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

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

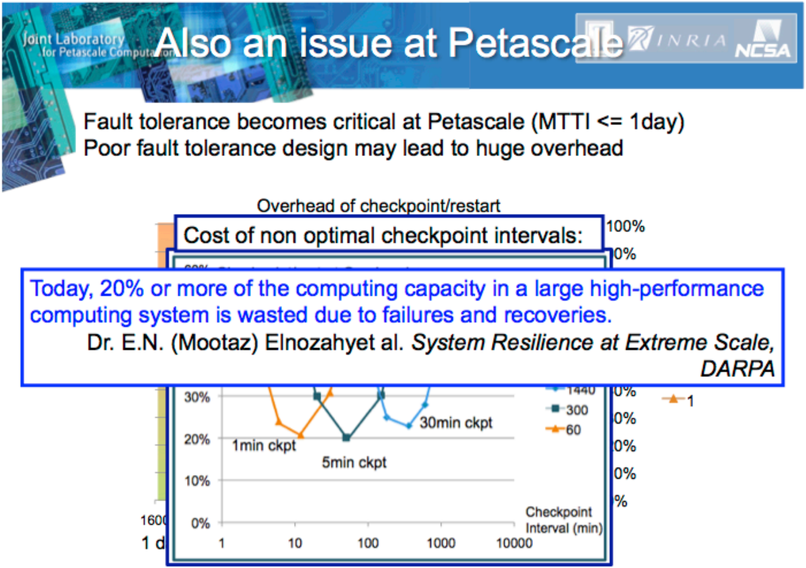
MTBF – o node	1 year	10 ars	120 years
MTBF atform of 10^6 nodes	30sec	5m	1h

Mor odes = een ilures)

Exascale

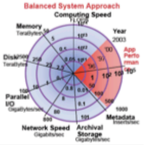
$\neq$ Petascale $\times 1000$

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

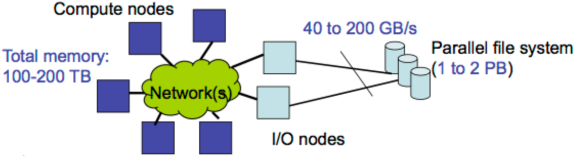


Even for today's platforms (courtesy F. Cappello)

Classic approach for FT: Checkpoint-Restart

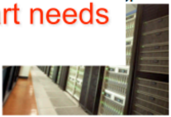


Typical "Balanced Architecture" for PetaScale Computers



RoadRunner

TACC Ranger



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
$\mu = 1$ week	2^8	91.56%
	2^{11}	73.75%
	2^{14}	20.07%
	2^{17}	2.51%
	2^{20}	0.31%

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

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

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

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

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

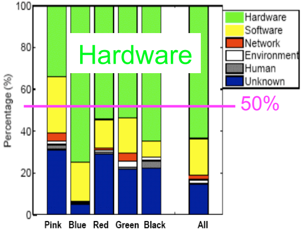
Outline

1. Introduction (15mn)
   - Large-scale computing platforms
   - Faults and failures
2. General-purpose fault-tolerance techniques (30mn)
   - Replication
   - Process Checkpointing
   - Coordinated Checkpointing
   - Uncoordinated checkpointing
3. Probabilistic models for checkpointing (45mn)
   - Young/Daly's approximation
   - Coordinated checkpointing
   - Hierarchical checkpointing
4. Application-specific fault-tolerance techniques (45mn)
   - Fault-Tolerant Middleware
   - Bags of tasks
   - Iterative algorithms and fixed-point convergence
   - ABFT for Linear Algebra applications
   - Composite approach: ABFT & Checkpointing
5. Other techniques (35mn)
   - Replication
   - Failure Prediction
   - In-memory checkpointing
   - Silent errors
6. Conclusion (10mn)

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.

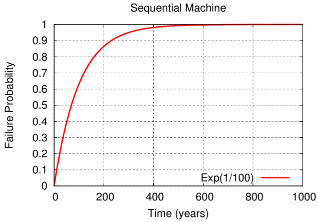
Hardware errors, Disks, processors, memory, network

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

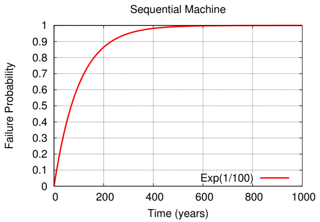
Failure distributions: (1) Exponential



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

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

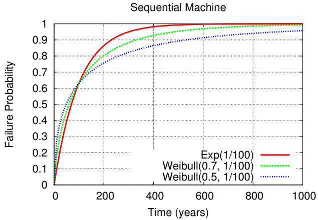
Failure distributions: (1) Exponential



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

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

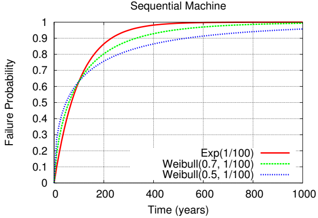
Failure distributions: (2) Weibull



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

- Pdf: $f(t) = k\lambda(t\lambda)^{k-1}e^{-(\lambda t)^k}dt$ 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 $Weibull(k, \lambda)$ failure inter-arrival times:

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

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

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

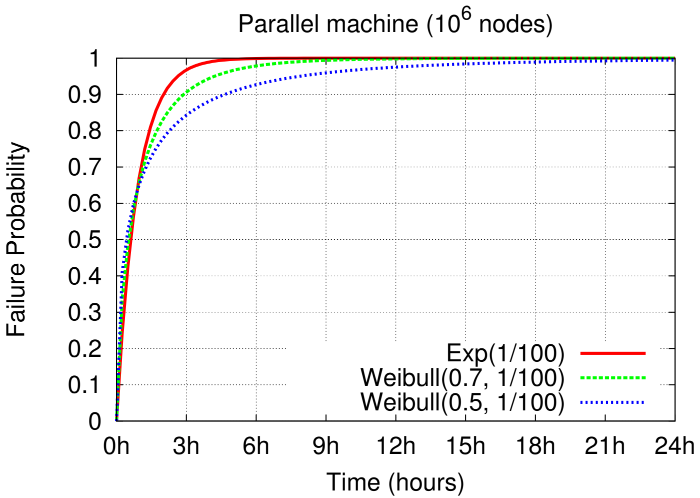
With p processors:

- $n(F)$ = 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)
- Y_i iid random variables for platform inter-arrival times, with $\mathbb{E}(Y_i) = \mu_p$
- $\lim_{F \to +\infty} \frac{n(F)}{F} = \frac{1}{\mu_p}$ as above
- $\lim_{F \to +\infty} \frac{n(F)}{F} = \frac{p}{\mu}$ because $n(F) = \sum_{q=1}^{p} n_q(F)$
- Hence $\mu_p = \frac{\mu}{p}$

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?



Outline

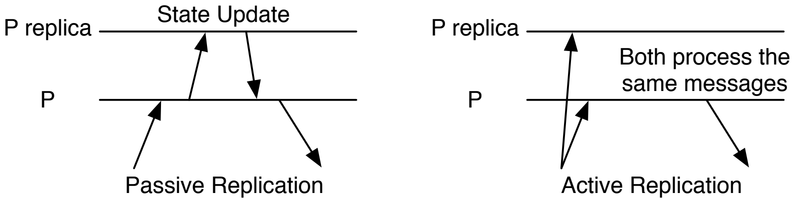
Maintaining Redundant Information

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

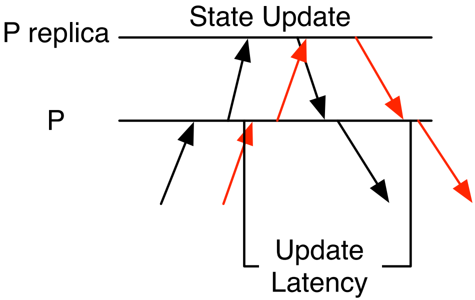
Replication



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

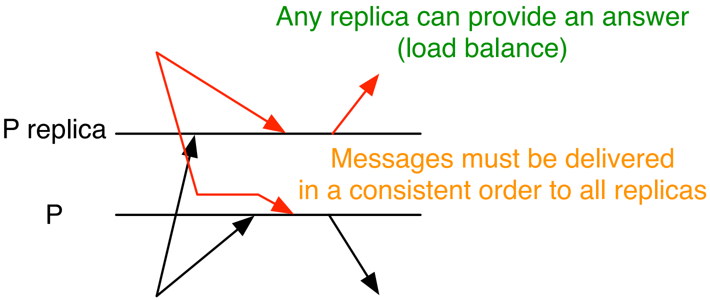
Replication



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`

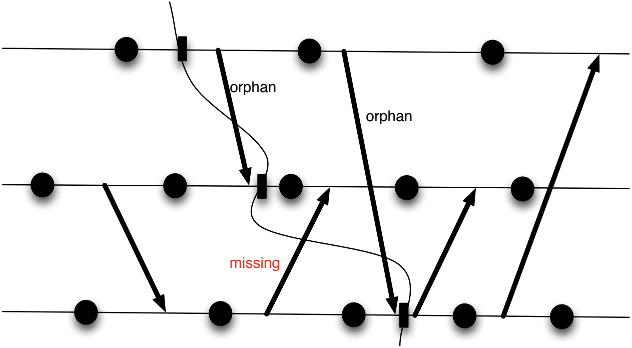
Checkpoint is valid when finished on reliable storage

Distributed Memory Storage

- In-memory checkpointing
- Disk-less checkpointing

Outline

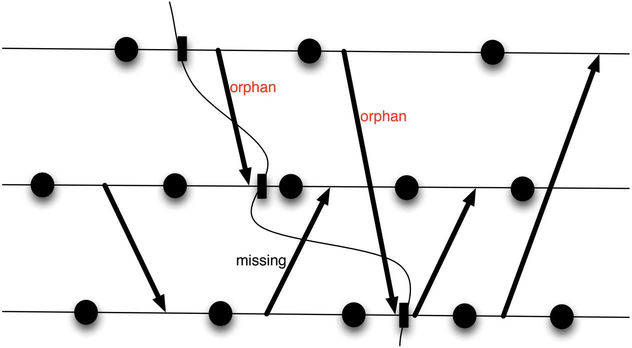
Coordinated checkpointing



Definition (Missing Message)

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

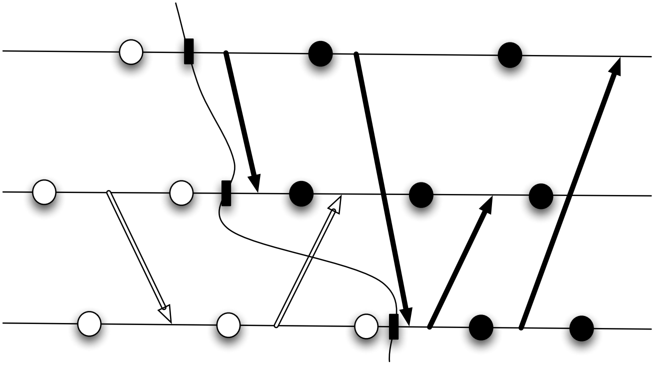
Coordinated checkpointing



Definition (Orphan Message)

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

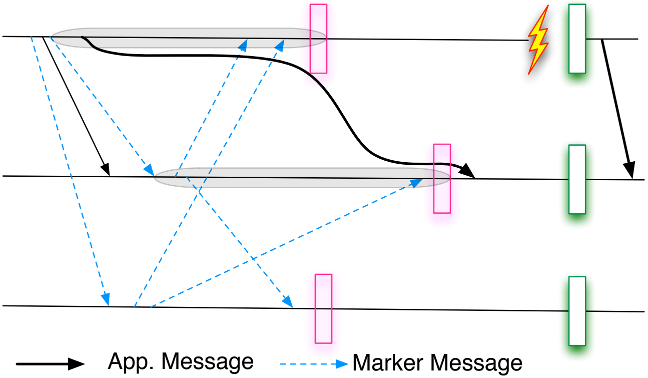
Coordinated Checkpointing Idea



Create a consistent view of the application

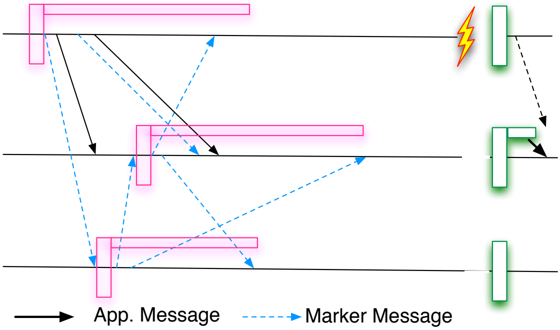
- Messages belong to a checkpoint wave or another
- All communication channels must be flushed (all2all)

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

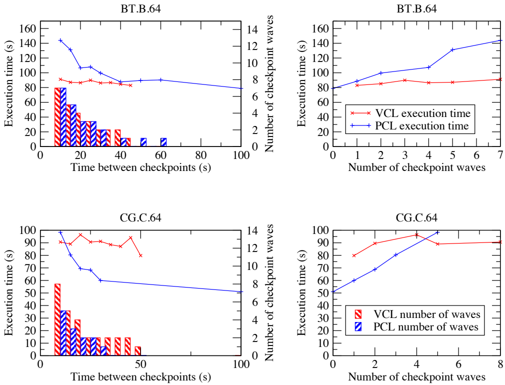
Communication Library

- 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 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

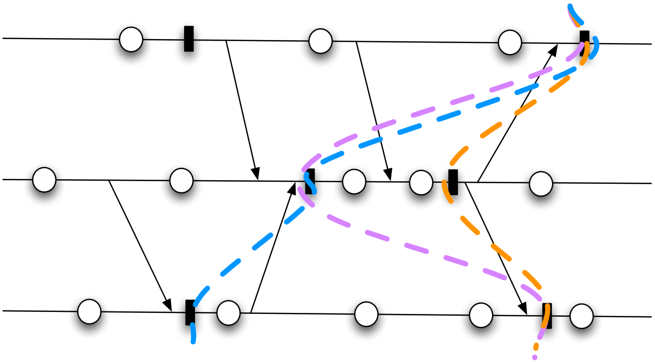


Coordinated Protocol Performance

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

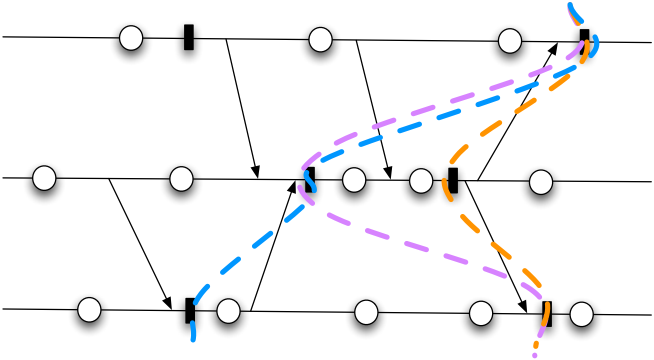
Outline

Uncoordinated Checkpointing Idea



Processes checkpoint independently

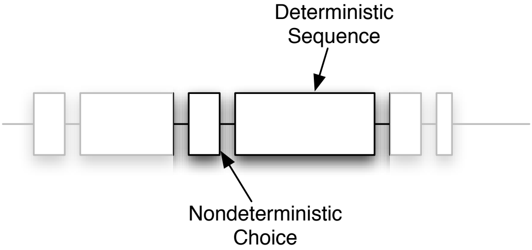
Uncoordinated Checkpointing Idea



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

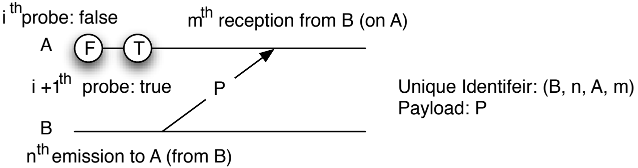
Piece-wise Deterministic Assumption



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

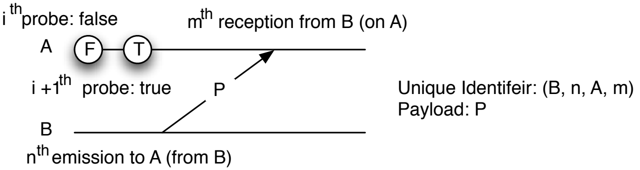
Message Logging



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

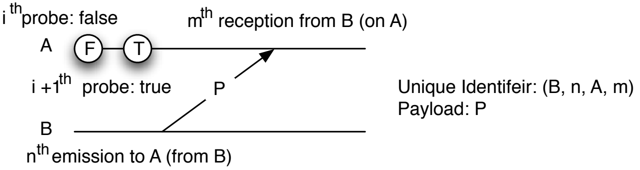
Message Logging



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

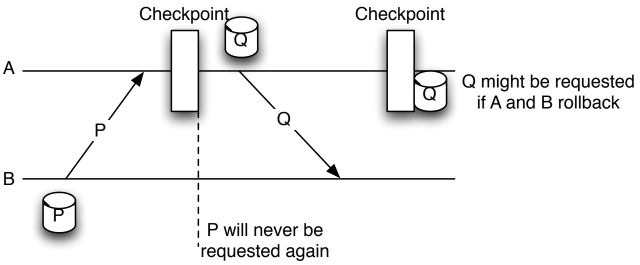
Message Logging



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

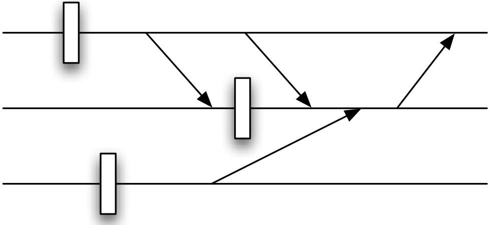
Message Logging



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

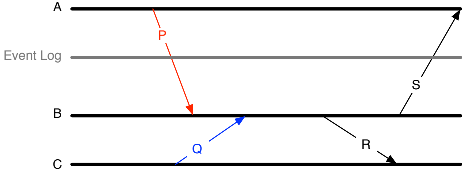
Message Logging



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)

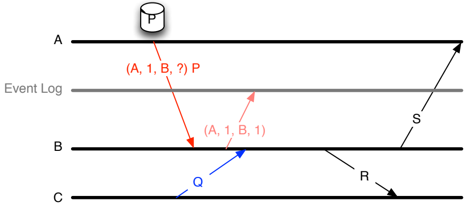
Optimistic Message Logging



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)

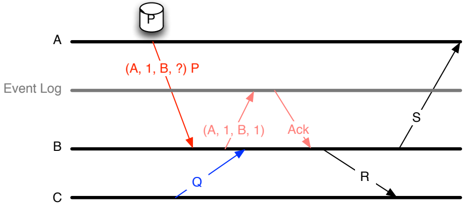
Optimistic Message Logging



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)

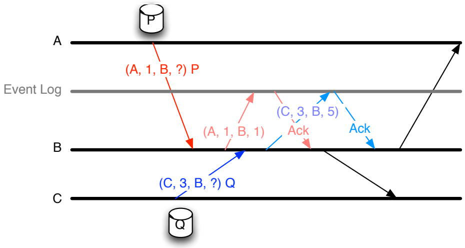
Optimistic Message Logging



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)

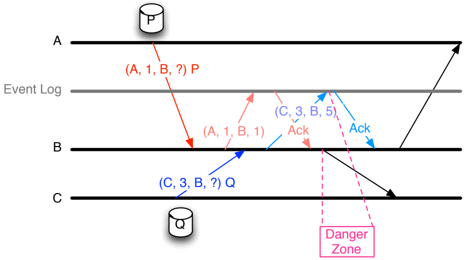
Optimistic Message Logging



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)

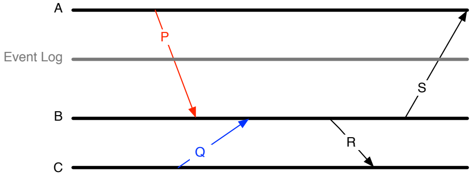
Optimistic Message Logging



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)

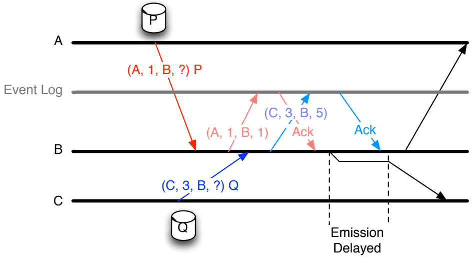
Pessimistic Message Logging



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

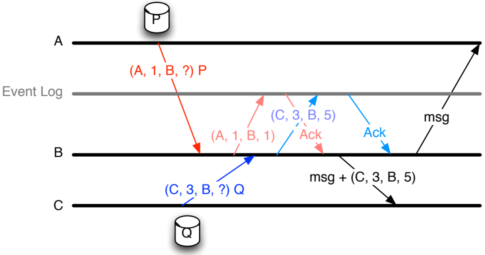
Pessimistic Message Logging



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

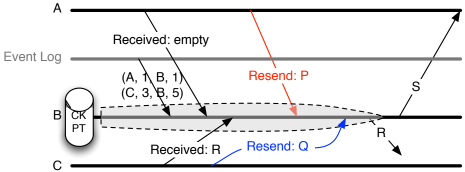
Causal Message Logging



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

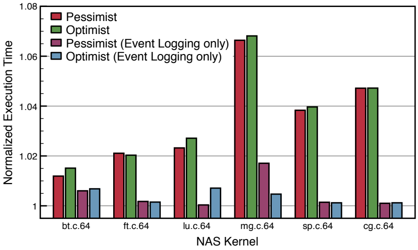
Recover in Message Logging

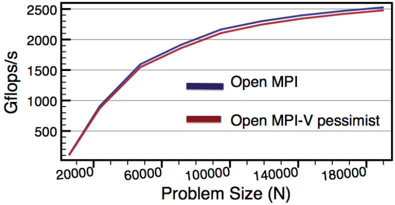


Recovery

- 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

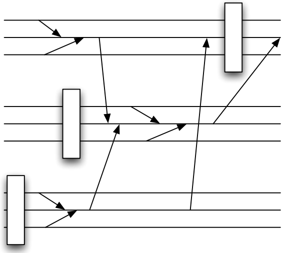
- NAS Parallel Benchmarks – 64 nodes
- High Performance Linpack
- Figures courtesy of A. Bouteiller, G. Bosilca

Hierarchical Protocols

Many Core Systems

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

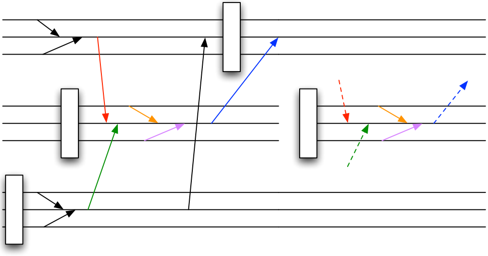
Hierarchical Protocols



Hierarchical Protocol

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

Hierarchical Protocols



Hierarchical Protocol

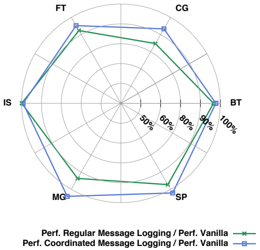
- Coordinated Checkpointing: the processes can behave as a non-deterministic entity (interactions between processes)
- Need to log the non-deterministic events: Hierarchical Protocols *are* uncoordinated protocols + event logging
- No need to log the payload

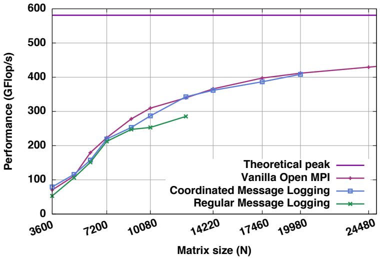
Event Log Reduction

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





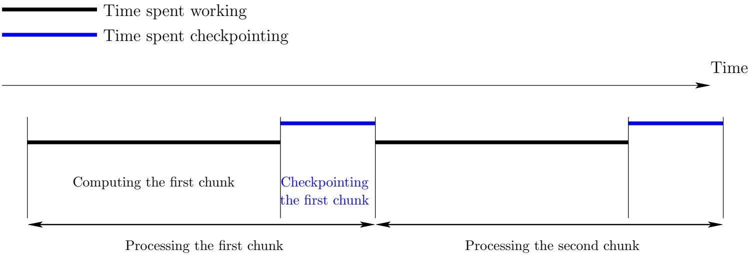
Hierarchical Protocol Performance

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

Outline

Outline

Checkpointing cost



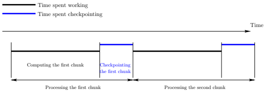
Blocking model: while a checkpoint is taken, no computation can be performed

Framework

- Periodic checkpointing policy of period T
- Independent and identically distributed 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

Waste: fraction of time not spent for useful computations

Waste in fault-free execution



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

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

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

$$\text{WASTE}[FF] = \frac{\text{TIME}_{\text{FF}} - \text{TIME}_{\text{base}}}{\text{TIME}_{\text{FF}}} = \frac{C}{T}$$

Waste due to failures

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

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

N_{faults} number of failures during execution
$T_{\textsf{lost}}$: average time lost par failures

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

$T_{\textsf{lost}}$?

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

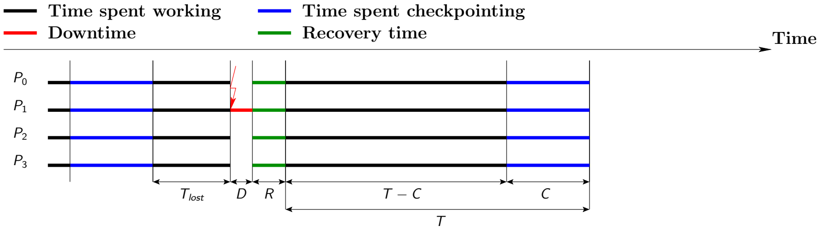
$$\textsc{Time}_{\text{final}} = \textsc{Time}_{\text{FF}} + N_{faults} \times T_{\text{lost}}$$

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

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

T_{lost}?

Computing T_{lost}



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

⇒ Instants when periods begin and failures strike are independent

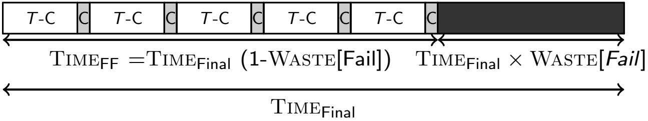
⇒ Valid for all distribution laws, regardless of their particular shape

Waste due to failures

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

$$\text{WASTE}[fail] = \frac{\text{TIME}_{\text{final}} - \text{TIME}_{\text{FF}}}{\text{TIME}_{\text{final}}} = \frac{1}{\mu}\left(D + R + \frac{T}{2}\right)$$

Total waste



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

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

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

Waste minimization

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

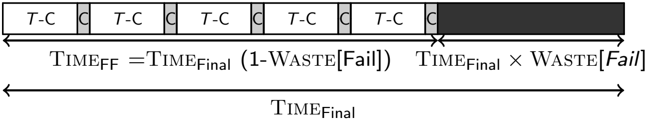
$$\textsc{Waste} = \frac{u}{T} + v + wT$$

$$u = C\left(1 - \frac{D+R}{\mu}\right) \qquad v = \frac{D + R - C/2}{\mu} \qquad w = \frac{1}{2\mu}$$

$\textsc{Waste}$ minimized for $T = \sqrt{\frac{u}{w}}$

$$T = \sqrt{2(\mu - (D + R))C}$$

Comparison with Young/Daly



$$\left(1 - \textsc{Waste}[\mathit{fail}]\right)\textsc{Time}_{\text{final}} = \textsc{Time}_{\text{FF}}$$
$$\Rightarrow T = \sqrt{2(\mu - (D+R))C}$$

Daly: $\textsc{Time}_{\text{final}} = \left(1 + \textsc{Waste}[\mathit{fail}]\right)\textsc{Time}_{\text{FF}}$
$$\Rightarrow T = \sqrt{2(\mu + (D+R))C} + C$$

Young: $\textsc{Time}_{\text{final}} = \left(1 + \textsc{Waste}[\mathit{fail}]\right)\textsc{Time}_{\text{FF}}$ and $D = R = 0$
$$\Rightarrow T = \sqrt{2\mu C} + C$$

Validity of the approach (1/3)

Technicalities

- $\mathbb{E}(N_{faults}) = \frac{\text{TIME}_{\text{final}}}{\mu}$ and $\mathbb{E}(T_{\text{lost}}) = D + R + \frac{T}{2}$
  but expectation of product is not product of expectations
  (not independent RVs here)

- Enforce $C \leq T$ to get $\text{WASTE}[FF] \leq 1$

- Enforce $D + R \leq \mu$ and bound T to get $\text{WASTE}[fail] \leq 1$
  but $\mu = \frac{\mu_{ind}}{p}$ too small for large p, regardless of μ_{ind}

Validity of the approach (2/3)

Several failures within same period?

- WASTE[fail] accurate only when two or more faults do not take place within same period
- Cap period: $T \leq \gamma\mu$, where γ is some tuning parameter
  - Poisson process of parameter $\theta = \frac{T}{\mu}$
  - Probability of having $k \geq 0$ failures : $P(X = k) = \frac{\theta^k}{k!} e^{-\theta}$
  - Probability of having two or more failures: $\pi = P(X \geq 2) = 1 - (P(X = 0) + P(X = 1)) = 1 - (1 + \theta)e^{-\theta}$
  - $\gamma = 0.27 \Rightarrow \pi \leq 0.03$
    $\Rightarrow$ overlapping faults for only 3% of checkpointing segments

Validity of the approach (3/3)

- Enforce $T \leq \gamma\mu$, $C \leq \gamma\mu$, and $D + R \leq \gamma\mu$
- Optimal period $\sqrt{2(\mu - (D + R))C}$ may not belong to admissible interval $[C, \gamma\mu]$
- Waste is then minimized for one of the bounds of this admissible interval (by convexity)

Wrap up

- Capping periods, and enforcing a lower bound on MTBF
  ⇒ mandatory for mathematical rigor ☹

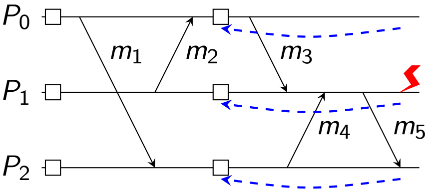
- Not needed for practical purposes ☺
  - actual job execution uses optimal value
  - account for multiple faults by re-executing work until success

- Approach surprisingly robust ☺

Outline

Background: coordinated checkpointing protocols

- Coordinated checkpoints over all processes
- Global restart after a failure



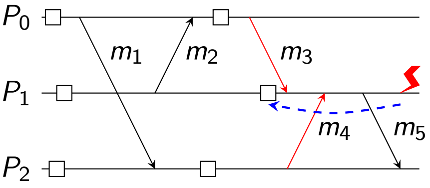
☺ No risk of cascading rollbacks

☺ No need to log messages

☹ All processors need to roll back

Background: message logging protocols

- Message content logging (sender memory)
- Restart of failed process only



☺ No cascading rollbacks

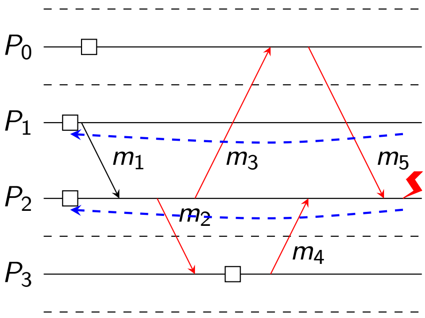
☺ Number of processes to roll back

☹ Memory occupation

☹ Overhead

Background: hierarchical protocols

- Clusters of processes
- Coordinated checkpointing protocol within clusters
- Message logging protocols between clusters
- Only processors from failed group need to roll back



☹ Need to log inter-groups messages
  - Slowdowns failure-free execution
  - Increases checkpoint size/time

☺ Faster re-execution with logged messages

Which checkpointing protocol to use?

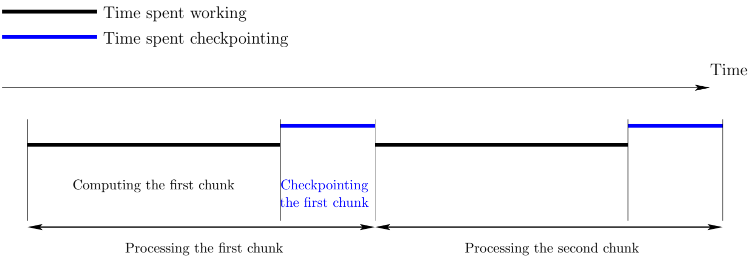
Coordinated checkpointing

- ☺ No risk of cascading rollbacks
- ☺ No need to log messages
- ☹ All processors need to roll back
- ☹ Rumor: May not scale to very large platforms

Hierarchical checkpointing

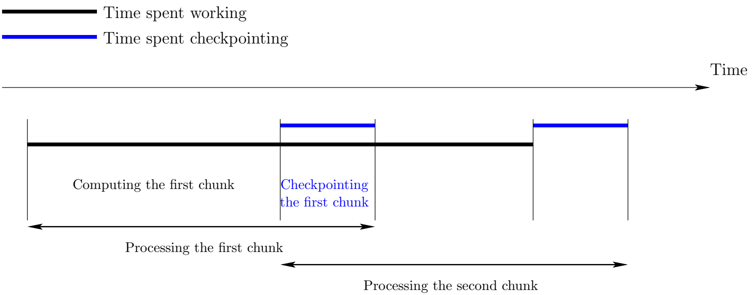
- ☹ Need to log inter-groups messages
  - Slowdowns failure-free execution
  - Increases checkpoint size/time
- ☺ Only processors from failed group need to roll back
- ☺ Faster re-execution with logged messages
- ☺ Rumor: Should scale to very large platforms

Coordinated checkpointing



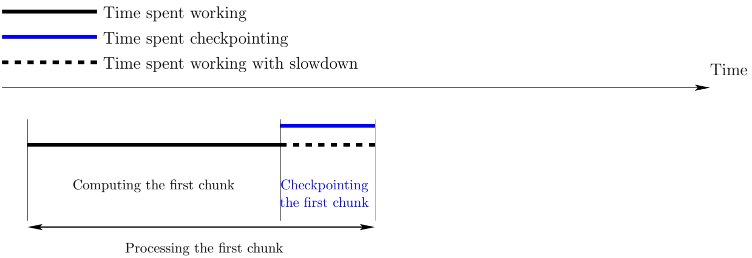
Blocking model: checkpointing blocks all computations

Coordinated checkpointing



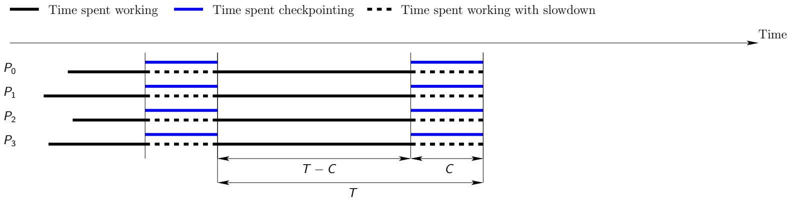
Non-blocking model: checkpointing has no impact on computations (e.g., first copy state to RAM, then copy RAM to disk)

Coordinated checkpointing



General model: checkpointing slows computations down: during a checkpoint of duration C, the same amount of computation is done as during a time αC without checkpointing ($0 \leq \alpha \leq 1$)

Waste in fault-free execution

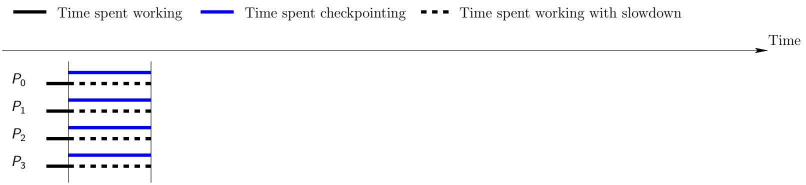


Time elapsed since last checkpoint: T

Amount of computations executed: $\textsc{Work} = (T - C) + \alpha C$

$\textsc{Waste}[FF] = \frac{T - \textsc{Work}}{T}$

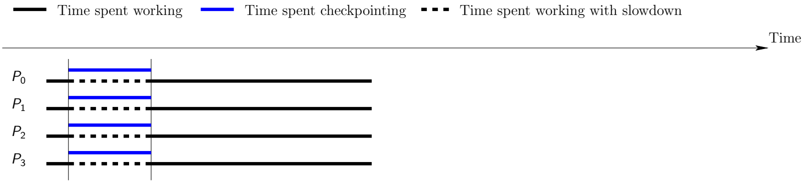
Waste due to failures



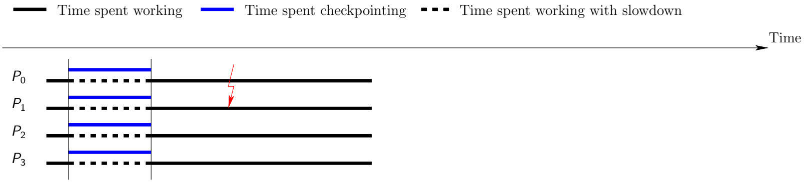
Failure can happen

1. During computation phase
2. During checkpointing phase

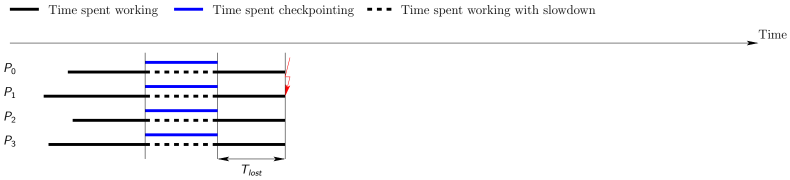
Waste due to failures



Waste due to failures

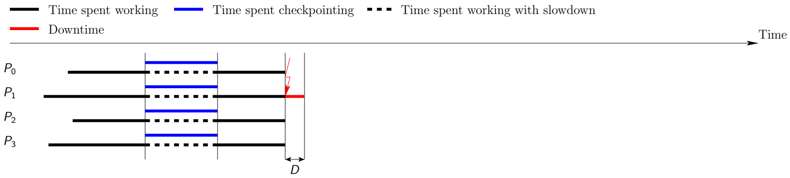


Waste due to failures

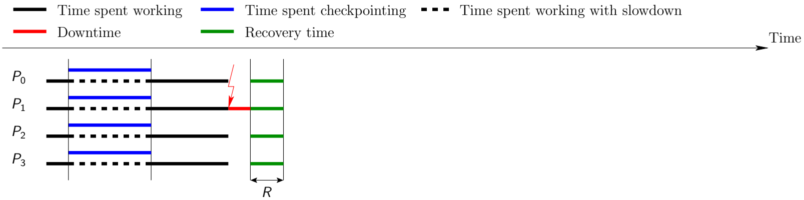


Coordinated checkpointing protocol: when one processor is victim of a failure, all processors lose their work and must roll back to last checkpoint

Waste due to failures in computation phase

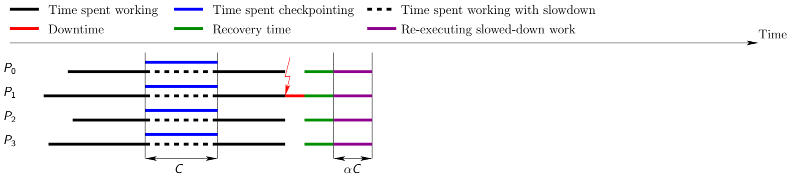


Waste due to failures in computation phase



Coordinated checkpointing protocol: all processors must recover from last checkpoint

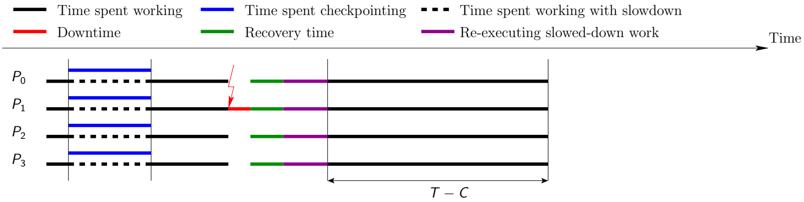
Waste due to failures in computation phase



Redo the work destroyed by the failure, that was done in the checkpointing phase before the computation phase

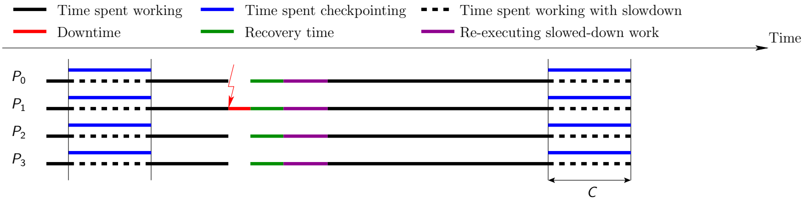
But no checkpoint is taken in parallel, hence this re-execution is faster than the original computation

Waste due to failures in computation phase



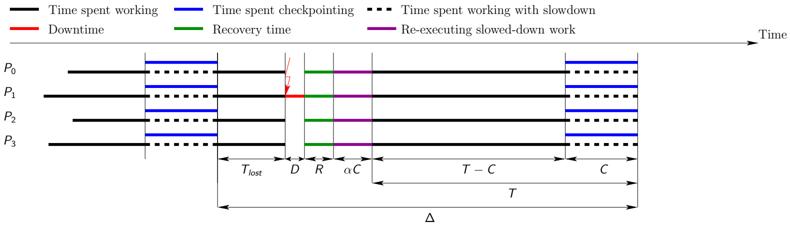
Re-execute the computation phase

Waste due to failures in computation phase



Finally, the checkpointing phase is executed

Total waste

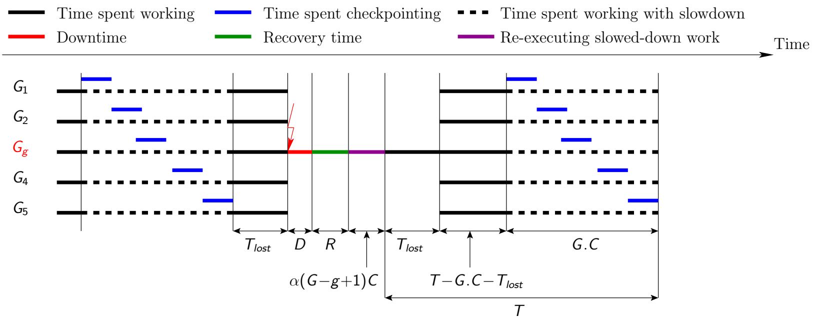


$$\text{WASTE}[fail] = \frac{1}{\mu}\left(D + R + \alpha C + \frac{T}{2}\right)$$

Optimal period $T_{\text{opt}} = \sqrt{2(1-\alpha)(\mu - (D+R))C}$

Outline

Hierarchical checkpointing



- Processors partitioned into G groups
- Each group includes q processors
- Inside each group: coordinated checkpointing in time $C(q)$
- Inter-group messages are logged

Accounting for message logging: Impact on work

- ☹ Logging messages slows down execution:
  $\Rightarrow$ $\textsc{Work}$ becomes $\lambda\textsc{Work}$, where $0 < \lambda < 1$
  Typical value: $\lambda \approx 0.98$
- ☺ Re-execution after a failure is faster:
  $\Rightarrow$ $\textsc{Re-Exec}$ becomes $\frac{\textsc{Re-Exec}}{\rho}$, where $\rho \in [1..2]$
  Typical value: $\rho \approx 1.5$

$$\textsc{Waste}[FF] = \frac{T - \lambda\textsc{Work}}{T}$$

$$\textsc{Waste}[fail] = \frac{1}{\mu}\left(D(q) + R(q) + \frac{\textsc{Re-Exec}}{\rho}\right)$$

Accounting for message logging: Impact on checkpoint size

- Inter-groups messages logged continuously
- Checkpoint size increases with amount of work executed before a checkpoint ☹
- $C_0(q)$: Checkpoint size of a group without message logging

$$C(q) = C_0(q)(1 + \beta \text{WORK}) \Leftrightarrow \beta = \frac{C(q) - C_0(q)}{C_0(q)\text{WORK}}$$

$$\text{WORK} = \lambda(T - (1-\alpha)GC(q))$$

$$C(q) = \frac{C_0(q)(1 + \beta\lambda T)}{1 + GC_0(q)\beta\lambda(1-\alpha)}$$

Three case studies

Coord-IO

Coordinated approach: $C = C_{\text{Mem}} = \frac{\text{Mem}}{\text{b}_{io}}$
where Mem is the memory footprint of the application

Hierarch-IO

Several (large) groups, *I/O-saturated*
$\Rightarrow$ groups checkpoint sequentially

$$C_0(q) = \frac{C_{\text{Mem}}}{G} = \frac{\text{Mem}}{G\text{b}_{io}}$$

Hierarch-Port

Very large number of smaller groups, *port-saturated*
$\Rightarrow$ some groups checkpoint in parallel
Groups of q_{min} processors, where $\text{q}_{\text{min}}\text{b}_{port} \geq \text{b}_{io}$

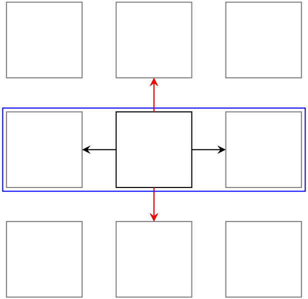
Three applications

1. 2D-stencil
2. Matrix product
3. 3D-Stencil
   - Plane
   - Line

Computing β for 2D-Stencil

$$C(q) = C_0(q) + Logged\_Msg = C_0(q)(1 + \beta \textsc{Work})$$

Real $n \times n$ matrix and $p \times p$ grid
$Work = \frac{9b^2}{s_p}$, $b = n/p$
Each process sends a block to its 4 neighbors



HIERARCH-IO:

- 1 group = 1 grid row
- 2 out of the 4 messages are logged
- $\beta = \frac{Logged\_Msg}{C_0(q)\textsc{Work}} = \frac{2pb}{pb^2(9b^2/s_p)} = \frac{2s_p}{9b^3}$

HIERARCH-PORT:

- β doubles

Four 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	Write	I/O Bandwidth (b_{port}) Read/Write per processor
Titan	299,008	16,688	16	32GB	300GB/s	300GB/s	20GB/s
K-Computer	705,024	88,128	8	16GB	150GB/s	96GB/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
K-Computer	Coord-IO	1 (14,688s)	/	/
	Hierarch-IO	296 (50s)	0.0002858	0.001113
	Hierarch-Port	17,626 (0.83s)	0.0005716	0.002227
Exascale-Slim	Coord-IO	1 (64,000s)	/	/
	Hierarch-IO	1,000 (64s)	0.0002599	0.001013
	Hierarch-Port	200,0000 (0.32s)	0.0005199	0.002026
Exascale-Fat	Coord-IO	1 (64,000s)	/	/
	Hierarch-IO	316 (217s)	0.00008220	0.0003203
	Hierarch-Port	33,3333 (1.92s)	0.00016440	0.0006407

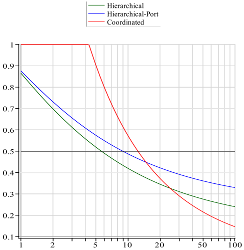
Checkpoint time

Name	C
K-Computer	14,688s
Exascale-Slim	64,000
Exascale-Fat	64,000

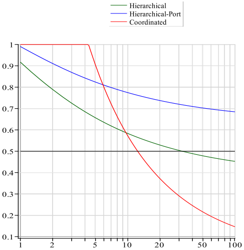
- Large time to dump the memory
- Using 1%C
- Comparing with 0.1%C for exascale platforms
- $\alpha = 0.3$, $\lambda = 0.98$ and $\rho = 1.5$

Plotting formulas – Platform: Titan

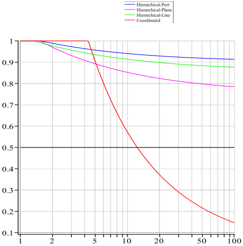
Stencil 2D



Matrix product



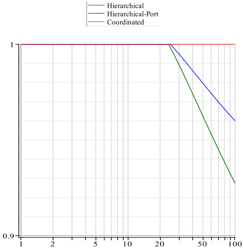
Stencil 3D



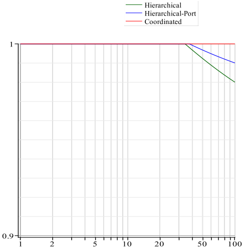
Waste as a function of processor MTBF μ_{ind}

Platform: K-Computer

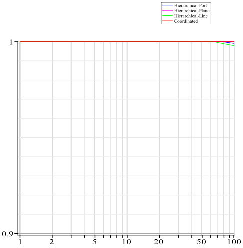
Stencil 2D



Matrix product



Stencil 3D



Waste as a function of processor MTBF μ_{ind}

Plotting formulas – Platform: Exascale

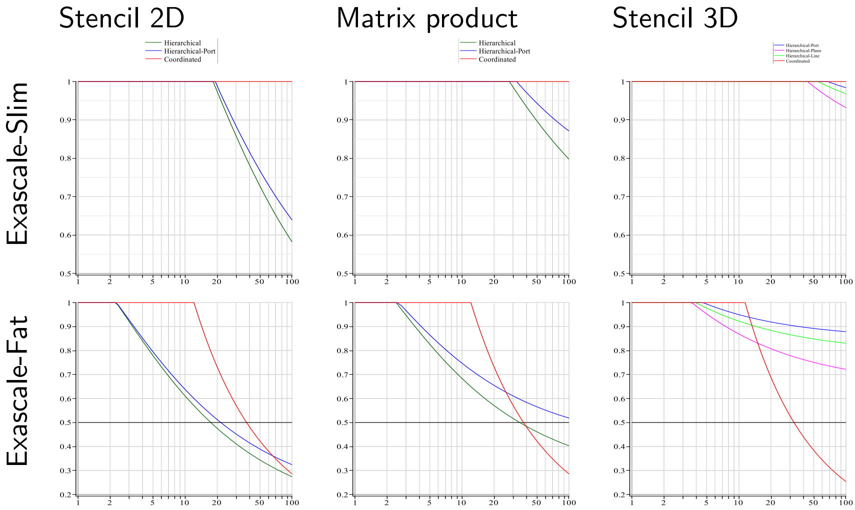
$\text{Waste} = 1$ for all scenarios!!!

Plotting formulas – Platform: Exascale

WASTE = 1 for all scenarios!!!

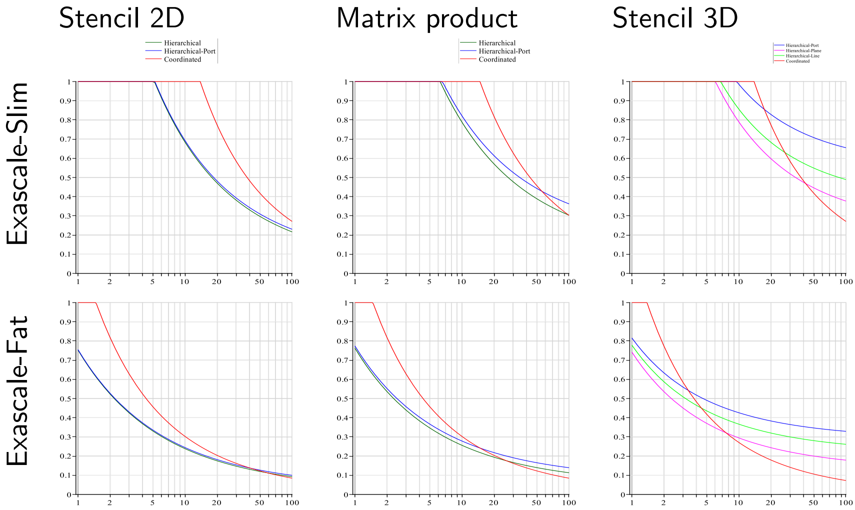
Goodbye Exascale?!

Plotting formulas – Platform: Exascale with $C = 1,000$



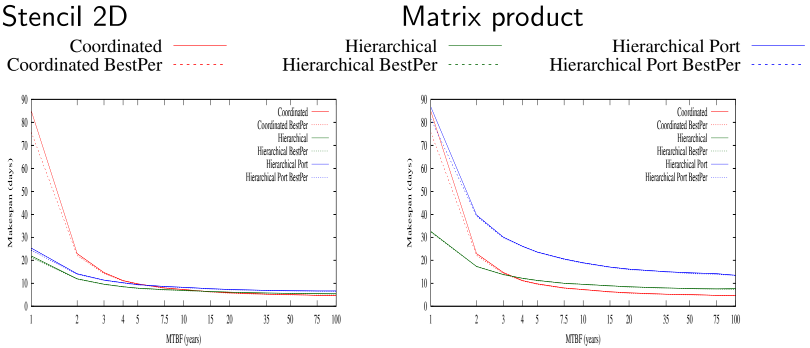
Waste as a function of processor MTBF μ_{ind}, $C = 1,000$

Plotting formulas – Platform: Exascale with $C = 100$



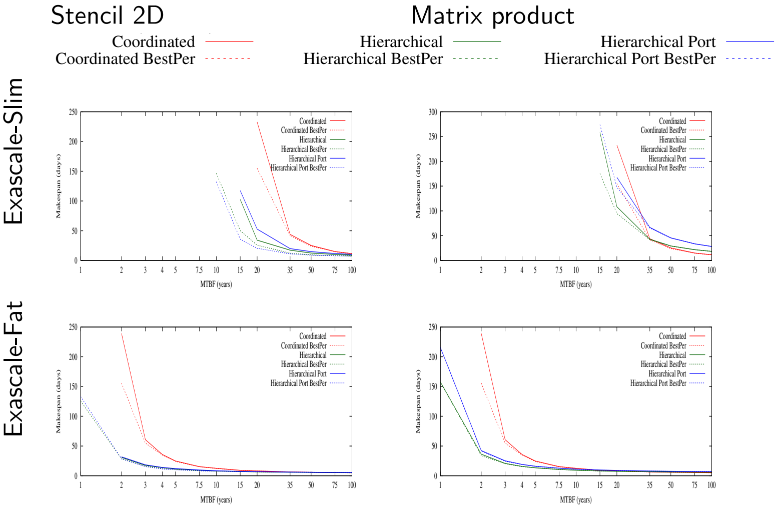
Waste as a function of processor MTBF μ_{ind}, $C = 100$

Simulations – Platform: Titan



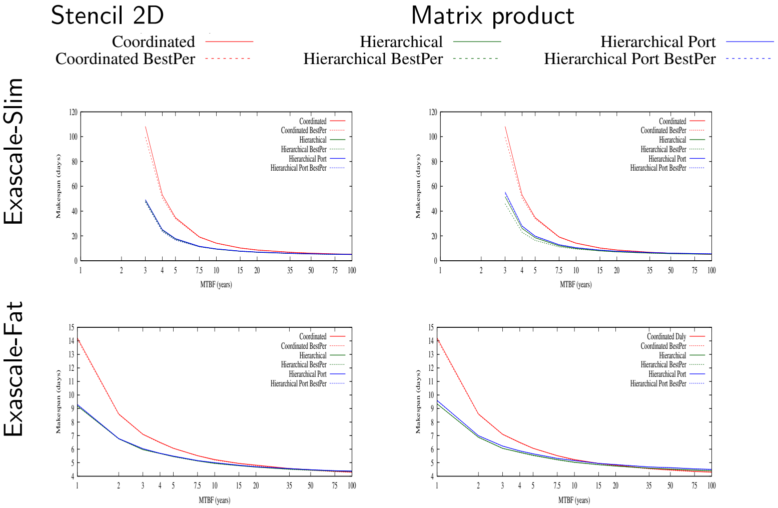
Makespan (in days) as a function of processor MTBF μ_{ind}

Simulations – Platform: Exascale with $C = 1,000$



Makespan (in days) as a function of processor MTBF μ_{ind}, $C = 1,000$

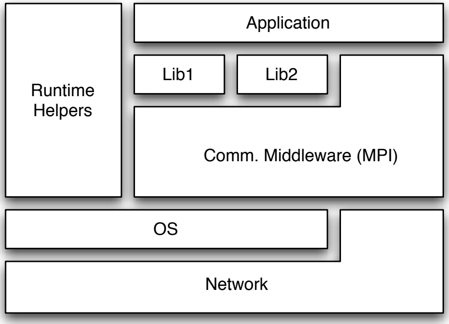
Simulations – Platform: Exascale with $C = 100$



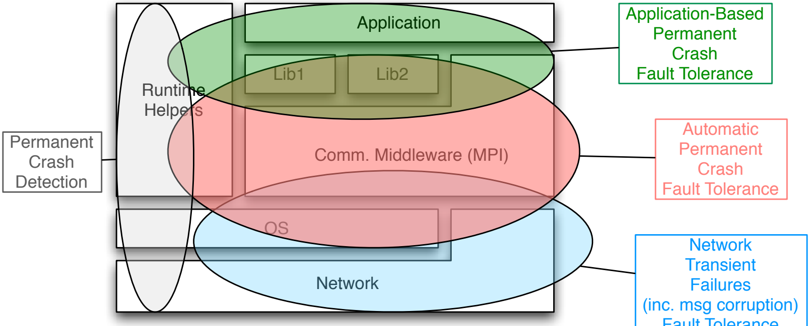
Makespan (in days) as a function of processor MTBF μ_{ind}, $C = 100$

Outline

Fault Tolerance Software Stack



Fault Tolerance Software Stack



Motivation

Motivation

- Generality can prevent Efficiency
- Specific solutions exploit more capability, have more opportunity to extract efficiency
- Naturally Fault Tolerant Applications

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
3. Probabilistic models for checkpointing (45mn)
   - Young/Daly's approximation
   - Coordinated checkpointing
   - Hierarchical checkpointing
4. Application-specific fault-tolerance techniques (45mn)
   - Fault-Tolerant Middleware
   - Bags of tasks
   - Iterative algorithms and fixed-point convergence
   - ABFT for Linear Algebra applications
   - Composite approach: ABFT & Checkpointing
5. Other techniques (35mn)
   - Replication
   - Failure Prediction
   - In-memory checkpointing
   - Silent errors
6. Conclusion (10mn)

HPC – MPI

HPC

- Most popular middleware for multi-node programming in HPC: Message Passing Interface (+Open MP +pthread +...)
- Fault Tolerance in MPI:

> *[...] it is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, such failures will be reflected as errors in the relevant communication call. Similarly, MPI itself provides no mechanisms for handling processor failures.*
>
> – MPI Standard 3.0, p. 20, l. 36:39

HPC – MPI

HPC

- Most popular middleware for multi-node programming in HPC: Message Passing Interface (+Open MP +pthread +...)
- Fault Tolerance in MPI:

> *This document does not specify the state of a computation after an erroneous MPI call has occurred.*

– MPI Standard 3.0, p. 21, l. 24:25

HPC – MPI

MPI Implementations

- Open MPI (`http://www.open-mpi.org`)
  - On failure detection, the runtime system kills all processes
  - trunk: error is never reported to the MPI processes.
  - ft-branch: the error is reported, MPI might be partly usable.
- MPICH (`http://www.mcs.anl.gov/mpi/mpich/`)
  - Default: on failure detection, the runtime kills all processes. Can be de-activated by a runtime switch
  - Errors might be reported to MPI processes in that case. MPI might be partly usable.

FT Middleware in HPC

- Not MPI. Sockets, PVM... CCI? `http://www.olcf.ornl.gov/center-projects/common-communication-interface/` UCCS?
- FT-MPI: `http://icl.cs.utk.edu/harness/`, 2003
- MPI-Next-FT proposal (Open MPI, MPICH): ULFM
  - User-Level Failure Mitigation
  - `http://fault-tolerance.org/ulfm/`
- Checkpoint on Failures: the rejuvenation in HPC

MPI-Next-FT proposal: ULFM

Goal

Resume Communication Capability for MPI (and nothing more)

- Failure Reporting
- Failure notification propagation / Distributed State reconciliation

$\Longrightarrow$ In the past, these operations have often been merged
$\Longrightarrow$ this incurs high failure free overheads
ULFM splits these steps and gives *control to the user*

- Recovery
- Termination

MPI-Next-FT proposal: ULFM

Goal

Resume Communication Capability for MPI (and nothing more)

- Error reporting indicates impossibility to carry an operation
  - State of MPI is unchanged for operations that can continue (i.e. if they do not involve a dead process)
- Errors are *non uniformly* returned
  - (Otherwise, synchronizing semantic is altered drastically with high performance impact)

New APIs

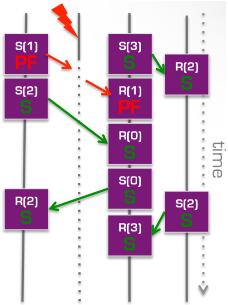
- REVOKE allows to resolve non-uniform error status
- SHRINK allows to rebuild error-free communicators
- AGREE allows to quit a communication pattern knowing it is fully complete

MPI-Next-FT proposal: ULFM

Errors are visible only for operations that cannot complete

Error Reporting

- Operations that cannot complete return
  - ERR_PROC_FAILED, or ERR_PENDING if appropriate
  - State of MPI Objects is unchanged (communicators etc.)
  - Repeating the same operation has the same outcome
- Operations that can be completed return MPI_SUCCESS
  - point to point operations between non-failed ranks can continue

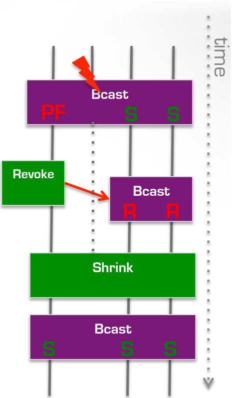


MPI-Next-FT proposal: ULFM

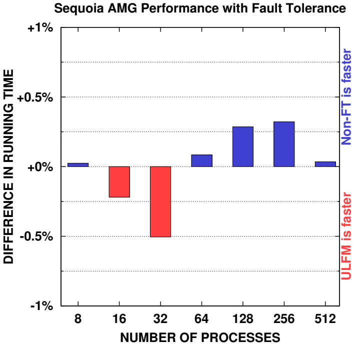
Inconsistent Global State and Resolution

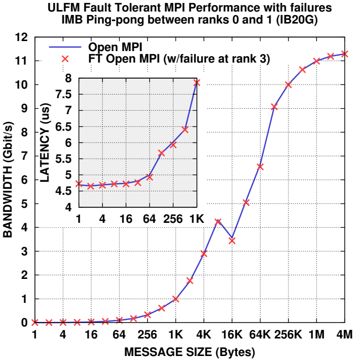
Error Reporting

- Operations that can't complete return
  - ERR_PROC_FAILED, or ERR_PENDING if appropriate
- Operations that can be completed return MPI_SUCCESS
  - Local semantic is respected (buffer content is defined), this does not indicate success at other ranks.
  - New constructs MPI_Comm_Revoke/MPI_Comm_shrink are a base to resolve inconsistencies introduced by failure



MPI-Next-FT proposal: ULFM



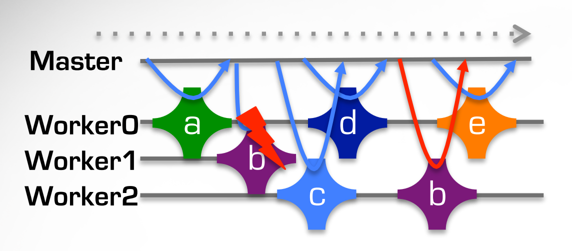


Open MPI - ULFM support

- Branch of Open MPI (`www.open-mpi.org`)
- Maintained on bitbucket: `https://bitbucket.org/icldistcomp/ulfm`

Outline

Master/Worker



Worker

```
while(1) {
    MPI_Recv( master, &work );
    if( work == STOP_CMD )
        break;
    process_work(work, &result);
    MPI_Send( master, result );
}
```

Master/Worker

Master

```
for(i = 0; i < active_workers; i++) {
    new_work = select_work();
    MPI_Send(i, new_work);
}
while( active_workers > 0 ) {
    MPI_Wait( MPI_ANY_SOURCE, &worker );
    MPI_Recv( worker, &work );
    work_completed(work);
    if( work_tocomplete() == 0 ) break;
    new_work = select_work();
    if( new_work) MPI_Send( worker, new_work );
}
for(i = 0; i < active_workers; i++) {
   MPI_Send(i, STOP_CMD);
}
```

FT Master

Fault Tolerant Master

```
/* Non-FT preamble */
for(i = 0; i < active_workers; i++) {
    new_work = select_work();
    rc = MPI_Send(i, new_work);
    if( MPI_SUCCESS != rc ) MPI_Abort(MPI_COMM_WORLD);
}
/* FT Section */
<...>
/* Non-FT epilogue */
for(i = 0; i < active_workers; i++) {
    rc = MPI_Send(i, STOP_CMD);
    if( MPI_SUCCESS != rc ) MPI_Abort(MPI_COMM_WORLD);
}
```

FT Master

Fault Tolerant Master

```
while( active_workers > 0 ) { /* FT Section */
   rc = MPI_Wait( MPI_ANY_SOURCE, &worker );
   switch( rc ) {
      case MPI_SUCCESS: /* Received a result */
      break;
      case MPI_ERR_PENDING:
      case MPI_ERR_PROC_FAILED: /* Worker died */
         <...>
         continue;
      break;
      default:
         /* Unknown error, not related to failure */
         MPI_Abort(MPI_COMM_WORLD);
   }
   <...>
```

FT Master

Fault Tolerant Master

```
    case MPI_ERR_PENDING:
    case MPI_ERR_PROC_FAILED:
        /* A worker died */
       MPI_Comm_failure_ack(comm);
       MPI_Comm_failure_get_acked(comm, &group);
       MPI_Group_difference(group, failed,
                            &newfailed);
       MPI_Group_size(newfailed, &ns);
       active_workers -= ns;
       /* Iterate on newfailed to mark the work
        * as not submitted */
       failed = group;
       continue;
```

FT Master

Fault Tolerant Master

```
rc = MPI_Recv( worker, &work );
switch( rc ) {
     /* Code similar to the MPI_Wait code */
     <...>
}
work_completed(work);
if( work_tocomplete() == 0 ) break;
new_work = select_work();
```

FT Master

Fault Tolerant Master

```
    if(new_work) {
        rc = MPI_Send( worker, new_work );
        switch( rc ) {
            /* Code similar to the MPI_Wait code */
            /* Re-submit the work somewhere */
            <...>
        }
    }
} /* End of while( active_workers > 0 ) */
MPI_Group_difference(comm, failed, &living);
/* Iterate on living */
for(i = 0; i < active_workers; i++) {
    MPI_Send(rank_of(comm, living, i), STOP_CMD);
}
```

Outline

Iterative Algorithm

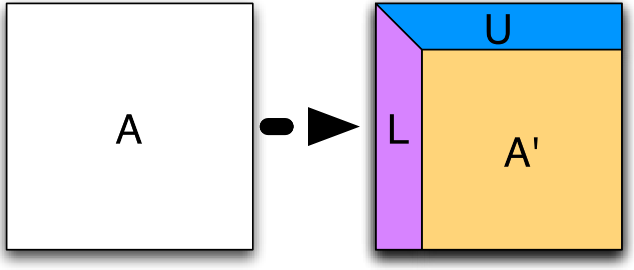
```
while( gnorm > epsilon ) {
     iterate();
     compute_norm(&lnorm);
     rc = MPI_Allreduce( &lnorm, &gnorm, 1,
                         MPI_DOUBLE, MPI_MAX, comm);
     if( (MPI_ERR_PROC_FAILED == rc) ||
         (MPI_ERR_COMM_REVOKED == rc) ||
         (gnorm <= epsilon) ) {

        if( MPI_ERR_PROC_FAILED == rc )
            MPI_Comm_revoke(comm);

        allsuceeded = (rc == MPI_SUCCESS);
        MPI_Comm_agree(comm, &allsuceeded);
```

Iterative Algorithm

```
        if( !allsucceeded ) {
            MPI_Comm_revoke(comm);
            MPI_Comm_shrink(comm, &comm2);
            MPI_Comm_free(comm);
            comm = comm2;
            gnorm = epsilon + 1.0;
        }
    }
}
```

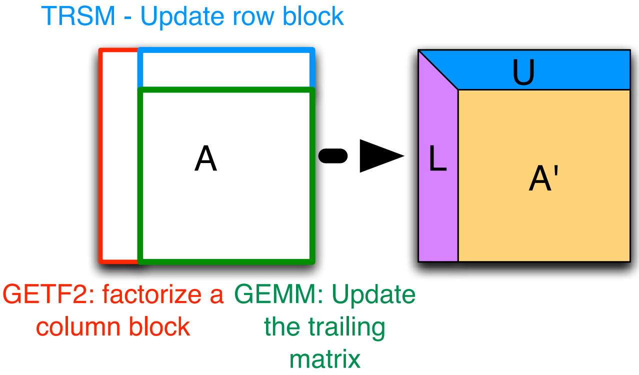
Outline

Example: block LU/QR factorization



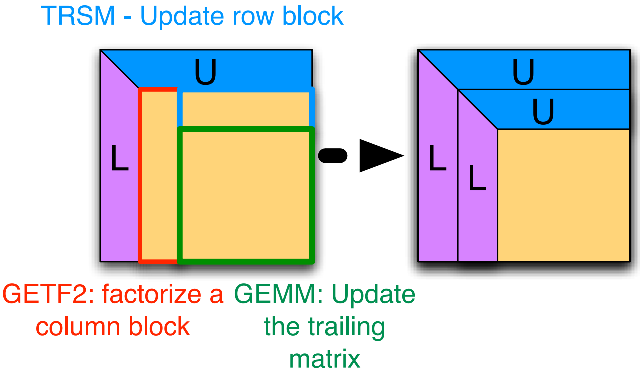
- Solve $A \cdot x = b$ (hard)
- Transform A into a LU factorization
- Solve $L \cdot y = B \cdot b$, then $U \cdot x = y$

Example: block LU/QR factorization



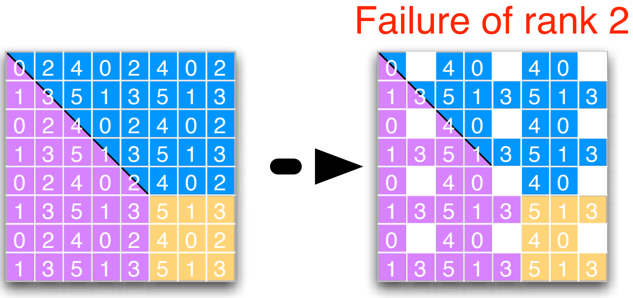
- Solve $A \cdot x = b$ (hard)
- Transform A into a LU factorization
- Solve $L \cdot y = B \cdot b$, then $U \cdot x = y$

Example: block LU/QR factorization



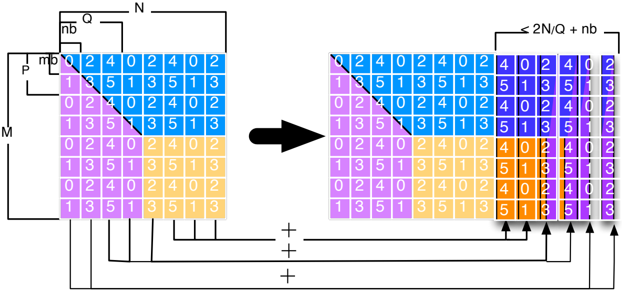
- Solve $A \cdot x = b$ (hard)
- Transform A into a LU factorization
- Solve $L \cdot y = B \cdot b$, then $U \cdot x = y$

Example: block LU/QR factorization



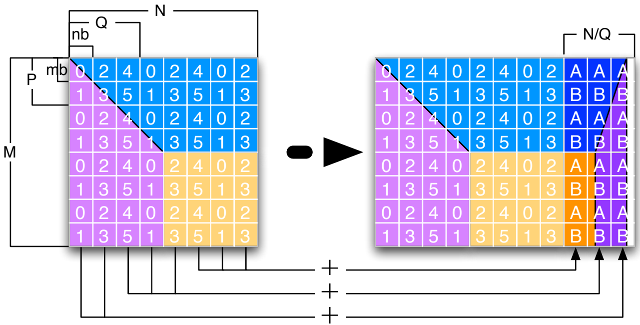
- 2D Block Cyclic Distribution (here 2×3)
- A single failure $\Rightarrow$ many data lost

Algorithm Based Fault Tolerant LU decomposition



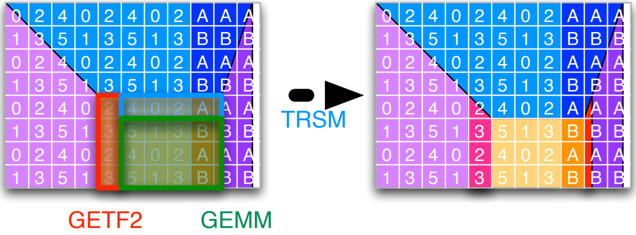
- Checksum: invertible operation on the data of the row / column
  - Checksum blocks are doubled, to allow recovery when data and checksum are lost together

Algorithm Based Fault Tolerant LU decomposition



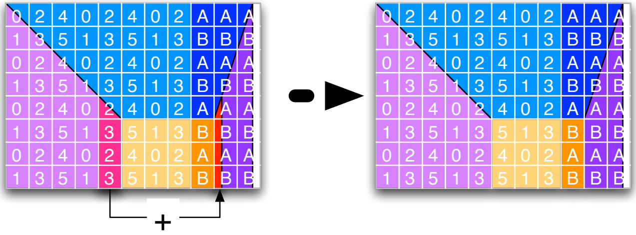
- Checksum: invertible operation on the data of the row / column
  - Checksum replication can be avoided by dedicating computing resources to checksum storage

Algorithm Based Fault Tolerant LU decomposition



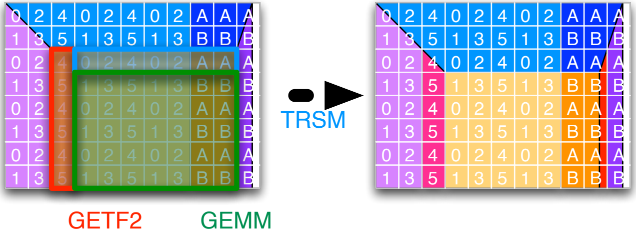
- Checksum: invertible operation on the data of the row / column
  - Idea of ABFT: applying the operation on data and checksum preserves the checksum properties

Algorithm Based Fault Tolerant LU decomposition



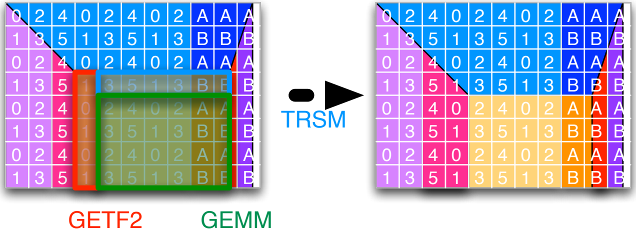
- Checksum: invertible operation on the data of the row / column
  - For the part of the data that is not updated this way, the checksum must be re-calculated

Algorithm Based Fault Tolerant LU decomposition



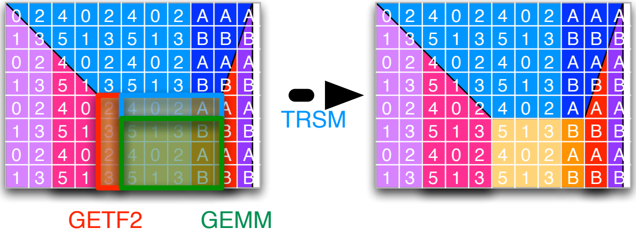
- Checksum: invertible operation on the data of the row / column
  - To avoid slowing down all processors and panel operation, group checksum updates every q block columns

Algorithm Based Fault Tolerant LU decomposition



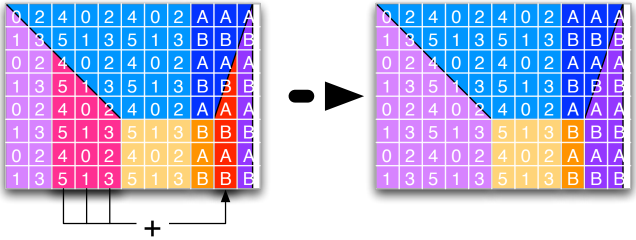
- Checksum: invertible operation on the data of the row / column
  - To avoid slowing down all processors and panel operation, group checksum updates every q block columns

Algorithm Based Fault Tolerant LU decomposition



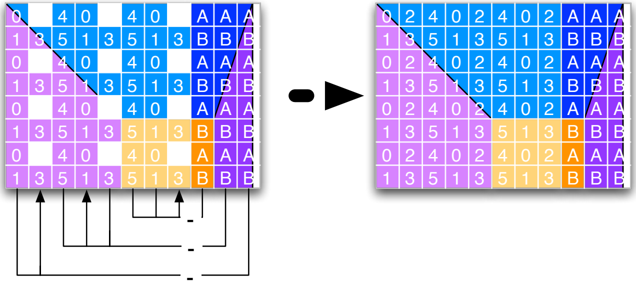
- Checksum: invertible operation on the data of the row / column
  - To avoid slowing down all processors and panel operation, group checksum updates every q block columns

Algorithm Based Fault Tolerant LU decomposition



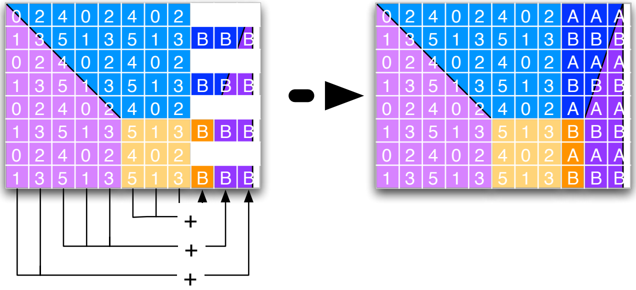
- Checksum: invertible operation on the data of the row / column
    - Then, update the missing coverage. Keep checkpoint block column to cover failures during that time

Algorithm Based Fault Tolerant LU decomposition



- In case of failure, conclude the operation, then
  - Missing Data = Checksum - Sum(Existing Data) s

Algorithm Based Fault Tolerant LU decomposition



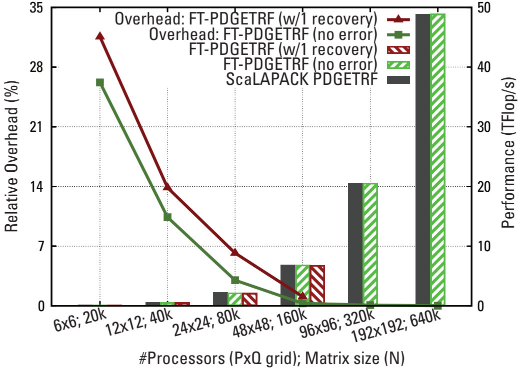
- In case of failure, conclude the operation, then
  - Missing Checksum = Sum(Existing Data)s

ABFT LU decomposition: implementation

MPI Implementation

- PBLAS-based: need to provide “Fault-Aware” version of the library
- Cannot enter recovery state at any point in time: need to complete ongoing operations despite failures
  - Recovery starts by defining the position of each process in the factorization and bring them all in a consistent state (checksum property holds)
- Need to test the return code of each and every MPI-related call

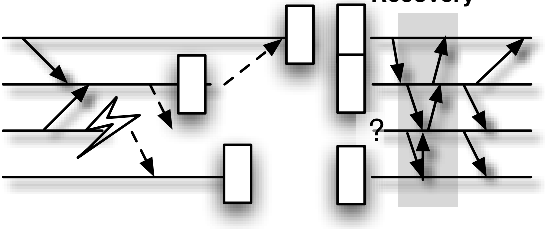
ABFT LU decomposition: performance



MPI-Next ULFM Performance

- Open MPI with ULFM; Kraken supercomputer;

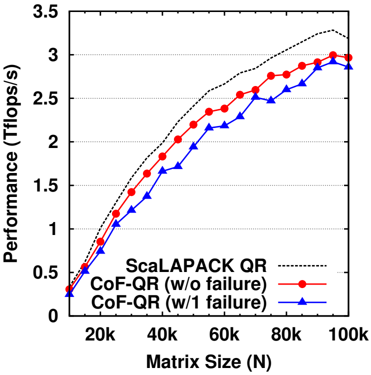
ABFT LU decomposition: implementation



Checkpoint on Failure - MPI Implementation

- FT-MPI / MPI-Next FT: not easily available on large machines
- Checkpoint on Failure = workaround

ABFT QR decomposition: performance



Checkpoint on Failure - MPI Performance

- Open MPI; Kraken supercomputer;

Outline

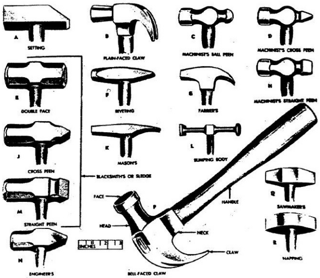
Fault Tolerance Techniques

General Techniques

- Replication
- Rollback Recovery
    - Coordinated Checkpointing
    - Uncoordinated Checkpointing & Message Logging
    - Hierarchical Checkpointing

Application-Specific Techniques

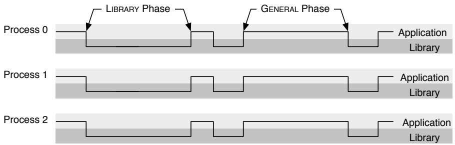
- Algorithm Based Fault Tolerance (ABFT)
- Iterative Convergence
- Approximated Computation



Application

Typical Application

```
for( aninsanenumber ) {
  /* Extract data from
   * simulation, fill up
   * matrix */
  sim2mat();

  /* Factorize matrix,
   * Solve */
  dgeqrf();
  dsolve();

  /* Update simulation
   * with result vector */
  vec2sim();
}
```



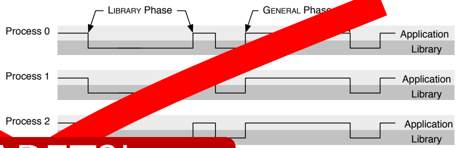
Characteristics

- ☺ Large part of (total) computation spent in factorization/solve
- Between LA operations:
  - ☹ use resulting vector / matrix with operations that do not preserve the checksums on the data
  - ☹ modify data not covered by ABFT algorithms

Application

Typical Applicatio

```
for( aninsanenumber ) {
  /* Extract dat
   * simulation ,
   * matrix */
  sim2mat();

  /* Factorize matri
   * Solve */
  dgeqrf();
  dsolve();

  /* Updat  imulation
   * with  esult vector */
  vec2s    ();
}
```



Goodbye ABFT?!

- ☺ Lar   part of (total) compu   ion spent in factorizat   /solve
- Between LA   erations:
  - ☹ use resultin   ector / matrix with operatio   that do not preserve the che   sums on the data
  - ☹ modify data not cov   d by ABFT algorithms

Application

Problem Statement

How to use fault tolerant operations(*) within a non-fault tolerant(**) application?(***)

(*) ABFT, or other application-specific FT

(**) Or within an application that does not have the same kind of FT

(***) And keep the application globally fault tolerant...

Typica

```
for( a
  /* I
   * s
   * r
  sim2

  /* I
   * S
  dge(
  dsol

  /* Update simulation
   * with result vector */
  vec2sim();
}
```

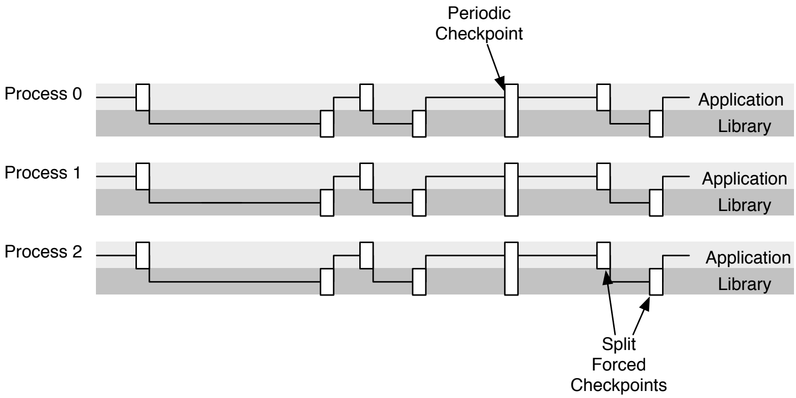
Application
Library
Application
Library
Application
Library

use resulting vector / matrix with operations that do not preserve the checksums on the data

☹ modify data not covered by ABFT algorithms

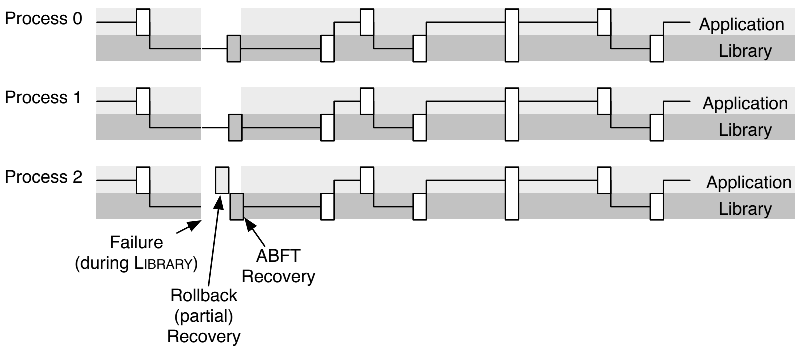
ABFT&PeriodicCkpt

ABFT&PeriodicCkpt: no failure



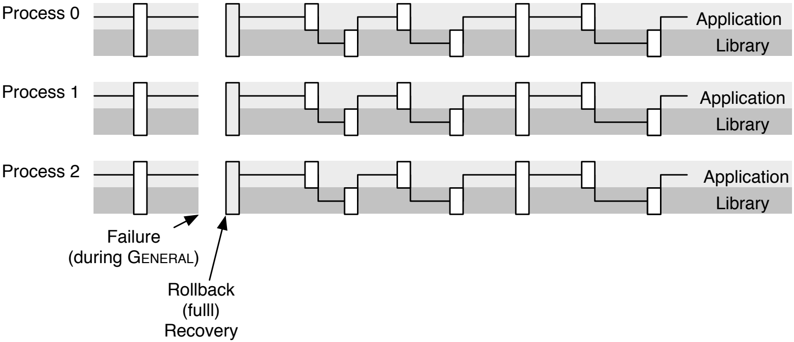
ABFT&PeriodicCkpt

ABFT&PeriodicCkpt: failure during Library phase

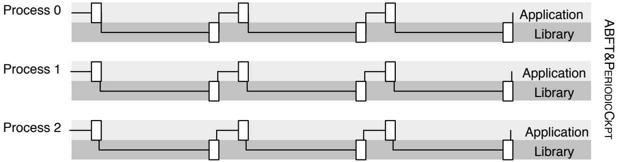


ABFT&PeriodicCkpt

ABFT&PeriodicCkpt: failure during General phase



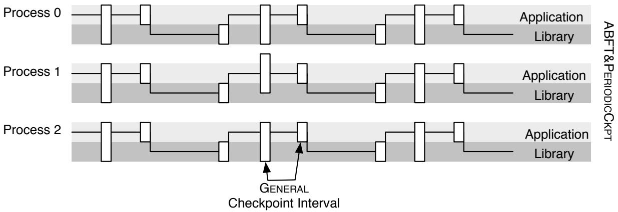
ABFT&PeriodicCkpt: Optimizations



ABFT&PeriodicCkpt: Optimizations

- If the duration of the General phase is too small: don't add checkpoints
- If the duration of the Library phase is too small: don't do ABFT recovery, remain in General mode
  - this assumes a performance model for the library call

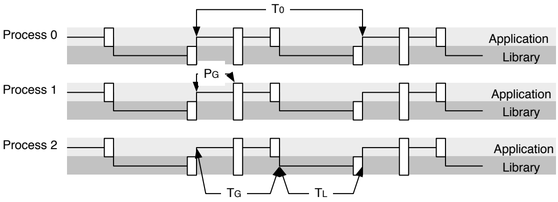
ABFT&PeriodicCkpt: Optimizations



ABFT&PeriodicCkpt: Optimizations

- If the duration of the General phase is too small: don't add checkpoints
- If the duration of the Library phase is too small: don't do ABFT recovery, remain in General mode
  - this assumes a performance model for the library call

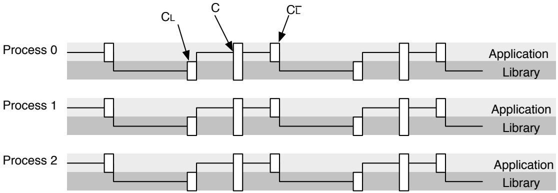
A few notations



Times, Periods

T_0: Duration of an Epoch (without FT)
$T_L = \alpha T_0$: Time spent in the LIBRARY phase
$T_G = (1 - \alpha) T_0$: Time spent in the GENERAL phase
P_G: Periodic Checkpointing Period
T^{ff}, T_G^{ff}, T_L^{ff}: "Fault Free" times
t_G^{lost}, t_L^{lost}: Lost time (recovery overheads)
T_G^{final}, T_L^{final}: Total times (with faults)

A few notations

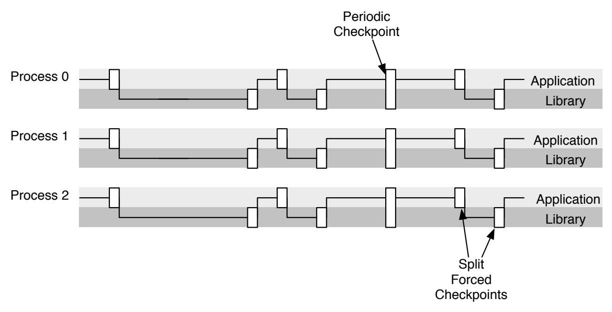


Costs

$C_L = \rho C$: time to take a checkpoint of the LIBRARY data set
$C_{\bar{L}} = (1-\rho)C$: time to take a checkpoint of the GENERAL data set
$R, R_{\bar{L}}$: time to load a full / GENERAL data set checkpoint
D: down time (time to allocate a new machine / reboot)
$\text{Recons}_{\text{ABFT}}$: time to apply the ABFT recovery
ϕ: Slowdown factor on the LIBRARY phase, when applying ABFT

GENERAL phase, fault free waste

GENERAL phase

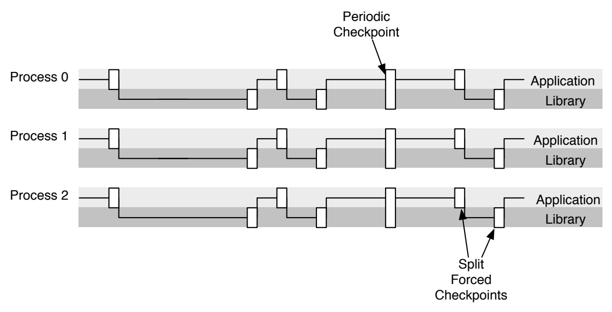


Without Failures

$$T_G^{\text{ff}} = \begin{cases} T_G + C_{\bar{L}} & \text{if } T_G < P_G \\ \frac{T_G}{P_G - C} \times P_G & \text{if } T_G \geq P_G \end{cases}$$

LIBRARY phase, fault free waste

LIBRARY phase

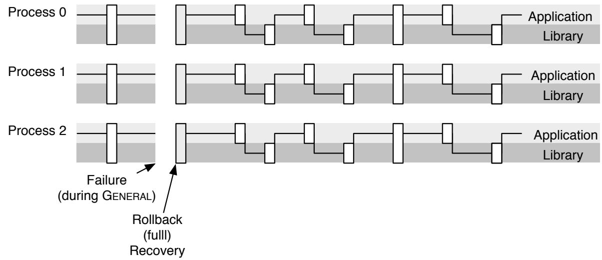


Without Failures

$$T_L^{\text{ff}} = \phi \times T_L + C_L$$

GENERAL phase, failure overhead

GENERAL phase

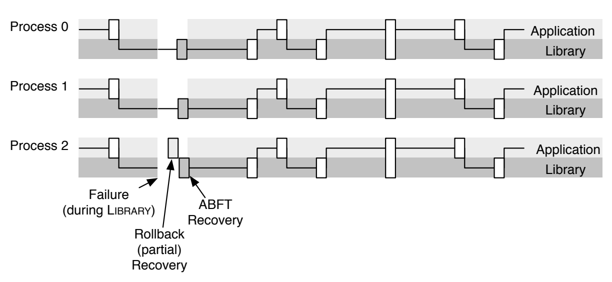


Failure Overhead

$$t_G^{\text{lost}} = \begin{cases} D + R + \frac{T_G^{\text{ff}}}{2} & \text{if } T_G < P_G \\ D + R + \frac{P_G}{2} & \text{if } T_G \geq P_G \end{cases}$$

LIBRARY phase, failure overhead

LIBRARY phase



Failure Overhead

$$t_L^{\text{lost}} = D + R_{\bar{L}} + \text{Recons}_{\text{ABFT}}$$

Overall

Overall

Time (with overheads) of LIBRARY phase is constant (in P_G):

$$T_L^{\text{final}} = \frac{1}{1 - \frac{D + R_{\bar{L}} + \text{Recons}_{\text{ABFT}}}{\mu}} \times (\alpha \times T_L + C_L)$$

Time (with overehads) of GENERAL phase accepts two cases:

$$T_G^{\text{final}} = \begin{cases} \frac{1}{1 - \frac{D + R + \frac{T_G + C_{\bar{L}}}{2}}{\mu}} \times (T_G + C_L) & \text{if } T_G < P_G \\ \frac{T_G}{(1 - \frac{C}{P_G})(1 - \frac{D + R + \frac{P_G}{2}}{\mu})} & \text{if } T_G \geq P_G \end{cases}$$

Which is minimal in the second case, if

$$P_G = \sqrt{2C(\mu - D - R)}$$

Waste

From the previous, we derive the waste, which is obtained by

$$\text{WASTE} = 1 - \frac{T_0}{T_G^{\text{final}} + T_L^{\text{final}}}$$

Toward Exascale, and Beyond!

Let's think at scale

- Number of components $\nearrow \Rightarrow$ MTBF $\searrow$
- Number of components $\nearrow \Rightarrow$ Problem Size $\nearrow$
- Problem Size $\nearrow \Rightarrow$
  Computation Time spent in LIBRARY phase $\nearrow$

☺ ABFT&PERIODICCKPT should perform better with scale

? By how much?

Competitors

FT algorithms compared

PeriodicCkpt Basic periodic checkpointing

Bi-PeriodicCkpt Applies incremental checkpointing techniques to save only the library data during the library phase.

ABFT&PeriodicCkpt The algorithm described above

Weak Scale #1

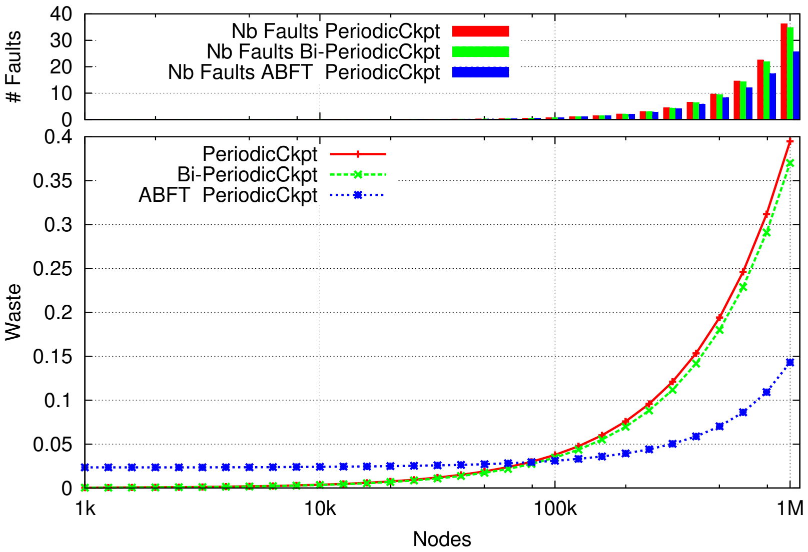
Weak Scale Scenario #1

- Number of components, n, increase
- Memory per component remains constant
- Problem Size increases in $O(\sqrt{n})$ (e.g. matrix operation)

- μ at $n = 10^5$: 1 day, is in $O(\frac{1}{n})$
- C $(=R)$ at $n = 10^5$, is 1 minute, is in $O(n)$
- α is constant at 0.8, as is ρ.

  (both LIBRARY and GENERAL phase increase in time at the same speed)

Weak Scale #1


40
30
20
10
0
Faults
Nb Faults PeriodicCkpt
Nb Faults Bi-PeriodicCkpt
Nb Faults ABFT PeriodicCkpt
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Waste
PeriodicCkpt
Bi-PeriodicCkpt
ABFT PeriodicCkpt
1k
10k
100k
1M
Nodes

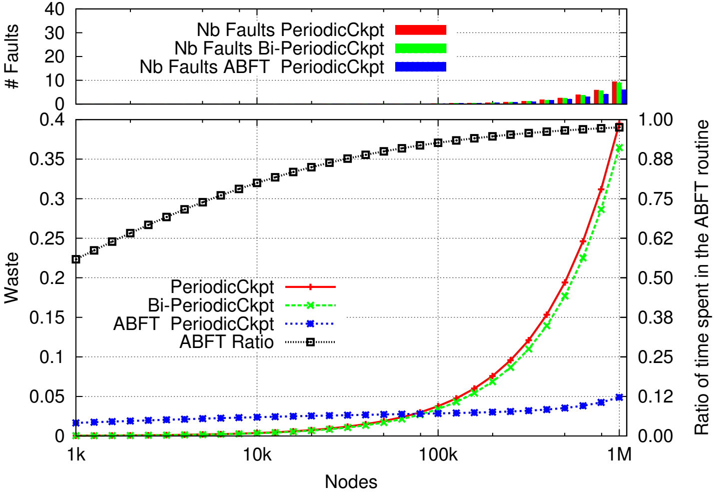
Weak Scale #2

Weak Scale Scenario #2

- Number of components, n, increase
- Memory per component remains constant
- Problem Size increases in $O(\sqrt{n})$ (e.g. matrix operation)

- μ at $n = 10^5$: 1 day, is $O(\frac{1}{n})$
- C $(=R)$ at $n = 10^5$, is 1 minute, is in $O(n)$
- ρ remains constant at 0.8, but LIBRARY phase is $O(n^3)$ when GENERAL phases progresses in $O(n^2)$ (α is 0.8 at $n = 10^5$ nodes).

Weak Scale #2



Faults
40
30
20
10
0
Nb Faults PeriodicCkpt
Nb Faults Bi-PeriodicCkpt
Nb Faults ABFT PeriodicCkpt
Waste
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
PeriodicCkpt
Bi-PeriodicCkpt
ABFT PeriodicCkpt
ABFT Ratio
1.00
0.88
0.75
0.62
0.50
0.38
0.25
0.12
0.00
Ratio of time spent in the ABFT routine
1k
10k
100k
1M
Nodes

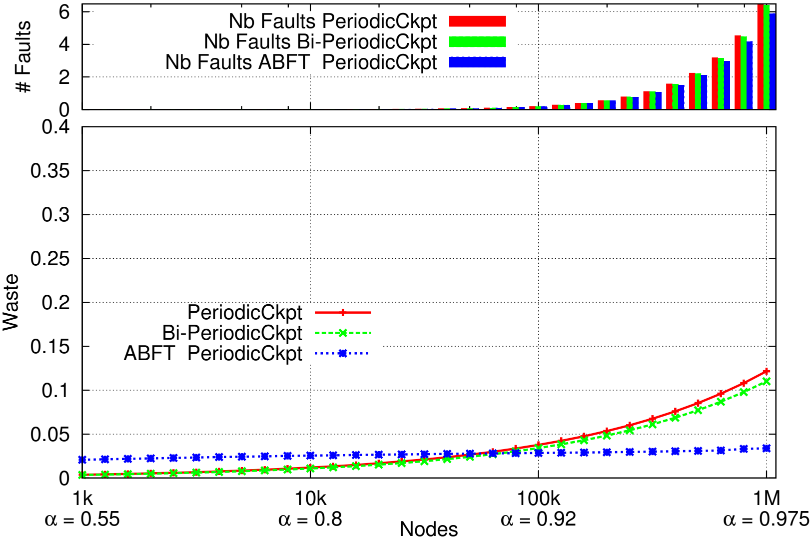
Weak Scale #3

Weak Scale Scenario #3

- Number of components, n, increase
- Memory per component remains constant
- Problem Size increases in $O(\sqrt{n})$ (e.g. matrix operation)

- μ at $n = 10^5$: 1 day, is $O(\frac{1}{n})$
- C $(=R)$ at $n = 10^5$, is 1 minute, stays independent of n $(O(1))$
- ρ remains constant at 0.8, but LIBRARY phase is $O(n^3)$ when GENERAL phases progresses in $O(n^2)$ (α is 0.8 at $n = 10^5$ nodes).

Weak Scale #3



Faults
6
4
2
0
Nb Faults PeriodicCkpt
Nb Faults Bi-PeriodicCkpt
Nb Faults ABFT PeriodicCkpt
Waste
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
PeriodicCkpt
Bi-PeriodicCkpt
ABFT PeriodicCkpt
1k
α = 0.55
10k
α = 0.8
100k
α = 0.92
1M
α = 0.975
Nodes

Outline

Outline

Replication

- Systematic replication: efficiency $< 50\%$
- Can replication+checkpointing be more efficient than checkpointing alone?
- Study by Ferreira et al. [SC'2011]: yes

Model by Ferreira et al. [SC' 2011]

- Parallel application comprising N processes
- Platform with $p_{total} = 2N$ processors
- Each process replicated $\rightarrow$ N *replica-groups*
- When a replica is hit by a failure, it is not restarted
- Application fails when both replicas in one replica-group have been hit by failures

The birthday problem

Classical formulation

What is the probability, in a set of m people, that two of them have same birthday ?

Relevant formulation

What is the average number of people required to find a pair with same birthday?

$$Birthday(N) = 1 + \int_0^{+\infty} e^{-x}(1 + x/N)^{N-1}dx$$

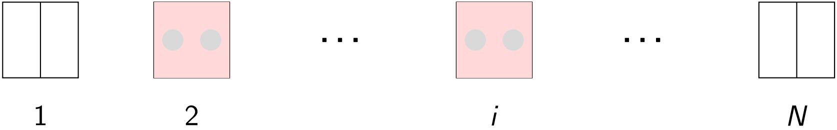
The analogy

Two people with same birthday

$\equiv$

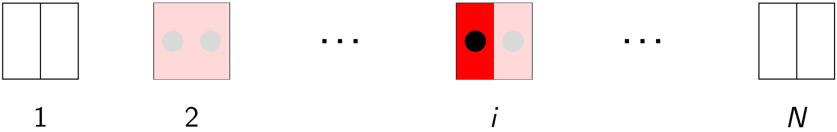
Two failures hitting same replica-group

Differences with birthday problem



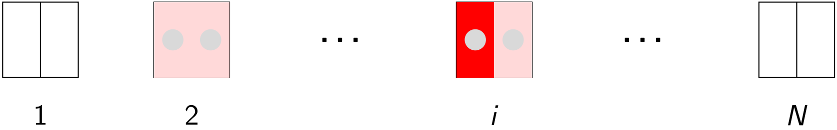
- N processes; each replicated twice
- Uniform distribution of failures
- First failure: each replica-group has probability $1/N$ to be hit
- Second failure

Differences with birthday problem



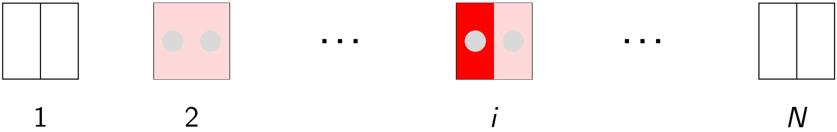
- N processes; each replicated twice
- Uniform distribution of failures
- First failure: each replica-group has probability $1/N$ to be hit
- Second failure

Differences with birthday problem



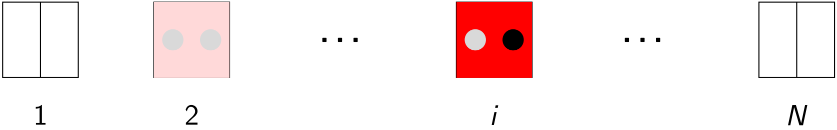
- N processes; each replicated twice
- Uniform distribution of failures
- First failure: each replica-group has probability $1/N$ to be hit
- Second failure: can failed PE be hit?

Differences with birthday problem



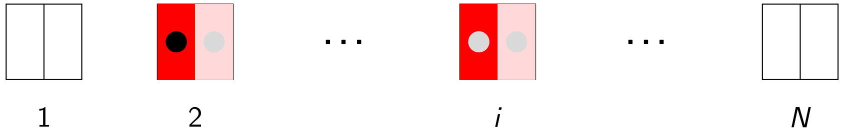
- N processes; each replicated twice
- Uniform distribution of failures
- First failure: each replica-group has probability $1/N$ to be hit
- Second failure cannot hit failed PE
  - Failure uniformly distributed over $2N-1$ PEs
  - Probability that replica-group i is hit by failure: $1/(2N-1)$
  - Probability that replica-group $\neq i$ is hit by failure: $2/(2N-1)$
  - Failure not uniformly distributed over replica-groups: this is not the birthday problem

Differences with birthday problem



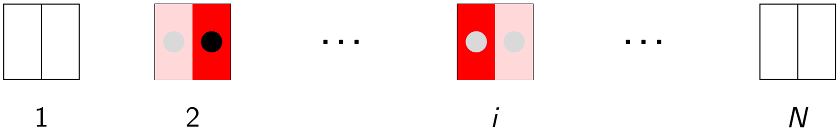
- N processes; each replicated twice
- Uniform distribution of failures
- First failure: each replica-group has probability $1/N$ to be hit
- Second failure cannot hit failed PE
  - Failure uniformly distributed over $2N-1$ PEs
  - Probability that replica-group i is hit by failure: $1/(2N-1)$
  - Probability that replica-group $\neq i$ is hit by failure: $2/(2N-1)$
  - Failure not uniformly distributed over replica-groups: this is not the birthday problem

Differences with birthday problem



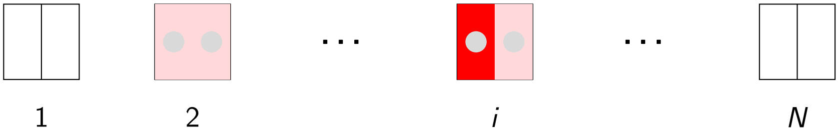
- N processes; each replicated twice
- Uniform distribution of failures
- First failure: each replica-group has probability $1/N$ to be hit
- Second failure cannot hit failed PE
  - Failure uniformly distributed over $2N-1$ PEs
  - Probability that replica-group i is hit by failure: $1/(2N-1)$
  - Probability that replica-group $\neq i$ is hit by failure: $2/(2N-1)$
  - Failure not uniformly distributed over replica-groups: this is not the birthday problem

Differences with birthday problem



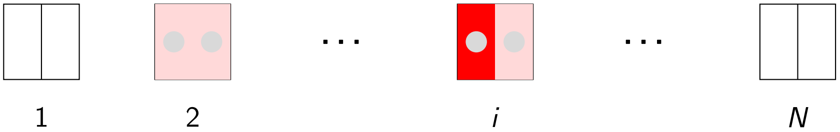
- N processes; each replicated twice
- Uniform distribution of failures
- First failure: each replica-group has probability $1/N$ to be hit
- Second failure cannot hit failed PE
  - Failure uniformly distributed over $2N - 1$ PEs
  - Probability that replica-group i is hit by failure: $1/(2N - 1)$
  - Probability that replica-group $\neq i$ is hit by failure: $2/(2N - 1)$
  - Failure not uniformly distributed over replica-groups: this is not the birthday problem

Differences with birthday problem



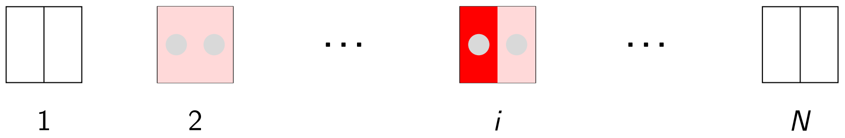
- N processes; each replicated twice
- Uniform distribution of failures
- First failure: each replica-group has probability $1/N$ to be hit
- Second failure cannot hit failed PE
  - Failure uniformly distributed over $2N - 1$ PEs
  - Probability that replica-group i is hit by failure: $1/(2N - 1)$
  - Probability that replica-group $\neq i$ is hit by failure: $2/(2N - 1)$
  - Failure not uniformly distributed over replica-groups: this is not the birthday problem

Differences with birthday problem



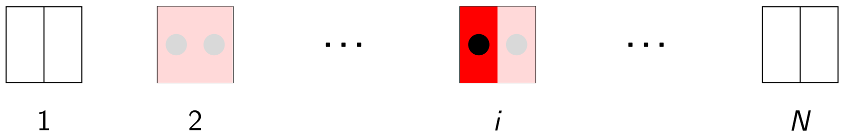
- N processes; each replicated twice
- Uniform distribution of failures
- First failure: each replica-group has probability $1/N$ to be hit
- Second failure can hit failed PE

Differences with birthday problem



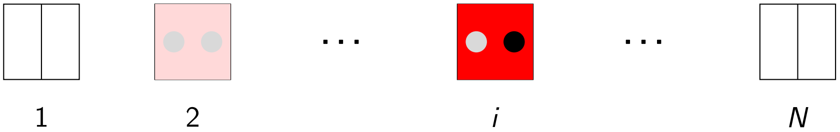
- N processes; each replicated twice
- Uniform distribution of failures
- First failure: each replica-group has probability $1/N$ to be hit
- Second failure can hit failed PE
  - Suppose failure hits replica-group i
  - If failure hits failed PE: application survives
  - If failure hits running PE: application killed
  - Not all failures hitting the same replica-group are equal: this is not the birthday problem

Differences with birthday problem



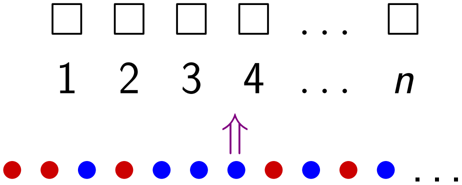
- N processes; each replicated twice
- Uniform distribution of failures
- First failure: each replica-group has probability $1/N$ to be hit
- Second failure can hit failed PE
  - Suppose failure hits replica-group i
  - If failure hits failed PE: application survives
  - If failure hits running PE: application killed
  - Not all failures hitting the same replica-group are equal: this is not the birthday problem

Differences with birthday problem



- N processes; each replicated twice
- Uniform distribution of failures
- First failure: each replica-group has probability $1/N$ to be hit
- Second failure can hit failed PE
  - Suppose failure hits replica-group i
  - If failure hits failed PE: application survives
  - If failure hits running PE: application killed
  - Not all failures hitting the same replica-group are equal: this is not the birthday problem

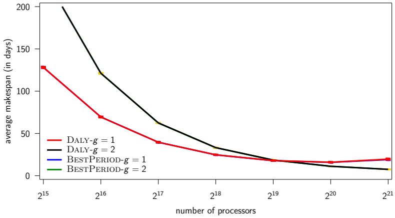
Correct analogy



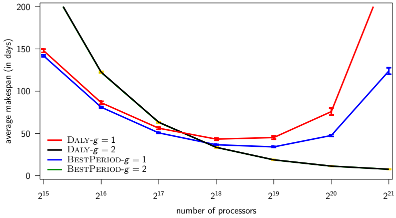
$N = n_{rg}$ bins, red and blue balls

Mean Number of Failures to Interruption (bring down application)
$MNFTI$ = expected number of balls to throw
until one bin gets one ball of each color

Failure distribution



(a) Exponential

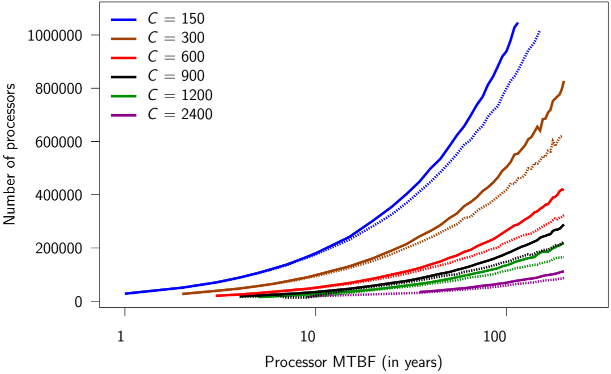


(b) Weibull, $k = 0.7$

Crossover point for replication when $\mu_{ind} = 125$ years

Weibull distribution with $k = 0.7$

Dashed line: Ferreira et al. Solid line: Correct analogy



- Study by Ferrreira et al. favors replication
- Replication beneficial if small μ + large C + big p_{total}

Outline

Framework

Predictor

- Exact prediction dates (at least C seconds in advance)
- Recall r: fraction of faults that are predicted
- Precision p: fraction of fault predictions that are correct

Events

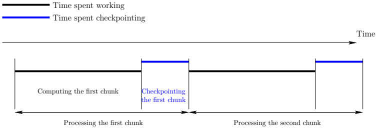
- *true positive*: predicted faults
- *false positive*: fault predictions that did not materialize as actual faults
- *false negative*: unpredicted faults

Algorithm

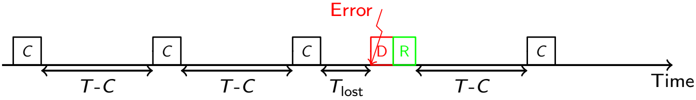
1. While no fault prediction is available:
   - checkpoints taken periodically with period T
2. When a fault is predicted at time t:
   - take a checkpoint ALAP (completion right at time t)
   - after the checkpoint, complete the execution of the period

Computing the waste

1. **Fault-free execution:** $\text{WASTE}[FF] = \frac{C}{T}$



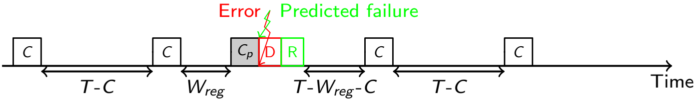
2. **Unpredicted faults:** $\frac{1}{\mu_{NP}}\left[D + R + \frac{T}{2}\right]$



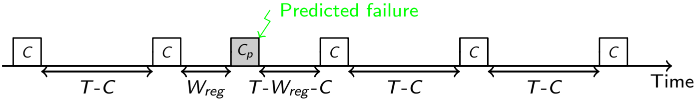
$$\text{WASTE}[fail] = \frac{1}{\mu}\left[(1-r)\frac{T}{2} + D + R + \frac{r}{p}C\right] \Rightarrow T_{opt} \approx \sqrt{\frac{2\mu C}{1-r}}$$

Computing the waste

3. **Predictions:** $\frac{1}{\mu_P}\left[p(C+D+R)+(1-p)C\right]$



with actual fault (true positive)

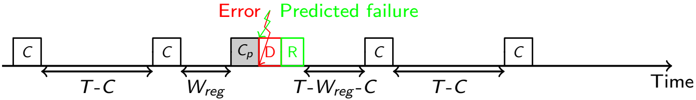


no actual fault (false negative)

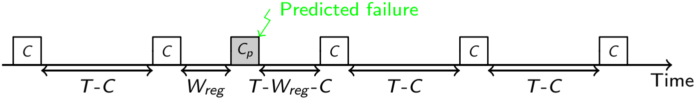
$$\text{WASTE}[fail]=\frac{1}{\mu}\left[(1-r)\frac{T}{2}+D+R+\frac{r}{p}C\right]\Rightarrow T_{opt}\approx\sqrt{\frac{2\mu C}{1-r}}$$

Computing the waste

3. **Predictions:** $\frac{1}{\mu_P}\left[p(C+D+R)+(1-p)C\right]$



with actual fault (true positive)



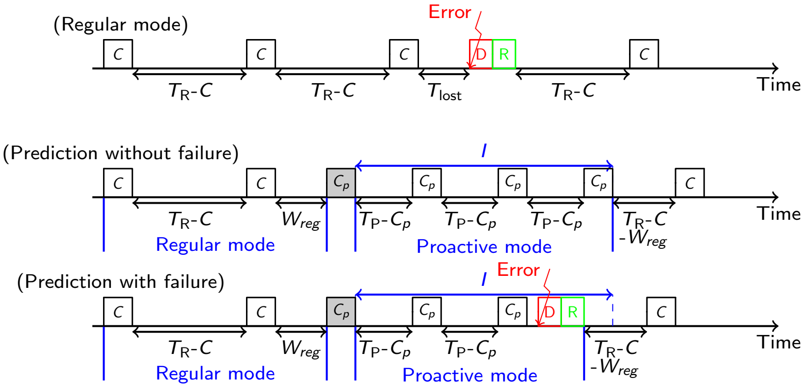
no actual fault (false negative)

$$\textsc{Waste}[fail]=\frac{1}{\mu}\left[(1-r)\frac{T}{2}+D+R+\frac{r}{p}C\right]\Rightarrow T_{opt}\approx\sqrt{\frac{2\mu C}{1-r}}$$

Refinements

- Use different value C_p for proactive checkpoints
- Avoid checkpointing too frequently for false negatives
  $\Rightarrow$ Only trust predictions with some fixed probability q
  $\Rightarrow$ Ignore predictions with probability $1 - q$
  Conclusion: trust predictor always or never ($q = 0$ or $q = 1$)
- Trust prediction depending upon position in current period
  $\Rightarrow$ Increase q when progressing
  $\Rightarrow$ Break-even point $\frac{C_p}{p}$

With prediction windows



Gets too complicated! ☹

Outline

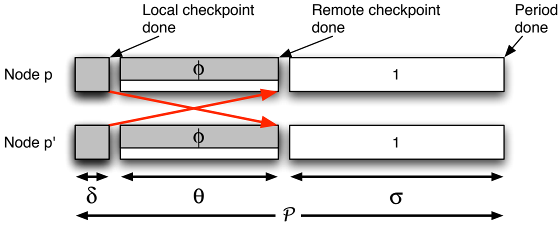
Motivation

- Checkpoint transfer and storage
  ⇒ critical issues of rollback/recovery protocols

- Stable storage: high cost

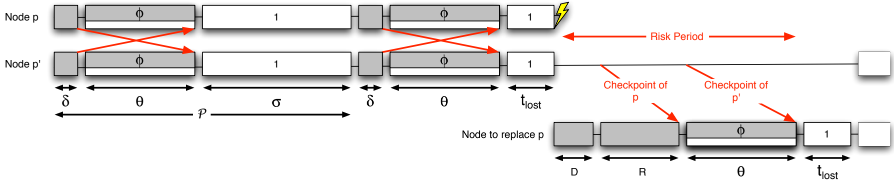
- Distributed in-memory storage:
  - Store checkpoints in local memory ⇒ no centralized storage
    ☺ Much better scalability
  - Replicate checkpoints ⇒ application survives single failure
    ☹ Still, risk of fatal failure in some (unlikely) scenarios

Double checkpoint algorithm (Kale et al., UIUC)



- Platform nodes partitioned into pairs
- Each node in a pair exchanges its checkpoint with its *buddy*
- Each node saves two checkpoints:
  - one locally: storing its own data
  - one remotely: receiving and storing its buddy's data

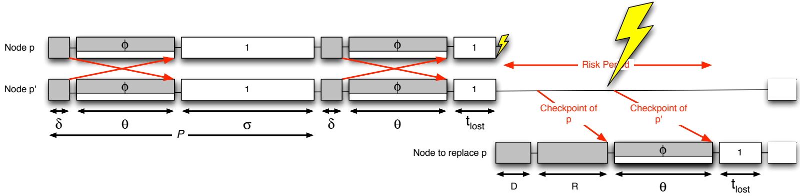
Failures



- After failure: downtime D and recovery from buddy node
- Two checkpoint files lost, must be re-sent to faulty processor

Best trade-off between performance and risk?

Failures



- After failure: downtime *D* and recovery from buddy node
- Two checkpoint files lost, must be re-sent to faulty processor
- Application at risk until complete reception of both messages

Best trade-off between performance and risk?

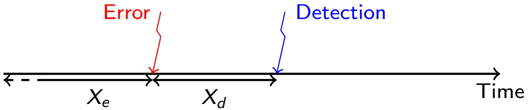
Outline

Silent errors

- Many types of faults: software error, hardware malfunction, memory corruption
- Many possible behaviors: silent, transient, unrecoverable
- Consider silent errors here
- This includes some software faults, some hardware errors (soft errors in L1 cache), bit flips (cosmic radiations)
- Silent error detected when corrupt data is activated

Detection latency

- Instantaneous error detection ⇒ fail-stop failures
- Silent errors (data corruption) ⇒ detection latency

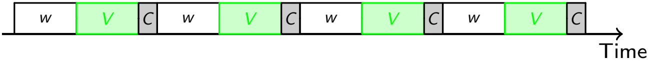


Error and detection latency

- Last checkpoint may have saved an already corrupted state
- Even when saving k checkpoints: which one to roll back to?
- Critical failure: all checkpoints contain corrupted data

Coupling checkpointing and verification

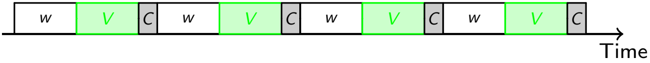
- Verification mechanism of cost V
- Simplest idea: verify work before each checkpoint



V large compared to w $\Rightarrow$ large WASTE_{ff}, can we improve that?

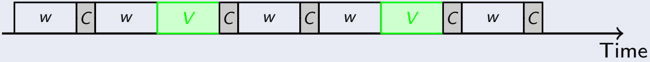
Coupling checkpointing and verification

- Verification mechanism of cost V
- Simplest idea: verify work before each checkpoint



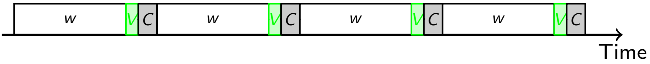
V large compared to $w \Rightarrow$ large WASTE_{ff}, can we improve that?

Is this better?



Coupling checkpointing and verification

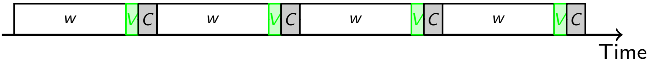
- Verification mechanism of cost V
- Simplest idea: verify work before each checkpoint



V small in front of $w \Rightarrow$ large $\text{WASTE}_{\text{fail}}$, can we improve that?

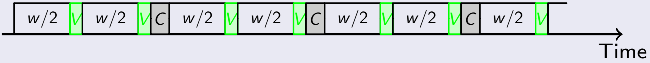
Coupling checkpointing and verification

- Verification mechanism of cost V
- Simplest idea: verify work before each checkpoint

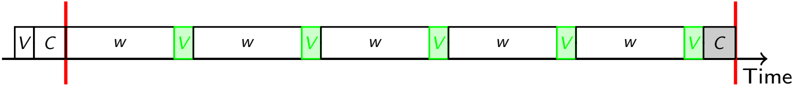


V small in front of w $\Rightarrow$ large $\text{WASTE}_{\text{fail}}$, can we improve that?

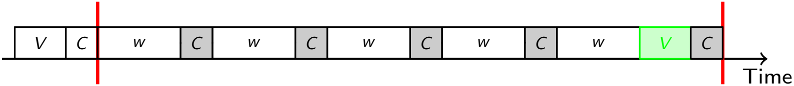
Is this better?



Coupling checkpointing and verification



Small cost V: 5 verifications for 1 checkpoint

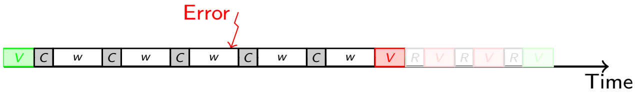


Large cost V: 5 checkpoints for 1 verification

More complicated periodic patterns? Different-size chunks?

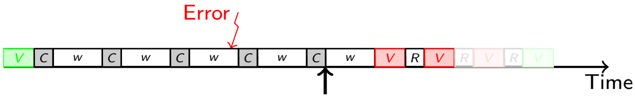
k checkpoints for 1 verification

Where did the error strike?



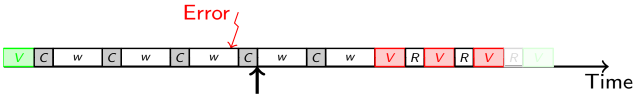
k checkpoints for 1 verification

Where did the error strike?



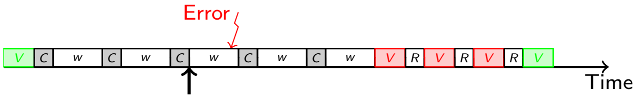
k checkpoints for 1 verification

Where did the error strike?



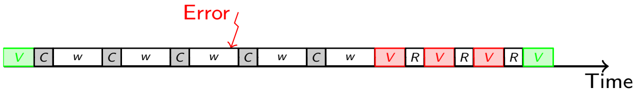
k checkpoints for 1 verification

Where did the error strike?



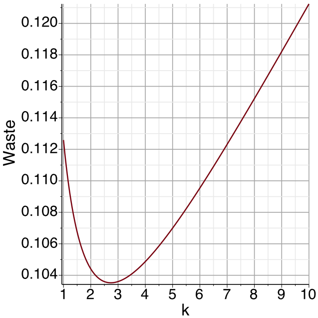
k checkpoints for 1 verification

Where did the error strike?



$$\text{RE-EXEC} = 2(w + C) + (w + V)$$

k checkpoints for 1 verification



Waste as function of k, using optimal period ($V = 100s$, $C = R = 6s$ and $\mu = \frac{10 years}{10^5}$)

Outline

Conclusion

- Multiple approaches to Fault Tolerance
- Application-Specific Fault Tolerance will always provide more benefits:
  - Checkpoint Size Reduction (when needed)
  - Portability (can run on different hardware, different deployment, etc..)
  - Diversity of use (can be used to restart the execution and change parameters in the middle)

Conclusion

- Multiple approaches to Fault Tolerance
- General Purpose Fault Tolerance is a required feature of the platforms
  - Not every computer scientist needs to learn how to write fault-tolerant applications
  - Not all parallel applications can be ported to a fault-tolerant version
- Faults are a feature of the platform. Why should it be the role of the programmers to handle them?

Conclusion

Application-Specific Fault Tolerance

- Fault Tolerance is introducing redundancy in the application
  - replication of computation
  - maintaining invariant in the data
- Requirements of a more Fault-friendly programming environment
  - MPI-Next evolution
  - Other programming environments?

Conclusion

General Purpose Fault Tolerance

- Software/hardware techniques to reduce checkpoint, recovery, migration times and to improve failure prediction
- Multi-criteria scheduling problem
  execution time/energy/reliability
  add replication
  best resource usage (performance trade-offs)
- Need combine all these approaches!

Several challenging algorithmic/scheduling problems ☺

Bibliography

Exascale
• Toward Exascale Resilience, Cappello F. et al., IJHPCA 23, 4 (2009)
• The International Exascale Software Roadmap, Dongarra, J., Beckman, P. et al., IJHPCA 25, 1 (2011)

ABFT Algorithm-based fault tolerance applied to high performance computing, Bosilca G. et al., JPDC 69, 4 (2009)

Coordinated Checkpointing Distributed snapshots: determining global states of distributed systems, Chandy K.M., Lamport L., ACM Trans. Comput. Syst. 3, 1 (1985)

Message Logging A survey of rollback-recovery protocols in message-passing systems, Elnozahy E.N. et al., ACM Comput. Surveys 34, 3 (2002)

Replication Evaluating the viability of process replication reliability for exascale systems, Ferreira K. et al, SC'2011

Models
• Checkpointing strategies for parallel jobs, Bougeret M. et al., SC'2011
• Unified model for assessing checkpointing protocols at extreme-scale, Bosilca G et al., INRIA RR-7950, 2012