPU Intel@2.53GHz, 4 cores/CPU, 16GB RAM
  - Infiniband-20G (Mellanox Technologies MT26418)
- Clustering tool
- Recovery / failure free for different cluster sizes

Expect speed up from:

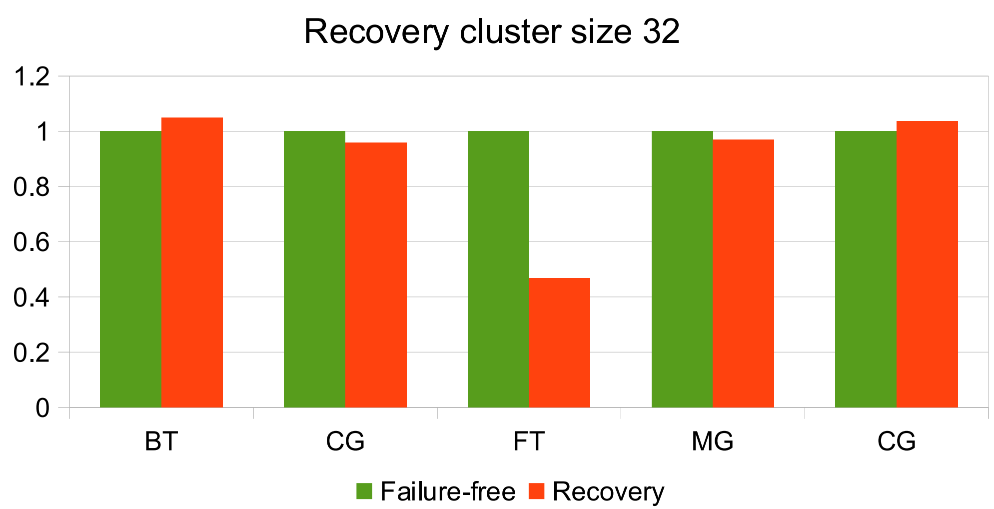
- Recovering ps doesn't send inter cluster messages
- Replaying ps deliver message earlier than recovering ps does receive call

Early results





Sometimes recovery is slower than failure free execution, hmm...



Conclusions & Future work

- More analysis needed to understand what impacts recovery speed
  - Number of channels?
  - Size of messages?
- What will happen on larger scale?
- Can we do better?
  - Send first *n* messages on channel and only then start completing

Collaboration

- Ability to do partial process restart with MPICH2?
- Communication pattern detection during compilation?

Thank you
Questions?