Fault tolerance techniques for high-performance computing Part 1

Anne Benoit

ENS Lyon

Anne.Benoit@ens-lyon.fr

http://graal.ens-lyon.fr/~abenoit

CR02 - 2016/2017

Outline

1. Faults and failures

2. Checkpoint and rollback recovery
   - Process checkpointing
   - Coordinated checkpointing
   - Hierarchical checkpointing

3. Probabilistic models
   - Young/Daly's approximation

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[illegible]
  - 10^5 or 10^6 node[illegible]
  - Each node equipped wi[illegible] 10^4 [illegible] 10^3 cores

- Failure-prone

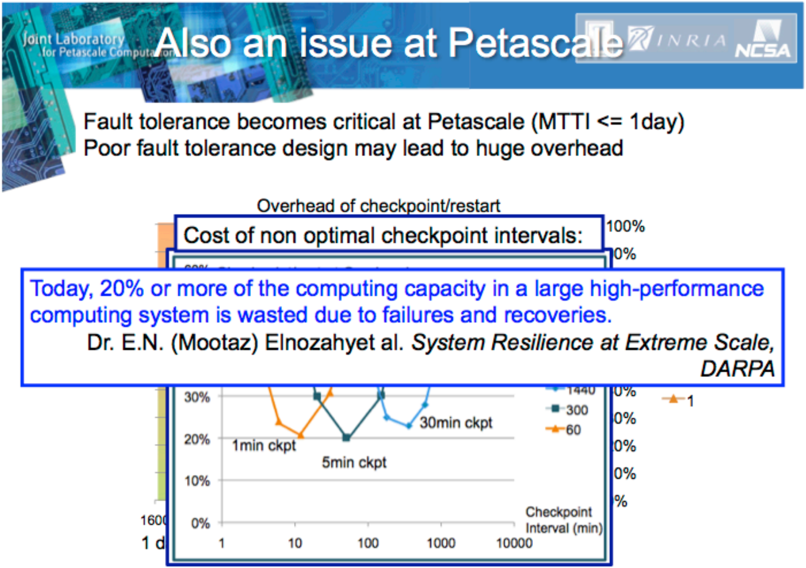
MTBF – o[illegible]node	1 year	10 [illegible]ars	120 years
MTBF – [illegible]latform [illegible]of 10^6 nodes	30sec	5m[illegible]	1h

Mor[illegible]odes = [illegible]een [illegible]lures)

Exascale

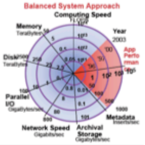
$\neq$ Petascale $\times 1000$

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

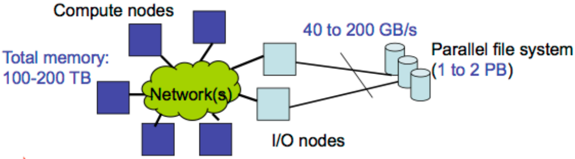


Even for today's platforms (courtesy F. Cappello)

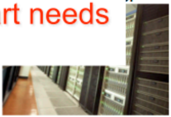
Classic approach for FT: Checkpoint-Restart



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

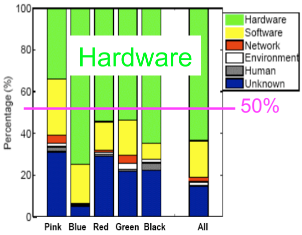
Outline

1. Faults and failures
2. Checkpoint and rollback recovery
3. Probabilistic models

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
- Silent errors (SDC) will be addressed later in the course

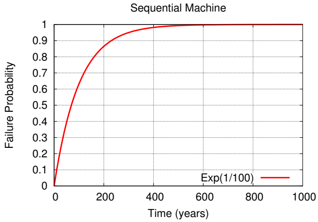
- First question: quantify the rate or frequency at which these faults strike!

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
- Silent errors (SDC) will be addressed later in the course

- First question: quantify the rate or frequency at which these faults strike!

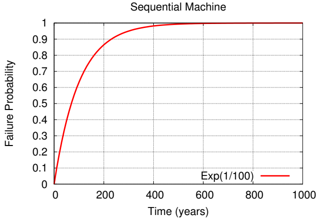
Failure distributions: (1) Exponential



Exp(λ): Exponential distribution law of parameter λ:

- Probability density function (pdf): $f(t) = \lambda e^{-\lambda t} dt$ for $t \geq 0$
- Cumulative distribution function (cdf): $F(t) = 1 - e^{-\lambda t}$
- Mean: $\mu = \frac{1}{\lambda}$

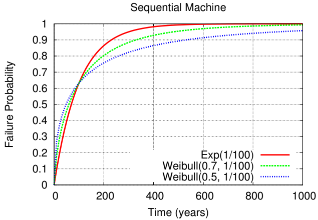
Failure distributions: (1) Exponential



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

- $\mathbb{P}(X \leq t) = 1 - e^{-\lambda t}dt$ (by definition)
- Memoryless property: $\mathbb{P}(X \geq t + s \,|\, X \geq s) = \mathbb{P}(X \geq t)$ (for all $t, s \geq 0$): at any instant, time to next failure does not depend upon time elapsed since last failure
- Mean Time Between Failures (MTBF) $\mu = \mathbb{E}(X) = \frac{1}{\lambda}$

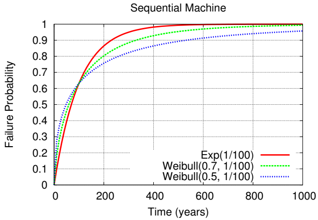
Failure distributions: (2) Weibull



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

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

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: (3) 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 ☹

Failure distributions: (3) 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 ☹

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)

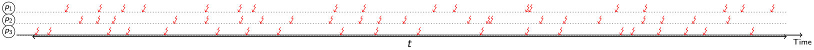
- Rebooting only faulty processor
- Platform failure distribution
  $\Rightarrow$ superposition of p IID processor distributions
  $\Rightarrow$ IID only for Exponential
- Define μ_p by

$$\lim_{F \to +\infty} \frac{n(F)}{F} = \frac{1}{\mu_p}$$

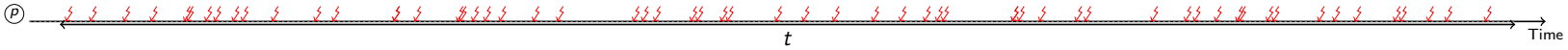
$n(F)$ = number of platform failures until time F is exceeded

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

Intuition



If three processors have around 20 faults during a time t ($\mu = \frac{t}{20}$)...



...during the same time, the platform has around 60 faults ($\mu_p = \frac{t}{60}$)

MTBF with p processors (1/2)

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

With one processor:

- $n(F)$ = number of failures until time F is exceeded
- X_i iid random variables for inter-arrival times, with $\mathbb{E}(X_i) = \mu$
- $\sum_{i=1}^{n(F)-1} X_i \leq F \leq \sum_{i=1}^{n(F)} X_i$
- Wald's equation: $(\mathbb{E}(n(F)) - 1)\mu \leq F \leq \mathbb{E}(n(F))\,\mu$
- $\lim_{F \to +\infty} \frac{\mathbb{E}(n(F))}{F} = \frac{1}{\mu}$

MTBF with p processors (2/2)

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

With p processors:

- $n(F)$ = number of platform failures until time F is exceeded
- $n_q(F)$ = number of those failures that strike processor q
- $n_q(F) + 1$ = number of failures on processor q until time F is exceeded (except for processor with last-failure)
- $\lim_{F \to +\infty} \frac{n_q(F)}{F} = \frac{1}{\mu}$ as above
- $\lim_{F \to +\infty} \frac{n(F)}{F} = \frac{1}{\mu_p}$ by definition
- Hence $\mu_p = \frac{\mu}{p}$ because $n(F) = \sum_{q=1}^{p} n_q(F)$

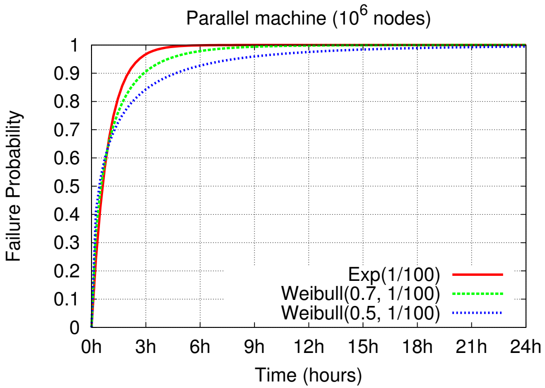
A little digression for afficionados

- X_i IID random variables for processor inter-arrival times
- Assume X_i continuous, with $\mathbb{E}(X_i) = \mu$
- Y_i random variables for platform inter-arrival times
- **Definition:** $\mu_p \stackrel{def}{=} \lim_{n\to+\infty} \frac{\sum_i^n \mathbb{E}(Y_i)}{n}$
- Limits always exists (superposition of renewal processes)
- **Theorem:** $\mu_p = \frac{\mu}{p}$

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?



After infant mortality and before aging,
instantaneous failure rate of computer platforms is almost constant

Summary for the road

- MTBF key parameter and $\mu_p = \frac{\mu}{p}$ ☺
- Exponential distribution OK for most purposes ☺
- Assume failure independence while not (completely) true ☹

Outline

1 Faults and failures

2 Checkpoint and rollback recovery
- Process checkpointing
- Coordinated checkpointing
- Hierarchical checkpointing

3 Probabilistic models

Outline

1 Faults and failures

2 Checkpoint and rollback recovery
- Process checkpointing
- Coordinated checkpointing
- Hierarchical checkpointing

3 Probabilistic models

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

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

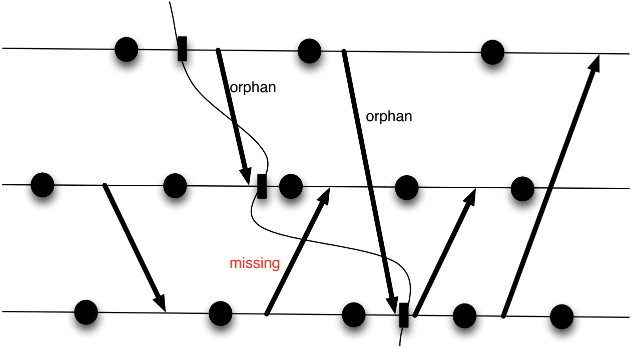
Outline

1 Faults and failures

2 Checkpoint and rollback recovery
- Process checkpointing
- Coordinated checkpointing
- Hierarchical checkpointing

3 Probabilistic models

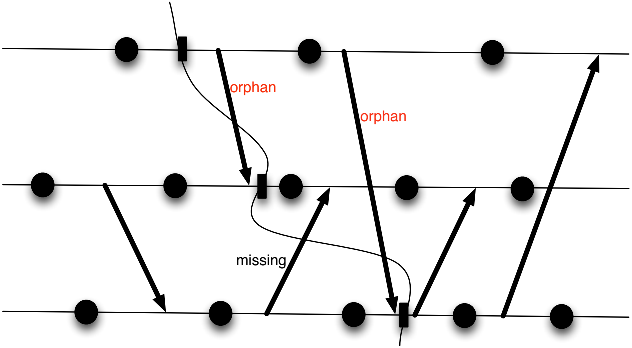
Coordinated checkpointing



Definition (Missing Message)

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

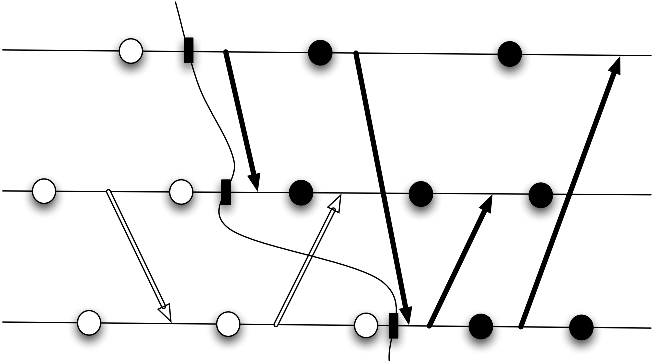
Coordinated checkpointing



Definition (Orphan Message)

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

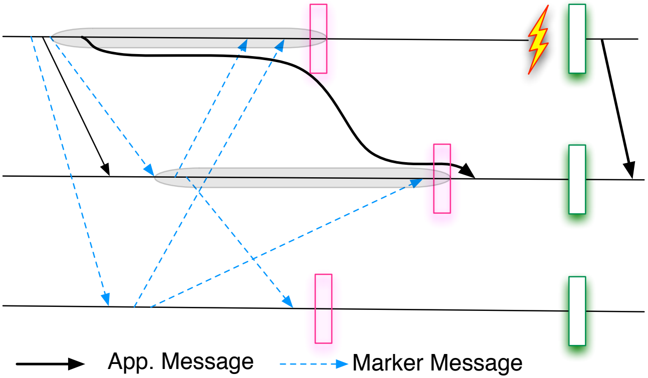
Coordinated checkpointing



Create a consistent view of the application (no orphan messages)

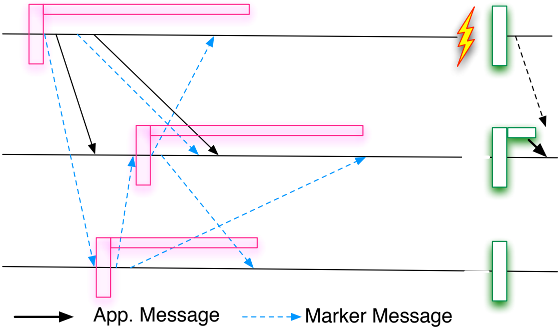
- Every message belongs to a single checkpoint wave
- All communication channels must be flushed (all2all)

Blocking coordinated checkpointing



- Silences the network during 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

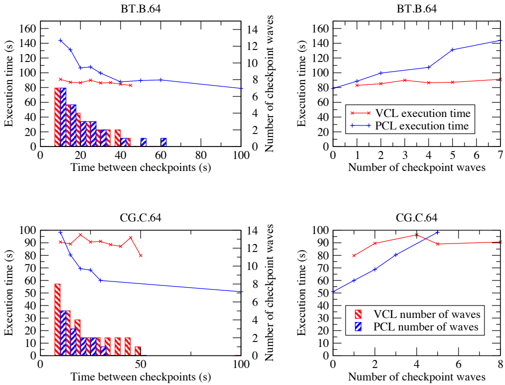
Communication Library

- 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 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

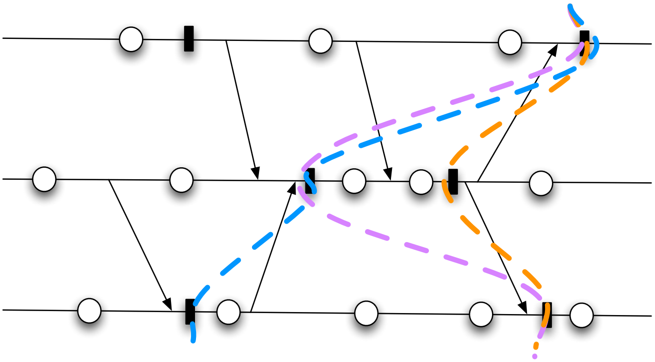
Outline

1 Faults and failures

2 Checkpoint and rollback recovery
- Process checkpointing
- Coordinated checkpointing
- Hierarchical checkpointing

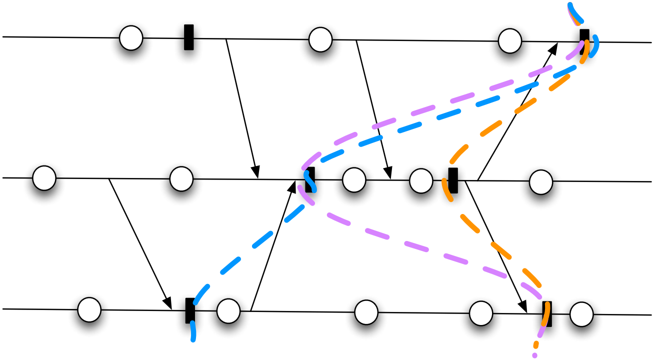
3 Probabilistic models

Uncoordinated Checkpointing Idea



Processes checkpoint independently

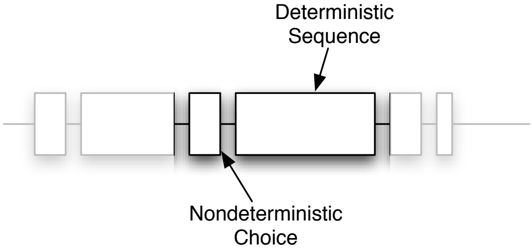
Uncoordinated Checkpointing Idea



Optimistic Protocol

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

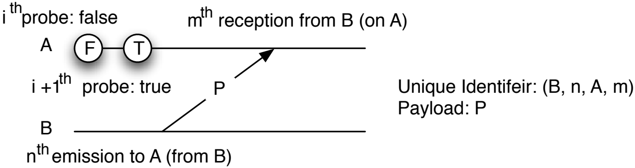
Piece-wise Deterministic Assumption



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

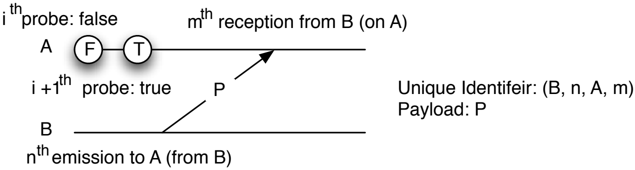
Message Logging



Message Logging

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

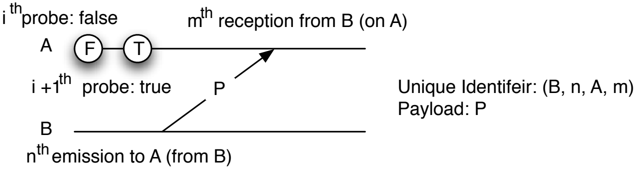
Message Logging



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

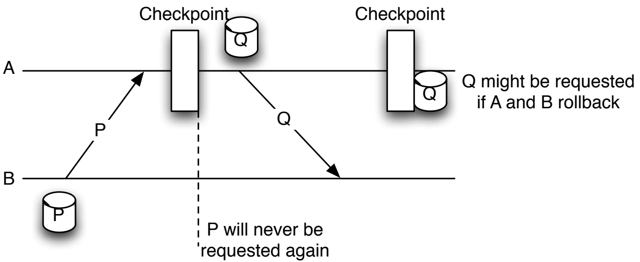
Message Logging



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

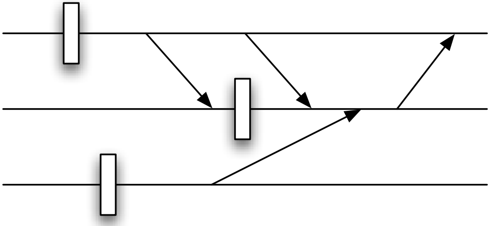
Message Logging



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

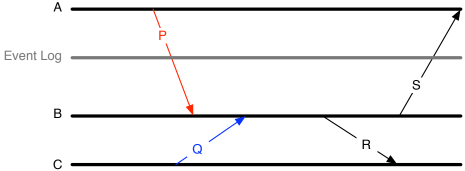
Message Logging



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)

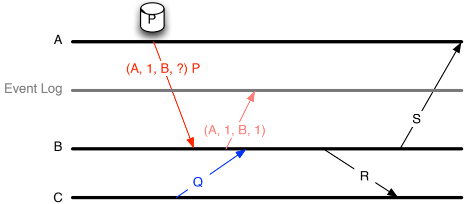
Optimistic Message Logging



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)

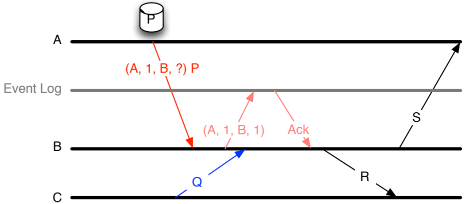
Optimistic Message Logging



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)

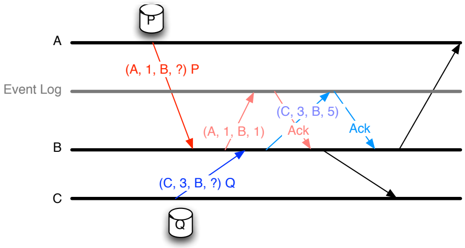
Optimistic Message Logging



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)

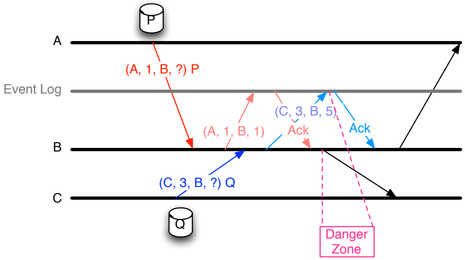
Optimistic Message Logging



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)

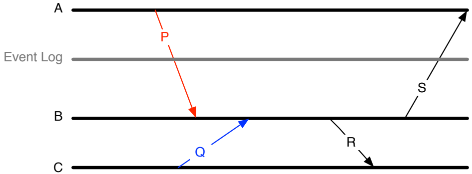
Optimistic Message Logging



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)

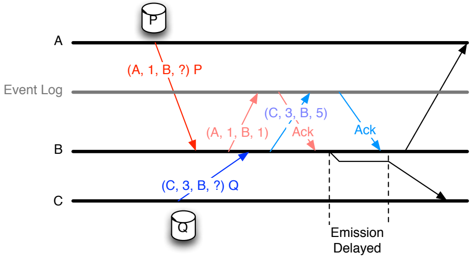
Pessimistic Message Logging



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

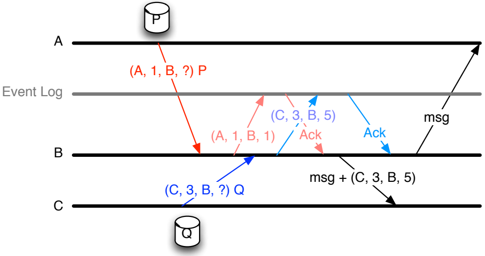
Pessimistic Message Logging



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

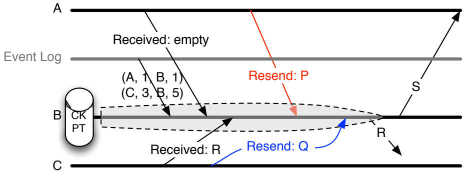
Causal Message Logging



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

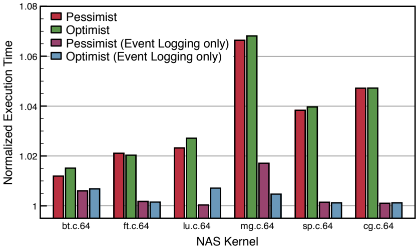
Recover in Message Logging

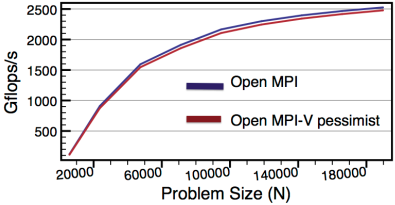


Recovery

- 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

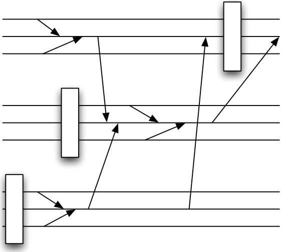
- NAS Parallel Benchmarks – 64 nodes
- High Performance Linpack
- Figures courtesy of A. Bouteiller, G. Bosilca

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

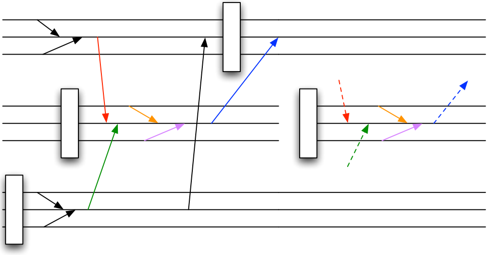
Hierarchical Protocols



Hierarchical Protocol

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

Hierarchical Protocols



Hierarchical Protocol

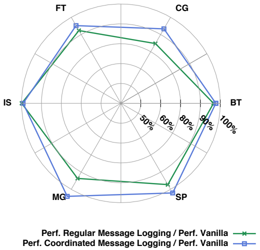
- Coordinated Checkpointing: the processes can behave as a non-deterministic entity (interactions between processes)
- Need to log the non-deterministic events: Hierarchical Protocols *are* uncoordinated protocols + event logging
- No need to log the payload

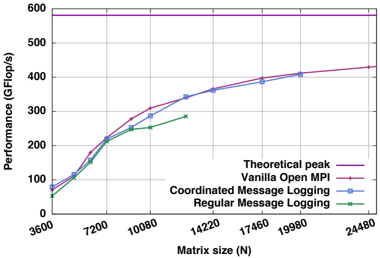
Event Log Reduction

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

Outline

1. Faults and failures
2. Checkpoint and rollback recovery
3. Probabilistic models
   - Young/Daly's approximation

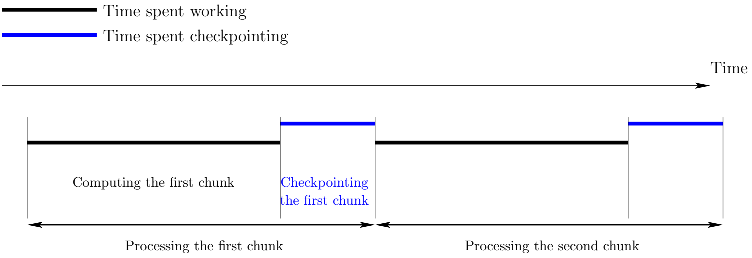
Outline

1 Faults and failures

2 Checkpoint and rollback recovery

3 Probabilistic models
- Young/Daly's approximation

Checkpointing cost



Blocking model: while a checkpoint is taken, no computation can be performed

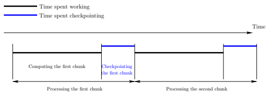
Framework

- Periodic checkpointing policy of period T
- Independent and identically distributed (IID) failures
- Applies to a single processor with MTBF $\mu = \mu_{ind}$
- Applies to a platform with p processors with MTBF $\mu = \frac{\mu_{ind}}{p}$
  - coordinated checkpointing
  - tightly-coupled application
  - progress $\Leftrightarrow$ all processors available

$\Rightarrow$ platform = single (powerful, unreliable) processor ☺

Waste: fraction of time not spent for useful computations

Waste in fault-free execution



- $\textsc{Time}_{\text{base}}$: application base time
- $\textsc{Time}_{\text{FF}}$: with periodic checkpoints but failure-free

$$\textsc{Time}_{\text{FF}} = \textsc{Time}_{\text{base}} + \#checkpoints \times C$$

$$\#checkpoints = \left\lceil \frac{\textsc{Time}_{\text{base}}}{T - C} \right\rceil \approx \frac{\textsc{Time}_{\text{base}}}{T - C} \text{ (valid for large jobs)}$$

$$\textsc{Waste}[FF] = \frac{\textsc{Time}_{\text{FF}} - \textsc{Time}_{\text{base}}}{\textsc{Time}_{\text{FF}}} = \frac{C}{T}$$

Waste due to failures

- $\textsc{Time}_{\text{base}}$: application base time
- $\textsc{Time}_{\text{FF}}$: with periodic checkpoints but failure-free
- $\textsc{Time}_{\text{final}}$: expectation of time with failures

$$\textsc{Time}_{\text{final}} = \textsc{Time}_{\text{FF}} + N_{faults} \times T_{\text{lost}}$$

N_{faults} number of failures during execution
T_{lost}: average time lost per failure

$$N_{faults} = \frac{\textsc{Time}_{\text{final}}}{\mu}$$

T_{lost}?

Waste due to failures

- $\text{TIME}_{\text{base}}$: application base time
- TIME_{FF}: with periodic checkpoints but failure-free
- $\text{TIME}_{\text{final}}$: expectation of time with failures

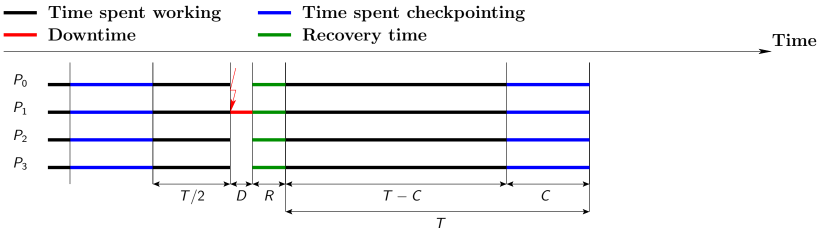
$$\text{TIME}_{\text{final}} = \text{TIME}_{\text{FF}} + N_{faults} \times T_{\text{lost}}$$

N_{faults} number of failures during execution
T_{lost}: average time lost per failure

$$N_{faults} = \frac{\text{TIME}_{\text{final}}}{\mu}$$

T_{lost}?

Computing T_{lost}



$$T_{\text{lost}} = D + R + \frac{T}{2}$$

Rationale

⇒ Instants when periods begin and failures strike are independent

⇒ Approximation used for all distribution laws

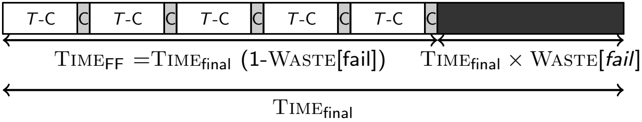
⇒ Exact for Exponential and uniform distributions

Waste due to failures

$$\textsc{Time}_{\mathsf{final}} = \textsc{Time}_{\mathsf{FF}} + N_{faults} \times T_{\mathsf{lost}}$$

$$\textsc{Waste}[fail] = \frac{\textsc{Time}_{\mathsf{final}} - \textsc{Time}_{\mathsf{FF}}}{\textsc{Time}_{\mathsf{final}}} = \frac{1}{\mu}\left(D + R + \frac{T}{2}\right)$$

Total waste



$$\text{WASTE} = \frac{\text{TIME}_{\text{final}} - \text{TIME}_{\text{base}}}{\text{TIME}_{\text{final}}}$$

$$1 - \text{WASTE} = (1 - \text{WASTE}[FF])(1 - \text{WASTE}[fail])$$

$$\text{WASTE} = \frac{C}{T} + \left(1 - \frac{C}{T}\right)\frac{1}{\mu}\left(D + R + \frac{T}{2}\right)$$

How do we minimize the waste? (use the goat's lemma!)