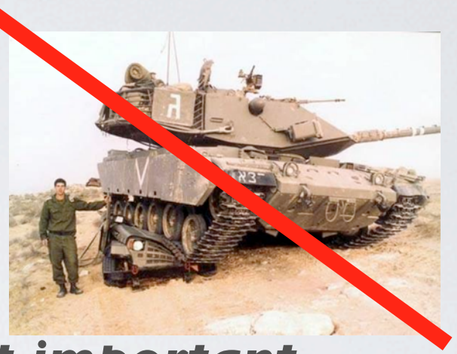
PROCESS FAILURE MITIGATION IN THE MPI 3.0 STANDARD

Aurelien Bouteiller, Thomas Herault

MPI Forum meeting - Chicago, March 5, 2012

STATEMENT OF INTENT



Failures are common enough that we care…

…but rare enough that failure free performance is most important

- The proposal's focus is to empower MPI *users* with *lightweight, efficient and extensible* failure mitigation
  - *Lightweight*: On reliable systems, application code and performance should be unchanged
  - *Efficient*: Applications that need low recovery consistency (e.g. master/worker) should pay minimal cost
  - *Extensible*: The standard MPI constructs should enable users to define their own *portable*, stronger consistency models (e.g. RTS proposal, transactions, synchronous phases, ...)
- No excessive, unrealistic requirements from the MPI implementors

CONSISTENT FAILURE NOTIFICATION: DRAWBACKS

- Game: what are the hidden requirements in the strongly consistent approach below?

```
For(i=0; i<n; i++) {
   value = compute();
   rc = MPI_Bcast(comm, value);
   if(rc != MPI_SUCCESS)
        recover();
}
```

the Bcast «agrees» on the return code: O(n^2)

```
while(morework) {
      rc = MPI_Recv(..., myrank-1, ...);
      if(rc != MPI_SUCCESS) recover();
      rc = MPI_Send(..., myrank+1, ...);
      if(rc != MPI_SUCCESS) recover();
}
```

A failure that happened on another process must be reported during an unrelated operation: asynchronous failure propagation and many special cases on the latency critical path

WEAK CONSISTENCY

A correct MPI code does not deadlock because of process failures

- Every operation must ***return control to the user if completion may be impossible*** due to the failure of a process *involved* in the operation
  - If an operation can complete, it does not raise an error
  - Rationale: if an operation deadlocks, the user cannot recover from the failure.
- An ***error indicates that an operation couldn't complete locally*** because of a process failure
  - Failure discovery is only a side effect
  - Succeeding an operation is insufficient to reason on the status of other processes
    - Rationale: strong consistency is expensive

RESOLVING WEAK CONSISTENCY

A correct MPI code does not deadlock because of process failures

- **Conditional branching after an error** is a typical use case
- What was a **correct code may diverge** into an incorrect deadlocking code after a failure (because of weak consistency)
  - MPI must empower the users with a routine to resolve such scenarios

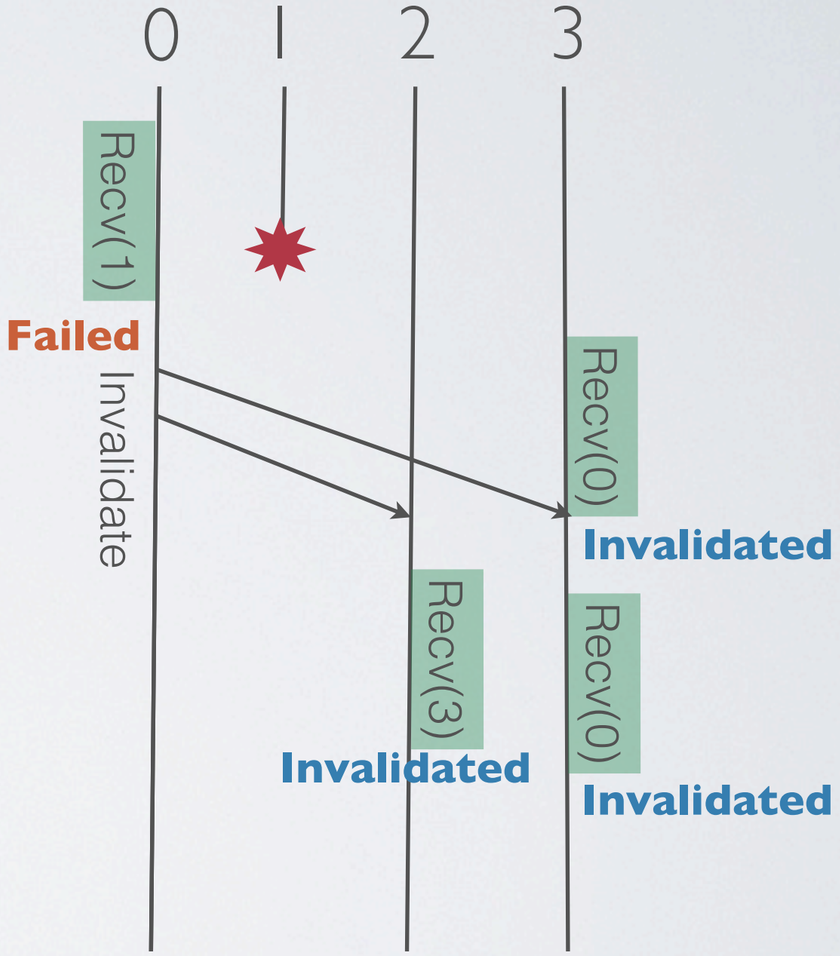
```
while(morework) {
      rc = MPI_Recv(..., myrank-1, ...);
      if(rc != MPI_SUCCESS) recover();
      rc = MPI_Send(..., myrank+1, ...);
      if(rc != MPI_SUCCESS) recover();
}
```

A failure that happened on another process is not reported during an unrelated operation, if a process goes into the conditional recover, some processes may deadlock

0:Recover() ; 1:Dead ; 2:Recover() ; 3:MPI_Recv(2)

MPI_COMM_INVALIDATE

- Weak consistency diverges into possible deadlocks behavior after a failure
- **MPI_COMM_INVALIDATE(comm)** is the **tool** provided by MPI **for the users to resolve deadlocks**
  - **non-collective** function (like MPI_Abort), it notifies all processes in comm of the invalidation
  - Any **ongoing operation** on *comm* raises an exception MPI_ERR_INVALIDATED and **returns immediately** (resolves deadlocks)
  - Future operations on comm are **local and keep raising** MPI_ERR_INVALIDATED



RULES FOR FAILURE NOTIFICATION

- An **operation cannot complete** because a participating peer is dead: **MPI_ERR_PROC_FAILED**
- A **non blocking completion *may* not complete** because a peer is dead: **MPI_ERR_PENDING**
  - during the completion, after an iRecv(MPI_ANY_SOURCE) which have not been matched yet
  - if the matching has happened, the peer is well known, the library can discard the request and raise MPI_ERR_PROC_FAILED instead
- **After an error** has been raised, the **state of communication objects is unchanged** (i.e. operations with failed processes continue to raise errors, operations that can complete succeed)
- **MPI_Success** does not mean that no failure occurred, or that a collective operation succeeded on all ranks; it **guarantees only that the operation had the expected local behavior** (i.e. buffers can be used)

FAILURE MITIGATION
IGNORING FAILURES

- In some case (i.e. master/slave), failed process can just be ignored
  - Point-to-point between non-failed peers work
  - **MPI_Comm_failure_ack(comm)**
    - Updates the list of (locally) acknowledged failures with (at least) all failures that have already raised an exception
    - Unmatched any-source reception do not raise exceptions anymore, unless a new (unacknowledged yet) failure happens
  - **MPI_Comm_failure_get_acked(comm, failedgrp)**
    - Returns the group of (locally) acknowledged failures (since startup)
    - The group is invariant as long as Failure_ack is not called again on the communicator

FAILURE MITIGATION
GETTING A WORKING COMMUNICATOR

- **MPI_Comm_shrink(comm, newcomm)** can be called to obtain a new functional communicator
  - Only on an invalidated communicator
  - **Collective** over the alive processes
  - An agreement is made among living processes on a new communicator ***newcomm*, similar to *comm*, but without the failed processes**
  - Future operations on *comm* still raise MPI_ERR_INVALIDATED; it may be freed though
  - *newcomm* can be used as a replacement for *comm*

FAILURE MITIGATION
RESTORING CONSISTENCY

- Processes may have a divergent view of the global state, even after deadlocks are resolved
  - **MPI_Comm_agreement(comm, flag)** enables application to reason about the global state
    - **Completes despite failures or invalidation** of the communicator
    - **Collective** between alive processes
    - **flag is the logical AND between provided values**, dead processes do not participate
- Can be used to bring back strong consistency when needed by setting flag to (rc == MPI_SUCCESS)
- Can be used by application to reason about other results, such as termination, soft errors, ...

RMA/FILES

- Same semantic for returning an error: only if the operation cannot satisfy its local specification
- New functions for windows:
  - **MPI_WIN_INVALIDATE(win)**
  - **MPI_WIN_GET_FAILED(win, failedgrp)**
    - Get the group of processes know to have failed in win
    - Can be used to continue performing active target operations and locks despite failures
- New functions for files:
  - **MPI_FILE_INVALIDATE(fh)**
- When a file or a window is invalidated, users should use the «parent» communicator to reconstruct a new file/window (or to agree on the global state)

THREAD SAFETY

- **Communication objects are invalidated forever.** A thread getting notified of an invalidation does not «eat» the invalidation, other threads will get it too.
- A ERR_PROC_FAILED error means that a particular operation could not complete. **If a failure impacts operations ongoing in concurrent threads, each will raise an error.**
- An ERR_PENDING error does not remove operations from the matching queue. **Matching order between concurrent threads is not disrupted.**
- **It is safe to call failure_get_acked and failure_ack concurrently**: the group of acknowledged failures is never trimmed, no knowledge is suppressed from a thread.
- However, It is the **responsibility of the user to synchronize threads acknowledging failures before resuming concurrent ANY_SOURCE receptions.**

BUT, MY APPLICATION NEEDS STRONG CONSISTENCY

- MPI_Comm_agreement can be used to reason on the success of a phase
  - Communicator creations are obvious targets for uniform, consistent return of errors
  - Transactions, Run Trough Stabilization, etc...
- All these approaches can be implemented in a **portable** manner, based only on standardized construct
- We encourage the development of MPI **user-level helper libraries to provide different/stronger consistency models** to applications

AT A GLANCE

- **Weak consistency of failure notification**
  - An error denotes that an operation could not complete its local specification
  - An operation focuses only on its own peers and communicators; no asynchronous progress, no asynchronous failure detection, implementation is concise
- **After a failure, semantic is unchanged**
  - Non-failed ranks can still communicate, repeating operations with failed ranks return the same errors again
  - MPI_Comm_failure_ack/get_acked allows pt-2-pt applications to ignore failed ranks

- **MPI_XXX_Invalidate is used to solve deadlocks resulting from diverging views of failures**
  - An invalidated communication object is broken forever (collective, pt-2-pt, etc).
  - Invalidation is per-communicator; composition of libraries is possible
  - MPI_Comm_shrink permits rebuilding a *new* working communicator
- **MPI_Comm_agreement enables reasoning on a shared condition** among all surviving processes
- Windows and Files share the same weak semantic, agreement on an overlapping communicator can be used to provide stronger semantics

IMPLEMENTORS CORNER

- Requirement: no operation that involves a failed process deadlocks
  - Operations that can complete are allowed to complete normally (hence, they do not need to check for error before going straight to the latency optimized path)
  - If a process does not communicate with another process, it does not need to detect its failure (no jitter prone asynchronous background failure detection, an operation probes only the status of participating peers)
  - Collective operation must complete (with error or success on different ranks). There is a simple implementation, based on unchanged, existing collective modules
- Most work has already been done for the high quality «Error return» mode in current implementations
- MPI_Comm_invalidate is similar to MPI_Abort, some code may be reused
- We have already implemented an agreement, it is short and easy to understand (to be released under BSD license)

PERFORMANCE

- Many «error return» MPI implementation already exist (high quality MPI2)
  - In many case, the proposal is expected to have a similar impact