Fault-tolerant Techniques for HPC: Theory and Practice

George Bosilca\textsuperscript{1}, Aurélien Bouteiller\textsuperscript{1}, Thomas Hérault\textsuperscript{1} & Yves Robert\textsuperscript{1,2}

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

\{bosilca,bouteiller,herault\}@icl.utk.edu | yves.robert@inria.fr
http://graal.ens-lyon.fr/~yrobert/sc14tutorial.pdf

SC’2014 Tutorial
Outline

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)
5. Hands-on: Designing a Resilient Application (90 mn)
6. Forward-recovery techniques (40mn)
7. Silent errors (35mn)
8. Conclusion (15mn)
Outline

1. **Introduction (15mn)**
   - Large-scale computing platforms
   - Faults and failures

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. Hands-on: Designing a Resilient Application (90 mn)

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)

8. Conclusion (15mn)
Outline

1. Introduction (15mn)
   - Large-scale computing platforms
   - Faults and failures

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. Hands-on: Designing a Resilient Application (90 mn)

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)

8. Conclusion (15mn)
Exascale platforms (courtesy Jack Dongarra)

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>System peak</td>
<td>10.5 Pflop/s</td>
<td>1 Eflop/s</td>
<td>O(100)</td>
</tr>
<tr>
<td>Power</td>
<td>12.7 MW</td>
<td>~20 MW</td>
<td></td>
</tr>
<tr>
<td>System memory</td>
<td>1.6 PB</td>
<td>32 - 64 PB</td>
<td>O(10)</td>
</tr>
<tr>
<td>Node performance</td>
<td>128 GF</td>
<td>1,2 or 15TF</td>
<td>O(10) – O(100)</td>
</tr>
<tr>
<td>Node memory BW</td>
<td>64 GB/s</td>
<td>2 - 4TB/s</td>
<td>O(100)</td>
</tr>
<tr>
<td>Node concurrency</td>
<td>8</td>
<td>O(1k) or 10k</td>
<td>O(100) – O(1000)</td>
</tr>
<tr>
<td>Total Node Interconnect BW</td>
<td>20 GB/s</td>
<td>200-400GB/s</td>
<td>O(10)</td>
</tr>
<tr>
<td>System size (nodes)</td>
<td>88,124</td>
<td>O(100,000) or O(1M)</td>
<td>O(10) – O(100)</td>
</tr>
<tr>
<td>Total concurrency</td>
<td>705,024</td>
<td>O(billion)</td>
<td>O(1,000)</td>
</tr>
<tr>
<td>MTTI</td>
<td>days</td>
<td>O(1 day)</td>
<td>- O(10)</td>
</tr>
</tbody>
</table>
## Toward Exascale Computing (My Roadmap)

*Based on proposed DOE roadmap with MTTI adjusted to scale linearly*

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

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

- **Hierarchical**
  - $10^5$ or $10^6$ nodes
  - Each node equipped with $10^4$ or $10^3$ cores

- **Failure-prone**

<table>
<thead>
<tr>
<th>MTBF – one node</th>
<th>1 year</th>
<th>10 years</th>
<th>120 years</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTBF – platform of $10^6$ nodes</td>
<td>30sec</td>
<td>5mn</td>
<td>1h</td>
</tr>
</tbody>
</table>

More nodes $\Rightarrow$ Shorter MTBF (Mean Time Between Failures)
Exascale platforms

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

- Failure-prone

<table>
<thead>
<tr>
<th>MTBF – one node</th>
<th>1 year</th>
<th>10^6 years</th>
<th>120 years</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTBF – platform of 10^6 nodes</td>
<td>30sec</td>
<td>5mn</td>
<td>1h</td>
</tr>
</tbody>
</table>

More nodes ≠ Petascale ×1000

{bosilca, bouteiller, herault}@icl.utk.edu | yves.robert@inria.fr
Even for today’s platforms (courtesy F. Cappello)

Fault tolerance becomes critical at Petascale (MTTI <= 1 day)
Poor fault tolerance design may lead to huge overhead

Today, 20% or more of the computing capacity in a large high-performance computing system is wasted due to failures and recoveries.

Dr. E.N. (Mootaz) Elnozahyet al. *System Resilience at Extreme Scale, DARPA*
Even for today’s platforms (courtesy F. Cappello)

Classic approach for FT: Checkpoint-Restart

Typical “Balanced Architecture” for PetaScale Computers

Compute nodes

Total memory: 100-200 TB

Network(s)

I/O nodes

40 to 200 GB/s

Parallel file system (1 to 2 PB)

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

<table>
<thead>
<tr>
<th>Systems</th>
<th>Perf.</th>
<th>Ckpt time</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoadRunner</td>
<td>1PF</td>
<td>~20 min.</td>
<td>Panasas</td>
</tr>
<tr>
<td>LLNL BG/L</td>
<td>500 TF</td>
<td>&gt;20 min.</td>
<td>LLNL</td>
</tr>
<tr>
<td>LLNL Zeus</td>
<td>11TF</td>
<td>26 min.</td>
<td>LLNL</td>
</tr>
<tr>
<td>YYY BG/P</td>
<td>100 TF</td>
<td>~30 min.</td>
<td>YYY</td>
</tr>
</tbody>
</table>
Optimistic Scenario

- Phase-Change memory
  - read bandwidth 100GB/sec
  - write bandwidth 10GB/sec
- Checkpoint size 128GB
- $C$: checkpoint save time: $C = 12$sec
- $R$: checkpoint recovery time: $R = 1.2$sec
- $D$: down/reboot time: $D = 15$sec
- $p$: 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

<table>
<thead>
<tr>
<th>$p$</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^8$</td>
<td>91.56%</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>73.75%</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>20.07%</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>2.51%</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>0.31%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$p$</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^8$</td>
<td>96.04%</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>88.23%</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>62.28%</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>10.66%</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>1.33%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$p$</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^8$</td>
<td>98.89%</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>98.80%</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>90.59%</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>70.46%</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>15.96%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$p$</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^8$</td>
<td>99.65%</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>99.00%</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>97.15%</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>91.63%</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>74.01%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$p$</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^8$</td>
<td>99.89%</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>99.69%</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>99.11%</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>97.45%</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>92.56%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$p$</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^8$</td>
<td>99.97%</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>99.90%</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>99.72%</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>99.20%</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>97.73%</td>
</tr>
</tbody>
</table>
Outline

1. Introduction (15mn)
   - Large-scale computing platforms
   - Faults and failures

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. Hands-on: Designing a Resilient Application (90 mn)

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)

8. Conclusion (15mn)
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):

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Raw</th>
<th>Filtered</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Count</td>
<td>%</td>
</tr>
<tr>
<td>Hardware</td>
<td>171,586,516</td>
<td>98.04</td>
</tr>
<tr>
<td>Software</td>
<td>144,899</td>
<td>0.08</td>
</tr>
<tr>
<td>Indeterminate</td>
<td>3,350,044</td>
<td>1.88</td>
</tr>
</tbody>
</table>

Relative frequency of root cause by system type.

Software errors: Applications, OS bug (kernel panic), communication libs, File system error and other.
Hardware errors, Disks, processors, memory, network

Conclusion: Both Hardware and Software failures have to be considered
A few definitions

- Many types of faults: software error, hardware malfunction, memory corruption
- Many possible behaviors: silent, transient, unrecoverable
- Restrict to faults that lead to application failures
- This includes all hardware faults, and some software ones
- Will use terms *fault* and *failure* interchangeably
- Silent errors (SDC) addressed later in the tutorial
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 $\text{Exp}(\lambda)$ failure inter-arrival times:

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

- 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

\[ X \text{ random variable for } Weibull(k, \lambda) \text{ failure inter-arrival times:} \]

- 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
  ⇒ approach **agnostic to granularity**

- If the MTBF is \( \mu \) with one processor, what is its value with \( p \) processors?

- Well, it depends 😊
Failure distributions: with several processors

- Processor (or node): any entity subject to failures
  ⇒ approach **agnostic to granularity**

- If the MTBF is $\mu$ 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 $\text{Exp}(\lambda)$:
  \[
  \min (\text{Exp}(\lambda_1), \text{Exp}(\lambda_2)) = \text{Exp}(\lambda_1 + \lambda_2)
  \]
  \[
  \mu = \frac{1}{\lambda} \Rightarrow \mu_p = \frac{\mu}{p}
  \]
- With $p$ distributions $\text{Weibull}(k, \lambda)$:
  \[
  \min_{1..p} (\text{Weibull}(k, \lambda)) = \text{Weibull}(k, p^{1/k} \lambda)
  \]
  \[
  \mu = \frac{1}{\lambda} \Gamma(1 + \frac{1}{k}) \Rightarrow \mu_p = \frac{\mu}{p^{1/k}}
  \]
Without rejuvenation (= real life)

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

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

With one processor:

- $n(F) = \text{number of failures until time } F \text{ is exceeded}$
- $X_i \text{ iid random variables for inter-arrival times, with } \mathbb{E}(X_i) = \mu$
- $\sum_{i=1}^{n(F)-1} X_i \leq F \leq \sum_{i=1}^{n(F)} X_i$
- Wald’s equation: $(\mathbb{E}(n(F)) - 1) \mu \leq F \leq \mathbb{E}(n(F)) \mu$
- $\lim_{F \to +\infty} \frac{\mathbb{E}(n(F))}{F} = \frac{1}{\mu}$
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
- \( \lim_{F \to +\infty} \frac{n(F)}{F} = \frac{p}{\mu} \) by definition
- Hence \( \mu_p = \frac{\mu}{p} \) because \( n(F) = \sum_{q=1}^{p} n_q(F) \)
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?

Parallel machine (10^6 nodes)

Failure Probability

Time (hours)

Exponential (1/100)
Weibull(0.7, 1/100)
Weibull(0.5, 1/100)

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr

Fault-tolerance for HPC
Outline

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)
   - Process Checkpointing
   - Coordinated Checkpointing
   - Hierarchical checkpointing

3. Checkpointing: Probabilistic models (45mn)

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. Hands-on: Designing a Resilient Application (90 mn)

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)

8. Conclusion (15mn)
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

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)
   - Process Checkpointing
   - Coordinated Checkpointing
   - Hierarchical checkpointing

3. Checkpointing: Probabilistic models (45mn)

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. Hands-on: Designing a Resilient Application (90 mn)

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)

8. Conclusion (15mn)
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 (or 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 (≈ 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


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

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
Outline

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)
   - Process Checkpointing
   - Coordinated Checkpointing
   - Hierarchical checkpointing

3. Checkpointing: Probabilistic models (45mn)

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. Hands-on: Designing a Resilient Application (90 mn)

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)

8. Conclusion (15mn)
Coordinated checkpointing

Definition (Missing Message)

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

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
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
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.
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

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

1. Introduction (15mn)

2. **Checkpointing: Protocols (30mn)**
   - Process Checkpointing
   - Coordinated Checkpointing
   - Hierarchical checkpointing

3. Checkpointing: Probabilistic models (45mn)

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. Hands-on: Designing a Resilient Application (90 mn)

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)

8. Conclusion (15mn)
Uncoordinated Checkpointing Idea

Processes checkpoint independently
Uncoordinated Checkpointing Idea

Optimistic Protocol

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

Uncoordinated Checkpointing Idea

Optimistic Protocol

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

Uncoordinated Checkpointing Idea

Optimistic Protocol

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

Uncoordinated Checkpointing Idea

Optimistic Protocol

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

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 = 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

- 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

\{bosilca,bouteiller,herault\}@icl.utk.edu | yves.robert@inria.fr

Fault-tolerance for HPC 45/211
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)

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
**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

- 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

Recovery

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr

Fault-tolerance for HPC 50/211
Overhead
Class C

First, when a non deterministic event is created, it has to
flag was used in order to generate a non-deterministic event
used a small ping-pong test with 2 processes. The
of measurements. On this faster network, the sender-based
only by less than 2%, which is close to the error margin
network (Open MPI=1).

NAS Kernel

mg.c.64  sp.c.64  cg.c.64

bt.c.64  ft.c.64  lu.c.64

Pessimist (Event Logging only)
Optimist (Event Logging only)
Optimist
Pessimist

Normalized Execution Time

Gflops/s

Problem Size (N)

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

Uncoordinated Protocol Performance

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

Many Core Systems

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

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

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
Hierarchical Protocols

- 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

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

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)
   - Young/Daly’s approximation
   - Coordinated checkpointing
   - Hierarchical checkpointing
   - In-memory checkpointing
   - Failure Prediction
   - Replication

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. Hands-on: Designing a Resilient Application (90 mn)

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)

8. Conclusion (15mn)
Outline

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
   - Young/Daly’s approximation
   - Coordinated checkpointing
   - Hierarchical checkpointing
   - In-memory checkpointing
   - Failure Prediction
   - Replication
4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)
5. Hands-on: Designing a Resilient Application (90 mn)
6. Forward-recovery techniques (40mn)
7. Silent errors (35mn)
8. Conclusion (15mn)
Checkpointing cost

Blockmodel: 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 $\mu = \mu_{\text{ind}}$
- Applies to a platform with $p$ processors with MTBF $\mu = \frac{\mu_{\text{ind}}}{p}$
  - coordinated checkpointing
  - tightly-coupled application
  - progress $\iff$ all processors available

Waste: fraction of time not spent for useful computations
Waste in fault-free execution

- **TIME\textsubscript{base}**: application base time
- **TIME\textsubscript{FF}**: with periodic checkpoints but failure-free

\[
\text{TIME}\textsubscript{FF} = \text{TIME}\textsubscript{base} + \#\text{checkpoints} \times C
\]

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

\[
\text{WASTE}[\text{FF}] = \frac{\text{TIME}\textsubscript{FF} - \text{TIME}\textsubscript{base}}{\text{TIME}\textsubscript{FF}} = \frac{C}{T}
\]
Waste due to failures

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

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

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

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

$T_{\text{lost}}$?
Waste due to failures

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

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

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

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

\( T_{\text{lost}} \)?
Computing $T_{\text{lost}}$

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

Rationale

⇒ Instants when periods begin and failures strike are independent
⇒ Approximation used for all distribution laws
⇒ Exact for Exponential and uniform distributions
Waste due to failures

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

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

\[
\text{TIME}_{FF} = \text{TIME}_{Final} \times (1 - \text{WASTE}[\text{Fail}]) \quad \text{TIME}_{Final} \times \text{WASTE}[\text{Fail}]
\]

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

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

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

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

\[ \text{WASTE} = \frac{u}{T} + v + wT \]

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

\[ \text{WASTE} \text{ minimized for } T = \sqrt{\frac{u}{w}} \]

\[ T = \sqrt{2(\mu - (D + R))C} \]
Comparison with Young/Daly

\[ (1 - \text{WASTE}[\text{fail}]) \cdot \text{TIME}_{\text{final}} = \text{TIME}_{\text{FF}} \]

\[ \Rightarrow T = \sqrt{2(\mu - (D + R))} \cdot C \]

Daly: \[ \text{TIME}_{\text{final}} = (1 + \text{WASTE}[\text{fail}]) \cdot \text{TIME}_{\text{FF}} \]

\[ \Rightarrow T = \sqrt{2(\mu + (D + R))} \cdot C + C \]

Young: \[ \text{TIME}_{\text{final}} = (1 + \text{WASTE}[\text{fail}]) \cdot \text{TIME}_{\text{FF}} \text{ and } D = R = 0 \]

\[ \Rightarrow T = \sqrt{2\mu} \cdot C + C \]
Validity of the approach (1/3)

Technicalities

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

2. Enforce \( C \leq T \) to get \( \text{WASTE}[FF] \leq 1 \)

3. Enforce \( D + R \leq \mu \) and bound \( T \) to get \( \text{WASTE}[\text{fail}] \leq 1 \)
   but \( \mu = \frac{\mu_{\text{ind}}}{p} \) too small for large \( p \), regardless of \( \mu_{\text{ind}} \)
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 $\gamma$ 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:
    \[ \pi = P(X \geq 2) = 1 - (P(X = 0) + P(X = 1)) = 1 - (1 + \theta)e^{-\theta} \]
  - $\gamma = 0.27 \Rightarrow \pi \leq 0.03$
    \[ \Rightarrow \text{overlapping faults for only 3\% of checkpointing segments} \]
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
  $\Rightarrow$ 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 😊
Lesson learnt for fail-stop failures

(Not so) Secret data

- Tsubame 2: 962 failures during last 18 months so \( \mu = 13 \text{ hrs} \)
- Blue Waters: 2-3 node failures per day
- Titan: a few failures per day
- Tianhe 2: wouldn’t say

\[
T_{opt} = \sqrt{2\mu C} \quad \Rightarrow \quad \text{WASTE}[opt] \approx \sqrt{\frac{2C}{\mu}}
\]

Petascale: \( C = 20 \text{ min} \quad \mu = 24 \text{ hrs} \quad \Rightarrow \quad \text{WASTE}[opt] = 17\%

Scale by 10: \( C = 20 \text{ min} \quad \mu = 2.4 \text{ hrs} \quad \Rightarrow \quad \text{WASTE}[opt] = 53\%

Scale by 100: \( C = 20 \text{ min} \quad \mu = 0.24 \text{ hrs} \quad \Rightarrow \quad \text{WASTE}[opt] = 100\% \)
Lesson learnt for fail-stop failures

(Not so) Secret data
- Tsubame 2: 962 failures during last 18 months so $\mu = 13$ hrs
- Blue Waters: 2-3 node failures per day
- Titan: a few failures per day
- Tianhe

Exascale $\neq$ Petascale $\times 1000$
Need more reliable components
Need to checkpoint faster

Petascale:
\[ C = 20 \text{ min} \quad \mu = 24 \text{ hrs} \quad \Rightarrow \quad \text{WASTE}[\text{opt}] = 17\% \]
Scale by 10:
\[ C = 20 \text{ min} \quad \mu = 2.4 \text{ hrs} \quad \Rightarrow \quad \text{WASTE}[\text{opt}] = 53\% \]
Scale by 100:
\[ C = 20 \text{ min} \quad \mu = 0.24 \text{ hrs} \quad \Rightarrow \quad \text{WASTE}[\text{opt}] = 100\% \]
Lesson learnt for fail-stop failures

(Not so) Secret data
- Tsubame 2: 962 failures during last 18 months so $\mu = 13$ hrs
- Blue Waters: 2-3 node failures per day
- Titan: a few failures per day
- Tianhe 2: wouldn’t say

Silent errors:
detection latency $\Rightarrow$ additional problems

Petascale: \[ C = 20 \text{ min} \quad \mu = 24 \text{ hrs} \quad \Rightarrow \quad \text{WASTE}[\text{opt}] = 17\% \]
Scale by 10: \[ C = 20 \text{ min} \quad \mu = 2.4 \text{ hrs} \quad \Rightarrow \quad \text{WASTE}[\text{opt}] = 53\% \]
Scale by 100: \[ C = 20 \text{ min} \quad \mu = 0.24 \text{ hrs} \quad \Rightarrow \quad \text{WASTE}[\text{opt}] = 100\% \]
Outline

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
   - Young/Daly’s approximation
   - Coordinated checkpointing
   - Hierarchical checkpointing
   - In-memory checkpointing
   - Failure Prediction
   - Replication
4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)
5. Hands-on: Designing a Resilient Application (90 mn)
6. Forward-recovery techniques (40mn)
7. Silent errors (35mn)
8. Conclusion (15mn)
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: hierarchical protocols

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

需

- Need to log inter-groups messages
  - Slowdowns failure-free execution
  - Increases checkpoint size/time
- Faster re-execution with logged messages

---

\{bosilca, bouteiller, herault\}@icl.utk.edu | yves.robert@inria.fr
Which checkpointing protocol to use?

**Coordinated checkpointing**
- 😊 No risk of cascading rollbacks
- 😊 No need to log messages
- 😞 All processors need to rollback
- 😞 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 rollback
- 😊 Faster re-execution with logged messages
- 😊 Rumor: Should scale to very large platforms
Coordinated checkpointing

![Diagram of 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 $\alpha C$ without checkpointing ($0 \leq \alpha \leq 1$)
Waste in fault-free execution

Time elapsed since last checkpoint: $T$

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

$\text{WASTE}[FF] = \frac{T - \text{WORK}}{T}$
Waste due to failures

Failure can happen

1. During computation phase
2. During checkpointing phase
Waste due to failures

- Time spent working
- Time spent checkpointing
- Time spent working with slowdown
Waste due to failures

- Time spent working
- Time spent checkpointing
- Time spent working with slowdown

- $P_0$
- $P_1$
- $P_2$
- $P_3$
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

- Time spent working
- Time spent checkpointing
- Time spent working with slowdown
- Downtime

$P_0$ $P_1$ $P_2$ $P_3$
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

\[
\begin{align*}
T - C
\end{align*}
\]
Waste due to failures in computation phase

Finally, the checkpointing phase is executed
Total waste

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

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

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
   - Young/Daly’s approximation
   - Coordinated checkpointing
   - Hierarchical checkpointing
   - In-memory checkpointing
   - Failure Prediction
   - Replication
4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)
5. Hands-on: Designing a Resilient Application (90 mn)
6. Forward-recovery techniques (40mn)
7. Silent errors (35mn)
8. Conclusion (15mn)
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:
  ⇒ \( \text{WORK} \) becomes \( \lambda \text{WORK} \), where \( 0 < \lambda < 1 \)
  Typical value: \( \lambda \approx 0.98 \)

- ☀️ Re-execution after a failure is faster:
  ⇒ \( \text{RE-EXEC} \) becomes \( \frac{\text{RE-EXEC}}{\rho} \), where \( \rho \in [1..2] \)
  Typical value: \( \rho \approx 1.5 \)

\[
\text{WASTE}[FF] = \frac{T - \lambda \text{WORK}}{T}
\]

\[
\text{WASTE}[\text{fail}] = \frac{1}{\mu} \left( D(q) + R(q) + \frac{\text{RE-EXEC}}{\rho} \right)
\]
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

\[
C(q) = C_0(q)(1 + \beta \text{WORK}) \iff \beta = \frac{C(q) - C_0(q)}{C_0(q) \text{WORK}}
\]

\[
\text{WORK} = \lambda(T - (1 - \alpha)G\text{C}(q))
\]

\[
C(q) = \frac{C_0(q)(1 + \beta \lambda T)}{1 + G\text{C}_0(q)\beta \lambda(1 - \alpha)}
\]
Three case studies

**Coord-IO**

Coordinated approach: \( C = C_{\text{Mem}} = \frac{\text{Mem}}{b_{\text{io}}} \)

where Mem is the memory footprint of the application

**Hierarch-IO**

Several (large) groups, *I/O-saturated*

\( \Rightarrow \) groups checkpoint sequentially

\[
C_0(q) = \frac{C_{\text{Mem}}}{G} = \frac{\text{Mem}}{G b_{\text{io}}}
\]

**Hierarch-Port**

Very large number of smaller groups, *port-saturated*

\( \Rightarrow \) some groups checkpoint in parallel

Groups of \( q_{\text{min}} \) processors, where \( q_{\text{min}} b_{\text{port}} \geq b_{\text{io}} \)
Three applications

1. 2D-stencil
2. Matrix product
3. 3D-Stencil
   - Plane
   - Line
Computing $\beta$ for 2D-Stencil

\[ C(q) = C_0(q) + \text{Logged}_\text{Msg} = C_0(q)(1 + \beta \text{Work}) \]

Real $n \times n$ matrix and $p \times p$ grid

\[ \text{Work} = \frac{9b^2}{sp}, \ b = n/p \]

Each process sends a block to its 4 neighbors

**Hierarch-IO:**
- 1 group = 1 grid row
- 2 out of the 4 messages are logged
- $\beta = \frac{\text{Logged}_\text{Msg}}{C_0(q)\text{Work}} = \frac{2pb}{pb^2(9b^2/s_p)} = \frac{2sp}{9b^3}$

**Hierarch-Port:**
- $\beta$ doubles
## Four platforms: basic characteristics

<table>
<thead>
<tr>
<th>Name</th>
<th>Number of cores</th>
<th>Number of processors $p_{total}$</th>
<th>Number of cores per processor</th>
<th>Memory</th>
<th>I/O Network Bandwidth (b$_{io}$)</th>
<th>Bandwidth (b$_{port}$)</th>
<th>I/O Bandwidth (b$_{io}$) Read/Write per processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Titan</td>
<td>299,008</td>
<td>16,688</td>
<td>16</td>
<td>32GB</td>
<td>300GB/s</td>
<td>300GB/s</td>
<td>20GB/s</td>
</tr>
<tr>
<td>K-Computer</td>
<td>705,024</td>
<td>88,128</td>
<td>8</td>
<td>16GB</td>
<td>150GB/s</td>
<td>96GB/s</td>
<td>20GB/s</td>
</tr>
<tr>
<td>Exascale-Slim</td>
<td>1,000,000,000</td>
<td>1,000,000</td>
<td>1,000</td>
<td>64GB</td>
<td>1TB/s</td>
<td>1TB/s</td>
<td>200GB/s</td>
</tr>
<tr>
<td>Exascale-Fat</td>
<td>1,000,000,000</td>
<td>100,000</td>
<td>10,000</td>
<td>640GB</td>
<td>1TB/s</td>
<td>1TB/s</td>
<td>400GB/s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Scenario</th>
<th>$G(C(q))$</th>
<th>$\beta$ for 2D-Stencil</th>
<th>$\beta$ for Matrix-Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Titan</td>
<td>COORD-IO</td>
<td>1 (2,048s)</td>
<td>0.00001098</td>
<td>0.0004280</td>
</tr>
<tr>
<td></td>
<td>HIERARCH-IO</td>
<td>136 (15s)</td>
<td>0.0002196</td>
<td>0.0008561</td>
</tr>
<tr>
<td></td>
<td>HIERARCH-PORT</td>
<td>1,246 (1.6s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-Computer</td>
<td>COORD-IO</td>
<td>1 (14,688s)</td>
<td>0.0002858</td>
<td>0.001113</td>
</tr>
<tr>
<td></td>
<td>HIERARCH-IO</td>
<td>296 (50s)</td>
<td>0.0005716</td>
<td>0.002227</td>
</tr>
<tr>
<td></td>
<td>HIERARCH-PORT</td>
<td>17,626 (0.83s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exascale-Slim</td>
<td>COORD-IO</td>
<td>1 (64,000s)</td>
<td>0.0002599</td>
<td>0.001013</td>
</tr>
<tr>
<td></td>
<td>HIERARCH-IO</td>
<td>1,000 (64s)</td>
<td>0.0005199</td>
<td>0.002026</td>
</tr>
<tr>
<td></td>
<td>HIERARCH-PORT</td>
<td>200,000 (0.32s)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exascale-Fat</td>
<td>COORD-IO</td>
<td>1 (64,000s)</td>
<td>0.00008220</td>
<td>0.003203</td>
</tr>
<tr>
<td></td>
<td>HIERARCH-IO</td>
<td>316 (217s)</td>
<td>0.00016440</td>
<td>0.0006407</td>
</tr>
<tr>
<td></td>
<td>HIERARCH-PORT</td>
<td>33,333 (1.92s)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Checkpoint time

<table>
<thead>
<tr>
<th>Name</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-Computer</td>
<td>14,688s</td>
</tr>
<tr>
<td>Exascale-Slim</td>
<td>64,000</td>
</tr>
<tr>
<td>Exascale-Fat</td>
<td>64,000</td>
</tr>
</tbody>
</table>

- 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 $\mu_{ind}$
Platform: K-Computer

Waste as a function of processor MTBF $\mu_{ind}$
WASTE = 1 for all scenarios!!!
WASTE = 1 for all scenarios!!!

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

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

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
Plotting formulas – Platform: Exascale with $C = 100$

Waste as a function of processor MTBF $\mu_{ind}$, $C = 100$
Simulations – Platform: Titan

Stencil 2D

- Coordinated
- Coordinated BestPer

Matrix product

- Hierarchical
- Hierarchical BestPer
- Hierarchical Port
- Hierarchical Port BestPer

Makespan (in days) as a function of processor MTBF $\mu_{ind}$
Simulations – Platform: Exascale with $C = 1,000$

**Stencil 2D**
- Coordinated
- Coordinated BestPer
- Hierarchical
- Hierarchical BestPer

**Matrix product**
- Hierarchical
- Hierarchical BestPer
- Hierarchical Port
- Hierarchical Port BestPer

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

---

{bosilca,bouteiller,herault}@icl.utk.edu  |  yves.robert@inria.fr

Fault-tolerance for HPC 93/211
Simulations – Platform: Exascale with $C = 100$

**Stencil 2D**
- Coordinated
- Coordinated BestPer
- Hierarchical
- Hierarchical BestPer

**Matrix product**
- Hierarchical
- Hierarchical BestPer
- Hierarchical Port
- Hierarchical Port BestPer

Makespan (in days) as a function of processor MTBF $\mu_{\text{ind}}, C = 100$
Outline

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)
   - Young/Daly’s approximation
   - Coordinated checkpointing
   - Hierarchical checkpointing
   - In-memory checkpointing
   - Failure Prediction
   - Replication

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. Hands-on: Designing a Resilient Application (90 mn)

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)

8. Conclusion (15mn)
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
After failure: downtime $D$ and recovery from buddy node

Two checkpoint files lost, must be re-sent to faulty processor
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?
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:** \( \text{WASTE}[FF] = \frac{C}{T} \)

   - Time spent working
   - Time spent checkpointing

   ![Diagram](chart1.png)

2. **Unpredicted faults:**

   \[
   \frac{1}{\mu NP} \left[ D + R + \frac{T}{2} \right]
   \]

   ![Diagram](chart2.png)

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

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr

Fault-tolerance for HPC 102/211
Computing the waste

Predictions: \( \frac{1}{\mu P} \left[ p(C + D + R) + (1 - p)C \right] \)

with actual fault (true positive)

no actual fault (false negative)

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

3 Predictions: \[ \frac{1}{\mu P} \left[ p(C + D + R) + (1 - p)C \right] \]

\begin{align*}
\text{Error} & \quad \text{Predicted failure} \\
\frac{C}{T-C} \quad \frac{C}{W_{\text{reg}}} \quad \frac{C_p}{T-W_{\text{reg}}-C} \quad \frac{C}{T-C} \quad \frac{C}{T-C} \\
\text{Time} & \\
\end{align*}

with actual fault (true positive)

\begin{align*}
\text{Predicted failure} \\
\frac{C}{T-C} \quad \frac{C}{W_{\text{reg}}} \quad \frac{C_p}{T-W_{\text{reg}}-C} \quad \frac{C}{T-C} \quad \frac{C}{T-C} \\
\text{Time} & \\
\end{align*}

no actual fault (false negative)

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

(Regular mode)

\[ T_{R-C} \]

(Regular mode)

\[ T_{R-C} \]

\[ T_{lost} \]

\[ T_{R-C} \]

Time

(Prediction without failure)

\[ T_{R-C} \]

\[ W_{reg} \]

Regular mode

Proactive mode

\[ T_{P-C_p} \]

\[ T_{P-C_p} \]

\[ T_{P-C_p} \]

\[ T_{R-C} \]

\[-W_{reg} \]

Time

(Prediction with failure)

\[ T_{R-C} \]

\[ W_{reg} \]

Regular mode

Proactive mode

\[ T_{P-C_p} \]

\[ T_{P-C_p} \]

\[ T_{P-C_p} \]

\[ T_{R-C} \]

\[-W_{reg} \]

Time

Error

\[ \text{Gets too complicated! 😞} \]

{boslca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
Outline

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)
   - Young/Daly’s approximation
   - Coordinated checkpointing
   - Hierarchical checkpointing
   - In-memory checkpointing
   - Failure Prediction
   - Replication

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. Hands-on: Designing a Resilient Application (90 mn)

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)

8. Conclusion (15mn)
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?

\[
Birthday(N) = 1 + \int_0^{+\infty} e^{-x} \left(1 + \frac{x}{N}\right)^{N-1} dx
\]

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

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
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

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
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

\{bosilca,bouteiller,herault\}@icl.utk.edu | yves.robert@inria.fr

Fault-tolerance for HPC 109/211
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

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
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

\[ \square \square \square \square \square \ldots \square \]

\[ 1 \ 2 \ 3 \ 4 \ \ldots \ n \]

\[ \uparrow \]

\[ \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \bullet \ldots \]

\[ N = n_{rg} \text{ bins, red and blue balls} \]

Mean Number of Failures to Interruption (bring down application)

\[ MNFTI = \text{expected number of balls to throw until one bin gets one ball of each color} \]
Failure distribution

(a) Exponential

(b) Weibull, $k = 0.7$

Crossover point for replication when $\mu_{ind} = 125$ years
Study by Ferreira et al. favors replication
Replication beneficial if small $\mu$ + large $C$ + big $p_{total}$
Outline

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)
   - Fault-Tolerant Middleware
   - Bags of tasks
5. Hands-on: Designing a Resilient Application (90 mn)
6. Forward-recovery techniques (40mn)
7. Silent errors (35mn)
8. Conclusion (15mn)
Fault Tolerance Software Stack

- Application
- Lib1
- Lib2
- Comm. Middleware (MPI)
- OS
- Network

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
Fault Tolerance Software Stack

- **Application**
  - Lib1
  - Lib2

- **Comm. Middleware (MPI)**
- **OS**
- **Network**

**Fault Tolerance**
- Application-Based
  - Permanent Crash Fault Tolerance
- Automatic Permanent Crash Fault Tolerance
- Network Transient Failures (inc. msg corruption) Fault Tolerance

**Permanent Crash Detection**

**Runtime Helpers**
Motivation

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

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)
   - Fault-Tolerant Middleware
   - Bags of tasks
5. Hands-on: Designing a Resilient Application (90 mn)
6. Forward-recovery techniques (40mn)
7. Silent errors (35mn)
8. Conclusion (15mn)
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

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](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.

  - 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?
- 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

⇒ In the past, these operations have often been merged
⇒ 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

- 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
Resilience Extensions for MPI: ULFM

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

Open MPI - ULFM support

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

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)
   • Fault-Tolerant Middleware
   • Bags of tasks

5. Hands-on: Designing a Resilient Application (90 mn)

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)

8. Conclusion (15mn)
Master/Worker

**Example: Master-worker**

```c
MPI_Irecv_init(comm, ANY_SOURCE, work_done)

While(more_work && workers) {
  submit_work(worker[i++ % workers])
  rc = MPI_Test(work_done)
  if(MPI_SUCCESS != RC)
    MPI_COMM_FAILURE_ACK(comm)
    MPI_COMM_FAILURE_GET_ACKED(comm, i)
    worker[i] = worker[workers--]
    resubmit_work(worker[i], i)
}
```

---

**Worker**

```c
while(1) {
  MPI_Recv(master, &work);
  if( work == STOP_CMD )
    break;
  process_work(work, &result);
  MPI_Send(master, result);
}
```
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);
}
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);
}
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);
    }
    <...>
    continue;
    break;
    default:
        /* Unknown error, not related to failure */
        MPI_Abort(MPI_COMM_WORLD);
    }
    <...>
Fault Tolerant Master

```c
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;
```
Fault Tolerant Master

```c
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();
```
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);
}
Hands-On

Material to support this part of the tutorial includes code skeletons.

It is available online:

http://fault-tolerance.org/sc14
Outline

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)
5. Hands-on: Designing a Resilient Application (90 mn)
   - The application (CG)
   - Using checkpoint and rollback recovery
   - In-memory checkpoint, spare-node & spawn
   - Lessons learned
6. Forward-recovery techniques (40mn)
7. Silent errors (35mn)
8. Conclusion (15mn)
Outline

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)
5. Hands-on: Designing a Resilient Application (90 mn)
   - The application (CG)
   - Using checkpoint and rollback recovery
   - In-memory checkpoint, spare-node & spawn
   - Lessons learned
6. Forward-recovery techniques (40mn)
7. Silent errors (35mn)
8. Conclusion (15mn)
Hands-On

Material to support this part of the tutorial includes code skeletons.

It is available online:
http://fault-tolerance.org/sc14
Outline

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. **Hands-on: Designing a Resilient Application (90 mn)**
   - The application (CG)
   - Using checkpoint and rollback recovery
   - In-memory checkpoint, spare-node & spawn
   - Lessons learned

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)

8. Conclusion (15mn)
Hands-On

Material to support this part of the tutorial includes code skeletons.

It is available online:
http://fault-tolerance.org/sc14
Outline

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. Hands-on: Designing a Resilient Application (90 mn)
   - The application (CG)
   - Using checkpoint and rollback recovery
   - In-memory checkpoint, spare-node & spawn
   - Lessons learned

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)

8. Conclusion (15mn)
Hands-On

Material to support this part of the tutorial includes code skeletons.

It is available online:

http://fault-tolerance.org/sc14
Outline

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. Hands-on: Designing a Resilient Application (90 mn)
   - The application (CG)
   - Using checkpoint and rollback recovery
   - In-memory checkpoint, spare-node & spawn
   - Lessons learned

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)

8. Conclusion (15mn)
Hands-On

Material to support this part of the tutorial includes code skeletons.

It is available online:

http://fault-tolerance.org/sc14
Outline

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)
5. Hands-on: Designing a Resilient Application (90 mn)
6. Forward-recovery techniques (40mn)
   - ABFT for Linear Algebra applications
   - Composite approach: ABFT & Checkpointing
7. Silent errors (35mn)
8. Conclusion (15mn)
Backward Recovery

- Rollback / Backward Recovery: returns in the history to recover from failures.
- Spends time to re-execute computations
- Rebuilds states already reached
- Typical: checkpointing techniques
Forward Recovery

- Forward Recovery: proceeds without returning.
- Pays additional costs during (failure-free) computation to maintain consistent redundancy
- Or pays additional computations when failures happen
- General technique: Replication
- Application-Specific techniques: Iterative algorithms with fixed point convergence, ABFT, ...
Outline

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)
5. Hands-on: Designing a Resilient Application (90 mn)
6. Forward-recovery techniques (40mn)
   - ABFT for Linear Algebra applications
   - Composite approach: ABFT & Checkpointing
7. Silent errors (35mn)
8. Conclusion (15mn)
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

GETF2: factorize a column block
TRSM - Update row block
GEMM: Update the trailing matrix

- 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

TRSM - Update row block

GETF2: factorize a column block

GEMM: Update the trailing matrix

- 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 $2 \times 3$)
- A single failure $\Rightarrow$ many data lost
Algorithm Based Fault Tolerant QR 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 QR 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 QR decomposition

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

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
Algorithm Based Fault Tolerant QR decomposition

- Idea of ABFT: applying the operation on data and checksum preserves the checksum properties
Algorithm Based Fault Tolerant QR decomposition

For the part of the data that is not updated this way, the checksum must be re-calculated.
Algorithm Based Fault Tolerant QR decomposition

To avoid slowing down all processors and panel operation, group checksum updates every $Q$ block columns.
Algorithm Based Fault Tolerant QR decomposition

To avoid slowing down all processors and panel operation, group checksum updates every $Q$ block columns.
Algorithm Based Fault Tolerant QR decomposition

To avoid slowing down all processors and panel operation, group checksum updates every $Q$ block columns.
Then, update the missing coverage. Keep checkpoint block column to cover failures during that time.
In case of failure, conclude the operation, then
• Missing Data = Checksum - Sum(Existing Data)
Algorithm Based Fault Tolerant QR decomposition

In case of failure, conclude the operation, then

- Missing Checksum = Sum(Existing Data)
Failures may happen while inside a $Q$–panel factorization.
Valid Checksum Information allows to recover most of the missing data, but not all: the checksum for the current
We use the checkpoint to restore the $Q$–panel in its initial state.
Algorithm Based Fault Tolerant QR decomposition

and re-execute that part of the factorization, without applying outside of the scope.
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 QR decomposition: performance

Figure 11 shows the overhead from these two cases for the LU factorization, along with the no-error overhead as a reference. In the "border" case, the failure is simulated to strike when the $96$th panel (which is a multiple of grid columns, $6, 12, \cdots, 48$) has just finished. In the "non-border" case, failure occurs during the $(Q+2)$th panel factorization. For example, when $Q=12$, the failure is injected when the trailing update for the step with panel $(1301,1301)$ finishes. From the result in Figure 10, the recovery procedure in both cases adds a small overhead that also decreases when scaled to large problem size and process grid. For largest setups, only 2-3 percent of the execution time is spent recovering from a failure.

7.4. Extension to Other factorization

The algorithm proposed in this work can be applied to a wide range of dense matrix factorizations other than LU. As a demonstration we have extended the fault tolerance functions to the ScaLAPACK QR factorization in double precision. Since QR uses a block algorithm similar to LU (and also similar to Cholesky), the integration of fault tolerance functions is mostly straightforward. Figure 11 shows the performance of QR with and without recovery. The overhead drops as the problem and grid size increase, although it remains higher than that of LU for the same problem size. This is expected: as the QR algorithm has a higher complexity than LU ($4 \times 3 \times N^3$ vs $2 \times 3 \times N^3$), the ABFT approach incurs more extra computation when updating checksums. Similar to the LU result, recovery adds an extra 2% overhead. At size 160,000 a failure incurs about 5.7% penalty to be recovered. This overhead becomes lower, the larger the problem or processor grid size considered.
ABFT LU decomposition: performance

As supercomputers grow ever larger in scale, the Mean Time to Failure becomes shorter and shorter, making the complete and successful execution of complex applications more and more difficult.

FT-LA delivers a new approach, utilizing Algorithm-Based Fault Tolerance (ABFT), to help factorization algorithms survive fail-stop failures.

The FT-LA software package extends ScaLAPACK with ABFT routines, and in sharp contrast with legacy checkpoint-based approaches, ABFT does not incur I/O overhead, and promises a much more scalable protection scheme.

**ABFT**

**THE IDEA**

Cost of ABFT comes only from extra flops (to update checksums) and extra storage. Cost decreases with machine scale (divided by $PQ$ when using $P \times Q$ processes).

**PROTECTION**

Matrix protected by block row checksum. The algorithm updates both the trailing matrix AND the checksums.

**RECOVERY**

Missing blocks reconstructed by inverting the checksum operation.

**FUNCTIONALITY**

**COVERAGE**

Linear Systems of Equations, Least Squares, Cholesky, LU, QR (with protection of the upper and lower factors).

**FEATURES**

- Work in progress
- Covering four precisions: double complex, single complex, double real, single real (ZCDS)
- Deploys on MPI FT draft (ULFM), or with “Checkpoint-on-failure”
- Allows toleration of permanent crashes
- Hessenber Reduction, Soft (silent) Errors

**MPI-Next ULFM Performance**

- Open MPI with ULFM; Kraken supercomputer;

**FIND OUT MORE AT**

http://icl.cs.utk.edu/ft-la

**Performance on Kraken**

<table>
<thead>
<tr>
<th>Processors (P x Q grid); Matrix size (N)</th>
<th>Overhead: FT-PDGETRF (w/1 recovery)</th>
<th>Overhead: FT-PDGETRF (no error)</th>
<th>Performance (TFlop/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6x6; 20k</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>12x12; 40k</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>24x24; 80k</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>48x48; 160k</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>96x96; 320k</td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>192x192; 640k</td>
<td></td>
<td></td>
<td>28</td>
</tr>
</tbody>
</table>

**Flops for the checksum update**

Matrix is extended with $2F$ columns every $Q$ columns.

**Overheads in**

$F / Q$.

Protection cost is inversely proportional to machine scale!
Checkpoint on Failure - MPI Implementation

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

boslca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
ABFT QR decomposition: performance

Figure 2. ABFT QR and one CoF recovery on Kraken (Lustre).

Figure 3. ABFT QR and one CoF recovery on Dancer (local SSD).

Figure 4. Time breakdown of one CoF recovery on Dancer (local SSD).

5.3 Checkpoint-on-Failure QR Performance

Supercomputer Performance:

Figure 2 presents the performance on the Kraken supercomputer. The process grid is 24 × 24 and the block size is 100. The CoF-QR (no failure) presents the performance of the CoF QR implementation, in a fault-free execution; it is noteworthy, that when there are no failures, the performance is exactly identical to the performance of the unmodified FT-QR implementation. The CoF-QR (with failure) curves present the performance when a failure is injected after the first step of the PDLARFB kernel. The performance of the non-fault tolerant ScaLAPACK QR is also presented for reference.

Without failures, the performance overhead compared to the regular ScaLAPACK is caused by the extra computation to maintain the checksums inherent to the ABFT algorithm [12]; this extra computation is unchanged between CoF-QR and FT-QR. Only on runs where a failure happened do the CoF protocols undergo the supplementary overhead of storing and reloading checkpoints. However, the performance of the CoF-QR remains very close to the no-failure case. For instance, at matrix size N=100,000, CoF-QR still achieves 2.86 Tflop/s after recovering from a failure, which is 90% of the performance of the non-fault tolerant ScaLAPACK QR. This demonstrates that the CoF protocol enables efficient, practical recovery schemes on supercomputers.

Impact of Local Checkpoint Storage:

Figure 3 presents the performance of the CoF-QR implementation on the Dancer cluster with a 8 × 16 process grid. Although a smaller test platform, the Dancer cluster features local storage on nodes and a variety of performance analysis tools unavailable on Kraken. As expected (see [12]), the ABFT method has a higher relative cost on this smaller machine. Compared to the Kraken platform, the relative cost of CoF failure recovery is smaller on Dancer. The CoF protocol incurs disk accesses to store and load checkpoints when a failure hits, hence the recovery overhead depends on I/O performance. By breaking down the relative cost of each recovery step in CoF, Figure 4 shows that checkpoint saving and loading only take a small percentage of the total run-time, thanks to the availability of solid state disks on every node. Since checkpoint reloading immediately follows checkpointing, the OS cache satisfy most disk access, resulting in high I/O performance. For matrices larger than...
Outline

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)
5. Hands-on: Designing a Resilient Application (90 mn)
6. Forward-recovery techniques (40mn)
   - ABFT for Linear Algebra applications
   - Composite approach: ABFT & Checkpointing
7. Silent errors (35mn)
8. Conclusion (15mn)
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**

```c
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
Typical Application

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

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

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

Googbye ABFT?!

- 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

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
Problem Statement

**Typical Application**

```c
for (a = 1; a < n; ++a)
  /* Extract data from simulation, fill up matrix */
  sim2mat();
  /* Factorize matrix, solve */
  dgeqrfs();
  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

**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...

**Notes:**

- \{bosilca,bouteiller,herault\}@icl.utk.edu
- yves.robert@inria.fr

---

Fault-tolerance for HPC 157/211
ABFT & Periodic Checkpoint: no failure

Process 0

Application
Library

Periodic Checkpoint

Process 1

Application
Library

Process 2

Application
Library

Split
Forced Checkpoints
ABFT & Periodic Ckpt: failure during Library phase

Process 0
Application
Library

Process 1
Application
Library

Process 2
Application
Library

Failure (during LIBRARY)
Rollback (partial)
Recovery
ABFT Recovery
ABFT & Periodic Checkpoint: failure during General phase

Process 0
Application
Library

Process 1
Application
Library

Process 2
Application
Library

Failure (during General)
Rollback (full)
Recovery

{bosilca, bouteiller, herault}@icl.utk.edu | yves.robert@inria.fr

Fault-tolerance for HPC
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

- 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
- $T_{ff}, T_{ff}^G, T_{ff}^L$: “Fault Free” times
- $t_{lost}^G, t_{lost}^L$: Lost time (recovery overhreads)
- $T_{final}^G, T_{final}^L$: Total times (with faults)
A few notations

Costs

\[ C_L = \rho C : \text{time to take a checkpoint of the } \textsc{Library} \text{ data set} \]
\[ C_L^\ominus = (1 - \rho)C : \text{time to take a checkpoint of the } \textsc{General} \text{ data set} \]
\[ R, R_L^\ominus : \text{time to load a full } / \text{ General data set checkpoint} \]
\[ D : \text{down time (time to allocate a new machine } / \text{ reboot)} \]
\[ \text{Recons}_{ABFT} : \text{time to apply the ABFT recovery} \]
\[ \phi : \text{Slowdown factor on the } \textsc{Library} \text{ phase, when applying ABFT} \]
GENERAL phase, fault free waste

GENERAL phase

Without Failures

\[ T_{G}^{ff} = \begin{cases} \frac{T_{G} + C_{L}}{P_{G} - C} \times P_{G} & \text{if } T_{G} < P_{G} \\ \frac{T_{G}}{P_{G} - C} \times P_{G} & \text{if } T_{G} \geq P_{G} \end{cases} \]
Library phase, fault free waste

Without Failures

\[ T_L^{ff} = \phi \times T_L + C_L \]
**GENERAL phase, failure overhead**

**GENERAL phase**

Process 0

Process 1

Process 2

Application

Library

Failure (during GENERAL)

Rollback (full)

Recovery

**Failure Overhead**

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

{bositca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
**Library** phase, failure overhead

**Library phase**

Process 0

- Application
- Library

Process 1

- Application
- Library

Process 2

- Application
- Library

Failure (during Library)

Rollback (partial)

Recovery

ABFT Recovery

**Failure Overhead**

\[ t_L^{\text{lost}} = D + R_L + \text{Recons}_{\text{ABFT}} \]
Time (with overheads) of **LIBRARY** phase is constant (in $P_G$):

$$T_{L_{final}} = \frac{1}{1 - \frac{D + R + \text{Recons}_{ABFT}}{\mu}} \times (\alpha \times T_L + C_L)$$

Time (with overheads) of **GENERAL** phase accepts two cases:

$$T_{G_{final}} = \begin{cases} \frac{1}{1 - \frac{T_G + C_L}{2}} \times (T_G + C_L) & \text{if } T_G < P_G \\ \frac{\mu}{T_G} & \text{if } T_G \geq P_G \end{cases}$$

Which is minimal in the second case, if

$$P_G = \sqrt{2C(\mu - D - R)}$$
From the previous, we derive the waste, which is obtained by

$$\text{WASTE} = 1 - \frac{T_0}{T_G^{\text{final}} + T_L^{\text{final}}}$$
Let's think at scale

- Number of components $\uparrow \Rightarrow$ MTBF $\downarrow$
- Number of components $\uparrow \Rightarrow$ Problem Size $\uparrow$
- Problem Size $\uparrow \Rightarrow$
  Computation Time spent in Library phase $\uparrow$

😊 ABFT & PeriodicCkpt should perform better with scale
❓ By how much?

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
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 Scenario #1

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

- $\mu$ at $n = 10^5$: 1 day, is in $O\left(\frac{1}{n}\right)$
- $C$ (= $R$) at $n = 10^5$, is 1 minute, is in $O(n)$
- $\alpha$ is constant at 0.8, as is $\rho$.
  (both Library and General phase increase in time at the same speed)
Weak Scale #1

![Graph showing fault tolerance metrics for different checkpointing protocols.]
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)

- $\mu$ at $n = 10^5$: 1 day, is $O(\frac{1}{n})$
- $C (= R)$ at $n = 10^5$, is 1 minute, is in $O(n)$
- $\rho$ remains constant at 0.8, but **Library** phase is $O(n^3)$ when **General** phases progresses in $O(n^2)$ ($\alpha$ is 0.8 at $n = 10^5$ nodes).
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)

- $\mu$ at $n = 10^5$: 1 day, is $O(\frac{1}{n})$
- $C (=R)$ at $n = 10^5$, is 1 minute, stays independent of $n$ ($O(1)$)
- $\rho$ remains constant at 0.8, but Library phase is $O(n^3)$ when General phases progresses in $O(n^2)$ ($\alpha$ is 0.8 at $n = 10^5$ nodes).
Weak Scale #3

![Graph showing fault tolerance metrics](image-url)

- **Number of Faults**
  - Periodic Checkpoint (Red)
  - Bi-Periodic Checkpoint (Green)
  - ABFT Periodic Checkpoint (Blue)

- **Waste Nodes**
  - Periodic Checkpoint (Red)
  - Bi-Periodic Checkpoint (Green)
  - ABFT Periodic Checkpoint (Blue)

- Parameters:
  - $\alpha = 0.55$
  - $\alpha = 0.8$
  - $\alpha = 0.92$
  - $\alpha = 0.975$

- Key:
  - (bosilca, bouteiller, herault)@icl.utk.edu
  - yves.robert@inria.fr

Fault-tolerance for HPC
Outline

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. Hands-on: Designing a Resilient Application (90 mn)

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)
   - Coupling checkpointing and verification
   - Application-specific methods

8. Conclusion (15mn)
Definitions

- Instantaneous error detection $\Rightarrow$ fail-stop failures, e.g. resource crash
- Silent errors (data corruption) $\Rightarrow$ detection latency

**Silent error detected only when the corrupt data is activated**

- Includes some software faults, some hardware errors (soft errors in L1 cache), double bit flip
- Cannot always be corrected by ECC memory
Soft Error: An unintended change in the state of an electronic device that alters the information that it stores without destroying its functionality, e.g. a bit flip caused by a cosmic-ray-induced neutron. (Hengartner et al., 2008)

SDC occurs when incorrect data is delivered by a computing system to the user without any error being logged (Cristian Constantinescu, AMD)

Silent errors are the black swan of errors (Marc Snir)
Should we be afraid? (courtesy Al Geist)

Fear of the Unknown

**Hard errors** – permanent component failure either HW or SW (hung or crash)

**Transient errors** – a blip or short term failure of either HW or SW

**Silent errors** – undetected errors either hard or soft, due to lack of detectors for a component or inability to detect (transient effect too short). Real danger is that answer may be incorrect but the user wouldn’t know.

**Statistically, silent error rates are increasing. Are they really? Its fear of the unknown**

Are silent errors really a problem or just monsters under our bed?
Probability distributions for silent errors

Theorem: $\mu_p = \frac{\mu_{\text{ind}}}{p}$ for arbitrary distributions
Probability distributions for silent errors

\[ \mu_p = \frac{\mu_{\text{ind}}}{p} \] for arbitrary distributions
Outline

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)
5. Hands-on: Designing a Resilient Application (90 mn)
6. Forward-recovery techniques (40mn)
7. Silent errors (35mn)
   - Coupling checkpointing and verification
   - Application-specific methods
8. Conclusion (15mn)
Error and detection latency

- Last checkpoint may have saved an already corrupted state
- Saving $k$ checkpoints (Lu, Zheng and Chien):
  1. Critical failure when all live checkpoints are invalid
  2. Which checkpoint to roll back to?
General-purpose approach

- Error and detection latency
  - Last checkpoint may have saved an already corrupted state
  - Saving $k$ checkpoints (Lu, Zheng and Chien):
    1. Critical failure when all live checkpoints are invalid
       - Assume unlimited storage resources
    2. Which checkpoint to roll back to?
       - Assume verification mechanism

Emails: {bosilca, bouteiller, herault}@icl.utk.edu | yves.robert@inria.fr
Optimal period?

- Error
- Detection

Error and detection latency

- $X_e$: inter arrival time between errors; mean time $\mu_e$
- $X_d$: error detection time; mean time $\mu_d$
- Assume $X_d$ and $X_e$ independent
Arbitrary distribution

\[
\text{WASTE}_{\text{ff}} = \frac{C}{T}
\]

\[
\text{WASTE}_{\text{fail}} = \frac{T}{2} + R + \frac{\mu_d}{\mu_e}
\]

Only valid if \( \frac{T}{2} + R + \mu_d \ll \mu_e \)

**Theorem**

- Best period is \( T_{\text{opt}} \approx \sqrt{2\mu_e C} \)
- Independent of \( X_d \)
Exponential distribution

**Theorem**

- **At the end of the day,**
  \[ \mathbb{E}(T(w)) = e^{\lambda R} (\mu_e + \mu_d) (e^{\lambda (w+C)} - 1) \]
- **Optimal period independent of \( \mu_d \)**
- **Good approximation is** \( T = \sqrt{2\mu_e C} \) (Young’s formula)
The case with limited resources

Assume that we can only save the last $k$ checkpoints

**Definition (Critical failure)**

Error detected when all checkpoints contain corrupted data. Happens with probability $P_{\text{risk}}$ during whole execution.

$P_{\text{risk}}$ decreases when $T$ increases (when $X_d$ is fixed). Hence, $P_{\text{risk}} \leq \varepsilon$ leads to a lower bound $T_{\text{min}}$ on $T$

Can derive an analytical form for $P_{\text{risk}}$ when $X_d$ follows an Exponential law. Use it as a good(?) approximation for arbitrary laws.
Limitation of the model

It is not clear how to detect when the error has occurred (hence to identify the last valid checkpoint) 😞 😞 😞

Need a verification mechanism to check the correctness of the checkpoints. This has an additional cost!
Verification mechanism of cost $V$
Silent errors detected only when verification is executed
Approach agnostic of the nature of verification mechanism (checksum, error correcting code, coherence tests, etc)
Fully general-purpose (application-specific information, if available, can always be used to decrease $V$)
On-line ABFT scheme for PCG

Zizhong Chen, PPoPP’13

- Iterate PCG
  - **Cost:** SpMV, preconditioner solve, 5 linear kernels
- Detect soft errors by checking orthogonality and residual
- Verification every \( d \) iterations
  - **Cost:** scalar product + SpMV
- Checkpoint every \( c \) iterations
  - **Cost:** three vectors, or two vectors + SpMV at recovery
- Experimental method to choose \( c \) and \( d \)
Base pattern (and revisiting Young/Daly)

<table>
<thead>
<tr>
<th></th>
<th>Fail-stop (classical)</th>
<th>Silent errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern</td>
<td>$T = W + C$</td>
<td>$S = W + V + C$</td>
</tr>
<tr>
<td>WASTE[$FF$]</td>
<td>$\frac{C}{T}$</td>
<td>$\frac{V+C}{S}$</td>
</tr>
<tr>
<td>WASTE[$fail$]</td>
<td>$\frac{1}{\mu}(D + R + \frac{W}{2})$</td>
<td>$\frac{1}{\mu}(R + W + V)$</td>
</tr>
<tr>
<td>Optimal</td>
<td>$T_{opt} = \sqrt{2C\mu}$</td>
<td>$S_{opt} = \sqrt{(C + V)\mu}$</td>
</tr>
<tr>
<td>WASTE[$opt$]</td>
<td>$\sqrt{\frac{2C}{\mu}}$</td>
<td>$2\sqrt{\frac{C+V}{\mu}}$</td>
</tr>
</tbody>
</table>
With \( p = 1 \) checkpoint and \( q = 3 \) verifications

Base Pattern  \( p = 1, q = 1 \)  \( \text{WASTE}[\text{opt}] = 2\sqrt{\frac{C+V}{\mu}} \)

New Pattern  \( p = 1, q = 3 \)  \( \text{WASTE}[\text{opt}] = 2\sqrt{\frac{4(C+3V)}{6\mu}} \)
Balanced Algorithm

- \( p \) checkpoints and \( q \) verifications, \( p \leq q \)
- \( p = 2, \ q = 5, \ S = 2C + 5V + W \)
- \( W = 10w \), six chunks of size \(w\) or \(2w\)
- May store invalid checkpoint (error during third chunk)
- After successful verification in fourth chunk, preceding checkpoint is valid
- Keep only two checkpoints in memory and avoid any fatal failure
Balanced Algorithm

\[ T_{\text{lost}}^1 = R + 2w + V \]
\[ T_{\text{lost}}^2 = R + 4w + 2V \]
\[ T_{\text{lost}}^3 = 2R + 6w + C + 4V \]
\[ T_{\text{lost}}^4 = R + w + 2V \]
\[ T_{\text{lost}}^5 = R + 3w + 2V \]
\[ T_{\text{lost}}^6 = R + 5w + 3V \]

\[
\text{WASTE}[\text{opt}] \approx 2\sqrt{\frac{7(2C + 5V)}{20\mu}}
\]
Key parameters

\( o_{\text{ff}} \) failure-free overhead per pattern

\( f_{\text{re}} \) fraction of work that is re-executed

- \( \text{WASTE}_{\text{ff}} = \frac{o_{\text{ff}}}{S} \), where \( o_{\text{ff}} = pC + qV \) and \( S = o_{\text{ff}} + pqw \ll \mu \)
- \( \text{WASTE}_{\text{fail}} = \frac{T_{\text{lost}}}{\mu} \), where \( T_{\text{lost}} = f_{\text{re}}S + \beta \)
  \( \beta \): constant, linear combination of \( C \), \( V \) and \( R \)
- \( \text{WASTE} \approx \frac{o_{\text{ff}}}{S} + \frac{f_{\text{re}}S}{\mu} \Rightarrow S_{\text{opt}} \approx \sqrt{\frac{o_{\text{ff}}}{f_{\text{re}}} \cdot \mu} \)

\[ \text{WASTE}[\text{opt}] = 2\sqrt{\frac{o_{\text{ff}}f_{\text{re}}}{\mu}} + o\left(\sqrt{\frac{1}{\mu}}\right) \]
The minimal value of $f_{re}(1, q)$ is obtained for same-size chunks

- $f_{re}(1, q) = \sum_{i=1}^{q} \left( \alpha_i \sum_{j=1}^{i} \alpha_j \right)$
- Minimal when $\alpha_i = 1/q$
- In that case, $f_{re}(1, q) = \frac{q+1}{2q}$
Computing $f_{re}$ when $p \geq 1$

Theorem

$$f_{re}(p, q) \geq \frac{p+q}{2pq}, \text{ bound is matched by } \text{BalancedAlgorithm.}$$

- Assess gain due to the $p - 1$ intermediate checkpoints
  $$f_{re}^{(1)} - f_{re}^{(p)} = \sum_{i=1}^{p} \left( \alpha_i \sum_{j=1}^{i-1} \alpha_j \right)$$
- Maximal when $\alpha_i = 1/p$ for all $i$
- In that case, $f_{re}^{(1)} - f_{re}^{(p)} = (p - 1)/p^2$
- Now best with equipartition of verifications too
  $$\text{In that case, } f_{re}^{(1)} = \frac{q+1}{2q} \text{ and } f_{re}^{(p)} = \frac{q+1}{2q} - \frac{p-1}{2p} = \frac{q+p}{2pq}$$
Choosing optimal pattern

- Let $V = \gamma C$, where $0 < \gamma \leq 1$
- $o_{ff}f_{re} = \frac{p+q}{2pq}(pC + qV) = C \times \frac{p+q}{2} \left(\frac{1}{q} + \frac{\gamma}{p}\right)$
- Given $\gamma$, minimize $\frac{p+q}{2} \left(\frac{1}{q} + \frac{\gamma}{p}\right)$ with $1 \leq p \leq q$, and $p, q$ taking integer values
- Let $p = \lambda \times q$. Then $\lambda_{opt} = \sqrt{\gamma} = \sqrt{\frac{V}{C}}$
**BalancedAlgorithm** optimal when $C, R, V \ll \mu$

- Keep only 2 checkpoints in memory/storage
- Closed-form formula for $WASTE[\text{opt}]$
- Given $C$ and $V$, choose optimal pattern
- Gain of up to 20% over base pattern
Outline

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)

4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)

5. Hands-on: Designing a Resilient Application (90 mn)

6. Forward-recovery techniques (40mn)

7. Silent errors (35mn)
   - Coupling checkpointing and verification
   - Application-specific methods

8. Conclusion (15mn)
- ABFT: dense matrices / fail-stop, extended to sparse / silent. Limited to one error detection and/or correction in practice.
- Asynchronous (chaotic) iterative methods (old work).
- Partial differential equations: use lower-order scheme as verification mechanism (detection only, Benson, Schmit and Schreiber).
- FT-GMRES: inner-outer iterations (Hoemmen and Heroux).
- PCG: orthogonalization check every $k$ iterations, re-orthogonalization if problem detected (Sao and Vuduc).
- ... Many others.
Dynamic programming for linear chains of tasks

- \{T_1, T_2, \ldots, T_n\} : linear chain of \(n\) tasks
- Each task \(T_i\) fully parametrized:
  - \(w_i\) computational weight
  - \(C_i, R_i, V_i\) : checkpoint, recovery, verification
- Error rates:
  - \(\lambda^F\) rate of fail-stop errors
  - \(\lambda^S\) rate of silent errors
**VC-only**

\[
\begin{align*}
\text{min}_{0 \leq k < n} & \quad \text{Time}^\text{rec}_C(n, k) \\
\text{Time}^\text{rec}_C(j, k) &= \min_{k \leq i < j} \{ \text{Time}^\text{rec}_C(i, k - 1) + T^\text{SF}_C(i + 1, j) \} \\
T^\text{SF}_C(i, j) &= p^F_{i,j} \left( T^{\text{lost}}_{i,j} + R_{i-1} + T^\text{SF}_C(i, j) \right) \\
&\quad + \left( 1 - p^F_{i,j} \right) \left( \sum_{\ell=i}^{j} w_\ell + V_j + p^S_{i,j} \left( R_{i-1} + T^\text{SF}_C(i, j) \right) + \left( 1 - p^S_{i,j} \right) C_j \right)
\end{align*}
\]
\[ T_{\text{opt}} = \sqrt{\frac{2(V + C)}{\lambda^F(s) + 2\lambda^S(s)}} \]

\[ \text{Waste} = \text{Waste}_{ef} + \text{Waste}_{fail} \]

\[ \text{Waste} = \frac{V + C}{T} + \lambda^F(s)(R + \frac{T}{2}) + \lambda^S(s)(R + T) \]

\[ \text{Time}_{FF} = \text{Time}_{Final}(1 - \text{Waste}_{Fail}) \times \text{Time}_{Final} \times \text{Waste}_{Fail} \]
Extensions

- VC-ONLY and VC+V
- Different speeds with DVFS, different error rates
- Different execution modes
- Optimize for time or for energy consumption

Current research

- Use verification to correct some errors (ABFT)
- Same analysis (smaller error rate but higher verification cost)
A few questions

Silent errors

- Error rate? MTBE?
- Selective reliability?
- New algorithms beyond iterative? matrix-product, FFT, ...

Resilient research on resilience

Models needed to assess techniques at scale without bias 😊
A few questions

Silent errors

- Error rate? MTBE?
- Selective reliability?
- New algorithms beyond iterative? matrix-product, FFT, ...

Resilient research on resilience

Models needed to assess techniques at scale without bias 😊
Silent errors

- Error rate? MTBE?
- Selective reliability?
- New algorithms beyond iterative? matrix-product, FFT, ...

Resilient research on resilience

Models needed to assess techniques at scale without bias 😊
A few questions

Silent errors

- Error rate? MTBE?
- Selective reliability?
- New algorithms beyond iterative? matrix-product, FFT, ...

Resilient research on resilience

Models needed to assess techniques at scale without bias 😊
Outline

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
4. Hands-on: First Implementation – Fault-Tolerant MPI (90 mn)
5. Hands-on: Designing a Resilient Application (90 mn)
6. Forward-recovery techniques (40mn)
7. Silent errors (35mn)
8. Conclusion (15mn)
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)
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?
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?
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 😊
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)


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