An overview of fault-tolerant techniques for HPC

Thomas Hérault[1] & Yves Robert[1,2]

1 – University of Tennessee Knoxville
2 – ENS Lyon & Institut Universitaire de France

herault@icl.utk.edu | yves.robert@ens-lyon.fr

http://graal.ens-lyon.fr/~yrobert/sc13tutorial.pdf

SC'2013 Tutorial

Thanks

INRIA & ENS Lyon

- Anne Benoit
- Frédéric Vivien
- PhD students (Guillaume Aupy, Dounia Zaidouni)

UT Knoxville

- George Bosilca
- Aurélien Bouteiller
- Jack Dongarra

Others

- Franck Cappello, Argonne and UIUC-Inria joint lab
- Henri Casanova, Univ. Hawai'i
- Amina Guermouche, UIUC-Inria joint lab

Outline

Outline

1. Introduction (15mn)
   - Large-scale computing platforms
   - Faults and failures
2. General-purpose fault-tolerance techniques (30mn)
   - Replication
   - Process Checkpointing
   - Coordinated Checkpointing
   - Uncoordinated checkpointing

Outline

Exascale platforms (courtesy Jack Dongarra)

Potential System Architecture with a cap of $200M and 20MW

Systems	2011 K computer	2019	Difference Today & 2019
System peak	**10.5 Pflop/s**	**1 Eflop/s**	**O(100)**
Power	**12.7 MW**	**~20 MW**	
System memory	1.6 PB	32 - 64 PB	O(10)
Node performance	128 GF	1,2 or 15TF	O(10) – O(100)
Node memory BW	64 GB/s	2 - 4TB/s	O(100)
Node concurrency	8	O(1k) or 10k	O(100) – O(1000)
Total Node Interconnect BW	20 GB/s	200-400GB/s	O(10)
System size (nodes)	88,124	O(100,000) or O(1M)	O(10) – O(100)
Total concurrency	705,024	O(billion)	O(1,000)
MTTI	days	O(1 day)	- O(10)

Exascale platforms (courtesy C. Engelmann & S. Scott)

Toward Exascale Computing (My Roadmap)

Based on proposed DOE roadmap with MTTI adjusted to scale linearly

Systems	2009	2011	2015	2018
System peak	2 Peta	20 Peta	100-200 Peta	1 Exa
System memory	0.3 PB	1.6 PB	5 PB	10 PB
Node performance	125 GF	200GF	200-400 GF	1-10TF
Node memory BW	25 GB/s	40 GB/s	100 GB/s	200-400 GB/s
Node concurrency	12	32	O(100)	O(1000)
Interconnect BW	1.5 GB/s	22 GB/s	25 GB/s	50 GB/s
System size (nodes)	18,700	100,000	500,000	O(million)
Total concurrency	225,000	3,200,000	O(50,000,000)	O(billion)
Storage	15 PB	30 PB	150 PB	300 PB
IO	0.2 TB/s	2 TB/s	10 TB/s	20 TB/s
MTTI	4 days	19 h 4 min	3 h 52 min	1 h 56 min
Power	6 MW	~10MW	~10 MW	~20 MW

Exascale platforms

- Hierarchical
  - 10^5 or 10^6 nodes
  - Each node equipped with 10^4 or 10^3 cores
- Failure-prone

MTBF – one node	1 year	10 years	120 years
MTBF – platform of 10^6 nodes	30sec	5mn	1h

More nodes ⇒ Shorter MTBF (Mean Time Between Failures)

Exascale platforms

- Hierarchica
  - 10^5 or 10^6 node
  - Each node equipped wi 10^4 10^3 cores
- Failure-prone

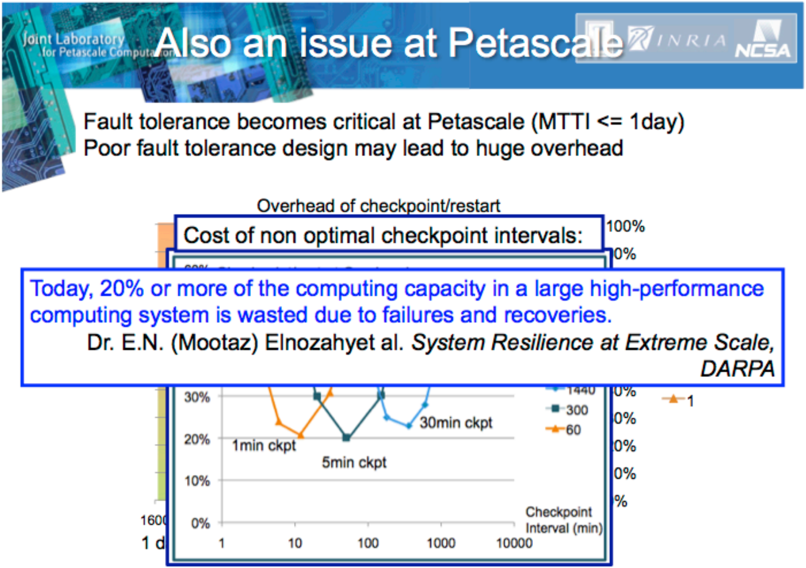
MTBF – or node	1 year	10 ars	120 years
MTBF – atform of 10^6 nodes	30sec	5m	1h

Mor odes = een ilures)

Exascale

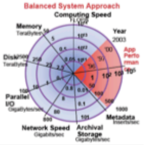
$\neq$ Petascale $\times 1000$

Even for today's platforms (courtesy F. Cappello)

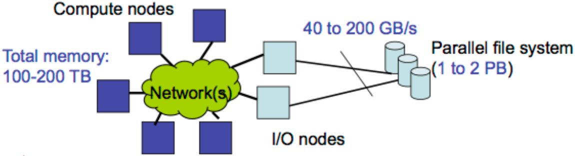


Even for today's platforms (courtesy F. Cappello)

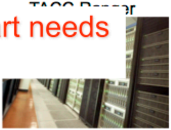
Classic approach for FT: Checkpoint-Restart



Typical "Balanced Architecture" for PetaScale Computers



RoadRunner



Without optimization, Checkpoint-Restart needs about 1h! (~30 minutes each)

Systems	Perf.	Ckpt time	Source
RoadRunner	1PF	~20 min.	Panasas
LLNL BG/L	500 TF	>20 min.	LLNL
LLNL Zeus	11TF	26 min.	LLNL
YYY BG/P	100 TF	~30 min.	YYY

LLNL BG/L

Scenario for 2015

- Phase-Change memory
  - read bandwidth 100GB/sec
  - write bandwidth 10GB/sec
- Checkpoint size 128GB
- C: checkpoint save time: $C = 12sec$
- R: checkpoint recovery time: $R = 1.2sec$
- D: down/reboot time: $D = 15sec$
- p: total number of (multicore) nodes: $p = 2^8$ to $p = 2^{20}$
- MTBF $\mu = 1$ week, 1 month, 1|10|100|1000 years (per node)

Distribution of parallel jobs

Number of processors required by typical jobs: *two-stage log-uniform distribution biased to powers of two* (says Dr. Feitelson)

- Let $p = 2^Z$ for simplicity
- Probability that a job is sequential: $\alpha_0 = p_1 \approx 0.25$
- Otherwise, the job is parallel, and uses 2^j processors with identical probability
- **Steady-state** utilization of whole platform:
  - all processors always active
  - constant proportion of jobs using any number of processors

Platform throughput with optimal checkpointing period

	p	**Throughput**
μ =1 week	2^8	91.56%
	2^{11}	73.75%
	2^{14}	20.07%
	2^{17}	2.51%
	2^{20}	0.31%

	p	**Throughput**
μ =1 month	2^8	96.04%
	2^{11}	88.23%
	2^{14}	62.28%
	2^{17}	10.66%
	2^{20}	1.33%

	p	**Throughput**
μ =1 year	2^8	98.89%
	2^{11}	96.80%
	2^{14}	90.59%
	2^{17}	70.46%
	2^{20}	15.96%

	p	**Throughput**
μ =10 years	2^8	99.65%
	2^{11}	99.00%
	2^{14}	97.15%
	2^{17}	91.63%
	2^{20}	74.01%

	p	**Throughput**
μ =100 years	2^8	99.89%
	2^{11}	99.69%
	2^{14}	99.11%
	2^{17}	97.45%
	2^{20}	92.56%

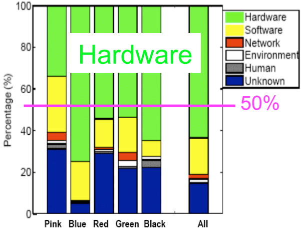
	p	**Throughput**
μ =1000 years	2^8	99.97%
	2^{11}	99.90%
	2^{14}	99.72%
	2^{17}	99.20%
	2^{20}	97.73%

Outline

Error sources (courtesy Franck Cappello)

Sources of failures

- Analysis of error and failure logs
- In 2005 (Ph. D. of CHARNG-DA LU) : "Software halts account for the most number of outages (59-84 percent), and take the shortest time to repair (0.6-1.5 hours). Hardware problems, albeit rarer, need 6.3-100.7 hours to solve."
- In 2007 (Garth Gibson, ICPP Keynote):



Relative frequency of root cause by system type.

- In 2008 (Oliner and J. Stearley, DSN Conf.):

Type	Raw Count	Raw %	Filtered Count	Filtered %
Hardware	174,586,516	98.04	1,999	18.78
Software	144,899	0.08	6,814	64.01
Indeterminate	3,350,044	1.88	1,832	17.21

Software errors: Applications, OS bug (kernel panic), communication libs, File system error and other.

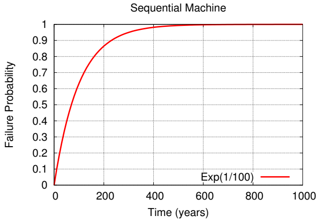
Hardware errors, Disks, processors, memory, network

Conclusion: Both Hardware and Software failures have to be considered

A few definitions

- Many types of faults: software error, hardware malfunction, memory corruption
- Many possible behaviors: silent, transient, unrecoverable
- Restrict to faults that lead to application failures
- This includes all hardware faults, and some software ones
- Will use terms *fault* and *failure* interchangeably

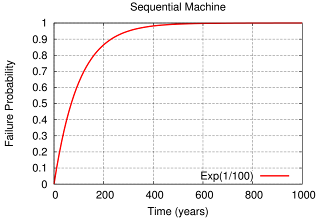
Failure distributions: (1) Exponential



$Exp(\lambda)$: Exponential distribution law of parameter λ:

- Pdf: $f(t) = \lambda e^{-\lambda t}$ for $t \geq 0$
- Cdf: $F(t) = 1 - e^{-\lambda t}$
- Mean $= \frac{1}{\lambda}$

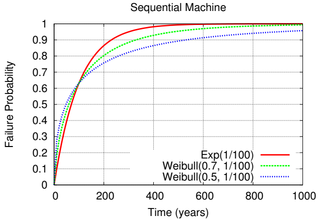
Failure distributions: (1) Exponential



X random variable for $Exp(\lambda)$ failure inter-arrival times:

- $\mathbb{P}(X \leq t) = 1 - e^{-\lambda t}$ (by definition)
- Memoryless property: $\mathbb{P}(X \geq t + s \mid X \geq s) = \mathbb{P}(X \geq t)$ at any instant, time to next failure does not depend upon time elapsed since last failure
- Mean Time Between Failures (MTBF) $\mu = \mathbb{E}(X) = \frac{1}{\lambda}$

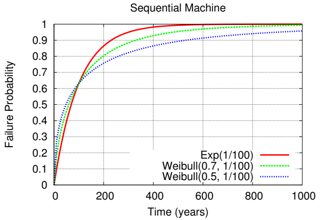
Failure distributions: (2) Weibull



Weibull(k, λ): Weibull distribution law of shape parameter k and scale parameter λ:

- Pdf: $f(t) = k\lambda(t\lambda)^{k-1}e^{-(\lambda t)^k}$ for $t \geq 0$
- Cdf: $F(t) = 1 - e^{-(\lambda t)^k}$
- Mean $= \frac{1}{\lambda}\Gamma(1 + \frac{1}{k})$

Failure distributions: (2) Weibull



X random variable for $\mathit{Weibull}(k, \lambda)$ failure inter-arrival times:

- If $k < 1$: failure rate decreases with time
  "infant mortality": defective items fail early
- If $k = 1$: $\mathit{Weibull}(1, \lambda) = \mathit{Exp}(\lambda)$ constant failure time

Failure distributions: with several processors

- Processor (or node): any entity subject to failures
  $\Rightarrow$ approach agnostic to granularity

- If the MTBF is μ with one processor,
  what is its value with p processors?

- Well, it depends ☹

Failure distributions: with several processors

- Processor (or node): any entity subject to failures
  ⇒ approach agnostic to granularity
- If the MTBF is μ with one processor,
  what is its value with p processors?
- Well, it depends ☹

With rejuvenation

- Rebooting all p processors after a failure
- Platform failure distribution
  $\Rightarrow$ minimum of p IID processor distributions
- With p distributions $Exp(\lambda)$:

$$\min\left(Exp(\lambda_1), Exp(\lambda_2)\right) = Exp(\lambda_1 + \lambda_2)$$

$$\mu = \frac{1}{\lambda} \Rightarrow \mu_p = \frac{\mu}{p}$$

- With p distributions $Weibull(k, \lambda)$:

$$\min_{1..p}\left(Weibull(k, \lambda)\right) = Weibull(k, p^{1/k}\lambda)$$

$$\mu = \frac{1}{\lambda}\Gamma(1 + \frac{1}{k}) \Rightarrow \mu_p = \frac{\mu}{p^{1/k}}$$

Without rejuvenation (= real life)

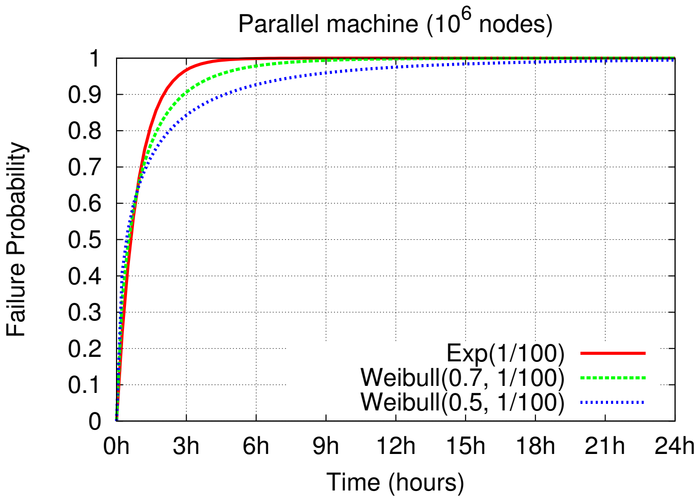
- Rebooting only faulty processor
- Platform failure distribution
  $\Rightarrow$ superposition of p IID processor distributions

Theorem: $\mu_p = \dfrac{\mu}{p}$ for arbitrary distributions

Values from the literature

- MTBF of one processor: between 1 and 125 years
- Shape parameters for Weibull: $k = 0.5$ or $k = 0.7$
- Failure trace archive from INRIA (`http://fta.inria.fr`)
- Computer Failure Data Repository from LANL (`http://institutes.lanl.gov/data/fdata`)

Does it matter?



Outline

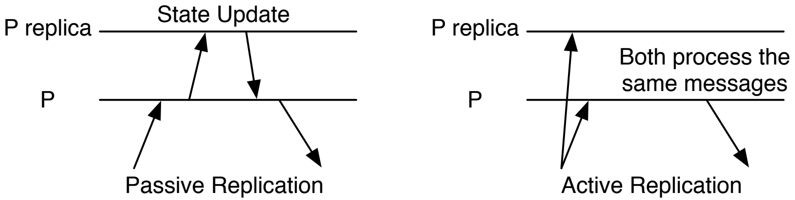
Maintaining Redundant Information

Goal

- General Purpose Fault Tolerance Techniques: work despite the application behavior
- Two adversaries: Failures & Application
- Use automatically computed redundant information
  - At given instants: checkpoints
  - At any instant: replication
  - Or anything in between: checkpoint + message logging

Outline

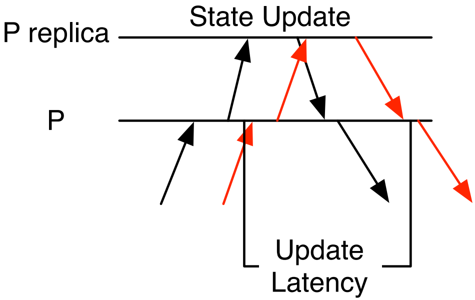
Replication



Idea

- Each process is replicated on a resource that has small chance to be hit by the same failure as its replica
- In case of failure, one of the replicas will continue working, while the other recovers
- Passive Replication / Active Replication

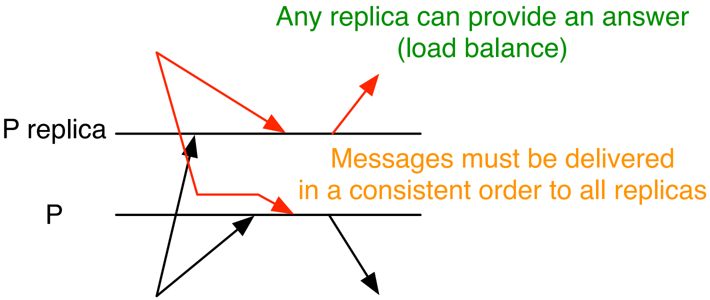
Replication



Challenges

- Passive replication: latency of state update
- Active replication: ordering of decision → internal additional communications

Replication



Challenges

- Passive replication: latency of state update
- Active replication: ordering of decision → internal additional communications

Outline

Process Checkpointing

Goal

- Save the current state of the *process*
    - FT Protocols save a *possible* state of the parallel *application*

Techniques

- User-level checkpointing
- System-level checkpointing
- Blocking call
- Asynchronous call

User-level checkpointing

User code serializes the state of the process in a file.

- Usually small(er than system-level checkpointing)
- Portability
- Diversity of use

- Hard to implement if preemptive checkpointing is needed
- Loss of the functions call stack
  - code full of jumps
  - loss of internal library state

System-level checkpointing

- Different possible implementations: OS syscall; dynamic library; compiler assisted
- Create a serial file that can be loaded in a process image. Usually on the same architecture, same OS, same software environment.

- Entirely transparent
- Preemptive (often needed for library-level checkpointing)

- Lack of portability
- Large size of checkpoint ($\approx$ memory footprint)

Blocking / Asynchronous call

Blocking Checkpointing

Relatively intuitive: `checkpoint(filename)`

Cost: no process activity during the whole checkpoint operation.

Can be linear in the size of memory and in the size of modified files

Asynchronous Checkpointing

System-level approach: make use of copy on write of `fork` syscall

User-level approach: critical sections, when needed

Storage

Remote Reliable Storage

Intuitive. I/O intensive. Disk usage.

Memory Hierarchy

- local memory
- local disk (SSD, HDD)
- remote disk
  - Scalable Checkpoint Restart Library
    `http://scalablecr.sourceforge.net`

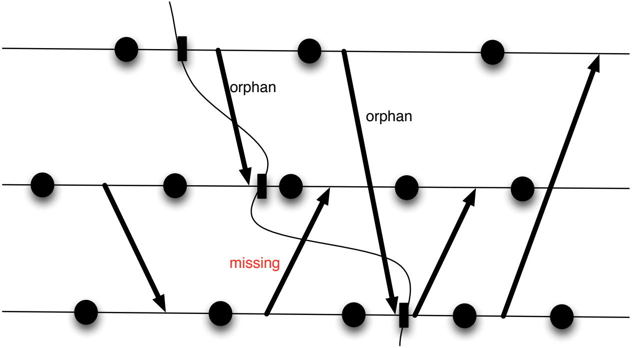
Checkpoint is valid when finished on reliable storage

Distributed Memory Storage

- In-memory checkpointing
- Disk-less checkpointing

Outline

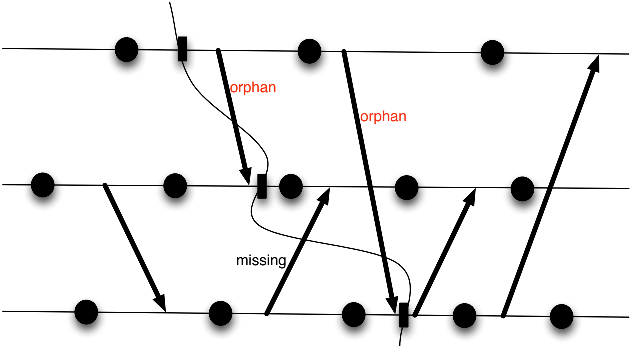
Coordinated checkpointing



Definition (Missing Message)

A message is missing if in the current configuration, the sender sent, while the receiver did not receive it

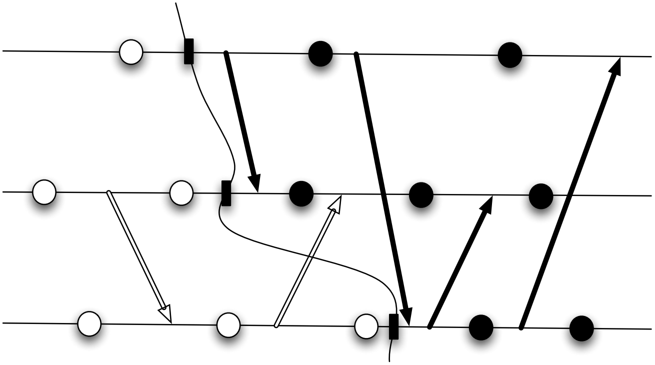
Coordinated checkpointing



Definition (Orphan Message)

A message is orphan if in the current configuration, the receiver received it, while the sender did not send it

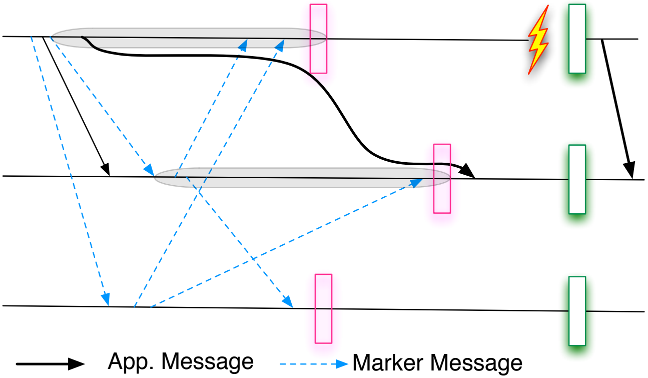
Coordinated Checkpointing Idea



Create a consistent view of the application

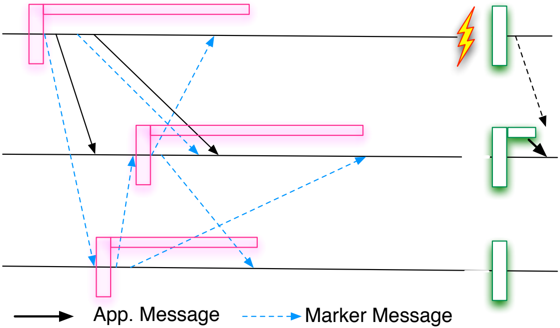
- Messages belong to a checkpoint wave or another
- All communication channels must be flushed (all2all)

Blocking Coordinated Checkpointing



- Silences the network during the checkpoint

Non-Blocking Coordinated Checkpointing



- Communications received after the beginning of the checkpoint and before its end are added to the receiver's checkpoint
- Communications inside a checkpoint are pushed back at the beginning of the queues

Implementation

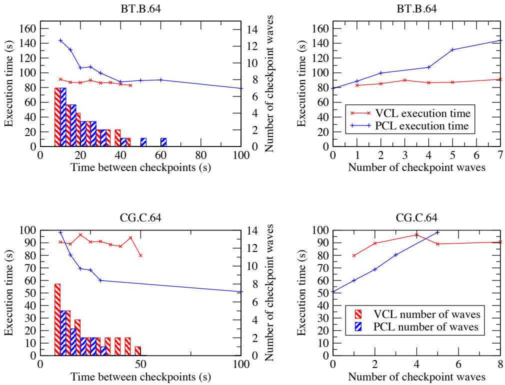
Communication Library

- Flush of communication channels
  - Conservative approach. One Message per open channel / One message per channel
- Preemptive checkpointing usually required
  - Can have a user-level checkpointing, but requires one that can be called any time

Application Level

- Flush of communication channels
  - Can be as simple as `Barrier(); Checkpoint();`
  - Or as complex as having a `quiesce();` function in all libraries
- User-level checkpointing

Coordinated Protocol Performance

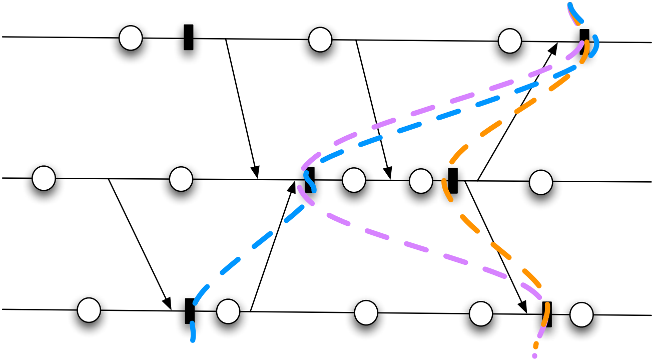


Coordinated Protocol Performance

- VCL = nonblocking coordinated protocol
- PCL = blocking coordinated protocol

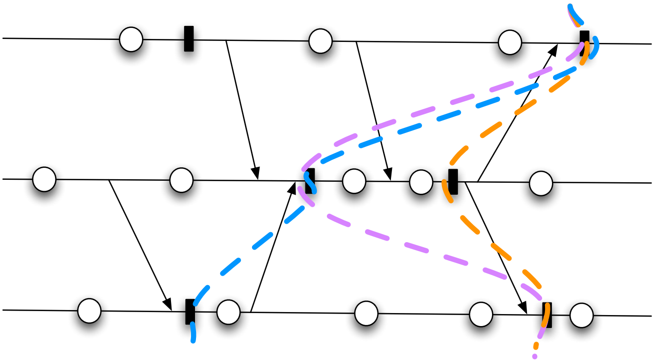
Outline

Uncoordinated Checkpointing Idea



Processes checkpoint independently

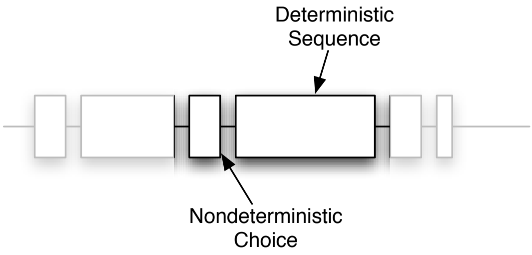
Uncoordinated Checkpointing Idea



Optimistic Protocol

- Each process i keeps some checkpoints C_i^j
- $\forall (i_1, \dots i_n), \exists j_k / \{C_{i_k}^{j_k}\}$ form a consistent cut?
- Domino Effect

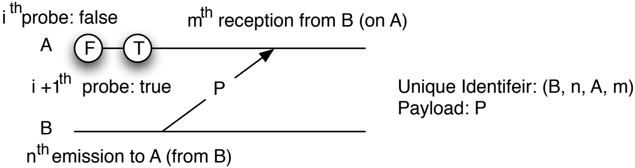
Piece-wise Deterministic Assumption



Piece-wise Deterministic Assumption

- Process: alternate sequence of non-deterministic choice and deterministic steps
- Translated in Message Passing:
  - Receptions / Progress test are non-deterministic (`MPI_Wait(ANY_SOURCE)`, `if( MPI_Test() )<...>; else <...>`)
  - Emissions / others are deterministic

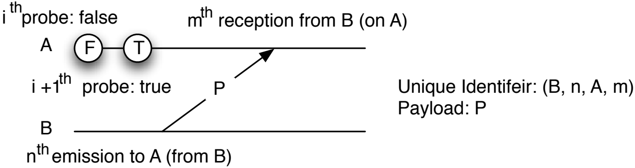
Message Logging



Message Logging

By replaying the sequence of messages and test/probe with the same result that it obtained in the initial execution (from the last checkpoint), one can guide the execution of a process to its exact state just before the failure

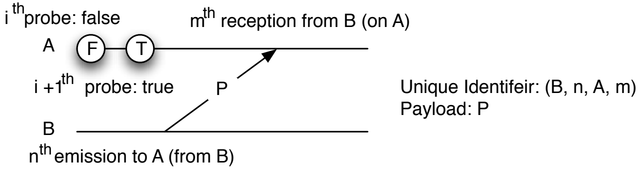
Message Logging



Message / Events

- Message = unique identifier (source, emission index, destination, reception index) + payload (content of the message)
- Probe = unique identifier (number of consecutive failed/success probes on this link)
- Event Logging: saving the unique identifier of a message, or of a probe

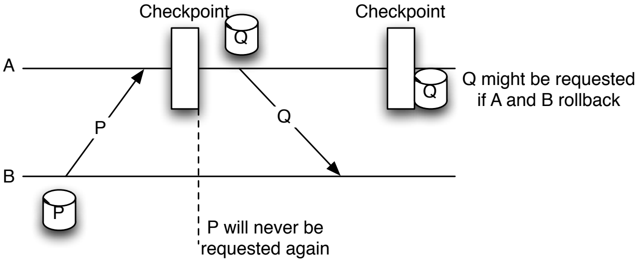
Message Logging



Message / Events

- Payload Logging: saving the content of a message
- Message Logging: saving the unique identifier and the payload of a message, saving unique identifiers of probes, saving the (local) order of events

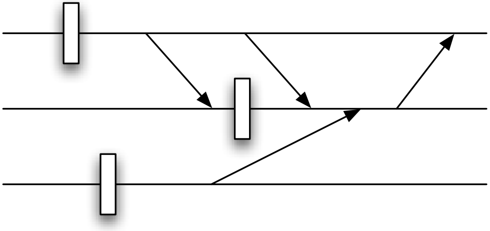
Message Logging



Where to save the Payload?

- Almost always as Sender Based
- Local copy: less impact on performance
- More memory demanding → trade-off garbage collection algorithm
- Payload needs to be included in the checkpoints

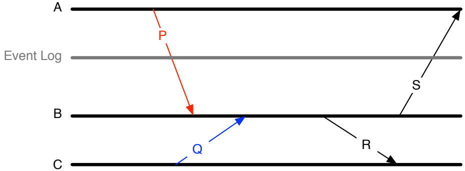
Message Logging



Where to save the Events?

- Events must be saved on a reliable space
- Must avoid: loss of events ordering information, for all events that can impact the outgoing communications
- Two (three) approaches: pessimistic + reliable system, or causal, (or optimistic)

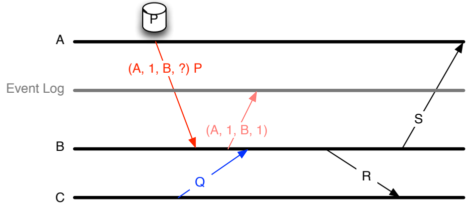
Optimistic Message Logging



Where to save the Events?

- On a reliable media, asynchronously
- “Hope that the event will have time to be logged” (before its loss is damageable)

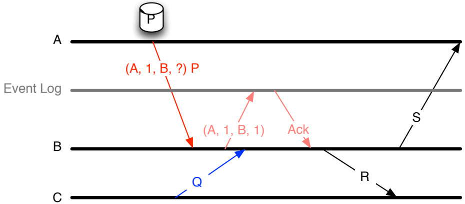
Optimistic Message Logging



Where to save the Events?

- On a reliable media, asynchronously
- "Hope that the event will have time to be logged" (before its loss is damageable)

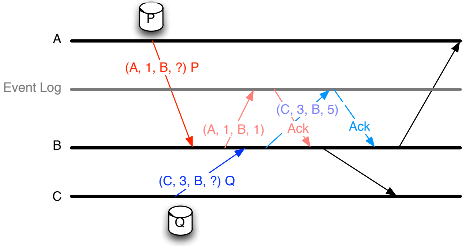
Optimistic Message Logging



Where to save the Events?

- On a reliable media, asynchronously
- “Hope that the event will have time to be logged” (before its loss is damageable)

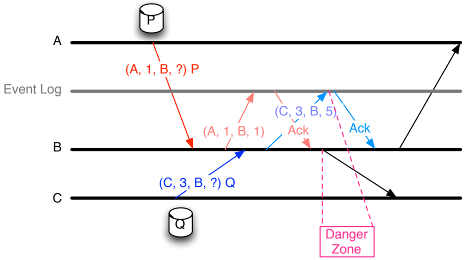
Optimistic Message Logging



Where to save the Events?

- On a reliable media, asynchronously
- "Hope that the event will have time to be logged" (before its loss is damageable)

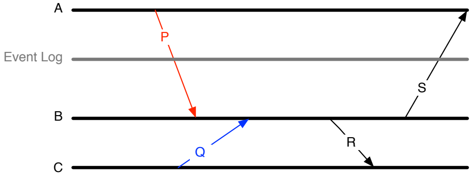
Optimistic Message Logging



Where to save the Events?

- On a reliable media, asynchronously
- "Hope that the event will have time to be logged" (before its loss is damageable)

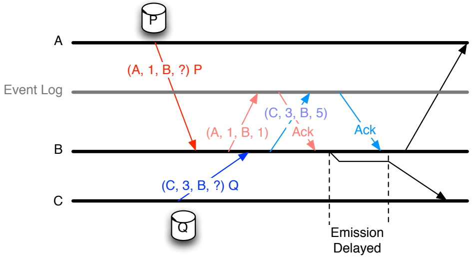
Pessimistic Message Logging



Where to save the Events?

- On a reliable media, synchronously
- Delay of emissions that depend on non-deterministic choices until the corresponding choice is acknowledged
- Recovery: connect to the storage system to get the history

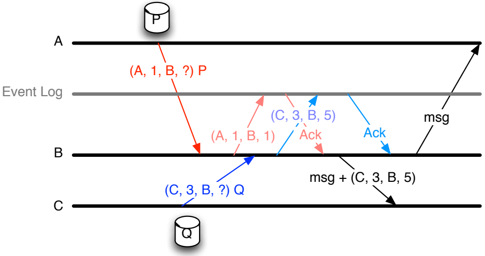
Pessimistic Message Logging



Where to save the Events?

- On a reliable media, synchronously
- Delay of emissions that depend on non-deterministic choices until the corresponding choice is acknowledged
- Recovery: connect to the storage system to get the history

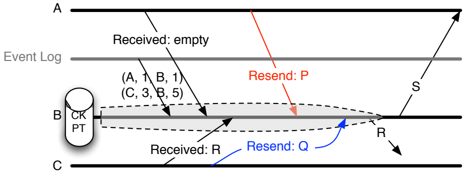
Causal Message Logging



Where to save the Events?

- Any message carries with it (piggybacked) the whole history of non-deterministic events that precede
- Garbage collection using checkpointing, detection of cycles
- Can be coupled with asynchronous storage on reliable media to help garbage collection
- Recovery: global communication + potential storage system

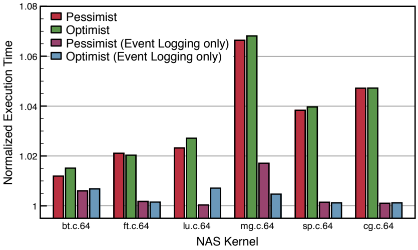
Recover in Message Logging

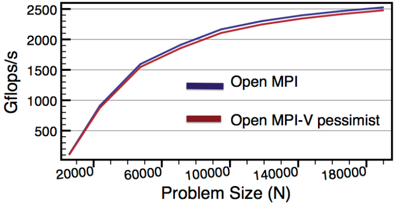


Recovery

- Collect the history (from event log / event log + peers for Causal)
- Collect Id of last message sent
- Emitters resend, deliver in history order
- Fake emission of sent messages

Uncoordinated Protocol Performance





Weak scalability of HPL (90 procs, 360 cores).

Uncoordinated Protocol Performance

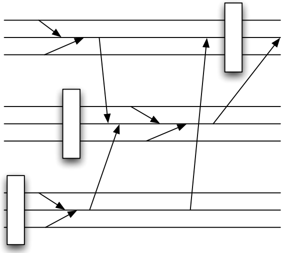
- NAS Parallel Benchmarks – 64 nodes
- High Performance Linpack
- Figures courtesy of A. Bouteiller, G. Bosilca

Hierarchical Protocols

Many Core Systems

- All interactions between threads considered as a message
- Explosion of number of events
- Cost of message payload logging $\approx$ cost of communicating $\rightarrow$ sender-based logging expensive
- Correlation of failures on the node

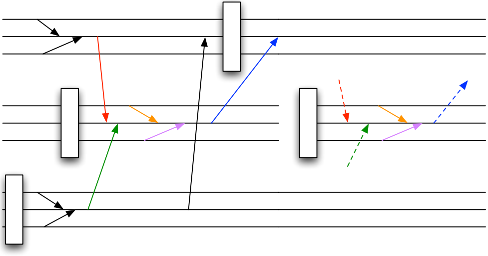
Hierarchical Protocols



Hierarchical Protocol

- Processes are separated in groups
- A group co-ordinates its checkpoint
- Between groups, use message logging

Hierarchical Protocols



Hierarchical Protocol

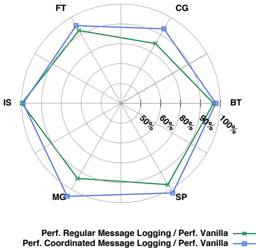
- Coordinated Checkpointing: the processes can behave as a non-deterministic entity (interactions between processes)
- Need to log the non-deterministic events: Hierarchical Protocols *are* uncoordinated protocols + event logging
- No need to log the payload

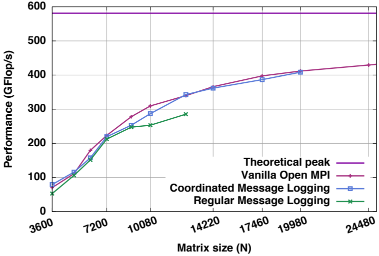
Event Log Reduction

Strategies to reduce the amount of event log

- Few HPC applications use message ordering / timing information to take decisions
- Many receptions (in MPI) are in fact deterministic: do not need to be logged
- For others, although the reception is non-deterministic, the order does not influence the interactions of the process with the rest (send-determinism). No need to log either
- Reduction of the amount of log to a few applications, for a few messages: event logging can be overlapped

Hierarchical Protocol Performance





Hierarchical Protocol Performance

- NAS Parallel Benchmarks – shared memory system, 32 cores
- HPL distributed system, 64 cores, 8 groups