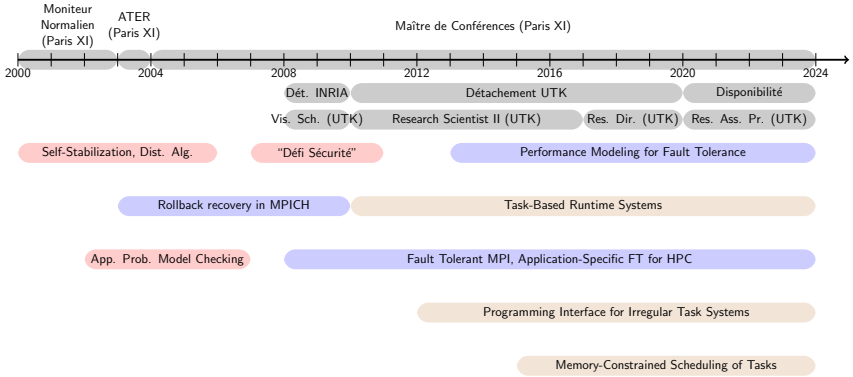
Fault-tolerance Techniques for HPC

Séminaire
Laboratoire de l'Informatique du Parallélisme
École Normale Supérieure de Lyon

Thomas Herault, University of Tennessee, Knoxville

Background


Moniteur Normalien (Paris XI)
ATER (Paris XI)
Maître de Conférences (Paris XI)
2000
2004
2008
2012
2016
2020
2024
Dét. INRIA
Détachement UTK
Disponibilité
Vis. Sch. (UTK)
Research Scientist II (UTK)
Res. Dir. (UTK)
Res. Ass. Pr. (UTK)
Self-Stabilization, Dist. Alg.
"Défi Sécurité"
Performance Modeling for Fault Tolerance
Rollback recovery in MPICH
Task-Based Runtime Systems
App. Prob. Model Checking
Fault Tolerant MPI, Application-Specific FT for HPC
Programming Interface for Irregular Task Systems
Memory-Constrained Scheduling of Tasks

Background (Research Topics: Fault-Tolerance)

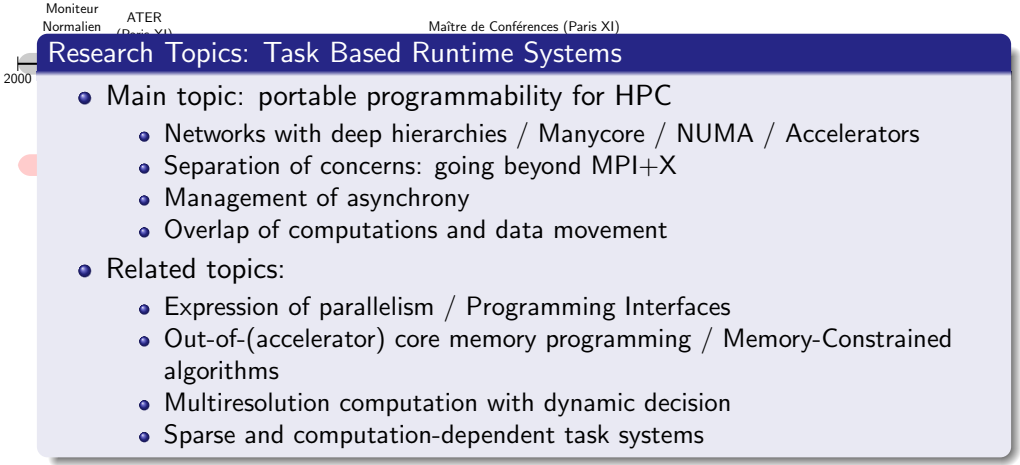
Research Topics: Fault Tolerance

2000

Two complementary approaches to Fault Tolerance

- General Purpose Fault Tolerance
  - Checkpointing and rollback-recovery
    - Experimental approach: implementing new protocols and evaluating them in 'real' applications / deployments
    - Performance models approach: designing mathematical models to project performance at scale or under new assumptions
- Application-Specific Fault Tolerance
  - Design / adapt communication middlewares to write fault-tolerant applications
  - Design and evaluate new fault-tolerance schemes for parallel applications

Background (Research Topics: Task-Based Runtime Systems)

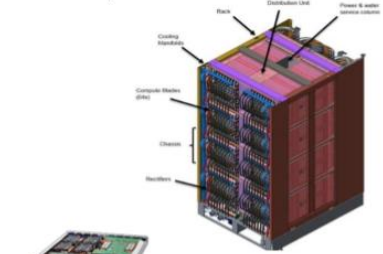


Outline

1 Introduction
2 Checkpointing Protocols
3 Application-Specific Fault Tolerance
4 Conclusion and Future Work

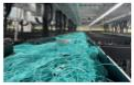
Supercomputers are large-scale hierarchical machines

Each AMD node:
- 1 AMD "Trento" CPU
- 4 AMD MI250X GPUs
- 512 GiB DDR4 memory on CPU
- 512 GiB HBM2e total per node (128 GiB HBM per GPU)
- Coherent memory across the node
- 4 TB NVM
- GPUs & CPU fully connected with AMD Infinity Fabric
- 4 Cassini NICs, 100 GB/s network BW



Each blade: 2 AMD nodes

Each rack:
- 28 AMD nodes
- 8,000 lbs/3,600 KG
- Supports 400 KW



Network:
Per rack:
- 512 NICs
- 32 Switches
- 18,816 L0 cables
- 1 Dragonfly group

Between racks:
- 9,472 L1 cables
- 5,402 L2

Frontier (Oak-Ridge Leadership Computing Facility)
- 1.7 EFlop/s DP peak
- 74 compute racks
- 29 MW Power Consumption
- 9,408 nodes
- 9.2 PB memory (4.6 PB HBM, 4.6 PB DDR4)
- Cray Slingshot network with dragonfly topology
- 37 PB Node Local Storage
- 716 PB Center-wide storage
- 360 m^2 footprint

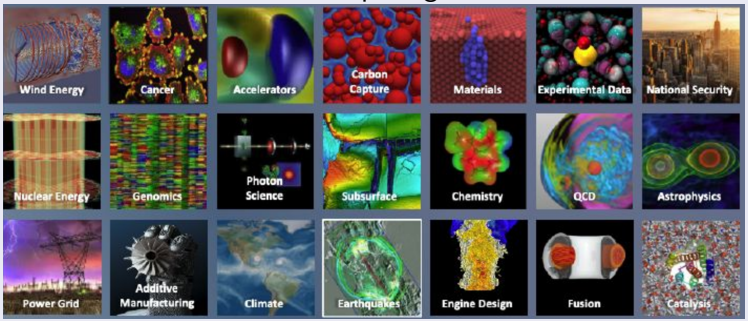
Supercomputers are large-scale hierarchical machines

TOP 500

Each AMD node:
- 1 AMD "Trento" C
- 4 AMD MI250X G
- 512 GiB DDR4 m
- 512 GiB HBM2e
- Coherent memory
- 4 TB NVM
- GPUs & CPU full
- 4 Cassini NICs, 1

What are those system used for?

Scientific simulation and computing



Each blade: 2 AMD nodes

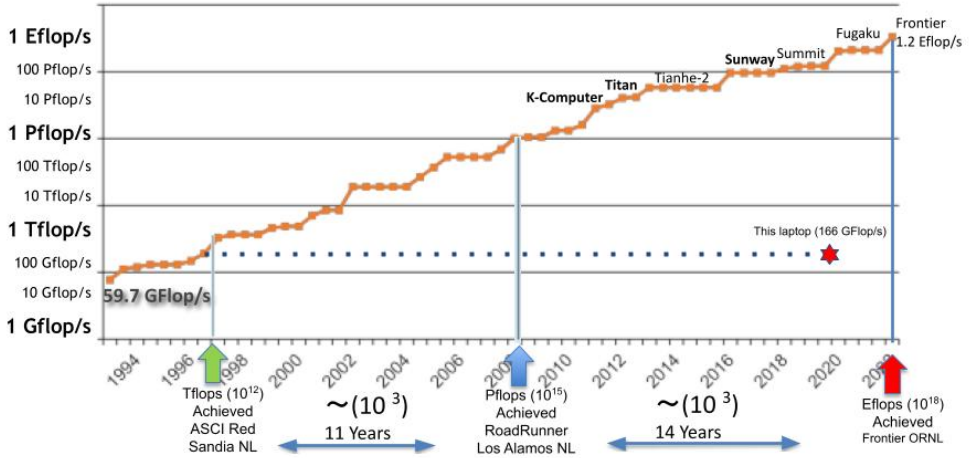
Each rack:
- 28 AMD nodes
- 8,000 lbs/3,600 KG
- Supports 400 KW

uting Facility)

.6 PB DDR4)
gonfly

topology
- 37 PB Node Local Storage
- 716 PB Center-wide storage
- 360 m^2 footprint

Performance development of HPC systems over the last 30+ years

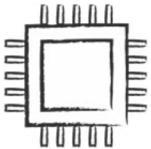


Focus of this talk

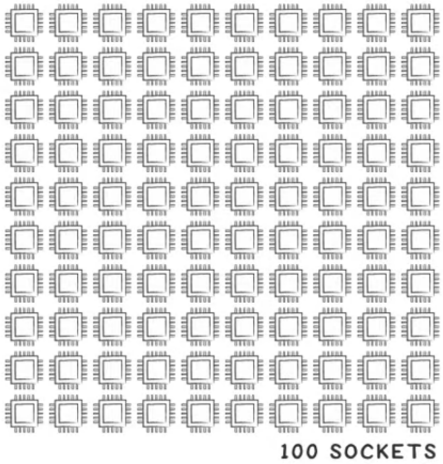
Resilience techniques for High Performance Computing

- Rollback-recovery protocols
  - Coordinated rollback-recovery
  - Noncoordinated rollback-recovery
  - Hierarchical protocols
- Application-specific fault-tolerance
  - User-Level Failure Mitigation
  - Failure detection and notification for HPC
  - Early Returning Agreement for ULFM

Effect of Scale



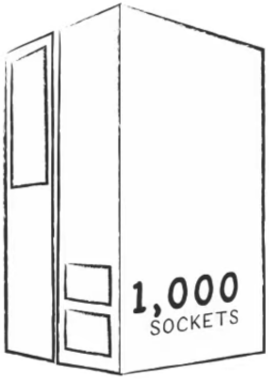
Effect of Scale



1
YEAR

MEAN TIME
BETWEEN FAILURES

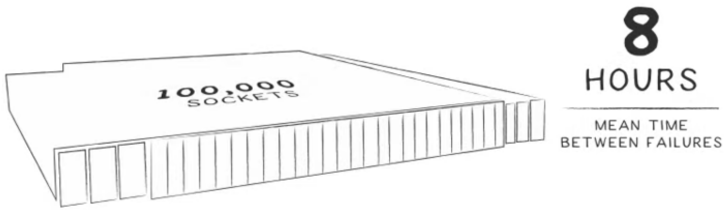
Effect of Scale



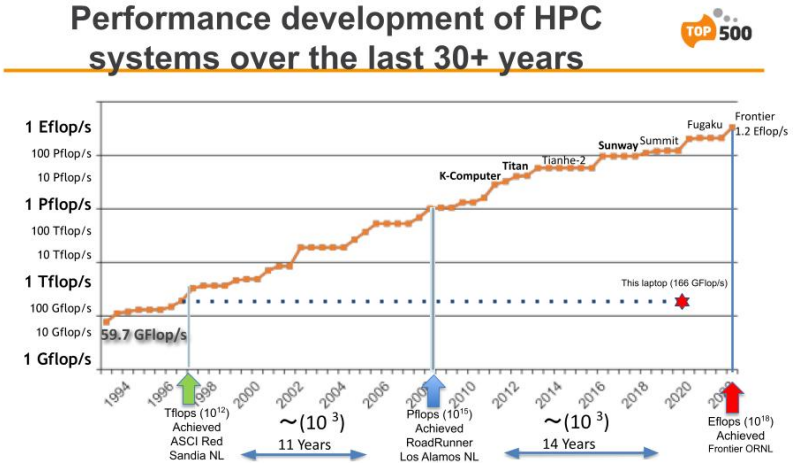
36
DAYS

MEAN TIME
BETWEEN FAILURES

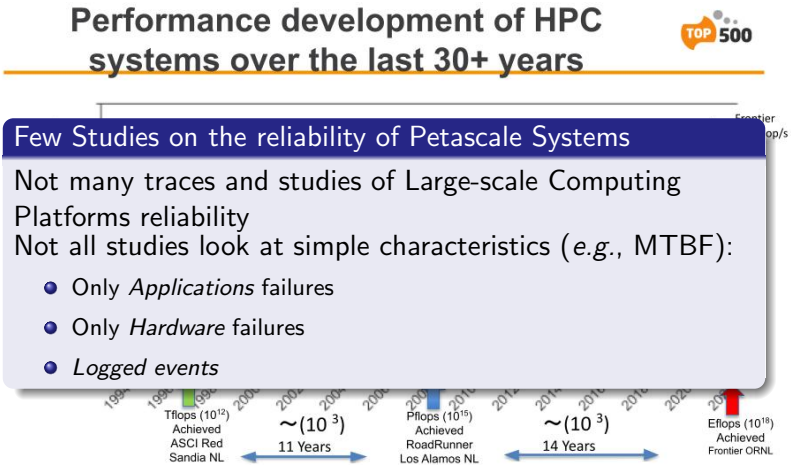
Effect of Scale



Petascale Computing Platforms



Petascale Computing Platforms



Petascale Platforms – K-Computer

#20 Top500 (June'19, Rmax = 10.510 PFlop/s); 06/11 — 08/19
864 cabinets; 88,128 nodes; 705,024 cores; 1.34PBytes; 12.6 MW;

Fumiyoshi Shoji, Shuji Matsui, Mitsuo Okamoto, Fumichika Sueyasu, Toshiyuki Tsukamoto, Atsuya Uno and Keiji Yamamoto. **Long term failure analysis of 10 petascale supercomputer**. Poster at ISC'15

MFR = Monthly Failure Rate
$(= \frac{\#\text{failures during month}}{\#\text{unit total}})$

CPU MFR	0.004%	$\rightleftharpoons$	0.017%
DIMM MFR	0.0017%	$\searrow$	0.001%
Sys. Board MFR	0.16%	$\searrow$	0.05%
Sys. Board MTBF	1.1 day	$\nearrow$	3 day

AFR = Annualized Failure Rate
$(= 1 - e^{\frac{-1year}{MTBF}})$
FIT = Failure In Time = #fail. / 10^9 h

	# Parts	AFR	MTBF	FIT
CPU	82,944	0.06%	7.33 days	72.00
DIMM	663,552	0.0016%	34.4 days	18.02

(Numbers in red are derived from reported numbers in black)

Petascale Platforms – Titan

#12 Top500 (June'19, Rmax = 17.590 PFlop/s); 10/12 — 08/19
200 cabinets; 18,688 nodes; 299,008 cores; 18,688 K20X; 693.6TBytes; 8.2 MW;

R. A. Ashraf and C. Engelmann, **Analyzing the Impact of System Reliability Events on Applications in the Titan Supercomputer**, 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS), 2018

Event log from Reliability, Availability and Serviceability (RAS) System

$68k$ apps have 1+ events (over $2 \cdot 10^6$)

95% of apps with 11k+ nodes have 1+ events
91% of apps with 125- nodes have 0 events
85% apps under 30min have 0 events
80% apps above 24h have 1+ events

Nature of first event hitting an application:

Parallel Filesystem	73.7%
Processor Failure	15.7 %
Machine Check Exception	6.5%
GPU Failure	1.5%
OOM / SEGFAULT	1.9%
Interconnect	0.8 %

Petascale Platforms – Titan

#12 Top500 (June'19, Rmax = 17.590 PFlop/s); 10/12 — 08/19
200 cabinets; 18,688 nodes; 299,008 cores; 18,688 K20X; 693.6TBytes; 8.2 MW;

E. Rojas, E. Meneses, T. Jones and D. Maxwell **Analyzing a Five-year Failure Record of a Leadership-class Supercomputer**, 31st Intl. Symp. on Computer Architecture and HPC (SBAC-PAD), 2019

System failure database (sys. admin.)

Year	2014	2015	2016	2017	2018
Events	161k	258k	444k	452k	1,348k

(About 30% of events are categorized as having a user origin)

5,625 hardware failures / 4 years (≈4/day – 6h MTBF)

GPU DBE represents 53% of hardware events ECC on these GPUs is single error correction double error detection, leads to interrupt in DBE.

Nature of failure:

GPU DBE	2070 (37%)
GPU BUS	797 (14%)
GPU DPR	308 (5.5%)
GPU XID (Soft.)	1370 (24%)
Machine Check Exc	653 (12%)
Other Failures	427 (7.5%)

(Numbers in red are derived from reported numbers in black)

Petascale Platforms – Sunway TaihuLight

#3 Top500 (June'19, Rmax = 93.014 PFlop/s); 06/16 —
40 cabinets; 20,480 nodes; 10,649,600 cores; 1.32PBytes; 15 MW;

Liu RT, Chen ZN. **A large-scale study of failures on petascale supercomputers.** Journal of Computer Science and Technology 33(1): 24-41, Jan. 2018

Nature of faults:

	All Failures	Fatal Failures
CPU	40%	68%
Memory	48%	21%
Power	9%	5%
MD$^{(*)}$	2%	5%

((∗) MD: Maintenance and Diagnosis system)

Effect of Application (1 cabinet, #faults/month):

	Mem Faults	CPU Faults
Comp. only	$7 \rightleftharpoons 93$	$5 \rightleftharpoons 33$
Comp. & Mem	$527 \rightleftharpoons 995$	$136 \rightleftharpoons 867$
Comm. & Mem	$115 \rightleftharpoons 284$	$20 \rightleftharpoons 85$

3 time spans; 1 CPU; 2 Computing Cards
Weibull Distribution best fits CDF

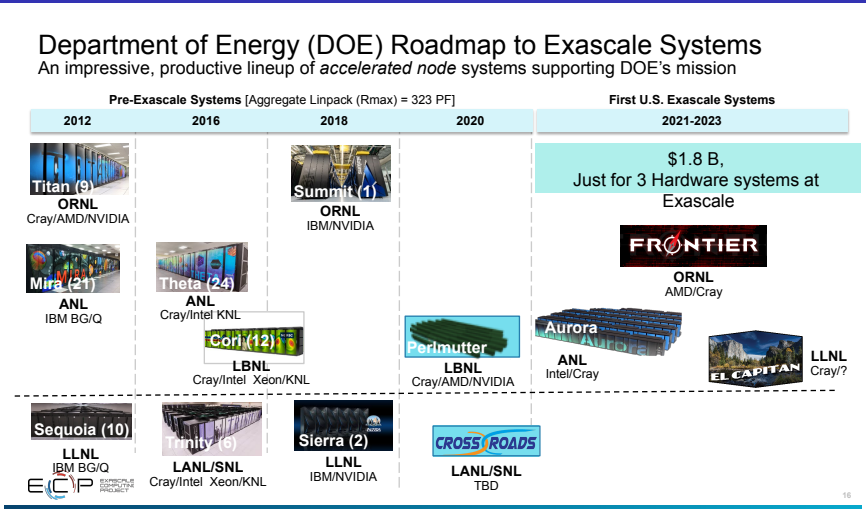
	P_1	P_2	P_3
CPU_1 MTBF	8 days	4 days	10 days
$Card_1$ MTBF	9.5 days		
$Card_2$ MTBF	10 days		

Data too sparse to give a system MTBF

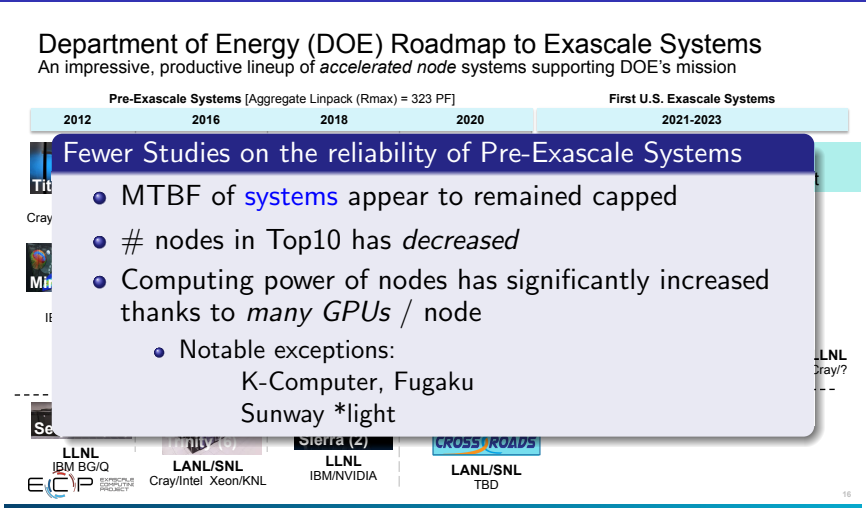
It was probably small

MTBF computed from η and m parameters given in the article:
PDF: $f(x) = \frac{m}{\eta}(\frac{x}{\eta})^{m-1}e^{-(x/\eta)^m}$;

Exascale platforms (courtesy Jack Dongarra)



Exascale platforms (courtesy Jack Dongarra)



Outline

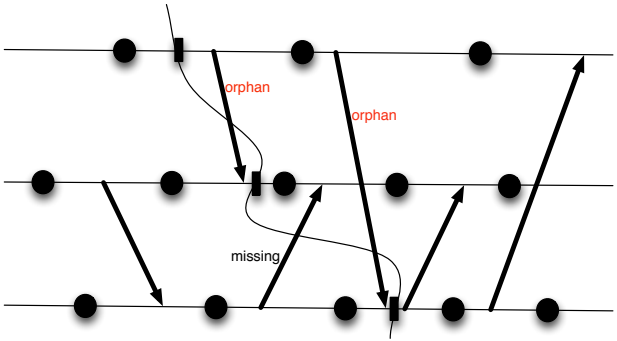
1 Introduction

2 Checkpointing Protocols

3 Application-Specific Fault Tolerance

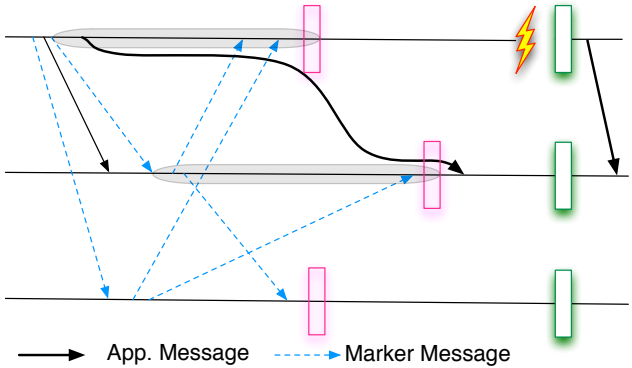
4 Conclusion and Future Work

Chandy-Lamport Lamport Algorithm (1984)



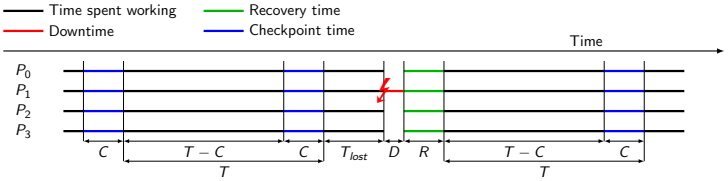
A message is *missing* if the sender sent it but the received did not receive it
A message is *orphan* if the receiver received it but the sender did not send it
A snapshot is consistent if there are no orphan messages and all missing messages are saved in the checkpoints

Blocking coordinated rollback-recovery protocol



The network is silenced during checkpoint waves, message deliveries are delayed until channels are flushed and checkpoints are taken

Sources of Waste

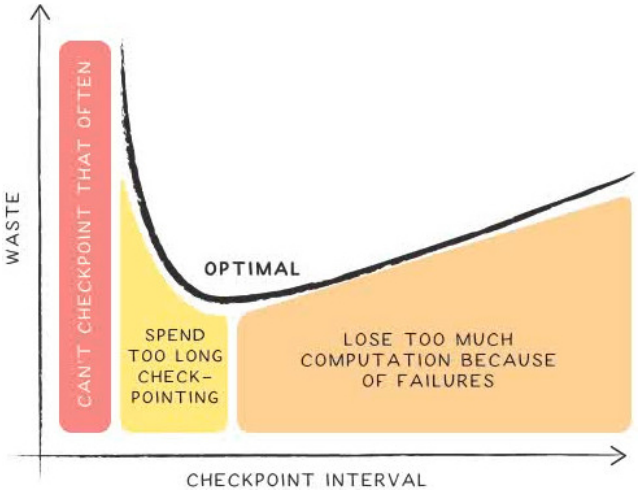


$$T_{\text{lost}} = \frac{T}{2} \text{ (true for large jobs)}$$

$$\text{WASTE}[\textit{FailureFree}] = \frac{C}{T}$$

$$\text{WASTE}[\textit{Failure}] = \frac{1}{\mu}\left(D + R + \frac{T}{2}\right)$$

Optimal checkpointing interval



First-order approximation [Young,Daly]:
Optimal period length:

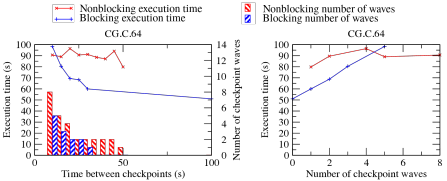
$$W^{\text{opt}} = \sqrt{2\mu C}$$

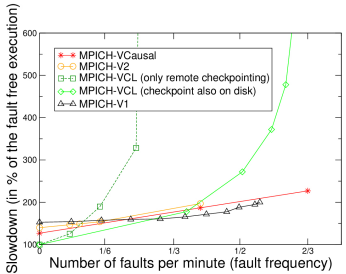
Corresponding overhead:

$$H_{\text{opt}} = \sqrt{\frac{2C}{\mu}}$$

In practice: checkpointing protocols

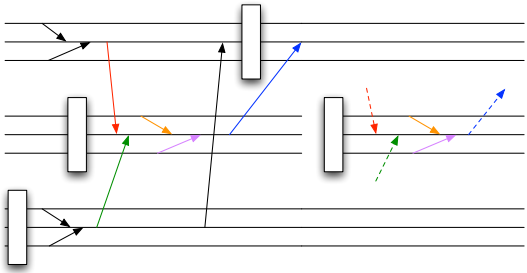
Rollback-Recovery in MPICH-V





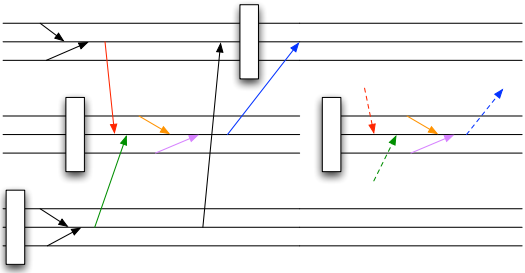
- MPICH-V: studied experimentally rollback-recovery protocols in a Message Passing Interface library (MPICH)
- Subject: how to checkpoint efficiently
  - Coordinated protocols (blocking, non-blocking)
  - Uncoordinated protocols (Optimistic, Pessimistic, Causal)

Hierarchical checkpointing



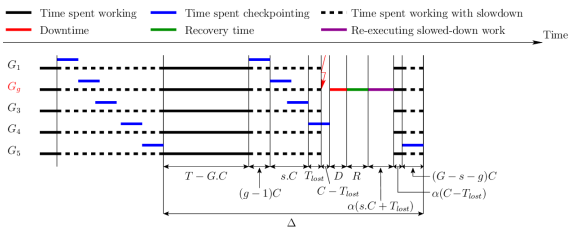
- Non-coordinated protocols with Message Logging:
  - Checkpoints are not coordinated
  - Messages (and other nondeterministic events) are logged and replayed
- Hierarchical Protocols:
  - Processors partitioned into G groups of q processors each
  - Coordinated checkpointing inside each group, noncoordinated between groups

Hierarchical checkpointing



- How to group processes:
  - Per physical resources (high probability of simultaneous failure)
  - Per communication group (high frequency of intra-group messages)
- Expected advantages:
  - ☺ no synchronization overheads
  - ☺ checkpoints are staggered
  - ☺ only failed processes need to restart

Modeling Hierarchical Protocols



Modeling the hierarchical protocol is challenging compared to the coordinated one

- Which group is subject to failure
- Failure between two checkpoint waves
- Failure within a wave
  - but before the group checkpoint
  - during the group checkpoint
  - after the group checkpoint
- Message logging impacts:
  - Work speed (re-execution is slightly faster)
  - Checkpoint size (and duration)
    - Checkpoint size increases with amount of work executed before the checkpoint

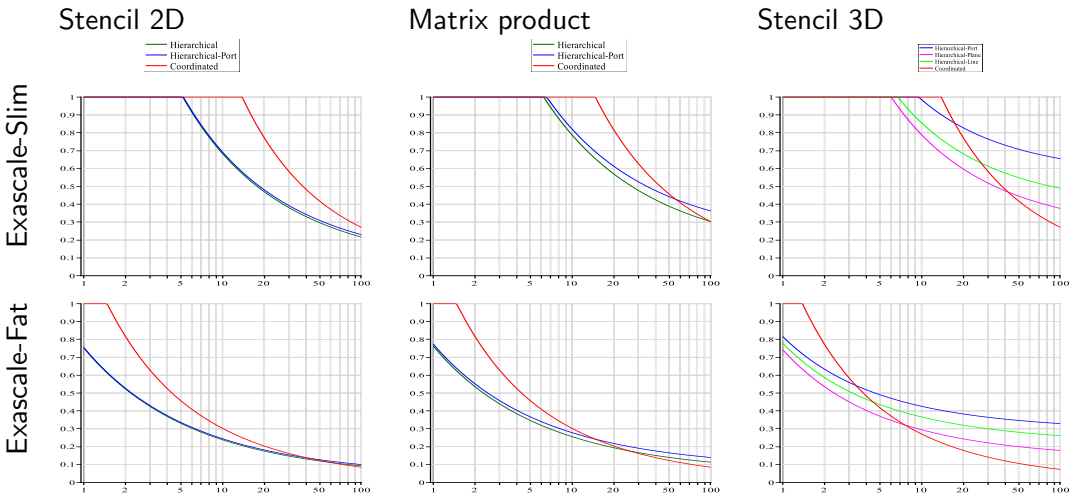
Platforms: basic characteristics

Name	Number of cores	Number of processors p_{total}	Number of cores per processor	Memory per processor	I/O Network Bandwidth (b_{io}) Read	I/O Network Bandwidth (b_{io}) Write	I/O Bandwidth (b_{port}) Read/Write per processor
Titan	299,008	16,688	16	32GB	300GB/s	300GB/s	20GB/s
Exascale-Slim	1,000,000,000	1,000,000	1,000	64GB	1TB/s	1TB/s	200GB/s
Exascale-Fat	1,000,000,000	100,000	10,000	640GB	1TB/s	1TB/s	400GB/s

Name	Scenario	G ($C(q)$)	β for 2D-Stencil	β for Matrix-Product
Titan	Coord-IO	1 (2,048s)	/	/
	Hierarch-IO	136 (15s)	0.0001098	0.0004280
	Hierarch-Port	1,246 (1.6s)	0.0002196	0.0008561
Exascale-Slim	Coord-IO	1 (1,000s/100s)	/	/
	Hierarch-IO	1,000 (64s)	0.0002599	0.001013
	Hierarch-Port	200,0000 (0.32s)	0.0005199	0.002026
Exascale-Fat	Coord-IO	1 (1,000s/100s)	/	/
	Hierarch-IO	316 (217s)	0.00008220	0.0003203
	Hierarch-Port	33,3333 (1.92s)	0.00016440	0.0006407

$\alpha = 0.3$, $\lambda = 0.98$ and $\rho = 1.5$

Waste for Prospective Exascale Platforms with $C = 100$



Waste as a function of processor MTBF μ_{ind}, $C = 100$

Conclusion on Hierarchical Checkpointing

- Model for Hierarchical Checkpointing is complex
- Validation via simulation and comparison with few deployments
- *All models are wrong, some are useful*
  - Study highlights better resilience for fat nodes for Exascale
  - Narrow gap (*application dependent*) of applicability for hierarchical approaches
- Message Logging is doomed? See future work

Outline

1 Introduction

2 Checkpointing Protocols

3 Application-Specific Fault Tolerance

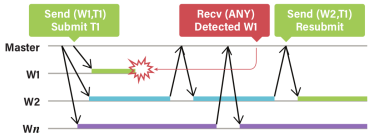
4 Conclusion and Future Work

ULFM: User-Level Failure Mitigation

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

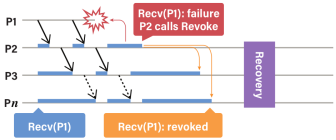
Continue Across Errors

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.



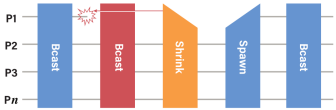
Exceptions in Contained Domains

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—that can be used to resume collective operations in malleable applications, spawn replacement processes in non-moldable applications, and rebuild RMA windows and files.



Outline

1 Introduction

2 Checkpointing Protocols

3 Application-Specific Fault Tolerance
- Failure Detection and Notification

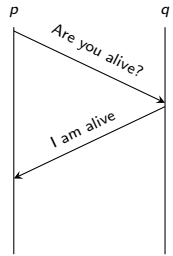
4 Conclusion and Future Work

ULFM: Failure Detection and Notification

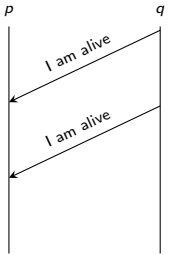
- Default Failure Detection: TCP time-out ($\sim 20mn$)
- Default Failure Notification: Admin network
- Work on fail-stop errors assumes *instantaneous* failure detection
- Goals:
  - Continue execution after crash of **several** nodes
  - Need *rapid* and *global* knowledge of group members
    1. Rapid: failure detection
    2. Global: failure notification
  - Resilience mechanism should have minimal impact

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)

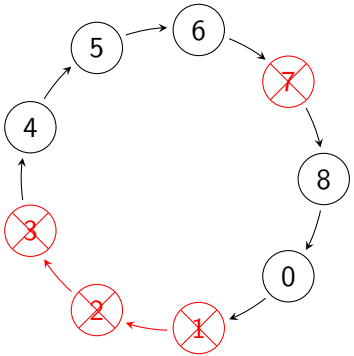


[1]: W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. IEEE Trans. Computers, 2002

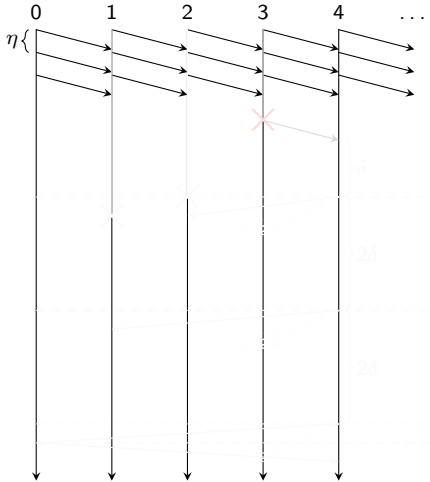
Algorithm for failure detection

- 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



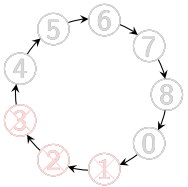
η: Heartbeat interval

δ: Timeout, $\delta >> \tau$

Heartbeat

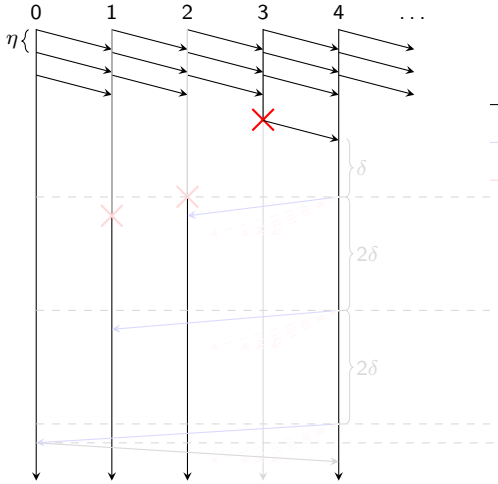
Reconnection message

Broadcast message



Ring reconnected

Reconnecting the ring



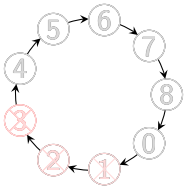
η: Heartbeat interval

δ: Timeout, $\delta >> \tau$

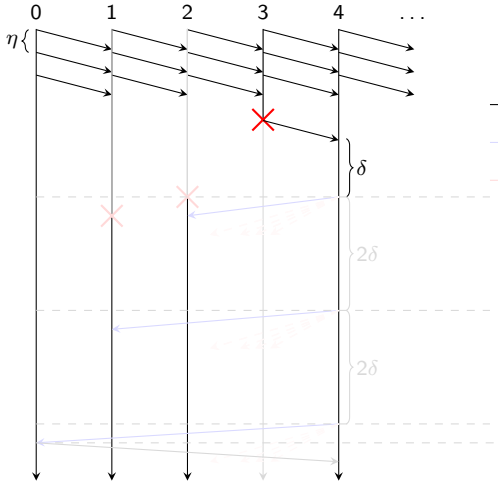
Heartbeat

Reconnection message

Broadcast message



Reconnecting the ring



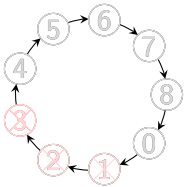
η: Heartbeat interval

δ: Timeout, $\delta >> \tau$

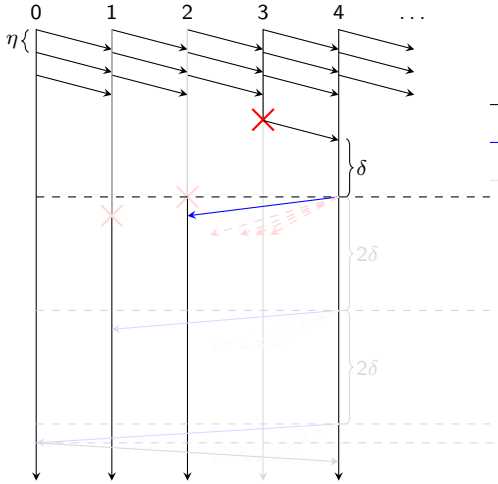
Heartbeat

Reconnection message

Broadcast message



Reconnecting the ring



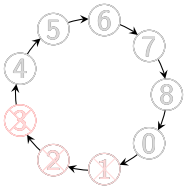
η: Heartbeat interval

δ: Timeout, $\delta >> \tau$

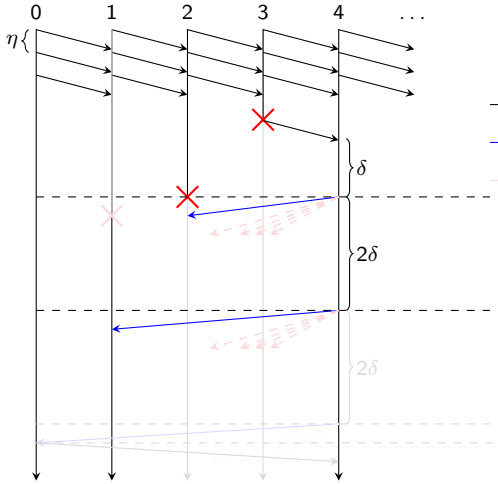
Heartbeat

Reconnection message

Broadcast message



Reconnecting the ring



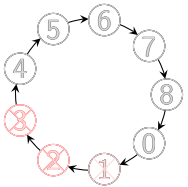
η: Heartbeat interval

δ: Timeout, $\delta >> \tau$

Heartbeat

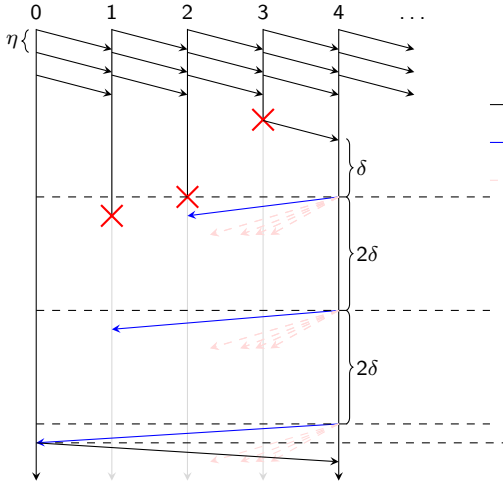
Reconnection message

Broadcast message



Ring reconnected

Reconnecting the ring



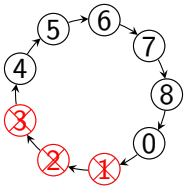
η: Heartbeat interval

δ: Timeout, $\delta >> \tau$

Heartbeat

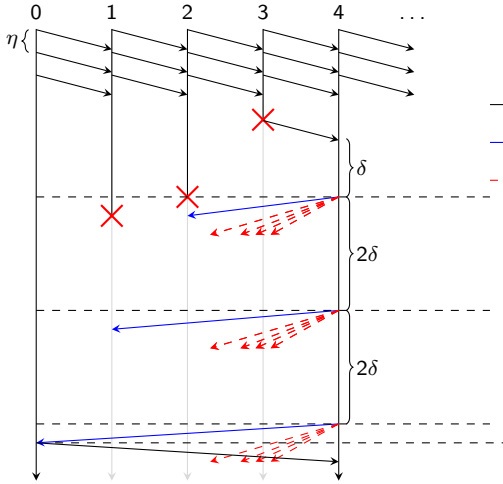
Reconnection message

Broadcast message



Ring reconnected

Reconnecting the ring



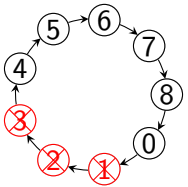
η: Heartbeat interval

δ: Timeout, $\delta >> \tau$

Heartbeat

Reconnection message

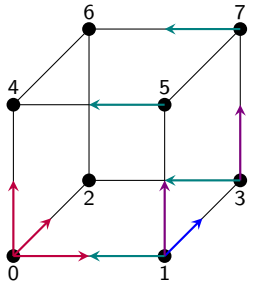
Broadcast message



Ring reconnected

Failure Notification

- Hypercube Broadcast Algorithm
  - 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)
  - 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)

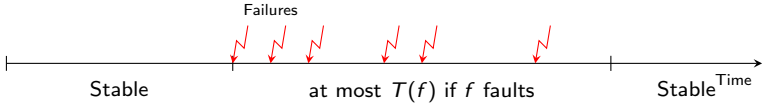


Node	Node1	Node2	Node4
1	0	0-2-3	0-4-5
2	0-1-3	0	0-4-6
3	0-1	0-2	0-4-5-7
4	0-1-5	0-2-6	0
5	0-1	0-2-6-7	0-4
6	0-1-3-7	0-2	0-4
7	0-1-3	0-2-6	0-4-5

Worst-case analysis

Definition

Stable configuration: all dead nodes are known to all processes



Theorem

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

$$T(f) \leq f(f+1)\delta + f\tau + \frac{f(f+1)}{2}(8\tau \log n)$$

- 2 sequential broadcasts: $4\tau logn$
- One-port model: broadcast messages and heartbeats interleaved

Worst-case scenario

$$T(f) \leq \underbrace{f(f+1)\delta + f\tau}_{\textit{reconstruction}} + \underbrace{\frac{f(f+1)}{2}(8\tau \log n)}_{\textit{broadcast}}$$

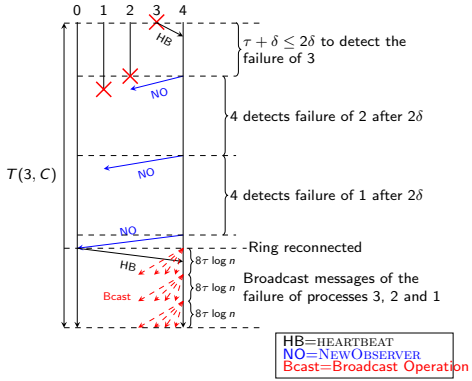
- $T(f) \leq$ ring reconstruction + broadcasts (for the proof)
- Process p discovers the death of q at most **once**
  $\Rightarrow i - th$ dead process discovered dead by at most $f - i + 1$ processes
  $\Rightarrow$ at most $\frac{f(f+1)}{2}$ broadcasts
- $R(f)$ ring reconstruction time
  For $1 \leq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

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

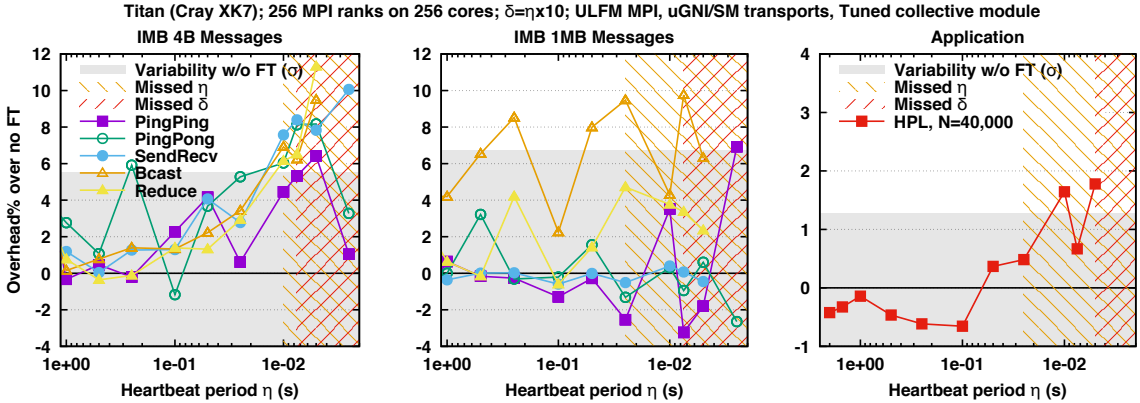
Ring reconnection

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

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

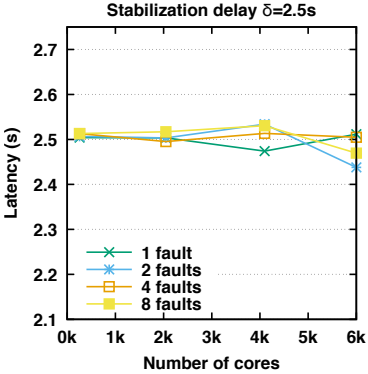


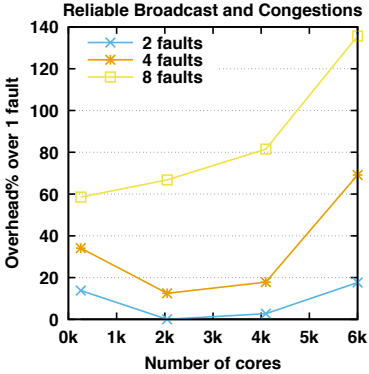
Noise

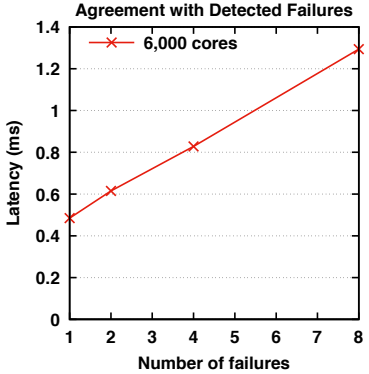

Titan (Cray XK7); 256 MPI ranks on 256 cores; δ=ηx10; ULFM MPI, uGNI/SM transports, Tuned collective module
IMB 4B Messages
IMB 1MB Messages
Application
Overhead% over no FT
Heartbeat period η (s)
Variability w/o FT (σ)
Missed η
Missed δ
PingPing
PingPong
SendRecv
Bcast
Reduce
HPL, N=40,000
1e+00
1e-01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

Detection and propagation delay

Titan (Cray XK7); 1 MPI rank/core; δ=ηx10; ULFM MPI, uGNI/SM transports, Tuned collective module







Outline

1 Introduction

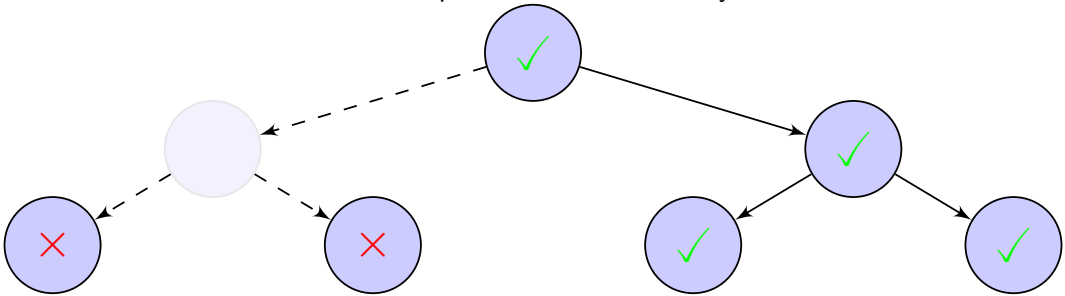
2 Checkpointing Protocols

3 Application-Specific Fault Tolerance
- Early Returning Agreement

4 Conclusion and Future Work

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.

Specification

Correctness

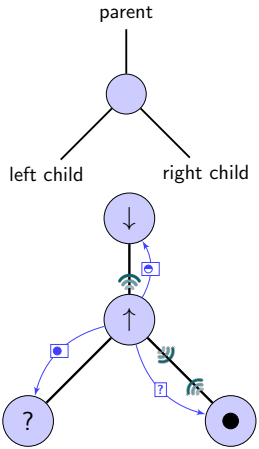
Termination	Every living process eventually decides.
Integrity	Once a living process decides a value, it remains decided on that value.
Agreement	No two living processes decide differently.
Participation	When a process decides upon a value, it contributed to the decided value.

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

Assumptions

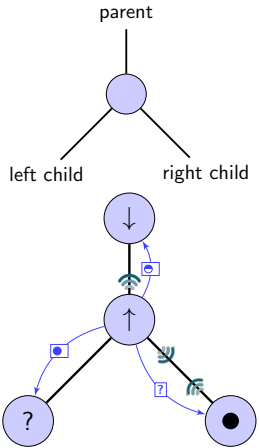
- Processes have totally ordered, unique identifiers
- Any process belonging to a group knows what processes belong to that group
- Any process may be subject to a permanent failure
- The network does not lose, modify, nor duplicate messages, but communication delays have *unknown* bounds
- The system 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*

Principle



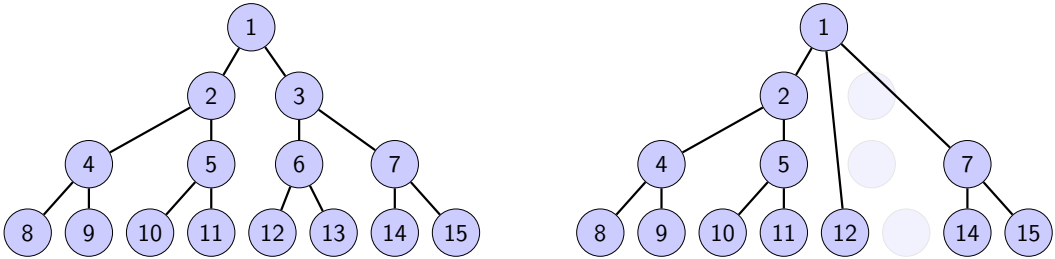
- Processes are arranged following a mendable tree topology: given a list of known dead processes, they communicate or monitor the liveliness of only their neighbors in that topology.
- The algorithm is a resilient version of Fan-in / Fan-out: all contributions (noted ◓) are reduced along the tree up 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.

Notation



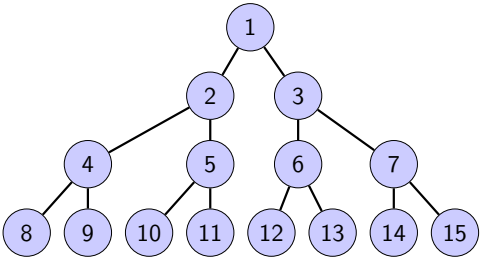
- Alive processes can be in 3 states:
  - ?, if they have not entered the consensus yet
  - ↑, if they are waiting from the contribution of their children
  - ↓, if they have sent their contribution to their parent and are waiting for the decision
  - ●, if they have received the decision
- There are 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

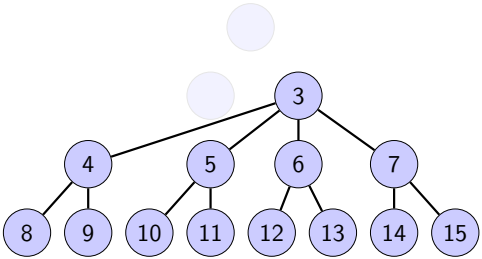
Mendable Tree for Consensus



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

Mendable Tree for Consensus

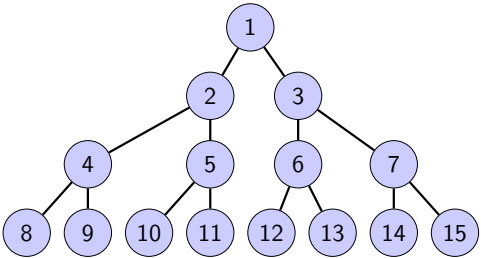


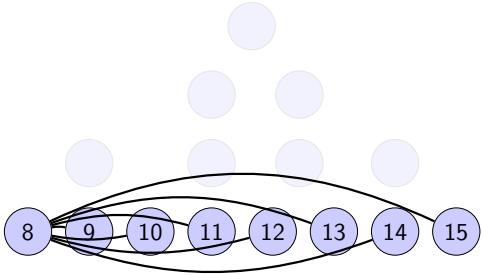


Nodes replace their parents by the highest-ranked alive ancester in the tree in case of failure.

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*

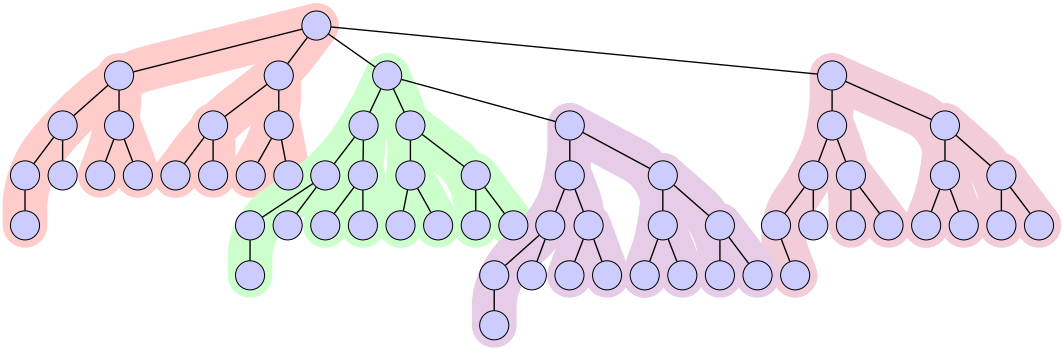
Mendable Tree for Consensus





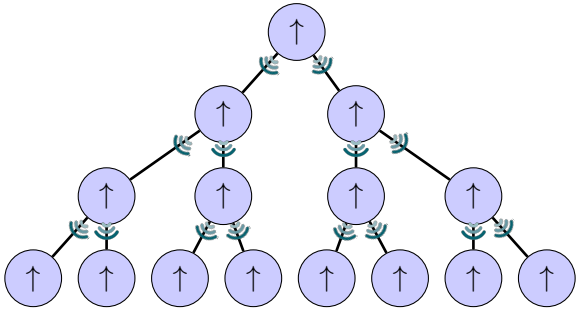
If half the processes die, the tree can, in the worst case, degenerate to a $np/2$-degree star

Architecture-Aware Tree



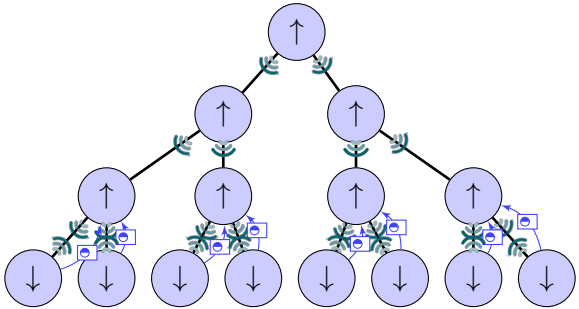
To map the hardware network hierarchy, two levels of trees are joined: In the example, *representative* processes of nodes (node0, node1, node2, node3) are interconnected following a *binary* tree, and processes belonging to the same node (16 process / node in this case) are also connected following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

No Failure



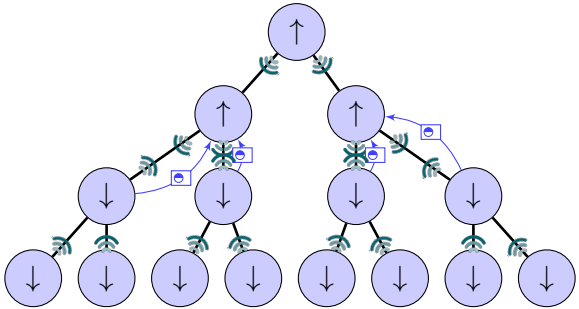
Initially, all processes are in the state ↑ to provide their participation, and the participation of their descendents to their ascendent. Each process monitors its descendents for possible failures ( ) until they have participated.

No Failure



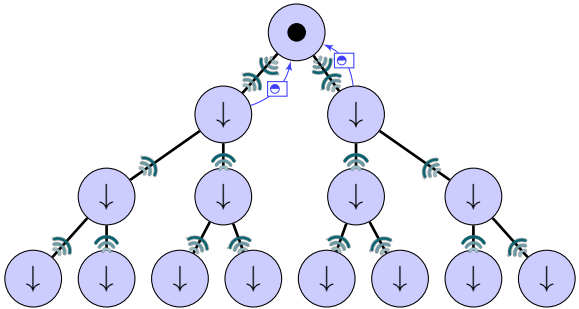
Leaves can send their participation (◓) to their parent, and enter the broadcasting state ↓. They start monitoring their parent for possible failures (📡)

No Failure



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

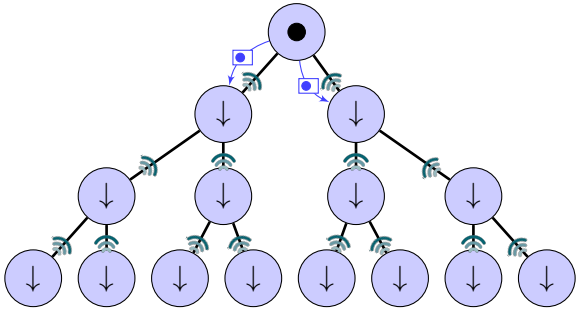
No Failure



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

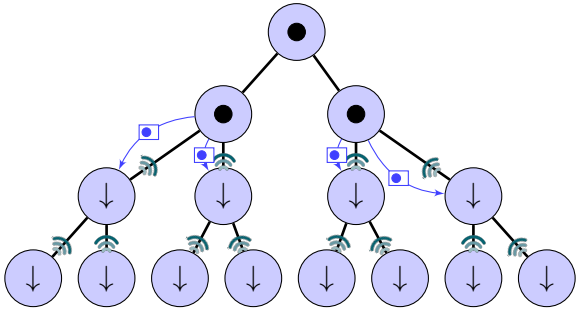
The root process can *decide* as soon as all descendents have contributed, it enters the decided state ●, starts broadcasting the decided message (●) to its descendents, and stops monitoring processes for failures

No Failure



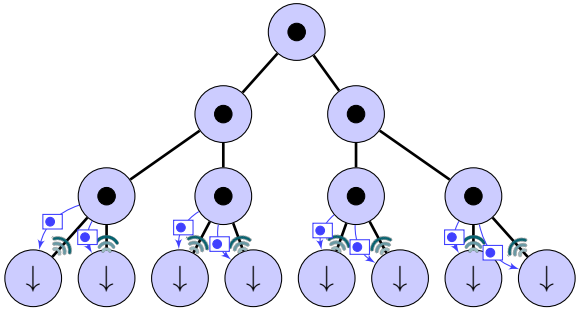
When a process receives a decision message (●), it decides, enters the decided state ●, and broadcasts the decision to its descendents, until all processes have decided

No Failure



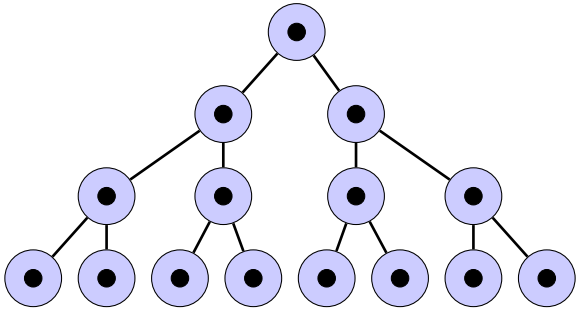
When a process receives a decision message (●), it decides, enters the decided state ●, and broadcasts the decision to its descendents, until all processes have decided

No Failure



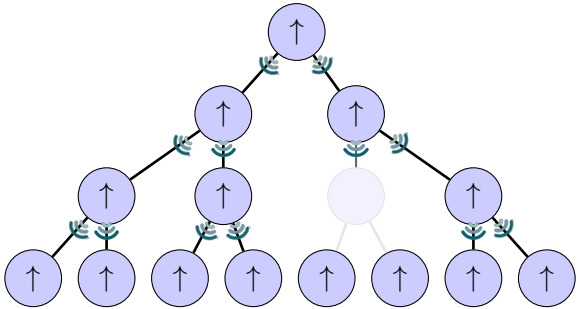
When a process receives a decision message (●), it decides, enters the decided state ●, and broadcasts the decision to its descendents, until all processes have decided

No Failure



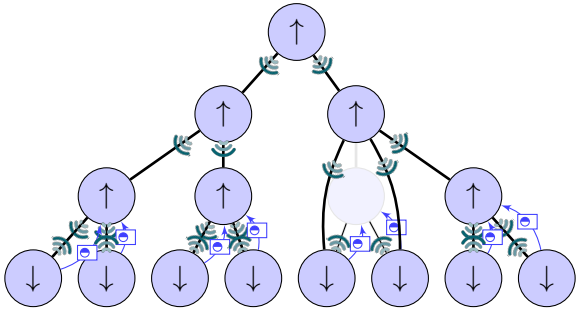
When a process receives a decision message (●), it decides, enters the decided state ●, and broadcasts the decision to its descendents, until all processes have decided

Failure before participating



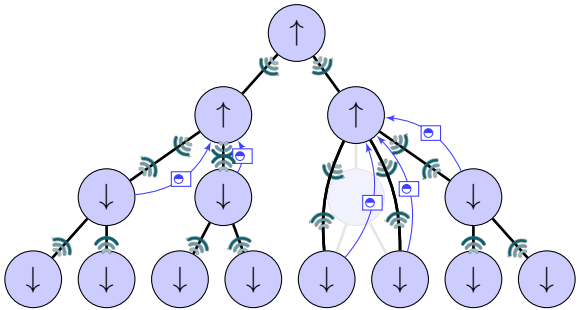
Process P_6 died before participating. P_3, its parent, started monitoring it ( ) when it entered the consensus (state $\uparrow$).

Failure before participating



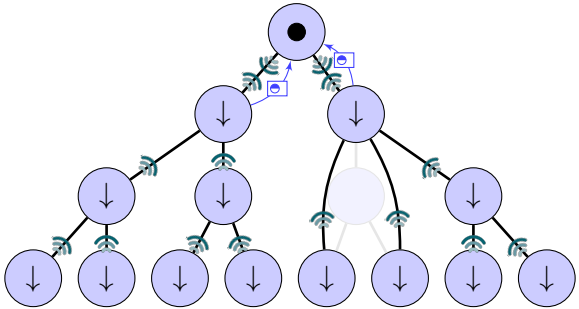
Processes P_{12} and P_{13} will send their participation (◓) to P_6, these messages are lost, and they start monitoring (📡) P_6. P_3 eventually discovers the death of P_6, and starts monitoring (📡) its new descendents P_{12} and P_{13}.

Failure before participating



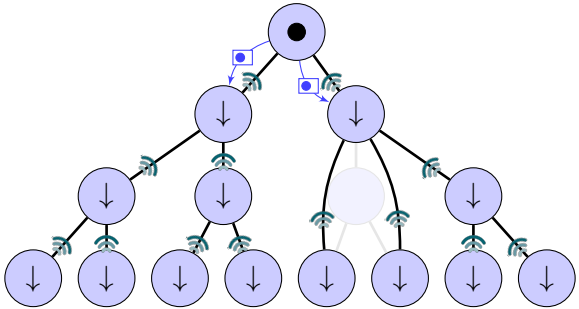
Processes P_{12} and P_{13} eventually discover the death of P_6, and take P_3 as their parent, sending it their participation (◓). They also start monitoring ((•)) their new parent, P_3.

Failure before participating



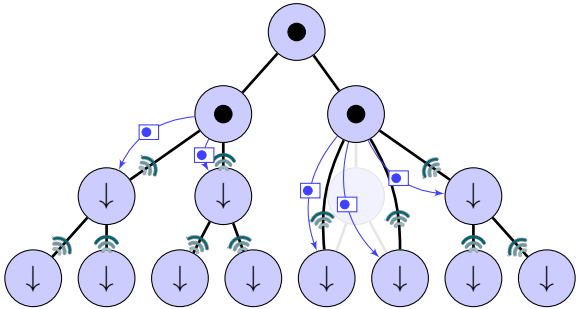
The tree being fixed, the information simply flows along the mended tree as initially.

Failure before participating



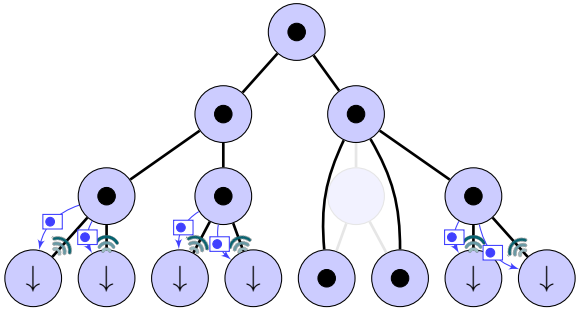
The tree being fixed, the information simply flows along the mended tree as initially.

Failure before participating



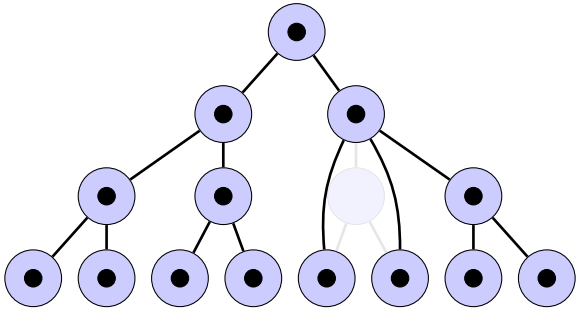
The tree being fixed, the information simply flows along the mended tree as initially.

Failure before participating



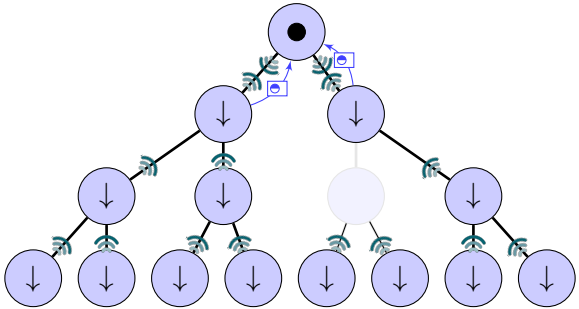
The tree being fixed, the information simply flows along the mended tree as initially.

Failure before participating



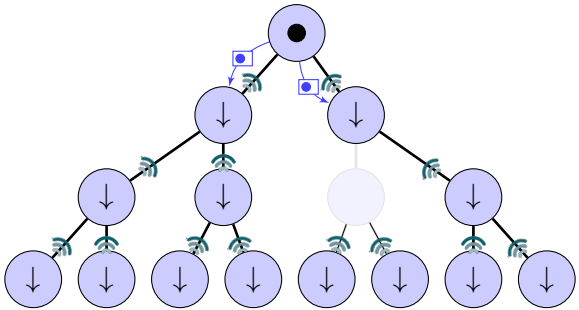
The tree being fixed, the information simply flows along the mended tree as initially.

Failure After Participating



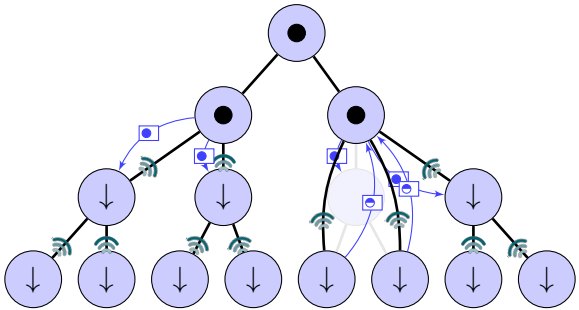
Process P_6 fails, but after participating to the current consensus.

Failure After Participating



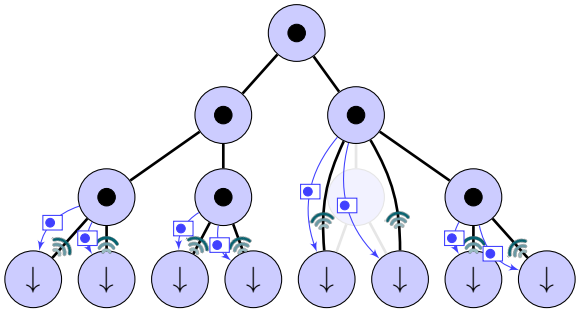
If it was a leaf, that would not prevent the consensus to complete. Since it has children, and they have not received the decision (●) yet, they are monitoring ( ) it, and eventually discover the death

Failure After Participating



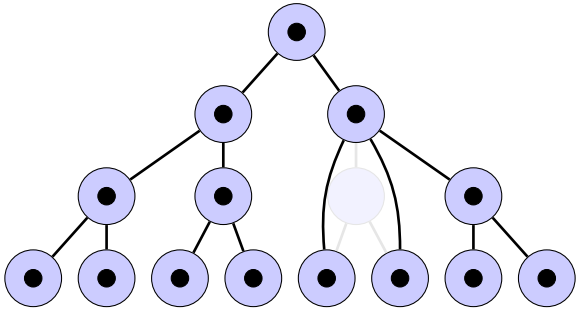
They send their participation (◓) back to their grand-parent, P_3, starting to monitor it (📡). This ensure that if P_6 died before forwarding it upward, their participartion (◓) is not lost. This also reconnects the tree.

Failure After Participating



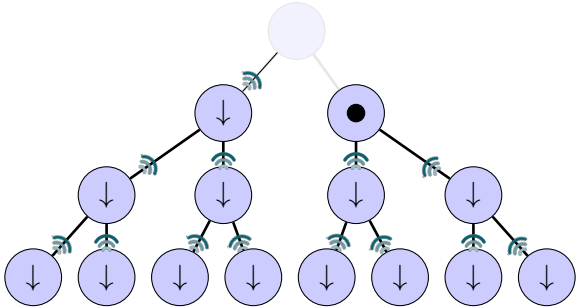
Even if P_3 is already done with the current consensus, it remembers the result (ERA property), and provides the result (●) again, allowing the information to continue flowing down the tree.

Failure After Participating



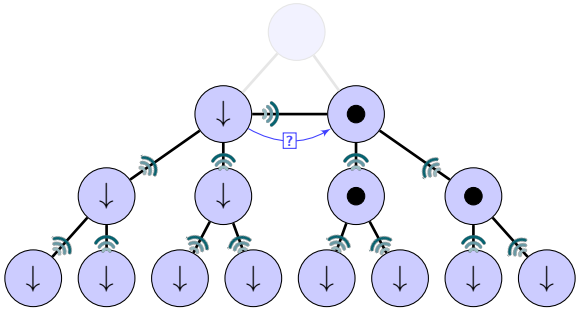
Even if P_3 is already done with the current consensus, it remembers the result (ERA property), and provides the result (●) again, allowing the information to continue flowing down the tree.

Failure of Root



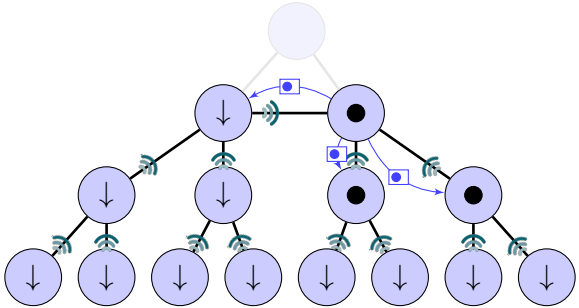
If the root of the tree dies after it started broadcasting the decision, but before it could reach all its children, the ones that did not receive the decision (●) are still monitoring that dead root (📡).

Failure of Root



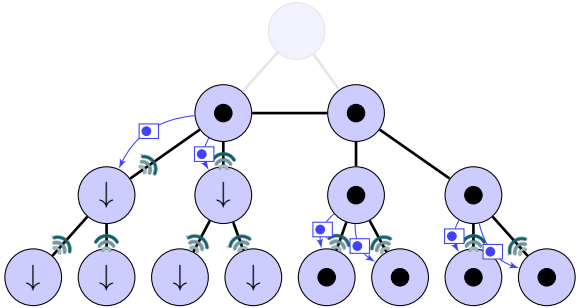
If a process becomes the root (lowest identifier), but was waiting for a decision, it asks all its new children if they received a decision before, by sending the message (?), and monitoring them ( ).

Failure of Root



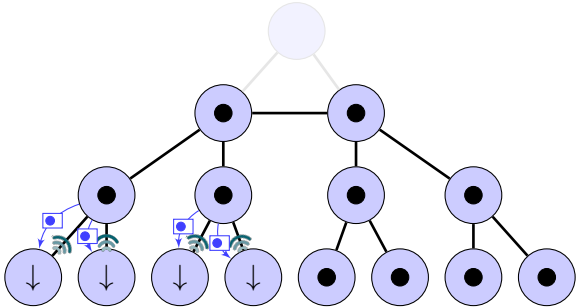
If one of them has the decision, it answers with it and the root can decide and broadcast (●). If none has it, they provide their participation (◓), if they reached that step, and wait for the decision of the new root.

Failure of Root



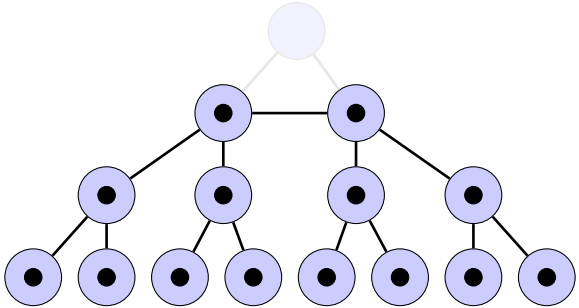
The broadcast of the decision (●) then continues along the tree

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

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.

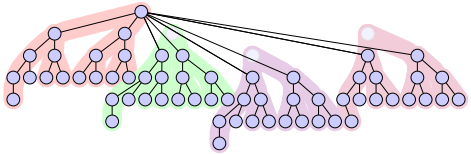
How to cleanup?

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

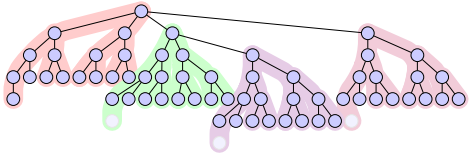
The result of this last ERA does not need to be remembered: if the communicator has been released, then all processes participated, and the return value is ignored.

Tree-Rebalancing

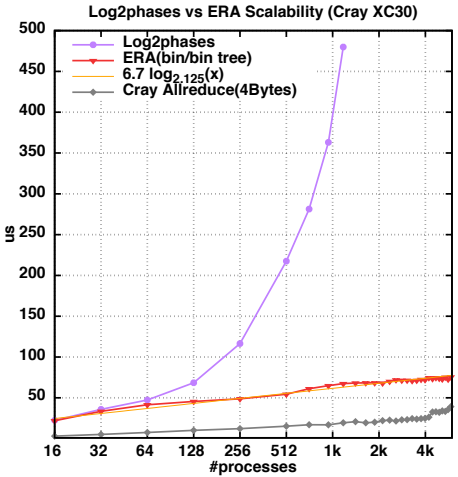
As processes crash, the Fan-in / Fan-out tree used to implement the two phases of the consensus can become unbalanced.



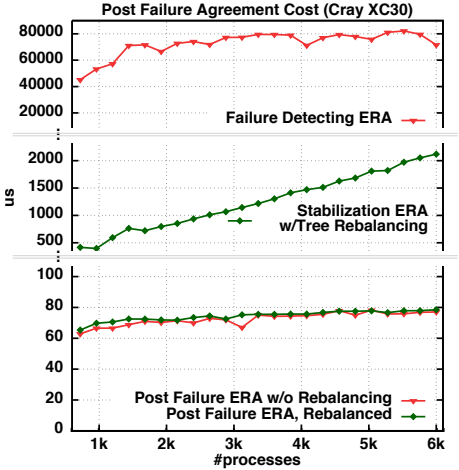
To implement the ULFM specification, all processes must agree on a list of failed nodes. Trees can be re-balanced when starting a new agreement based on that information.



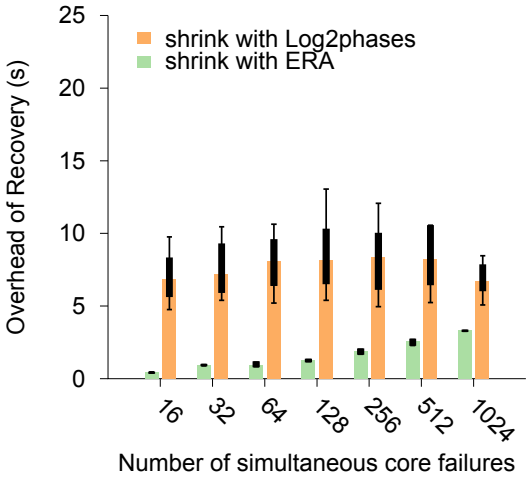
Agreement scalability in the failure-free case



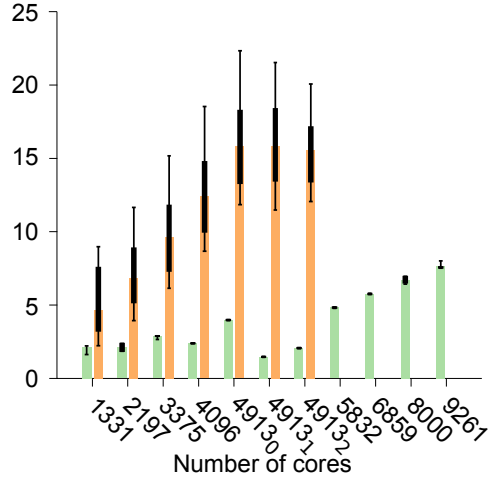
Post Failure Agreement Cost



FENIX & S3D Performance



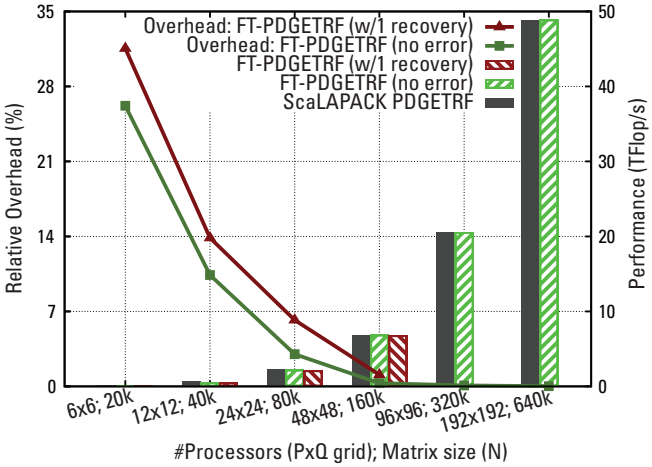
Simultaneous failures on an increasing number of cores, over 2197 total cores



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

Algorithm-Based Fault Tolerant LU over ULFM

- Algorithm-Based Fault Tolerant LU (PDGETRF) based on ULFM
  - Replication to ensure checksum availability
  - Partial checkpoint to rollback Q-panel operation
  - ABFT to recover most of the missing data
  - Partial re-execution to recover panel



Conclusion on Application-Specific Fault Tolerance

- Many applications are amenable to 'Roll-Forward' approaches
  - No Re-Execution, no TLost! (or very little)
- Performance of roll-forward is orders of magnitude higher than checkpoint/restart
  - And it scales better!
- Roll-forward is only for specific parts of real life applications
  - But we have composition techniques!
- Roll-forward requires a resilient communication middleware
  - ULFM is integrated in Open MPI and MPICH
  - Overhead on failure-free is non measurable
  - Overhead on failure-specific operation is very small
  - It is slowly being integrated in the MPI Standard

Outline

1 Introduction

2 Checkpointing Protocols

3 Application-Specific Fault Tolerance

4 Conclusion and Future Work

Current Practice for Fault Tolerance in HPC

Most applications do not have any fault-tolerance mechanisms
Long running applications mostly implement

- application-level checkpointing
- On disk (PFS)
  - Often using libraries (VeloC, HDF5, ADIOS2, ...)
  - Often with local cache / local staging
- Coordinated checkpoint and rollback recovery
- With full application-level synchronization
- More or less periodic (application-dependent)
- But not often using the optimal checkpointing period
  - MTBF is often unknown, and *allocations* define the period

Future Research on Fault Tolerance for HPC

- Scale remains the adversary
  - End of Dennard scaling (2006)
  - Moore's Law survives (more or less) by doubling cores (manycore)
  - or thanks to addition of accelerators
- 🤔☹ Will we continue to double the MTBF of nodes at the same speed we double their power?
- 🤔☹ Market argument: no-one will commission a supercomputer with $MTBF <\sim 12h$
  - ☹ See anecdotal evidence with Sunway TaihuLight
  - ☺ What is the cost of keeping up with the component reliability increase? What is the cost of introducing software fault tolerance? (SW_Qsim, GB 2021, uses a resilient all-reduce because of failures in new Sunway)
  - ☺'the likelyhood for a supercomputer running for weeks on end is approximately zero' – Jensen Huang's GTC Keynote, March 18, 2024.

Silent Data Corruption

SDCs are not well evaluated[1]
Intuition:

- They happen more often as the amount of memory grows
- They happen more often as the computational intensity grows

If the intuition is right, they will happen more and more frequently
Application-specific fault tolerance might help

- Not necessarily (or not only) to correct failures
- Mostly to detect failures

Overhead is negligible!

Advocating for hardened/trustworthy version of popular computational libraries

[1] In general we need more studies on faults in leadership systems and high-end systems

Future Research on Fault Tolerance for HPC (cont.)

- Energy consumption becomes a problem (both economical and ecological)
  - Current policy is to not bill allocation if a node within the allocation fails
  - Some platforms are switching to bill-per-Joule
  - Does it make sense to get 'free' Joules thanks to a node failure?
  - Cloud-HPC convergence: CPUs are billed as soon as allocated, even if the application cannot be deployed
- Using stranded energy in Cloud/HPC
  - Non-dispatchable renewable energy sources to power HPC
  - $\Rightarrow$ Power cost varies with time (soft version)
  - $\Rightarrow$ Power capping varies with time (hard version)
  - $\Rightarrow$ Job scheduler may have to kill apps. Resilience can help.

Resilience Techniques Applied to Manage Stragglers

- Stragglers (slow processes) $\sim$ Failed Processes
  - (In fully asynchronous systems, they are indistinguishable)
  - ABFT techniques: obtain $n - k$ results over n to decide
  - Extend MPI/ULFM for relaxed collectives / cancellable sends?
    - e.g. (All)Reduce with only $k < np$ participants?
    - e.g. Discard contributions from slow processes?

Applying ABFT to Task-Based Runtime Systems

- ABFT: bulk-synchronous 'by nature'
  - Need to keep the cheksum valid at each step
  - Efficient *because* checksum is updated by only extending synchronous operation
- Task-Based Runtime Systems: asynchronous 'by nature'
  - Add strict control flow to keep checksum up to date? ☹
  - How to react to failures? Conditional DAG?
- ABFT + Tasks:
  - Extend DAG with checksum / checksum update tasks
  - Introduce scheduling priorities to make sure checksums are updated
  - Define alternative paths to produce data, associate cost with paths
    - A block can be computed by inversing the checksum
    - but preferable path is to update directly

Looking back at hierarchical/noncoordinated checkpointing

'Narrow Gap' of applicability for hierarchical checkpointing
Advantages of noncoordinated checkpointing:

- no synchronization overheads (😕– but communications are impacted)
- checkpoints are staggered (😕– but staging-out of checkpoints works in coordinated)
- only failed processes need to restart (😕– but surviving processes need to wait in tightly-coupled applications)
  - 🙂 noncoordinated should be revisited for *moldable* applications
  - 🙂 Task Systems with work migration are ideal candidates

Future Work

- Energy! Silent Errors!
- Stragglers
- ABFT + Tasks
- Optimality of checkpointing in Task Systems
- Noncoordinated Checkpointing in Task Systems

Thank you

Questions?

Future Work

Nothing about AI !?!

Roll-forward approaches are probably interesting for AI applications:

- Relaxed collectives / stragglers management
- Completely ignore slow processes contribution?

Using Deep Learning / Reinforced Learning:

- Place checkpoints in Task Systems?
- Generate missing data for iterative/converging applications
- Silent Data Corruption Detector?

Questions?