Fault tolerant algorithms for a fault tolerant MPI

Cupseli Project Introductory Talks

Thomas Herault, TOPAL Inria, thomas.herault@inria.fr

December 8, 2025

Thomas Herault

Distributed Systems	Fault Tolerance in MPI	Fault Tolerance with MPI	Linear Algebra	Task Systems
Self-Stabilization	Rollback Recovery	Consensus, Failure Detection	Resilient LA	Resilient Task Systems

Motivation: Fault Tolerant Approaches in HPC

- Checkpoint/Restart (C/R) is the most widely used approach for fault tolerance in HPC
- However, C/R has limitations:
  - High overhead due to frequent checkpointing
  - Scalability issues as system size increases
  - Inefficiency in handling transient faults
- Alternative approaches are needed to address these challenges

- There are classes of algorithms that can inherently tolerate faults
- Other algorithms can be adapted to be fault-tolerant
- We need to explore these algorithms and their integration with MPI

What is the status of FT in MPI (2.0, 3.0?)

- Total denial

  "After an error is detected, the state of MPI is undefined. An MPI implementation is free to allow MPI to continue after an error but is not required to do so."

- Two forms of failure management
  - Return codes: all MPI functions return either `MPI_SUCCESS` or an error coode related to the error class encountered (e.g., `MPI_ERR_ARG`)
  - *"High quality implementations may provide mechanisms to continue execution after certain errors have occurred, but such mechanisms are outside the scope of this standard."*
  - Error handlers: a callback automatically invoked when an error is detected
  - Status of the MPI library after an error is returned: **undefined**

Introducing User-Level Failure Mitigation (ULFM)

- An extension to the MPI standard that provides a set of interfaces for fault tolerance
- Key features:
  - Specify behavior of the MPI library upon process failures
  - Provides failure detection: mechanisms to detect and notify process failures
  - Agreement operations: collective operations that can handle failures
  - Communicator repair: functions to create new communicators excluding failed processes
- Allows applications to implement their own fault-tolerance strategies

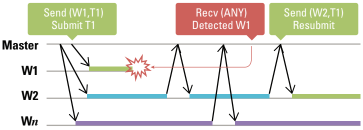
- About 40% of ULFM is now part of the official MPI standard (MPI 5.0)
- It is fully implemented in several MPI libraries (e.g., Open MPI, MPICH)
- It provides a foundation for building fault-tolerant applications in HPC

ULFM at a glance

ULFM provides targeted interfaces to empower recovery strategies with adequate options to restore communication capabilities and global consistency, at the necessary levels only.

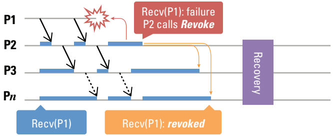
CONTINUE ACROSS ERRORS

In ULFM, failures do not alter the state of MPI communicators. Point-to-point operations can continue undisturbed between non-faulty processes. ULFM imposes no recovery cost on simple communication patterns that can proceed despite failures.



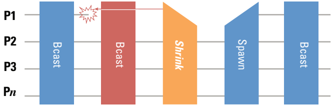
EXCEPTIONS IN CONTAINED DOMAINS

Consistent reporting of failures would add an unacceptable performance penalty. In ULFM, errors are raised only at ranks where an operation is disrupted; other ranks may still complete their operations. A process can use MPI_[Comm,Win,File]_revoke to propagate an error notification on the entire group, and could, for example, interrupt other ranks to join a coordinated recovery.



FULL-CAPABILITY RECOVERY

Allowing collective operations to operate on damaged MPI objects (Communicators, RMA windows or Files) would incur unacceptable overhead. The MPI_Comm_shrink routine builds a replacement communicator, excluding failed processes, which can be used to resume collective communications, spawn replacement processes, and rebuild RMA Windows and Files.



Outline

1/3 - Failure Detection and Notification

Outline

Failure detection: why?

- Nodes do crash at scale (you've heard the story before)
- Current solution:
  1. Detection: TCP time-out ($\approx 20mn$)
  2. Knowledge propagation: Admin network
- Work on fail-stop errors assumes *instantaneous* failure detection
- Seems we put the cart before the horse ☹

Resilient applications

- Continue execution after crash of one node

Resilient applications

- Continue execution after crash of **several** nodes

Resilient applications

- Continue execution after crash of **several** nodes
- Need *rapid* and *global* knowledge of group members
  1. Rapid: failure detection
  2. Global: failure knowledge propagation

Resilient applications

- Continue execution after crash of **several** nodes
- Need *rapid* and *global* knowledge of group members
  1. Rapid: failure detection
  2. Global: failure knowledge propagation
- Resilience mechanism should come for free

Resilient applications

- Continue execution after crash of **several** nodes
- Need *rapid* and *global* knowledge of group members
  1. Rapid: failure detection
  2. Global: failure knowledge propagation
- Resilience mechanism should have minimal impact

Contribution

- Failure-free overhead constant per node (memory, communications)
- Failure detection with minimal overhead
- Knowledge propagation based on fault-tolerant broadcast overlay
- Tolerate an arbitrary number of failures
  (but with bounded frequency of occurrence)
- Logarithmic worst-case repair time

Outline

Outline

Framework

- Large-scale platform with (dense) interconnection graph (physical links)
- One-port message passing model
- Reliable links (messages not lost/duplicated/modified)
- Communication time on each link:
  randomly distributed but bounded by τ
- Permanent node crashes

Failure detector

Definition

Failure detector: distributed service able to return the state of any node, alive or dead. Perfect if:

1. any failure is eventually detected by all alive nodes and
2. no alive node suspects another alive node of being dead

Definition

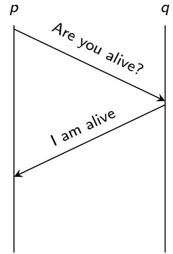
Stable configuration: all dead nodes are known to all processes (nodes may not be aware they are in a stable configuration).

Vocabulary

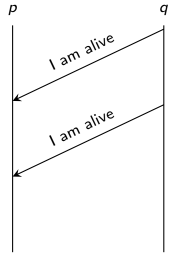
- Node = physical resource
- Process = program running on node
- Thread = part of a process that can run on a single core
- Failure detector will detect both process and node failures
- Failure detector mandatory to detect some node failures

Outline

Timeout techniques: p observes q



- Pull technique
  - Observer p sends a *Are you alive* message to q
  - ☹ More messages
  - ☹ Long timeout



- Push technique [1]
  - Observed q periodically sends heartbeats to p
  - ☺ Less messages
  - ☺ Faster detection (shorter timeout)

[1]: W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. IEEE Trans. Computers, 2002

Timeout techniques: platform-wide notification

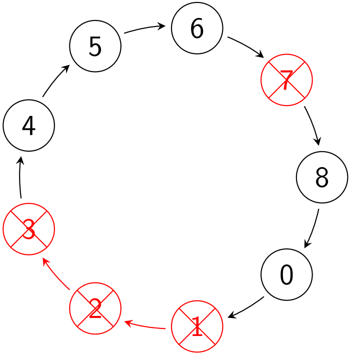
- All-to-all:
  - ☺ Immediate knowledge propagation
  - ☹ Dramatic overhead

- Random nodes and gossip:
  - ☺ Quick knowledge propagation
  - ☹ Redundant/partial failure information (more later)
  - ☹ Difficult to define timeout
  - ☹ Difficult to bound detection latency

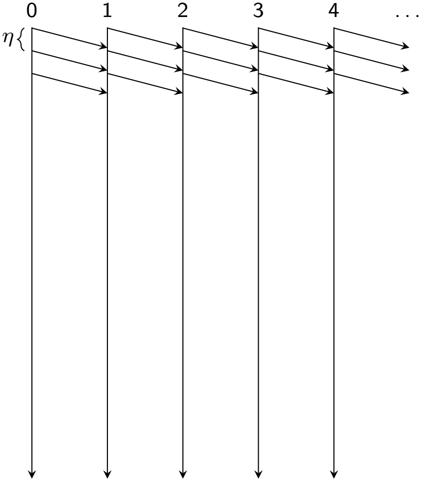
Algorithm for failure detection

- Processes arranged as a ring
- Periodic heartbeats from a node to its successor

- Maintain ring of alive nodes
  - → Reconnect ring after a failure
  - → Inform all processes

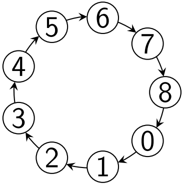


Reconnecting the ring

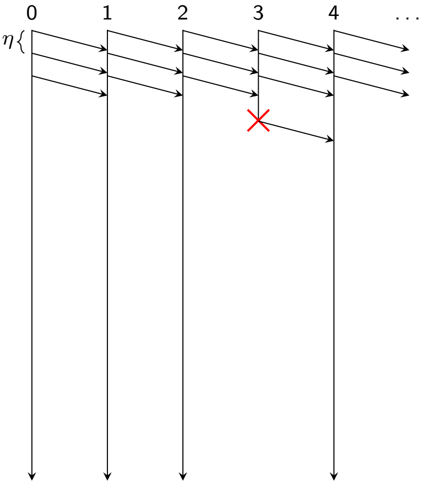


η: Heartbeat interval

→ Heartbeat

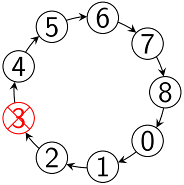


Reconnecting the ring

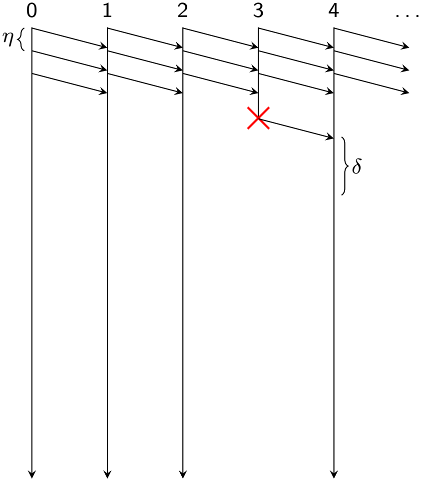


η: Heartbeat interval

Heartbeat



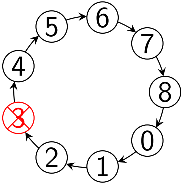
Reconnecting the ring



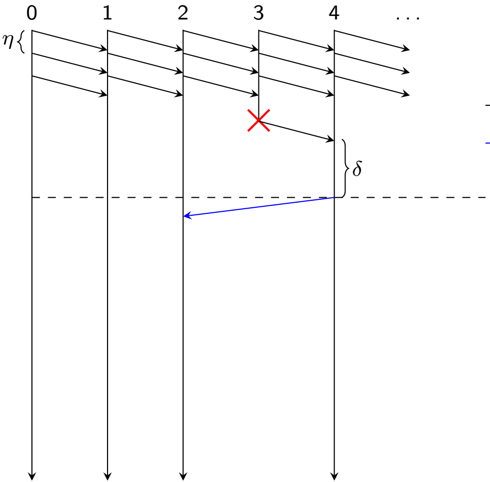
η: Heartbeat interval

δ: Timeout, $\delta >> \tau$

$\longrightarrow$ Heartbeat



Reconnecting the ring

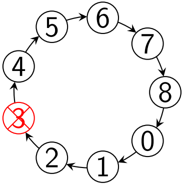


η: Heartbeat interval

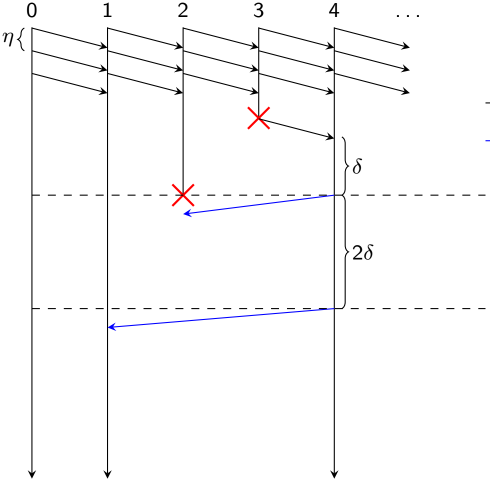
δ: Timeout, $\delta >> \tau$

→ Heartbeat

→ Reconnection message



Reconnecting the ring

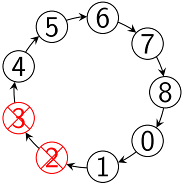


η: Heartbeat interval

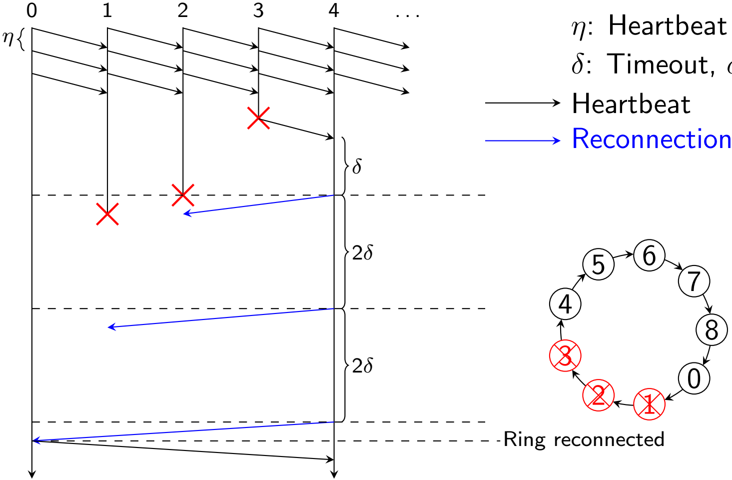
δ: Timeout, $\delta >> \tau$

Heartbeat

Reconnection message



Reconnecting the ring



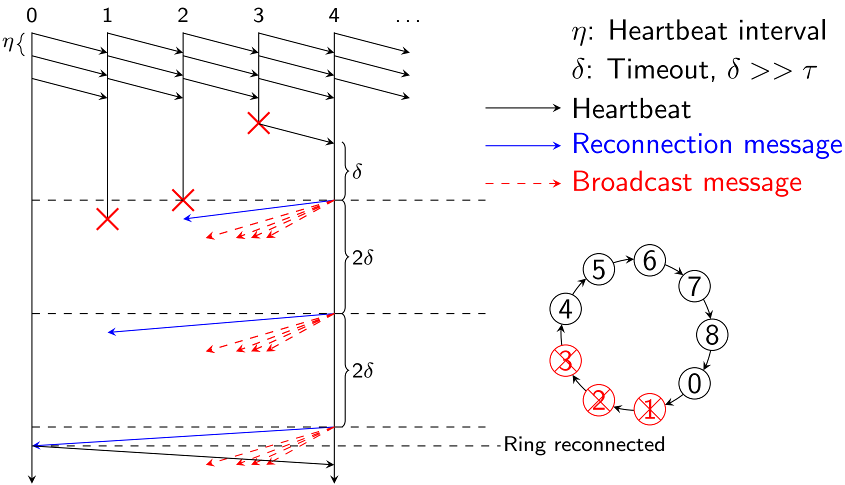
η: Heartbeat interval

δ: Timeout, $\delta >> \tau$

→ Heartbeat

→ Reconnection message

Reconnecting the ring



Algorithm

task Initialization
  $\texttt{emitter}_i \leftarrow (i-1) \bmod N$
  $\texttt{observer}_i \leftarrow (i+1) \bmod N$
  $\textsc{HB-Timeout} \leftarrow \eta$
  $\textsc{Susp-Timeout} \leftarrow \delta$
  $\mathcal{D}_i \leftarrow \emptyset$
end task

task T1: When $\textsc{HB-Timeout}$ expires
  $\textsc{HB-Timeout} \leftarrow \eta$
  Send $\textsc{heartbeat}(i)$ to $\texttt{observer}_i$
end task

task T2: upon reception of $\textsc{heartbeat}(\texttt{emitter}_i)$
  $\textsc{Susp-Timeout} \leftarrow \delta$
end task

task T3: When $\textsc{Susp-Timeout}$ expires
  $\textsc{Susp-Timeout} \leftarrow 2\delta$
  $\mathcal{D}_i \leftarrow \mathcal{D}_i \cup \texttt{emitter}_i$
  $\texttt{dead} \leftarrow \texttt{emitter}_i$
  $\texttt{emitter}_i \leftarrow \textit{FindEmitter}(\mathcal{D}_i)$
  Send $\textsc{NewObserver}(i)$ to $\texttt{emitter}_i$
  Send $\textsc{BcastMsg}(\texttt{dead}, i, \mathcal{D}_i)$ to
    $\textit{Neighbors}(i, \mathcal{D}_i)$
end task

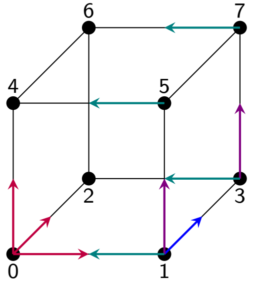
task T4: upon reception of $\textsc{NewObserver}(j)$
  $\texttt{observer}_i \leftarrow j$
  $\textsc{HB-Timeout} \leftarrow 0$
end task

task T5: upon reception of $\textsc{BcastMsg}(\texttt{dead}, s, \mathcal{D})$
  $\mathcal{D}_i \leftarrow \mathcal{D}_i \cup \{\texttt{dead}\}$
  Send $\textsc{BcastMsg}(\texttt{dead}, s, \mathcal{D})$ to
    $\textit{Neighbors}(s, \mathcal{D})$
end task

function $\textit{FindEmitter}(\mathcal{D}_i)$
  $k \leftarrow \texttt{emitter}_i$
  while $k \in \mathcal{D}_i$ **do**
    $k \leftarrow (k-1) \bmod N$
  return k
end function

Broadcast algorithm

- Hypercube Broadcast Algorithm [1]
  - Disjoint paths to deliver multiple broadcast message copies
  - Recursive doubling broadcast algorithm by each node
  - Completes if $f \leq \lfloor log(n) \rfloor - 1$ (f: number of failures, n: number of alive processes)



Node	Node1	Node2	Node4
1	0	0-2-3	0-4-5
2	0-1-3	0	0-4-6
3	0-1	0-2	0-4-5-7
4	0-1-5	0-2-6	0
5	0-1	0-2-6-7	0-4
6	0-1-3-7	0-2	0-4
7	0-1-3	0-2-6	0-4-5

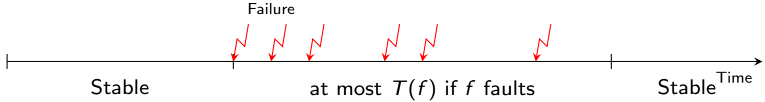
[1] P. Ramanathan and Kang G. Shin, 'Reliable Broadcast Algorithm', IEEE Trans. Computers, 1998

Failure propagation

- Hypercube Broadcast Algorithm
  - Completes if $f \leq \lfloor log(n) \rfloor - 1$ (f: number of failures, n: number of alive processes)
  - Completes within $2\tau log(n)$
- Application to failure detector
  - If $n \neq 2^l$
    - $k = \lfloor log(n) \rfloor$
    - $2^k \leq n \leq 2^{k+1}$
    - Initiate two successive broadcast operations
  - Source s of broadcast sends its current list D of dead processes
  - No update of D during broadcast initiated by s
    (do NOT change broadcast topology on the fly)

Outline

Worst-case analysis



Theorem

With $n \leq N$ alive nodes, and for any $f \leq \lfloor \log n \rfloor - 1$, we have

$$T(f) \leq f(f+1)\delta + f\tau + \frac{f(f+1)}{2}B(n)$$

where $B(n) = 8\tau \log n$.

- *2 sequential broadcasts: $4\tau log(n)$*
- *One-port model: broadcast messages and heartbeats interleaved*

Worst-case scenario

$$T(f) \leq \underbrace{f(f+1)\delta + f\tau}_{reconstruction} + \underbrace{\frac{f(f+1)}{2}B(n)}_{broadcast}$$

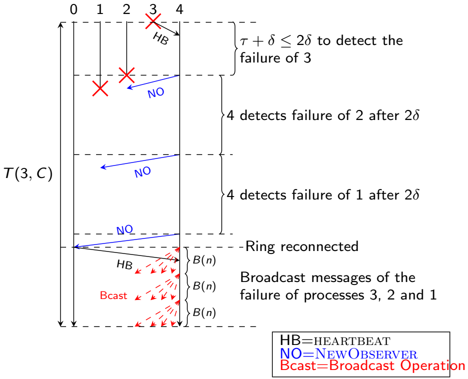
- $T(f) \leq$ ring reconstruction + broadcasts (for the proof)
- Process p discovers the death of q at most **once**
  $\Rightarrow$ $i-th$ dead process discovered dead by at most $f - i + 1$ processes
  $\Rightarrow$ at most $\frac{f(f+1)}{2}$ broadcasts
- $R(f)$ ring reconstruction time
  For $1 \leq f \leq \lfloor \log n \rfloor - 1$,

$$R(f) \leq R(f-1) + 2f\delta + \tau$$

Ring reconnection

$$R(f) \leq R(f-1) + 2f\delta + \tau$$

- $R(1) \leq 2\tau + \delta \leq 2\delta + \tau$
- $R(f) \leq R(f-1) + R(1)$ if next failure *non-adjacent* to previous ones
- Worst-case when failing nodes consecutive in the ring
- Build the ring by *"jumping"* over platform to avoid correlated failures



Worst-case scenario

$$T(f) \leq f(f+1)\delta + f\tau + \frac{f(f+1)}{2}B(n)$$

Worst-case scenario

$$(f) \leq f(f+1)\delta + f\tau \quad \frac{f(f+1)}{2} B(n)$$

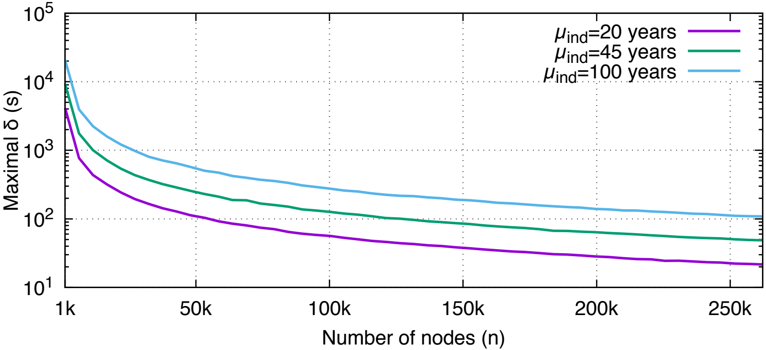
Too pessimistic!?

Worst-case scenario

1. If time between two consecutive faults is larger than $T(1)$, then average stabilization time is $T(1) = O(\log n)$
2. If f quickly overlapping faults hit non-consecutive nodes, $T(f) = O(\log^2 n)$
3. If f quickly overlapping faults hit f consecutive nodes in the ring, $T(f) = O(log^3 n)$

Large platforms: two successive faults strike consecutive nodes with probability $2/(n-1)$

Risk assessment with $\tau = 1\mu s$



$$\mathbb{P}(\geq \lfloor \log_2(n) \rfloor \text{ failures in } T(\lfloor \log_2(n) \rfloor - 1)) < 0.000000001$$

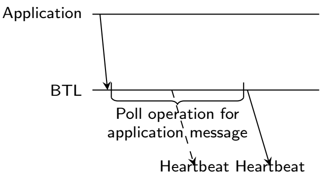
- With $\mu_{\text{ind}} = 45$ years, $\delta \leq 60s \Rightarrow$ timely convergence
- Detector causes negligible overhead to applications (e.g., $\eta = \delta/10$)

Outline

Implementation

- Observation ring and propagation topology implemented in Byte Transport Layer (BTL)
- No missing heartbeat period:
  - Implemented in MPI internal thread independently from application communications
  - RDMA put channel to directly raise a flag at receiver memory

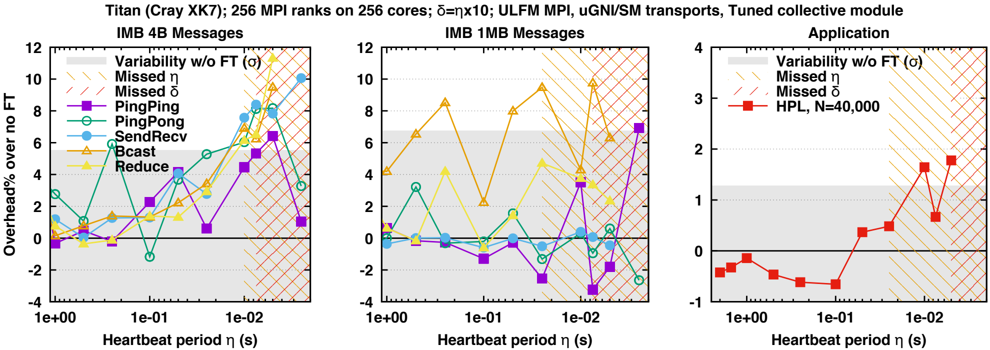
  $\rightarrow$ No allocated memory, no message wait queue
- Implementation in ULFM / Open MPI



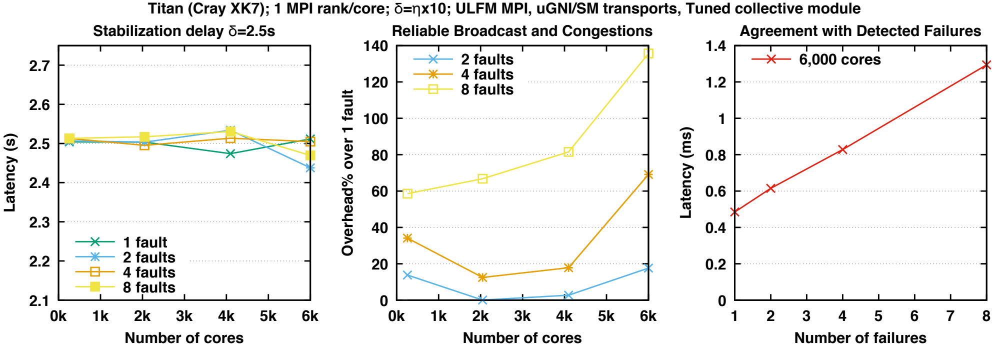
Experimental setup

- Titan ORNL Supercomputer
  - 16-core AMD Opteron processors
  - Cray Gemini interconnect
- ULFM
  - OpenMPI 2.x
  - Compiled with MPI_THREAD_MULTIPLE
- One MPI rank per core
- Up to $6,000$ cores
- Average of 30 times

Noise


Titan (Cray XK7); 256 MPI ranks on 256 cores; δ=ηx10; ULFM MPI, uGNI/SM transports, Tuned collective module
IMB 4B Messages
IMB 1MB Messages
Application
Overhead% over no FT
Variability w/o FT (σ)
Missed η
Missed δ
PingPing
PingPong
SendRecv
Bcast
Reduce
HPL, N=40,000
1e+00
1e-01
1e-02
Heartbeat period η (s)

Detection and propagation delay



50*X* improvement with failure detector ☺☺☺

Related work

- Some have a logical ring (Chord, Gulfstream, ...)
- Some separate detection and propagation (SWIM, consensus algorithms, ...)
- Most have non-deterministic strategies
  - at best: expectation of detection/propagation time for single failure
  - no quantitative assessment for several consecutive failures

- Our work is 100% deterministic
  - detection with single observer and easy-to-define time-out
  - minimal impact on failure-free execution of the application
  - logarithmic worst-case propagation
  - logarithmic worst-case repair time with consecutive failures

Conclusion on Failure Detection

Conclusion

- Failure detector based on timeout and heartbeats
- Tolerate arbitrary number of failures (but not too frequent)
- Complicated trade off between overhead, detection and risks (of not detecting failures)
- 100% deterministic
  $\Rightarrow$ First worst-case analysis of repair time with cascading failures
  $\Rightarrow$ 100X faster detection time over random rounds
- Unique implementation in ULFM
  $\Rightarrow$ Negligible overhead, quick failure information dissemination
  $\Rightarrow$ 50X improvement for consensus

More recently

- Failure detector service provided by MPI process manager (PMIx) instead of MPI library

Outline

2/3 - Consensus in ULFM

Outline

Consensus

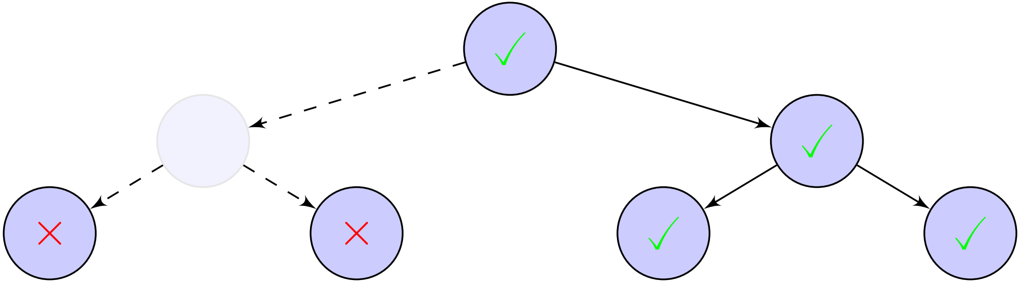
> *[consensus] is fundamental to distributed computing unreliable environments: it consists in* **agreeing** *on a piece of data upon which the computation depends*
>
> *M.Fischer, Brief Survey on Consensus*

D.Davies, J.F.Wakerly "*Synchronization and Matching in Redundant Systems*", IEEE Trans. on Comp., 1978. Context: Triple Modular Redundancy. Conclusion: Agreement through voting can tolerate only a minority of faulty processors.

Consensus is ubiquitous in distributed systems with high-availability (e.g. distributed database). It is a critical component in Fault-Tolerant HPC systems.

Consensus in the context of HPC

Consider the case of a broadcast implemented with a binary tree.



Failures, that happen during the execution, introduce inconsistencies: not all processes know that the broadcast operation failed.

Consensus (or agreement) allows to reconcile inconsistent / non-uniform states due to failures.

It must be reliable.

It must be efficient, especially in the failure-free case.

ULFM Agreement Specification

```
int MPIX_Comm_agree(MPI_Comm comm, int *flag);
MPIX_COMM_AGREE(COMM, FLAG, IERROR)
         INTEGER COMM, FLAG, IERROR
```

`comm` the communicator on which to apply the consensus

`flag` An in/out integer: in input, the process participation, in output, the result of the agreement on these ints (bitwise and)

`return value` An error code if new process failures were discovered during the agreement, or success

The operation implements an agreement on the couple (`flag, return code`): all surviving process, despite any failure have the same values in each (even if the return code is an error, flag is defined).

Specification

Correctness

Termination Every living process eventually decides.

Integrity Once a living process decides a value, it remains decided on that value.

Agreement No two living processes decide differently.

Participation When a process decides upon a value, it contributed to the decided value.

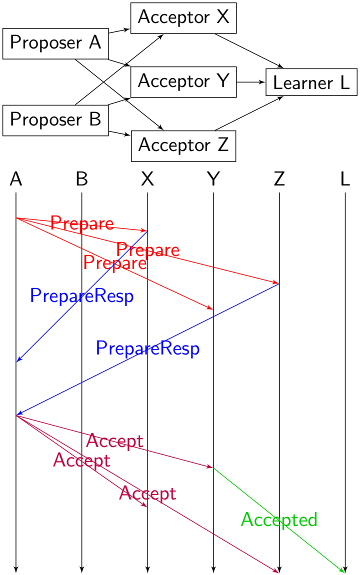
Traditional consensus relies on Validity
This is because one value is chosen.

ULFM does not require the consensus to be uniform (failed processes may have returned different values)

Assumptions

- Processes have totally ordered, unique identifiers
- Any process belonging to a group knows what processes belong to that group
- Any process may be subject to a permanent failure
- The network does not lose, modify, nor duplicate messages, but communication delays have *unknown* bounds
- The system provides a Perfect Failure Detector ($\mathcal{P}$):
  - All incorrect processes are eventually suspected by all correct processes
  - No correct process is ever suspected by any process
- The operation of the consensus is associative and commutative, and idempotent, with a *known neutral element*

Why not use Paxos?



- PAXOS provides reliability in persistant environments (intermittent failures and persistent storage space; message loss and dupplication)
- It relies on replication of information: requests are sent to multiple processes, and a majority must acknowledge
- Given our different requirements, we can achieve lower latencies in the failure-free case,
- Decision in PAXOS is upon one proposed value, while we need a combination of proposed values

Multiple Phase Commit Agreements

D. Buntinas, "Scalable distributed consensus to support MPI fault tolerance", IPDPS'12:

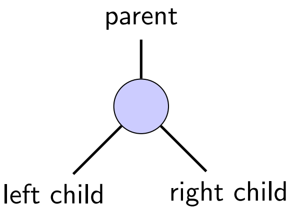
- Three Phase Commit:
  - Ballot number is chosen
  - Value is proposed
  - Value is committed
- Reliable P.I.F. ($O(log_2(n))$ comm., $O(1)$ comp.)

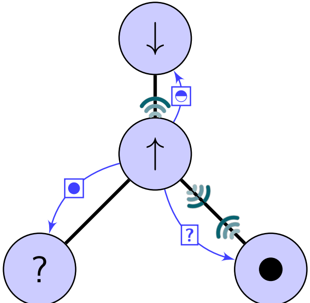
J. Hursey, T. Naughton, G. Vallee, R. Graham, "A Log-scaling Fault Tolerant Agreement Algorithm for a Fault Tolerant MPI", EuroMPI'11:

- Two Phase Commit
  - Fan-in / Fan-out approach
- Fatal errors when the root dies during the agreement
- $O(log_2(n))$ comm., but $O(n)$ comp.

Outline

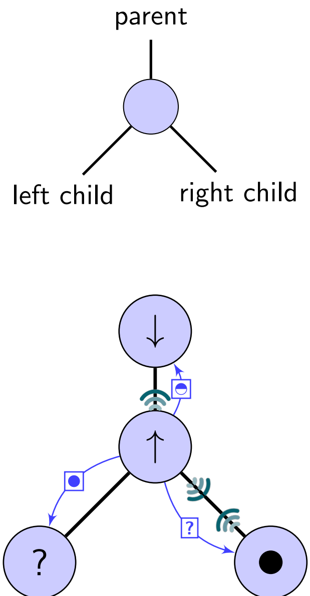
Principle and Notation





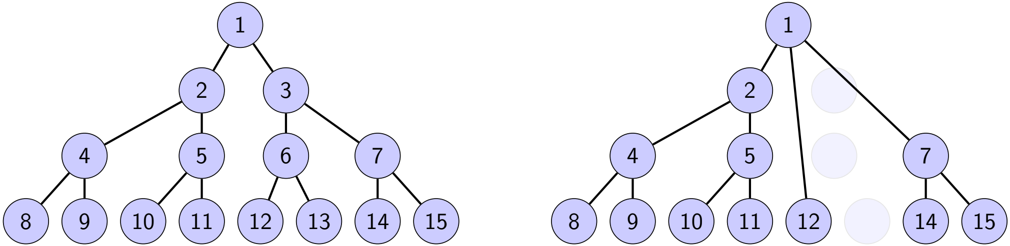
- Processes are arranged following a mendable tree topology: given a list of known dead processes, they communicate or monitor the liveliness of only their neighbors in that topology.
- The algorithm is a resilient version of Fan-in / Fan-out: all contributions (noted ◓) are reduced along the tree up to the root, that broadcasts it
- *Deciding* the result of the consensus for a given process consists in remembering the return value of the consensus, broadcasting it to the current children, and returning *as if* the consensus was completed.

Principle and Notation



- Alive processes can be in 3 states:
  - ?, if they have not entered the consensus yet
  - ↑, if they are waiting from the contribution of their children
  - ↓, if they have sent their contribution to their parent and are waiting for the decision
  - ●, if they have received the decision
- There are 3 types of messages:
  - ◓, when a process sends its participation to a parent
  - ●, when a process broadcasts the decision to its children
  - ?, when a process enquired about a possible result of a completed consensus
- Processes can monitor ( ) other processes for failures

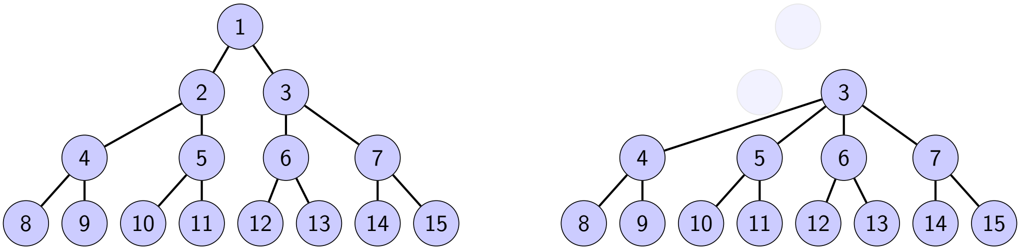
Mendable Tree for Consensus



The Fan-in Fan-out tree used during the consensus is mended, as failures are discovered during the execution.

The mending rule is simple: processes are arranged according to their (MPI) rank following a breath-first search of the tree, assuming no failure (left tree)

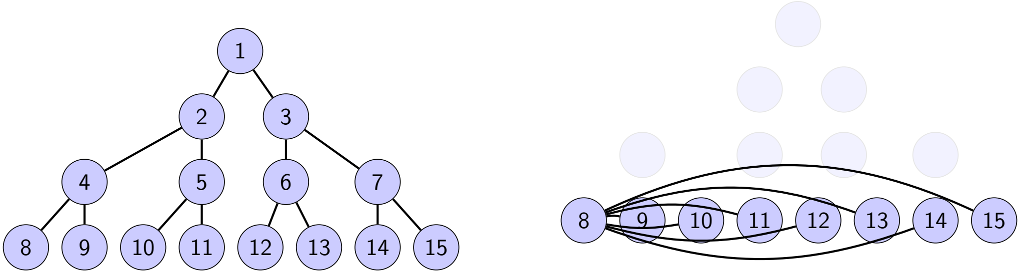
Mendable Tree for Consensus



Nodes replace their parents by the highest-ranked alive ancester in the tree in case of failure.

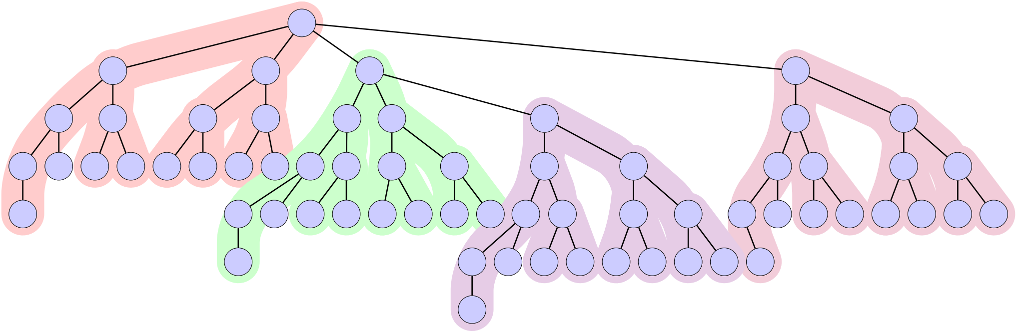
Processes without an alive ancestor in the original tree connect to the lowest alive processor as their parent. *The lowest alive processor is always the root of the tree*

Mendable Tree for Consensus



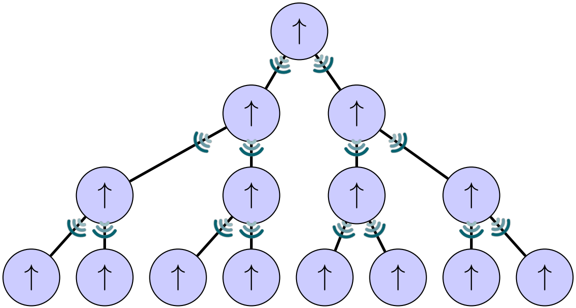
If half the processes die, the tree can, in the worst case, degenerate to a $np/2$-degree star

Architecture-Aware Tree



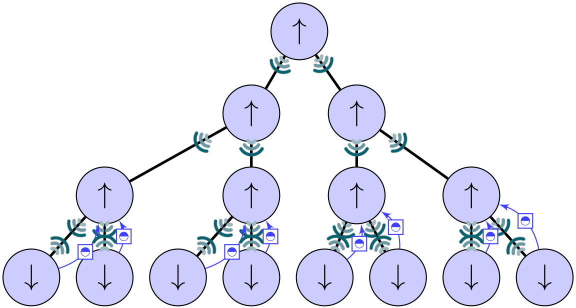
To map the hardware network hierarchy, two levels of trees are joined: In the example, *representative* processes of nodes (node0, node1, node2, node3) are interconnected following a *binary* tree, and processes belonging to the same node (16 process / node in this case) are also connected following independent *binary* trees.

No Failure



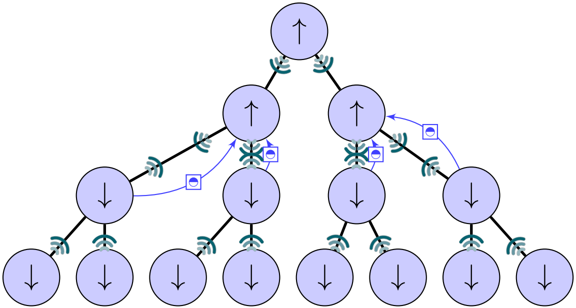
Initially, all processes are in the state ↑ to provide their participation, and the participation of their descendents to their ascendent. Each process monitors its descendents for possible failures ( ) until they have participated.

No Failure



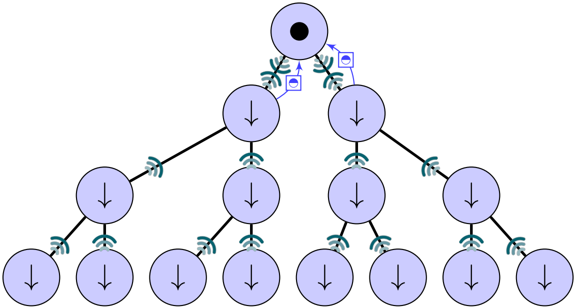
Leaves can send their participation (◓) to their parent, and enter the broadcasting state $\downarrow$. They start monitoring their parent for possible failures ( )

No Failure



Once a process has aggregated the participation of all its descendents, it can forward the information upward and do the same

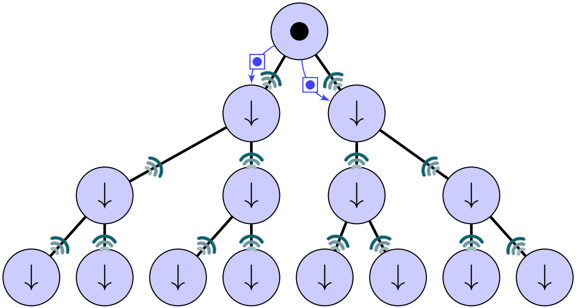
No Failure



Once a process has aggregated the participation of all its descendents, it can forward the information upward and do the same

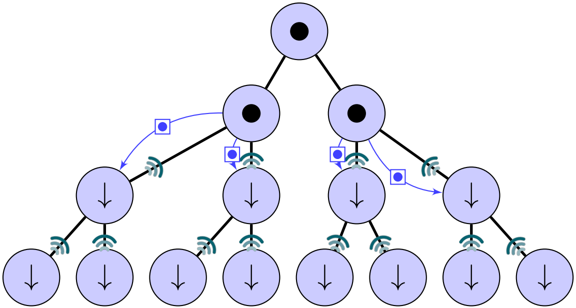
The root process can *decide* as soon as all descendents have contributed, it enters the decided state ●, starts broadcasting the decided message (●) to its descendents, and stops monitoring processes for failures

No Failure



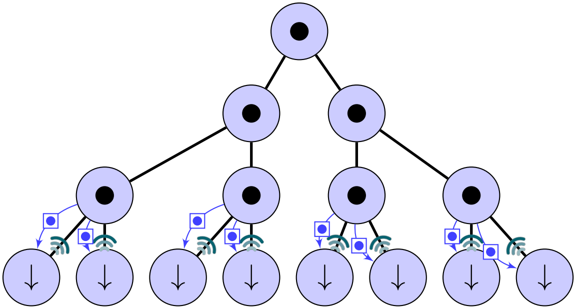
When a process receives a decision message (●), it decides, enters the decided state ●, and broadcasts the decision to its descendents, until all processes have decided

No Failure



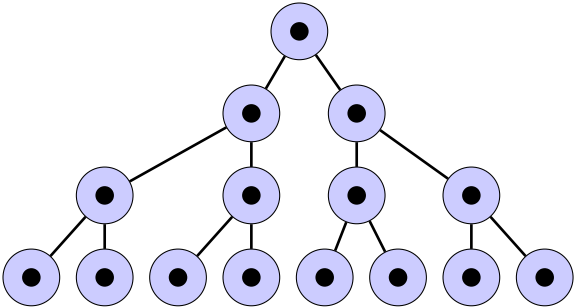
When a process receives a decision message (●), it decides, enters the decided state ●, and broadcasts the decision to its descendents, until all processes have decided

No Failure



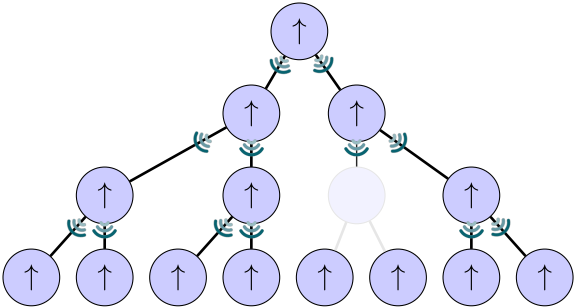
When a process receives a decision message (●), it decides, enters the decided state ●, and broadcasts the decision to its descendents, until all processes have decided

No Failure



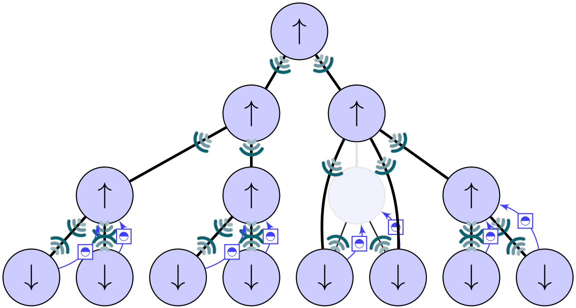
When a process receives a decision message (●), it decides, enters the decided state ●, and broadcasts the decision to its descendents, until all processes have decided

Failure before participating



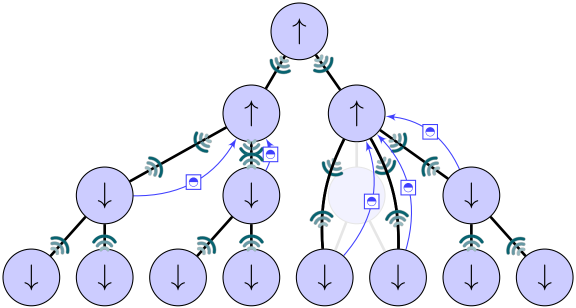
Process P_6 died before participating. P_3, its parent, starts monitoring it ( ) when it enters the consensus (state $\uparrow$).

Failure before participating



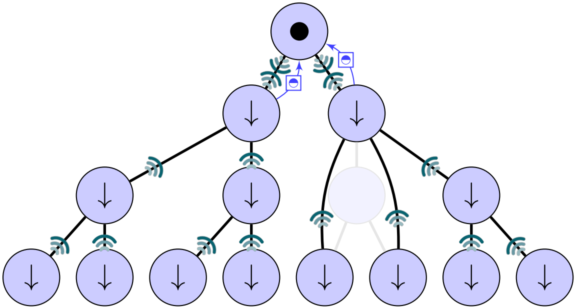
Processes P_{12} and P_{13} will send their participation (◓) to P_6, these messages are lost, and they start monitoring ()) P_6. P_3 eventually discovers the death of P_6, and starts monitoring ()) its new descendents P_{12} and P_{13}.

Failure before participating



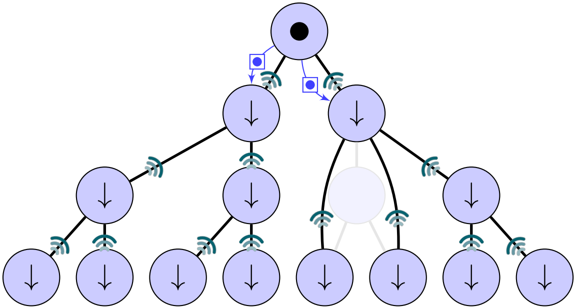
Processes P_{12} and P_{13} eventually discover the death of P_6, and take P_3 as their parent, sending it their participation (◓). They also start monitoring ())) their new parent, P_3.

Failure before participating



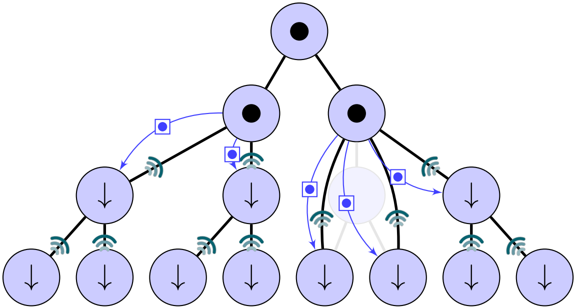
The tree being fixed, the information simply flows along the mended tree as initially.

Failure before participating



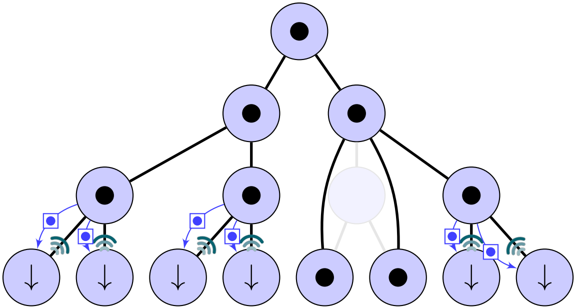
The tree being fixed, the information simply flows along the mended tree as initially.

Failure before participating



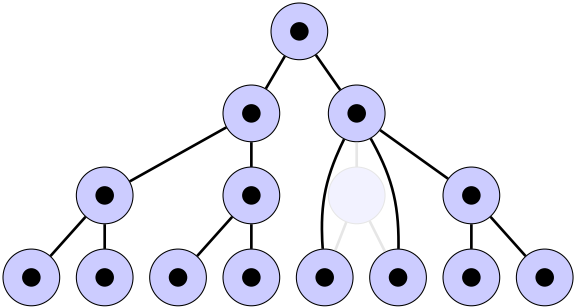
The tree being fixed, the information simply flows along the mended tree as initially.

Failure before participating



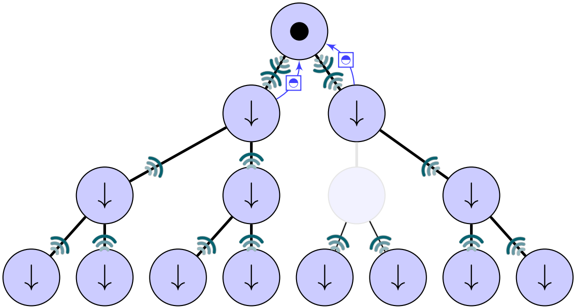
The tree being fixed, the information simply flows along the mended tree as initially.

Failure before participating



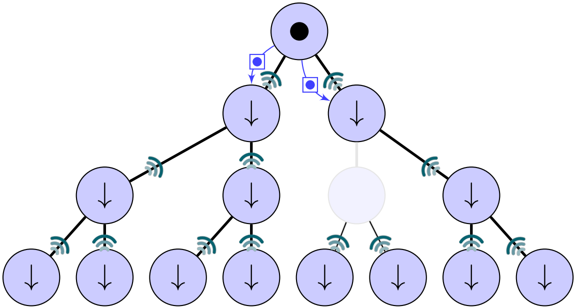
The tree being fixed, the information simply flows along the mended tree as initially.

Failure After Participating



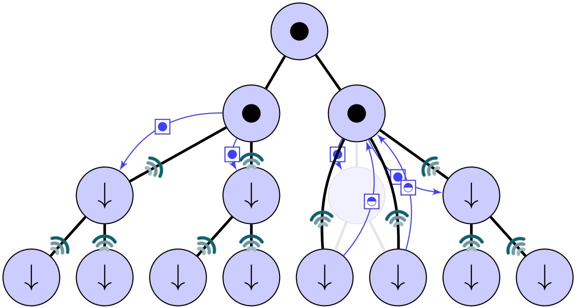
Process P_6 fails, but after participating to the current consensus.

Failure After Participating



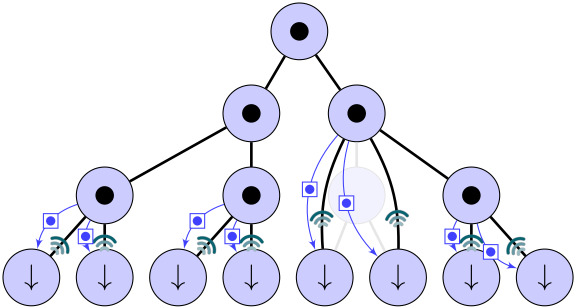
If it was a leaf, that would not prevent the consensus to complete. Since it has children, and they have not received the decision (●) yet, they are monitoring ( ) it, and eventually discover the death

Failure After Participating



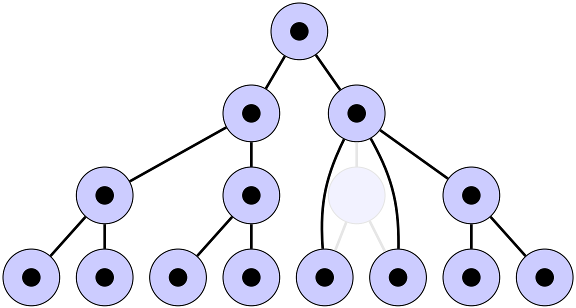
They send their participation (◒) back to their grand-parent, P_3, starting to monitor it ( ). This ensure that if P_6 died before forwarding it upward, their participartion (◒) is not lost. This also reconnects the tree.

Failure After Participating



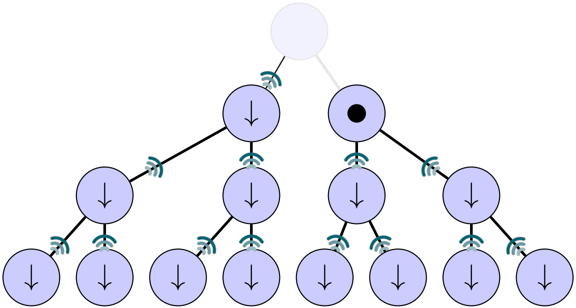
Even if P_3 is already done with the current consensus, it remembers the result (ERA property), and provides the result (●) again, allowing the information to continue flowing down the tree.

Failure After Participating



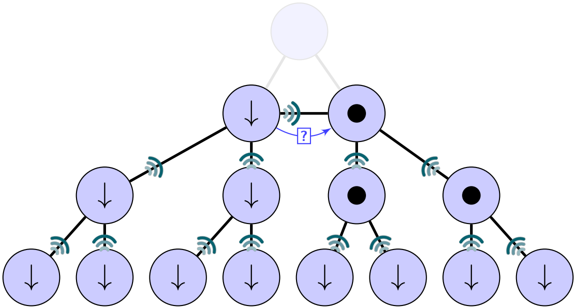
Even if P_3 is already done with the current consensus, it remembers the result (ERA property), and provides the result (●) again, allowing the information to continue flowing down the tree.

Failure of Root



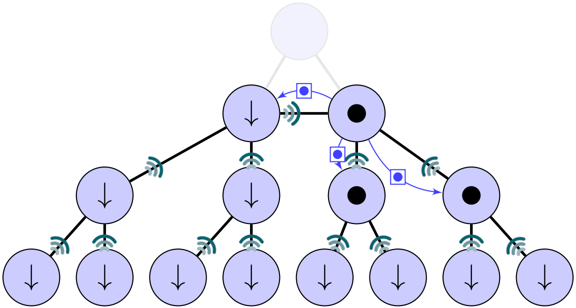
If the root of the tree dies after it started broadcasting the decision, but before it could reach all its children, the ones that did not receive the decision ( ) are still monitoring that dead root ( ).

Failure of Root



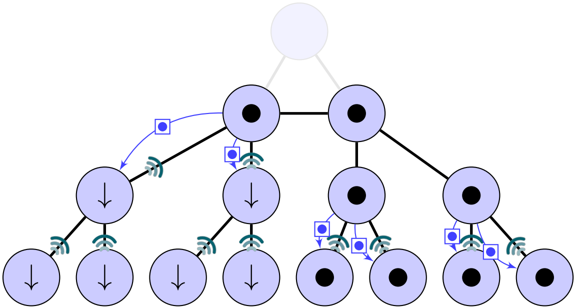
If a process becomes the root (lowest identifier), but was waiting for a decision, it asks all its new children if they received a decision before, by sending the message (?), and monitoring them ( ).

Failure of Root



If one of them has the decision, it answers with it and the root can decide and broadcast (●). If none has it, they provide their participation (◓), if they reached that step, and wait for the decision of the new root.

Failure of Root

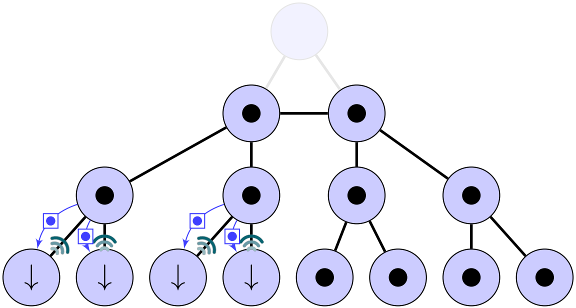


The broadcast of the decision (●) then continues along the tree

Fault tolerant algorithms for a fault tolerant MPI

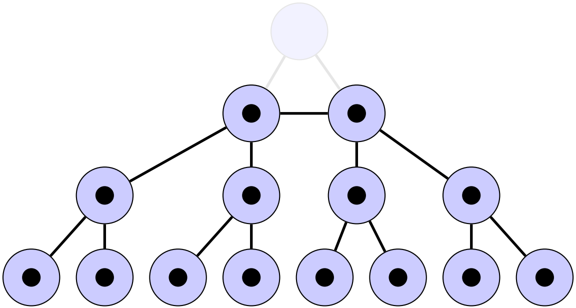
Thomas Herault

Failure of Root



The broadcast of the decision (●) then continues along the tree

Failure of Root



The broadcast of the decision (●) then continues along the tree

Implementation

Agreements are identified by a tuple (CID, $CEPOCH$, $ANUMBER$):

- CID is the communicator Identifier
- $CEPOCH$ Epoch of the communicator – Epochs are changed every time a new communicator is created, and reflect how many failures were known at the time of creation
- $ANUMBER$ is the sequence number of the current agreement.

Current values of the agreements, progress status, and past values of past agreements are stored in hash tables.

The ERA is implemented at the *BTL level*, below the matching and message layer mechanisms.

Garbage Collection

When multiple consensus are executed on the same group of processes, processes executing ERA need to remember each consensus result. This can lead to memory exhaustion.

ERA implements a Garbage Collection mechanism to forget past consensus that *will not* be requested in the future.

That mechanism is implemented using the consensus operation itself: in addition to the consensus value, processes agree in the ● message on past consensus that can be collected.

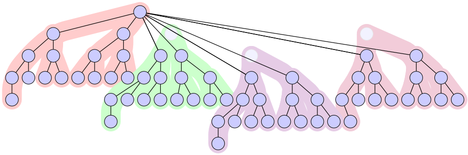
How to cleanup?

The last consensus is cleaned up by introducing an asynchronous ERA in the destructor of the communicator.

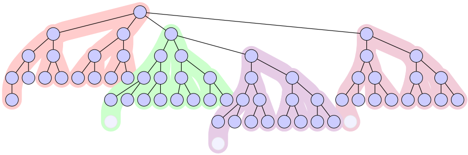
The result of this last ERA does not need to be remembered: if the communicator has been released, then all processes participated, and the return value is ignored.

Tree-Rebalancing

As processes crash, the Fan-in / Fan-out tree used to implement the two phases of the consensus can become unbalanced.



To implement the ULFM specification, all processes must agree on a list of failed nodes. Trees can be re-balanced when starting a new agreement based on that information.

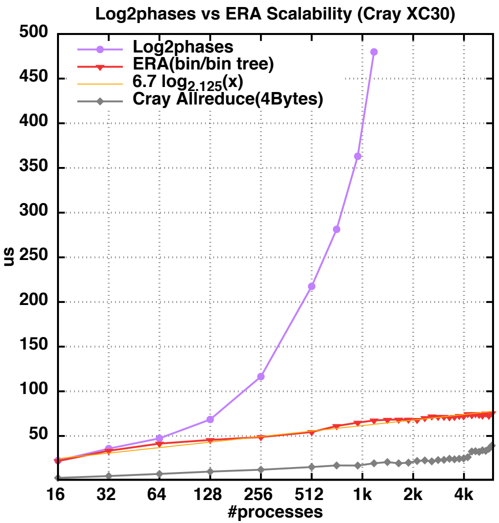


Outline

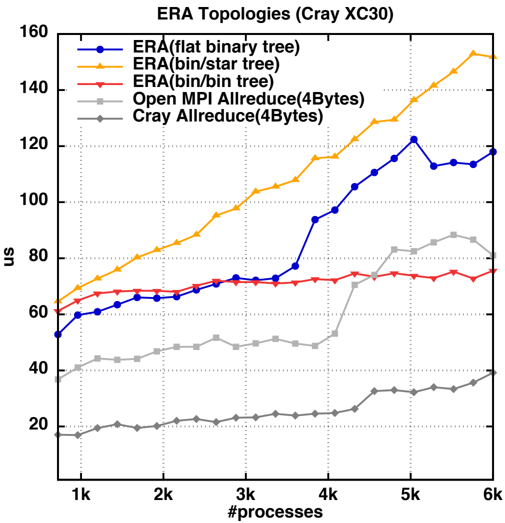
Environment

- NICS Darter: Cray XC30 (cascade)
  - `ugni` transport layer, with Aries interconnect
  - `sm` transport layer for shared memory
  - Scalability runs: 16 - 6,500 processes
- Benchmark:
  - `MPIX_COMM_AGREE` in loop
  - Measure duration:
    - before failure
    - during failure
    - stabilizing after failure
    - after stabilization

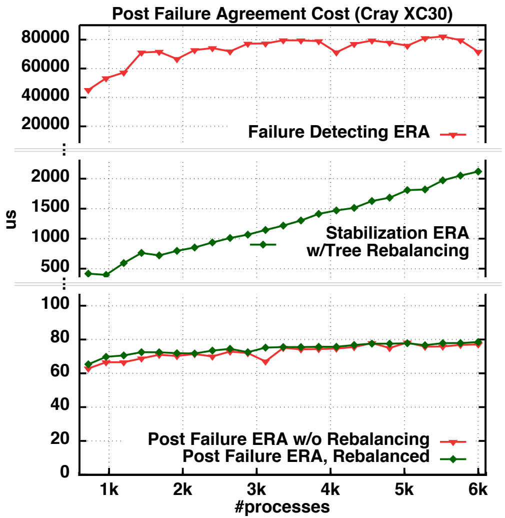
Agreement scalability in the failure-free case



ERA performance depending on the tree topology



Post Failure Agreement Cost



S3D and FENIX

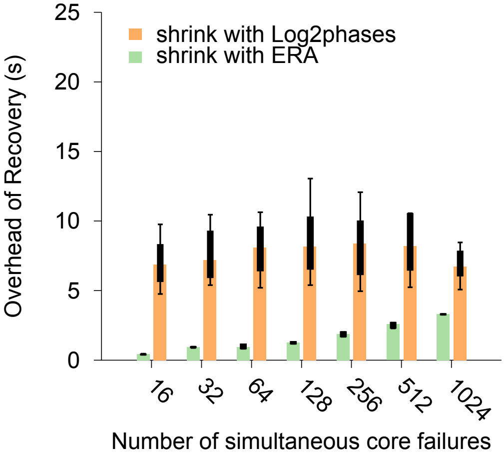
S3D

- Highly parallel method-of-lines solver for partial differential equations
- first-principles-based direct numerical simulations of turbulent combustion
- ported to all major platforms, demonstrates good scalability up to nearly 200K cores,

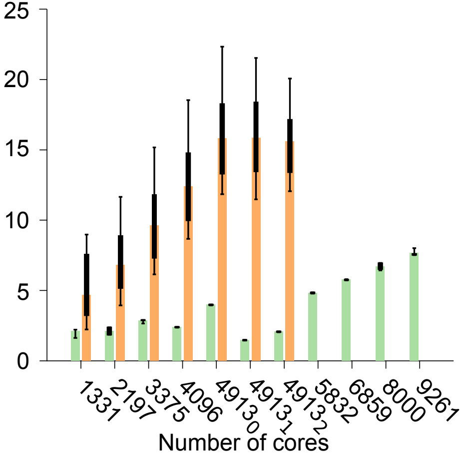
FENIX

- Online, Transparent recovery framework
- Encapsulates mechanisms to transparently
  - capture failures through ULFM return codes,
  - re-spawn new processes on spare nodes when possible,
  - fix failed communicators using ULFM capabilities,
  - restore application state, and return the execution control back to the application

FENIX & S3D Performance



Simultaneous failures on an increasing number of cores, over 2197 total cores



256-cores failure (*i.e.*, 16 nodes) on an increasing number of total cores

MiniFE and LFLR Framekwork

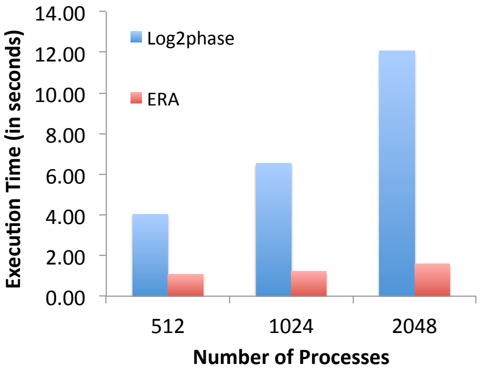
MiniFE

- Part of Mantevo mini-applications suite
- MiniFE performs a linear system solution with relatively quick mesh generation and matrix assembly steps.
- Modified version: performs a time-dependent PDE solution, where each time step involves a solution of a sparse linear system with the Conjugate Gradient (CG) method

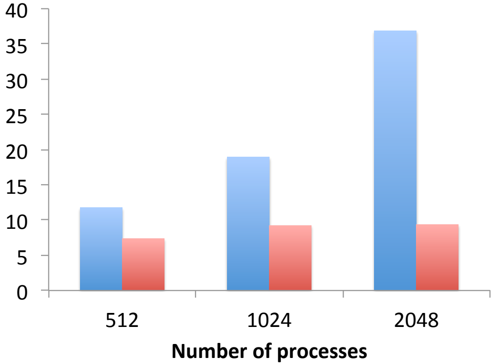
LFLR Framework

- Local Failure Local Recovery is a resilient application framework
- leverages ULFM to allow on-line application recovery from process loss without the traditional checkpoint/restart
- layer of abstraction classes to support `commit` and `restore` methods
- Works with active spare processes pool

MiniFE and LFLR Performance



Process and communicator recovery



Global agreement during 20 time steps.

Outline

Conclusion on the ERA algorithm

- ERA is a logarithmic agreement, in number of messages and in computation
- ERA allows processes to return early from the routine itself, serving potential late requests in the background
- Its implementation in ULFM / Open MPI shows performance comparable to an optimized non-fault-tolerant AllReduce
- Improvement of agreement translates into improvement of other routines (`shrink`).

Outline

3/3 - Fault-Tolerant LU Factorization using ULFM

Outline

Algorithm Based Fault Tolerance (ABFT)

Principle

- Limited to Linear Algebra computations
- Matrices are extended with rows and/or columns of checksums

$$M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 4 & 3 & 5 & 12 \\ 4 & 6 & 9 & 19 \end{pmatrix}$$

ABFT and failures: detect and correct

Missing checksum data

$$M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 4 & 3 & 5 & \\ 4 & 6 & 9 & 19 \end{pmatrix}$$

Simple recomputation: $4+3+5 = 12$.

ABFT and failures: detect and correct

Missing checksum data

$$M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 4 & 3 & 5 & \\ 4 & 6 & 9 & 19 \end{pmatrix}$$

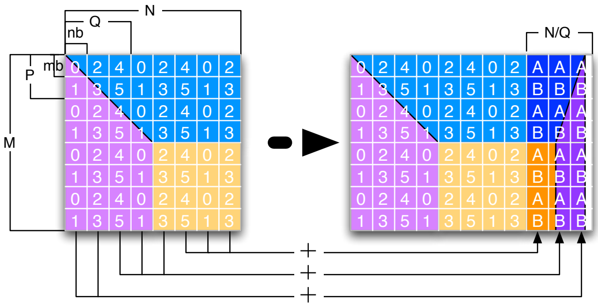
Simple recomputation: $4+3+5 = 12$.

Missing original data

$$M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 4 & & 5 & 12 \\ 4 & 6 & 9 & 19 \end{pmatrix}$$

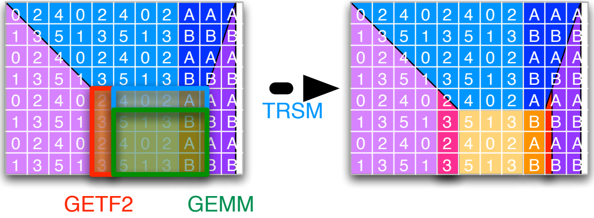
Simple recomputation: $12\text{-}(4+5) = 3$.

Algorithm Based Fault Tolerant LU decomposition



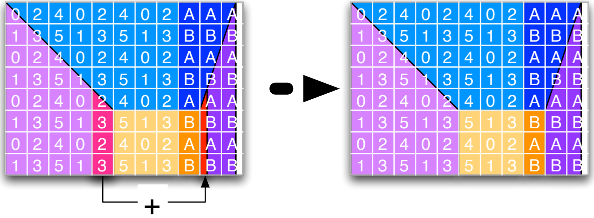
- Checksum: invertible operation on the data of the row / column
  - Checksum replication can be avoided by dedicating computing resources to checksum storage

Algorithm Based Fault Tolerant LU decomposition



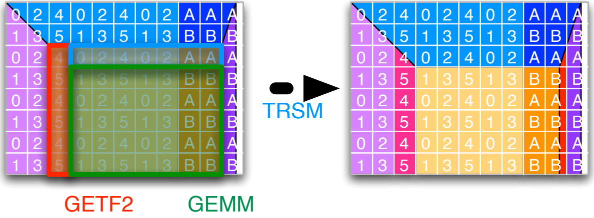
- Idea of ABFT: applying the operation on data and checksum preserves the checksum properties

Algorithm Based Fault Tolerant LU decomposition



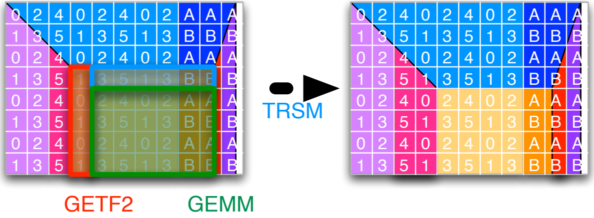
- For the part of the data that is not updated this way, the checksum must be re-calculated

Algorithm Based Fault Tolerant LU decomposition



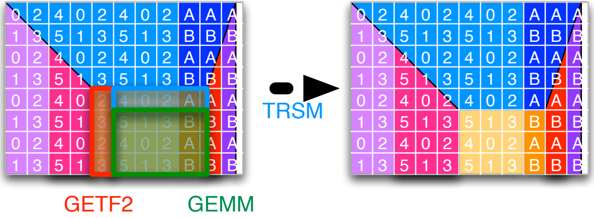
- To avoid slowing down all processors and panel operation, group checksum updates every Q block columns

Algorithm Based Fault Tolerant LU decomposition



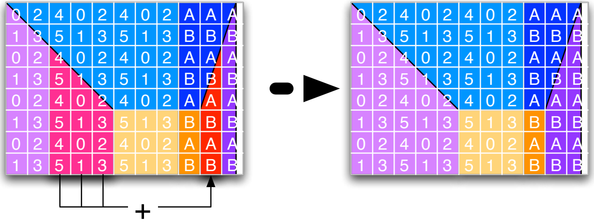
- To avoid slowing down all processors and panel operation, group checksum updates every Q block columns

Algorithm Based Fault Tolerant LU decomposition



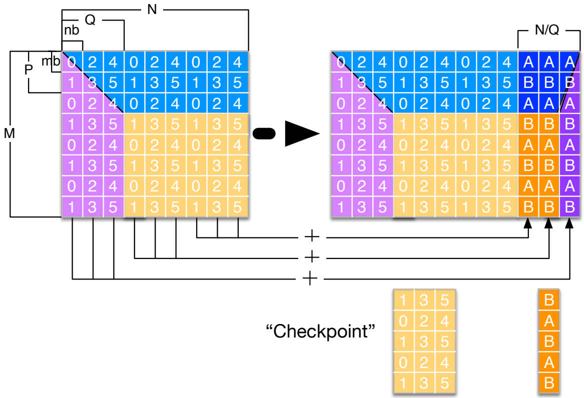
- To avoid slowing down all processors and panel operation, group checksum updates every Q block columns

Algorithm Based Fault Tolerant LU decomposition



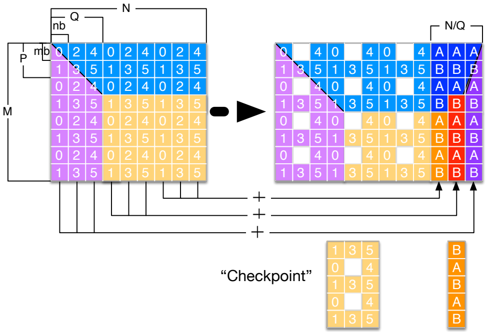
- Then, update the missing coverage. Keep checkpoint block column to cover failures during that time

Algorithm Based Fault Tolerant LU decomposition



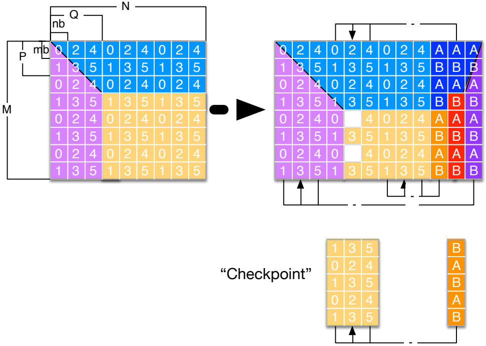
- Checkpoint the next set of Q-Panels to be able to return to it in case of failures

Algorithm Based Fault Tolerant LU decomposition



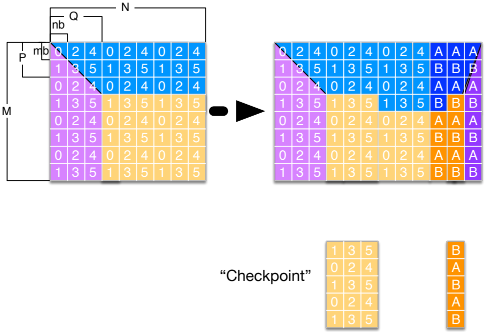
- Checksum blocks are made resilient by replication

Algorithm Based Fault Tolerant LU decomposition



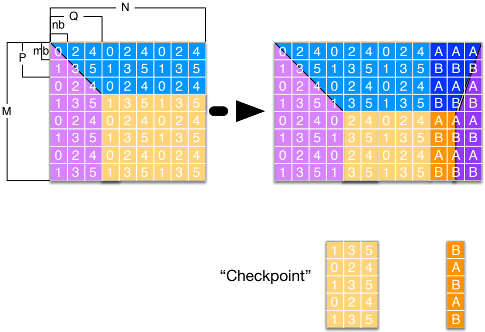
- Valid Checksum Information allows to recover most of the missing data, but not all: the checksum for the current $Q-$panels are not valid

Algorithm Based Fault Tolerant LU decomposition



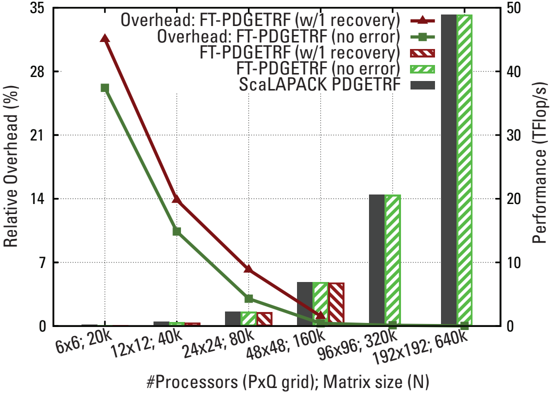
- We use the checkpoint to restore the $Q-$panel in its initial state

Algorithm Based Fault Tolerant LU decomposition



- and re-execute that part of the factorization

ABFT LU decomposition: performance



MPI-ULFM Performance

- Open MPI with ULFM; Kraken supercomputer;

Conclusion

ULFM Consensus and Failure Detection – Conclusions

- **Consensus (MPI_Comm_agree / ERA):** Early-Returning Agreement on a tree overlay for crash failures in pseudo-synchronous systems.
- Decision is a reduction with an associative, commutative, idempotent operator; processes may return as soon as the global value is fixed, yet can still serve late messages.
- Proven strong progress and **logarithmic** message/time complexity in the failure-free case; ULFM/Open MPI implementation scales and significantly outperforms log-scaling 2-phase agreement in both microbenchmarks and applications (e.g., MiniFE).
- **Failure detector:** virtual observation ring (one observer per process) plus non-uniform reliable broadcast on a circulant / hypercube-like overlay for propagation.
- Provides a practically *perfect* failure detector with deterministic logarithmic stabilization time, constant degree, and very low heartbeat traffic; validated by simulations and experiments on Titan and against SWIM.

ABFT LU and Dense Factorizations – Conclusions

- **Hybrid ABFT framework** for one-sided dense factorizations (LU with partial pivoting, QR, Cholesky): combines checksums and algorithm-driven checkpointing.
- Right factor (trailing matrix) protected by maintaining a checksum relationship $C = GA$ or $C = AG$ throughout the factorization; proven invariant even with full-matrix updates and pivoting.
- Left factor (panels already applied) protected by a scalable **vertical checkpointing** scheme that reuses checksum storage and overlaps checkpoints across iterations.
- Extended scheme tolerates **multiple simultaneous fail-stop failures**: $2f$ checksum blocks protect against f failures, with recovery that avoids error propagation.
- Theoretical analysis and Kraken experiments show *decreasing overhead with scale* and solution accuracy comparable to failure-free LU/QR, even after repeated failures and recoveries.

Ongoing and Future Work: The Cupseli Challenge

Extending the ULFM methodology to geo-distributed systems

- Apply the same end-to-end approach: *design* new distributed algorithms, *prove* their correctness and performance, *evaluate* them through simulation, and *validate* with real implementations on deployed platforms.
- Geo-distribution fundamentally changes the model: heterogeneous links, asymmetric bandwidth/latency, and weaker timing assumptions require a **complete redesign of the failure detector**.
- Platform nodes exhibit more variability and lower reliability, increasing the need for robust consensus and recovery.

Target applications shift: Deep Neural Networks

- Focus on DNN training and inference workloads, whose communication patterns and resilience needs differ substantially from HPC linear algebra.
- In TOPAL (with Philippe Swartvagher, Lionel Eyraud-Dubois, and Olivier Beaumont), early work on a geo-distributed **AllReduce** algorithm has begun with the new PhD student, **Fares Boudjaoui**.
- Current design is failure-free; **fault-tolerant AllReduce** is a central objective for the next phase of the project.