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/sc15tutorial.pdf

SC’2015 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 (90mn)

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

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

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

More nodes = $\neq$ Petascale $\times 1000$
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)
- 40 to 200 GB/s
- Parallel file system (1 to 2 PB)
- I/O nodes

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>

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
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 Count</th>
<th>Raw %</th>
<th>Filtered Count</th>
<th>Filtered %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>174,586,516</td>
<td>98.04</td>
<td>1,999</td>
<td>18.78</td>
</tr>
<tr>
<td>Software</td>
<td>144,899</td>
<td>0.08</td>
<td>6,814</td>
<td>64.01</td>
</tr>
<tr>
<td>Indeterminate</td>
<td>3,350,044</td>
<td>1.88</td>
<td>1,832</td>
<td>17.21</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
**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:

- \( \mathbb{P}(X \leq t) = 1 - e^{-\lambda t} \) (by definition)

- **Memoryless property**: \( \mathbb{P}(X \geq t + s | X \geq s) = \mathbb{P}(X \geq t) \)
  - at any instant, time to next failure does not depend upon time elapsed since last failure

- **Mean Time Between Failures (MTBF)** \( \mu = \mathbb{E}(X) = \frac{1}{\lambda} \)
**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*
Failure distributions: (2) Weibull

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

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

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

- If the MTBF is $\mu$ with one processor, what is its value with $p$ processors?
If three processors have around 20 faults during a time $t$ ($\mu = \frac{t}{20}$)...

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

Platform failure distribution
  ⇒ superposition of $p$ IID processor distributions
  ⇒ IID only for Exponential

Define $\mu_p$ by

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

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

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

#### Time (hours)

<table>
<thead>
<tr>
<th>Time (hours)</th>
<th>Failure Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0h</td>
<td>0</td>
</tr>
<tr>
<td>3h</td>
<td>0.1</td>
</tr>
<tr>
<td>6h</td>
<td>0.2</td>
</tr>
<tr>
<td>9h</td>
<td>0.3</td>
</tr>
<tr>
<td>12h</td>
<td>0.4</td>
</tr>
<tr>
<td>15h</td>
<td>0.5</td>
</tr>
<tr>
<td>18h</td>
<td>0.6</td>
</tr>
<tr>
<td>21h</td>
<td>0.7</td>
</tr>
<tr>
<td>24h</td>
<td>0.8</td>
</tr>
</tbody>
</table>

#### Parallel machine \((10^6 \text{ nodes})\)

- **Exp(1/100)**
- **Weibull(0.7, 1/100)**
- **Weibull(0.5, 1/100)**

---

After infant mortality and before aging, instantaneous failure rate of computer platforms is almost constant.
Summary for the road

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

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)
   - Process Checkpointing
   - Coordinated Checkpointing
   - Application-Level 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
   - Application-Level 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, or creates a copy in memory.

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

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

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

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

- Lack of portability
- Large size of checkpoint ($\approx$ memory footprint)
Blocking / Asynchronous call

### Blocking Checkpointing
Relatively intuitive: `checkpoint(filename)`
Cost: no process activity during the whole checkpoint operation.
Can be linear in the size of memory and in the size of modified files

### Asynchronous Checkpointing
System-level approach: make use of copy on write of `fork` syscall
User-level approach: critical sections, when needed
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
Outline

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)
   - Process Checkpointing
   - Coordinated Checkpointing
   - Application-Level 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 it, while the receiver did not receive it.
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

- Every message belongs to a single checkpoint wave
- 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.
Implementation

Communication Library

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

Application Level

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

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

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)
   - Process Checkpointing
   - Coordinated Checkpointing
   - Application-Level 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)
Application-Level Checkpointing

- Flush All Communication Channels
  - 'Natural Synchronization Point of the Application'
  - May need `quiesce()` interface for asynchronous libraries (unusual)

- Take User-Level Process Checkpoint
  - Serialize the state
  - Some frameworks can help – FTI

- Store the Checkpoint
  - In files (Some frameworks can help – SCR, FTI)
  - In memory (Some frameworks can help – FTI)

- Remove unused checkpoints
  - Atomic Commit
Application-Level Checkpointing

Application-Level Restart

- Synchronize processes
- Load the checkpoints
  - Decide which checkpoints to load
- Jump to the end of the corresponding checkpoint synchronization
  - Don’t forget to save the progress information in the checkpoint
Example: MPI-1D Stencil

```c
int main (int argc, char *argv[])
{
    double locals[NBLOCALS], /* The local values */
        *globals, /* all values, defined only for 0 */
        local_error, global_error; /* Estimates of the error */
    int    taskid, numtasks; /* rank and world size */
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
    /** Read the local domain from an input file */
    if( taskid == 0 ) globals = ReadFile("input");
    /** And distribute it on all nodes */
    MPI_Scatter(globals, NBLOCALS, MPI_DOUBLE,
                locals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    do {
        /** Update the domain, exchanging information with neighbors */
        UpdateLocals(locals, NBLOCALS, taskid, numtasks);
        /** Compute the local error */
        local_error = LocalError(locals, NBLOCALS);
        /** Compute the global error */
        MPI_AllReduce(&local_error, &global_error, 1, MPI_DOUBLE,
                      MPI_MAX, MPI_COMM_WORLD);
    } while( global_error > THRESHOLD );
    /** Output result to output file */
    MPI_Gather(locals, NBLOCALS, MPI_DOUBLE,
               globals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    if( taskid == 0 ) SaveFile("Result", globals);
    MPI_Finalize();
    return 0;
}
```
Example: MPI-1D Stencil

```c
int main (int argc, char *argv[]) {
    double locals[NBLOCALS], /* The local values */
                *globals, /* all values, defined only for 0 */
                local_error, global_error; /* Estimates of the error */
    int    taskid, numtasks; /* rank and world size */
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
    /** Read the local domain from an input file */
    if( taskid == 0 ) globals = ReadFile("input");
    /** And distribute it on all nodes */
    MPI_Scatter(globals, NBLOCALS, MPI_DOUBLE,
                  locals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    do {
        /** Update the domain, exchanging information with neighbors */
        UpdateLocals(locals, NBLOCALS, taskid, numtasks);
        /** Compute the local error */
        local_error = LocalError(locals, NBLOCALS);
        /** Compute the global error */
        MPI_AllReduce(&local_error, &global_error, 1, MPI_DOUBLE,
                       MPI_MAX, MPI_COMM_WORLD);
    } while( global_error > THRESHOLD );
    /** Output result to output file */
    MPI_Gather(locals, NBLOCALS, MPI_DOUBLE,
               globals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    if( taskid == 0 ) SaveFile("Result", globals);
    MPI_Finalize();
    return 0;
}
```

Bosilca, Bouteiller, Herault

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

Fault-tolerance for HPC
Example: MPI-1D Stencil

User-Level Checkpointing

```c
do {
    /** Update the domain, exchanging information with neighbors */
    UpdateLocals(locals, NBLOCALS, taskid, numtasks);
    /** Compute the local error */
    local_error = LocalError(locals, NBLOCALS);
    /** Compute the global error */
    MPI_AllReduce(&local_error, &global_error, 1, MPI_DOUBLE,
                MPI_MAX, MPI_COMM_WORLD);
    if( global_error > THRESHOLD && WantToCheckpoint() ) {
        MPI_Gather(locals, NBLOCALS, MPI_DOUBLE,
                  globals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        if( taskid == 0 ) {
            SaveFile("Checkpoint.new", globals);
            rename("Checkpoint.new", "Checkpoint.last");
        }
    }
} while( global_error > THRESHOLD );
/** Output result to output file */
MPI_Gather(locals, NBLOCALS, MPI_DOUBLE,
           globals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if( taskid == 0 ) SaveFile("Result", globals);
MPI_Finalize();
return 0;
```
**Example: MPI-1D Stencil**

User-Level Checkpointing

```c
1 do {
2     /** Update the domain, exchanging information with neighbors */
3     UpdateLocals(locals, NBLOCALS, taskid, numtasks);
4     /** Compute the local error */
5     local_error = LocalError(locals, NBLOCALS);
6     /** Compute the global error */
7     MPI_AllReduce(&local_error, &global_error, 1, MPI_DOUBLE,
8                     MPI_MAX, MPI_COMM_WORLD);
9     if( global_error > THRESHOLD && WantToCheckpoint() ) {
10        MPI_Gather(locals, NBLOCALS, MPI_DOUBLE,
11                     globals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
12        if( taskid == 0 ) {
13            SaveFile("Checkpoint.new", globals);
14            rename("Checkpoint.new", "Checkpoint.last")
15        }
16    }
17    } while( global_error > THRESHOLD );
18    /** Output result to output file */
19    MPI_Gather(locals, NBLOCALS, MPI_DOUBLE,
20                 globals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
21    if( taskid == 0 ) SaveFile("Result", globals);
22    MPI_Finalize();
23    return 0;
24}
```

**Atomic Commit of the Valid Checkpoint**
Example: MPI-1D Stencil

User-Level Rollback

```c
int main (int argc, char *argv[])
{
    double locals[NBLOCALS], /* The local values */
    *globals, /* all values, defined only for 0 */
    local_error, global_error; /* Estimates of the error */
    int     taskid, numtasks; /* rank and world size */

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD ,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD ,&taskid);

    /** Read the local domain from an input file */
    if( taskid == 0 ) globals = ReadFile(argv[1]);

    /** And distribute it on all nodes */
    MPI_Scatter(globals, NBLOCALS, MPI_DOUBLE,
                locals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Read Checkpoint or Input
User-Level Checkpointing

- Gather approach requires for one node to hold the entire checkpoint data
- Basic UNIX File Operations provide tools to manage the risk of failure during checkpoint creation

User-Level Rollback

- In general, rollback is more complex:
  - Need to remember the progress of computation
  - Need to jump to the appropriate part of the code when rollbacking

Time Overheads

- Checkpoint time includes Gather time
- Rollback time includes Scatter time
Application-Level Checkpointing – Distributed Checkpointing Approach

User-Level Distributed Checkpointing

- In files: one file per node, or shared file accessed by MPI_File_*
  - Atomic Commit of the last checkpoint might be a challenge
- In Memory
  - Can be very fast (no I/O)
  - Need a Fault-Tolerant MPI for hard failures (see hands on)
  - Need to store 3 checkpoints in processes memory space (for atomic commit)
User-Level Distributed Checkpointing

- **In files:** one file per node, or shared file accessed by MPI_File_ *
  - Atomic Commit of the last checkpoint might be a challenge
- **In Memory**
  - Can be very fast (no I/O)
  - Need a Fault-Tolerant MPI for hard failures (see hands on)
  - Need to store 3 checkpoints in processes memory space (for atomic commit)
Helping Libraries – SCR

Scalable Checkpoint Restart

- Manages Reliability of Storage for the user
- Manages Atomic Commit of Checkpoints
- Entirely based on Files
- Use local storage of files, as much as possible
  - Efficiency of local I/O
  - Risk of loosing data $\implies$ Fault Tolerant storage (Replication, or XOR)

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

SCR Example – Init

```c
int main (int argc, char *argv[])
{
    double locals[NBLOCALS], /* The local values */
        *globals, /* all values, defined only for 0 */
        local_error, global_error; /* Estimates of the error */
    int taskid, numtasks; /* rank and world size */
    char name[256], scr_file_name[SCR_MAX_FILENAME];
    FILE *f;
    size_t n;
    int rc, scr_want_to_checkpoint;

    MPI_Init(&argc,&argv);
    SCR_Init();
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);

    snprintf(name, "Checkpoint-%d", taskid);
    if( SCR_Route_file("MyCheckpoint", scr_file_name) != SCR_SUCCESS ) {
        fprintf(stderr, "Checkpoint disabled -- aborting
");
        MPI_Abort(MPI_COMM_WORLD);
    }
}
```

SCR Example – Fini

```c
if( taskid == 0 ) SaveFile("Result", globals);
SCR_Finalize();
MPI_Finalize();
return 0;
```

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

SCR Example – Checkpoint

```c
int local_error,
double global_error,
double threshold = 1.0;
int need_checkpoint = 0;

while( global_error > threshold ) {
    /** Update the domain, exchanging information with neighbors */
    UpdateLocals(locals, NBLOCALS, taskid, numtasks);
    /** Compute the local error */
    local_error = LocalError(locals, NBLOCALS);
    /** Compute the global error */
    MPI_AllReduce(&local_error, &global_error, 1, MPI_DOUBLE,
                   MPI_MAX, MPI_COMM_WORLD);
    SCR_Need_checkpoint(&scr_want_to_checkpoint);
    if( global_error > threshold && scr_want_to_checkpoint ) {
        SCR_Start_checkpoint();
        f = fopen(scr_file_name, "w");
        if( NULL != f ) {
            n = fwrite(f, locals, NBLOCALS * sizeof(double));
            rc = fclose(f);
        }
        SCR_Complete_checkpoint(f != NULL &&
                                 n == NBLOCALS * sizeof(double) &&
                                 rc == 0);
    }
} while( global_error > threshold );
```
SCR Example – Restart

```c
if( argc > 1 && (strcmp(argv[1], "-restart") == 0) ) {
    f = fopen(scr_file_name, "r");
    if( NULL != f ) {
        n = fread(f, locals, NBLOCALS * sizeof(double));
        rc = fclose(f);
    }
    if( f == NULL ||
        n != NBLOCALS * sizeof(double) ||
        rc != 0 ) {
        fprintf(stderr, "Unable to read checkpoint file\n");
        MPI_Abort(MPI_COMM_WORLD);
    } else {
        /** Read the local domain from an input file */
        if( taskid == 0 ) globals = ReadFile(argv[1]);
        /** And distribute it on all nodes */
        MPI_Scatter(globals, NBLOCALS, MPI_DOUBLE,
                    locals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }
}
```
Helping Libraries – FTI

Fault Tolerance Interface

- Manages Reliability of Storage for the user
- Manages Atomic Commit of Checkpoints
- Manages Transparent Restarts for the user

- Spawns new MPI processes to shadow the existing ones, and manage in-memory checkpoints
  - Relies on implementation-specific behaviors for MPI
  - Falls back on files in case of non-compliant MPI implementation

- Storage hierarchy: memory, local file, distributed file system
  - Fault Tolerant Storage algorithms: replication, Reed-Solomon Encoding
  - Computation might be offloaded to GPUs
FTI Example – Init

```c
int main (int argc, char *argv[])
{
  double locals[NBLOCALS], /* The local values */
  *globals, /* all values, defined only for 0 */
  local_error, global_error; /* Estimates of the error */
  int taskid, numtasks; /* rank and world size */

  MPI_Init(&argc,&argv);
  FTI_Init("conf.fti", MPI_COMM_WORLD);
  MPI_Comm_size(MPI_COMM_WORLD ,&numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD ,&taskid);
}
```

SCR Example – Fini

```c
if( taskid == 0 ) SaveFile("Result", globals);
FTI_Finalize();
MPI_Finalize();
return 0;
}
```
```c
/** Read the local domain from an input file */
if( taskid == 0 ) globals = ReadFile(argv[1]);
/** And distribute it on all nodes */
MPI_Scatter(globals, NBLOCALS, MPI_DOUBLE,
            locals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);

                FTI_Protect(0, locals, NBLOCALS * sizeof(double), FTI_DFLT);

            do {
                /** Update the domain, exchanging information with neighbors */
                UpdateLocals(locals, NBLOCALS, taskid, numtasks);
                /** Compute the local error */
                local_error = LocalError(locals, NBLOCALS);
                /** Compute the global error */
                MPI_AllReduce(&local_error, &global_error, 1, MPI_DOUBLE,
                               MPI_MAX, MPI_COMM_WORLD);
                FTI_Snapshot();
            } while( global_error > THRESHOLD );
```

- **FTI_Snapshot** decides if checkpoint is needed or not, and:
  - sets a jump point to the current position in the executable
  - saves 'protected' variables
Helping Libraries – FTI

FTI Example – Restart

```c
FTI_Protect(0, locals, NBLOCALS * sizeof(double), FTI_DFLT);

do {
    /** Update the domain, exchanging information with neighbors */
    UpdateLocals(locals, NBLOCALS, taskid, numtasks);
    /** Compute the local error */
    local_error = LocalError(locals, NBLOCALS);
    /** Compute the global error */
    MPI_AllReduce(&local_error, &global_error, 1, MPI_DOUBLE,
                  MPI_MAX, MPI_COMM_WORLD);
    FTI_Snapshot();
} while( global_error > THRESHOLD );
```

- **FTI_Init** jumps, if needed, to the checkpoint’s jump point, making the restart transparent
  - Non-protected variables are not restored: the code should not depend on them
  - Restoration assumes that the memory map is restored to the same (OS-dependent)
Helping Libraries – GVR

**Global View Resilience**

- Manages Reliability of Storage for the user

- Global View Resilience provides a reliable tuple-space for users to store persistent data. E.g., checkpoints

- Storage is entirely in memory, in independent processes accessible through the GVR API.
  - Spatial redundancy – coding at multiple levels
  - Temporal redundancy - Multi-version memory, integrated memory and NVRAM management

- Partitionned Global Address Space approach

- Data resides in the global GVR space, local values for specific versions are pulled for rollback, pushed for checkpoints

- Code is very different from the ones seen above, and outside the scope of this tutorial
Outline

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)
   - Process Checkpointing
   - Coordinated Checkpointing
   - Application-Level 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 $C_i^j$
- $\forall(i_1, \ldots, i_n), \exists j_k / \{C_{i_k}^j\}$ form a consistent cut?
- Domino Effect
Piece-wise Deterministic Assumption

- **Process**: alternate sequence of non-deterministic choice and deterministic steps
- **Transcribed in Message Passing**:
  - Receptions / Progress test are non-deterministic
    ```c
    (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 result obtained during the initial execution (from the last checkpoint), one can guide the execution of a process to its exact state just before the failure.
Message Logging

- **i^{th} probe:** false
- **m^{th} reception from B (on A):**

   A \[ \rightarrow \] T

- **i +1^{th} probe:** true
- **P**

- **n^{th} emission to A (from B):**

- Unique Identifier: (B, n, A, m)
- Payload: P

Message / Events

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

- 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

P will never be requested again

Q might be requested if A and B rollback
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)
Where to save the Events?

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

Where to save the Events?

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

Where to save the Events?

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

Where to save the Events?
- On a reliable media, asynchronously
- “Hope that the event will have time to be logged” (before its loss is damageable)
Optimistic Message Logging

Where to save the Events?

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

Where to save the Events?

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

Where to save the Events?

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

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

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

Recovery

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

First, when a non-deterministic event is created, it has to be handled by the protocols. Results are presented in Table I.

Fig. 6. Normalized performance of the NAS kernels on the Myrinet 10G network (Open MPI=1).

To evaluate the cost of event logging in the protocols, we used a small ping-pong test with 2 processes. The overhead clearly dominates the performance and flattens any differences between the protocols on the benchmarks.

As expected, the performance of event logging exhibits the worst case scenario, the executions of the two protocols are very similar. In this phase of the comparison we focus on widely uncoordinated protocol performance. We show that when there is no non-deterministic event, the performance varies only slightly. Even on those benchmarks where there is no non-deterministic events, the performance varies almost no differences between the protocols on the benchmarks.

The only benchmark showing a different scalability pattern is NAS Parallel Benchmarks – 64 nodes. In this case, the optimistic protocol with event logging is faster than the pessimistic. To understand this result, we evaluated the comparative scalability of the protocols we plot the normalized execution time of the benchmarks.

The benchmark with the worst scalability pattern is Optimist (Event Logging only). To understand the cost of event logging in the protocols, we evaluated the event logging overhead. We measured the execution time of the benchmarks with and without the sender-based overhead mechanism.

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

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

Fault-tolerance for HPC  67/ 235
Hierarchical Protocol Performance

- NAS Parallel Benchmarks – shared memory system, 32 cores
- HPL distributed system, 64 cores, 8 groups
General Techniques for Rollback Recovery – Conclusion

Summary

- Checkpointing is a general mechanism that is used for many reasons, *including* rollback-recovery fault-tolerance.
- There is a variety of protocols that coordinate (or not) the checkpoints, and complement them with necessary information.
- A critical element of performance of General Purpose Rollback-Recovery is how often checkpoints are taken.
- Other critical elements are the time to checkpoint (dominated by size of the data to checkpoint), and how processes are synchronized.

Coming Next

To understand how each element impacts the performance of rollback-recovery, we need to build *performance models* for these protocols.
Outline

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)
   - Young/Daly’s approximation
   - Exponential distributions
   - Assessing protocols at scale
   - 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
   - Exponential distributions
   - Assessing protocols at scale
   - 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)
Periodic checkpointing

- Time spent working
- Time spent checkpointing

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

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

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

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

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

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

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

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

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

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

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

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

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

WASTE minimized for \( T = \sqrt{\frac{u}{w}} \)

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

\[ (1 - \text{WASTE}[\text{fail}]) \text{TIME}_{\text{final}} = \text{TIME}_{\text{FF}} \]
\[ \Rightarrow T = \sqrt{2(\mu - (D + R))C} \]

Daly: \( \text{TIME}_{\text{final}} = (1 + \text{WASTE}[\text{fail}]) \text{TIME}_{\text{FF}} \)
\[ \Rightarrow T = \sqrt{2(\mu + (D + R))C + C} \]

Young: \( \text{TIME}_{\text{final}} = (1 + \text{WASTE}[\text{fail}]) \text{TIME}_{\text{FF}} \) and \( D = R = 0 \)
\[ \Rightarrow T = \sqrt{2\mu C + C} \]

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
Validity of the approach (1/3)

Technicalities

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

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

- Enforce $D + R \leq \mu$ and bound $T$ to get $\text{WASTE}[fail] \leq 1$
  but $\mu = \frac{\mu_{ind}}{p}$ too small for large $p$, regardless of $\mu_{ind}$
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:
    $$\pi = P(X \geq 2) = 1 - (P(X = 0) + P(X = 1)) = 1 - (1 + \theta)e^{-\theta}$$
  - $\gamma = 0.27 \Rightarrow \pi \leq 0.03$
    $$\Rightarrow$$ 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)
Capping periods, and enforcing a lower bound on MTBF
⇒ mandatory for mathematical rigor 😞

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

Approach surprisingly robust 😊
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

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

Petascale: $C = 20$ min, $\mu = 24$ hrs $\Rightarrow$ WASTE[opt] = 17%
Scale by 10: $C = 20$ min, $\mu = 2.4$ hrs $\Rightarrow$ WASTE[opt] = 53%
Scale by 100: $C = 20$ min, $\mu = 0.24$ hrs $\Rightarrow$ 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 ≠ Petascale $\times 1000$

Need more reliable components
Need to checkpoint faster

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

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

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

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)
   - Young/Daly’s approximation
   - Exponential distributions
   - Assessing protocols at scale
   - 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)
Exponential failure distribution

1. Expected execution time for a single chunk
2. Expected execution time for a sequential job
3. Expected execution time for a parallel job
Expected execution time for a single chunk

Compute the expected time $\mathbb{E}(T(W, C, D, R, \lambda))$ to execute a work of duration $W$ followed by a checkpoint of duration $C$.

Recursive Approach

$\mathbb{E}(T(W)) =$

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
Expected execution time for a single chunk

Compute the expected time $\mathbb{E}(T(W, C, D, R, \lambda))$ to execute a work of duration $W$ followed by a checkpoint of duration $C$.

**Recursive Approach**

$$\mathbb{E}(T(W)) = \mathbb{E}(T(W)) =$$

$P_{\text{succ}}(W + C)(W + C)$
Compute the expected time \( \mathbb{E}(T(W, C, D, R, \lambda)) \) to execute a work of duration \( W \) followed by a checkpoint of duration \( C \).

**Recursive Approach**

\[
\mathbb{E}(T(W)) = \mathcal{P}_{\text{succ}}(W + C)(W + C)
\]
Expected execution time for a single chunk

Compute the expected time $\mathbb{E}(T(W, C, D, R, \lambda))$ to execute a work of duration $W$ followed by a checkpoint of duration $C$.

Recursive Approach

$$\mathbb{E}(T(W)) = \mathcal{P}_{\text{suc}}(W + C)(W + C) + (1 - \mathcal{P}_{\text{suc}}(W + C))(\mathbb{E}(T_{\text{lost}}(W + C)) + \mathbb{E}(T_{\text{rec}}) + \mathbb{E}(T(W)))$$

Probability of failure
Expected execution time for a single chunk

Compute the expected time $\mathbb{E}(T(W, C, D, R, \lambda))$ to execute a work of duration $W$ followed by a checkpoint of duration $C$.

**Recursive Approach**

$$\begin{align*}
\mathbb{E}(T(W)) &= P_{\text{suc}}(W + C)(W + C) \\
&\quad + (1 - P_{\text{suc}}(W + C))(\mathbb{E}(T_{\text{lost}}(W + C)) + \mathbb{E}(T_{\text{rec}}) + \mathbb{E}(T(W)))
\end{align*}$$

Time elapsed before failure

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
Expected execution time for a single chunk

Compute the expected time $\mathbb{E}(T(W, C, D, R, \lambda))$ to execute a work of duration $W$ followed by a checkpoint of duration $C$.

**Recursive Approach**

$$
\mathbb{E}(T(W)) = P_{\text{succ}}(W + C)(W + C) + (1 - P_{\text{succ}}(W + C))(\mathbb{E}(T_{\text{lost}}(W + C)) + \mathbb{E}(T_{\text{rec}}) + \mathbb{E}(T(W)))
$$

Time needed to perform downtime and recovery
Expected execution time for a single chunk

Compute the expected time $\mathbb{E}(T(W, C, D, R, \lambda))$ to execute a work of duration $W$ followed by a checkpoint of duration $C$.

**Recursive Approach**

$$\mathbb{E}(T(W)) = \mathcal{P}_{\text{suc}}(W + C) (W + C) + (1 - \mathcal{P}_{\text{suc}}(W + C)) (\mathbb{E}(T_{\text{lost}}(W + C)) + \mathbb{E}(T_{\text{rec}}) + \mathbb{E}(T(W)))$$

Time needed to compute $W$ anew
Computation of $\mathbb{E}(T(W, C, D, R, \lambda))$

$\mathbb{E}(T(W)) = P_{\text{suc}}(W + C)(W + C) + (1 - P_{\text{suc}}(W + C))(\mathbb{E}(T_{\text{lost}}(W + C)) + \mathbb{E}(T_{\text{rec}}) + \mathbb{E}(T(W)))$

- $P_{\text{suc}}(W + C) = e^{-\lambda(W+C)}$
- $\mathbb{E}(T_{\text{lost}}(W + C)) = \int_0^\infty xP(X = x|X < W + C)dx = \frac{1}{\lambda} - \frac{W+C}{e^{\lambda(W+C)}-1}$
- $\mathbb{E}(T_{\text{rec}}) = e^{-\lambda R}(D+R) + (1-e^{-\lambda R})(D+\mathbb{E}(T_{\text{lost}}(R)) + \mathbb{E}(T_{\text{rec}}))$

$\mathbb{E}(T(W, C, D, R, \lambda)) = e^{\lambda R} \left( \frac{1}{\lambda} + D \right) (e^{\lambda(W+C)} - 1)$
Checkpointing a sequential job

- $E(T(W)) = e^{\lambda R} \left( \frac{1}{\lambda} + D \right) \left( \sum_{i=1}^{K} e^{\lambda (W_i+C)} - 1 \right)$
- Optimal strategy uses same-size chunks (convexity)
- $K_0 = \frac{\lambda W}{1 + \mathbb{L}(-e^{-\lambda C-1})}$ where $\mathbb{L}(z)e^{\mathbb{L}(z)} = z$ (Lambert function)
- Optimal number of chunks $K^*$ is max(1, $\lfloor K_0 \rfloor$) or $\lceil K_0 \rceil$

$$E_{opt}(T(W)) = K^* \left( e^{\lambda R} \left( \frac{1}{\lambda} + D \right) \right) \left( e^{\lambda \left( \frac{W}{K^*} + C \right)} - 1 \right)$$

- Can also use Daly’s second-order approximation
Checkpointing a parallel job

- \( p \) processors \( \Rightarrow \) distribution \( \text{Exp}(\lambda_p) \), where \( \lambda_p = p\lambda \)
- Use \( W(p) \), \( C(p) \), \( R(p) \) in \( F_{\text{opt}}(T(W)) \) for a distribution \( \text{Exp}(\lambda_p = p\lambda) \)
- Job types
  - Perfectly parallel jobs: \( W(p) = W/p \).
  - Generic parallel jobs: \( W(p) = W/p + \delta W \)
  - Numerical kernels: \( W(p) = W/p + \delta W^{2/3}/\sqrt{p} \)
- Checkpoint overhead
  - Proportional overhead: \( C(p) = R(p) = \delta V/p = C/p \)
    (bandwidth of processor network card/link is I/O bottleneck)
  - Constant overhead: \( C(p) = R(p) = \delta V = C \)
    (bandwidth to/from resilient storage system is I/O bottleneck)
Weibull failure distribution

- No optimality result known
- Heuristic: maximize expected work before next failure
- Dynamic programming algorithms
  - Use a time quantum
  - Trim history of previous failures
Outline

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)
   - Young/Daly’s approximation
   - Exponential distributions
   - Assessing protocols at scale
   - 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)
Which checkpointing protocol to use?

Coordinated checkpointing

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

Hierarchical checkpointing

- 😞 Need to log inter-groups messages
  - ● Slowdowns failure-free execution
  - ● Increases checkpoint size/time
- 😊 Only processors from failed group need to roll back
- 😊 Faster re-execution with logged messages
- 😊 Rumor: Should scale to very large platforms
Blocking vs. non-blocking

Blocking model: checkpointing blocks all computations
Blocking vs. Non-blocking

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

**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: $\text{WORK} = (T - C) + \alpha C$

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

\[ P_0, P_1, P_2, P_3 \]
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
Waste due to failures in computation phase

Finally, the checkpointing phase is executed
Total waste

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

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

\[ \alpha(G-g+1)C \quad T-G.C-T_{\text{lost}} \quad T \]
Accounting for message logging: Impact on work

- Logging messages slows down execution:
  \[ \text{WORK} \text{ becomes } \lambda \text{WORK}, \text{ where } 0 < \lambda < 1 \]
  Typical value: \( \lambda \approx 0.98 \)

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

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

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

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

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

\[
C(q) = \frac{C_0(q)(1 + \beta \lambda T)}{1 + GC_0(q)\beta \lambda(1 - \alpha)}
\]
Three case studies

**Coord-IO**

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

where Mem is the memory footprint of the application

**Hierarch-IO**

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

⇒ groups checkpoint sequentially

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

**Hierarch-Port**

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

⇒ some groups checkpoint in parallel

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

1. 2D-stencil
2. Matrix product
3. 3D-Stencil
   - Plane
   - Line
### 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 per processor</th>
<th>I/O Network Bandwidth (b$_{io}$)</th>
<th>Bandwidth (b$_{io}$)</th>
<th>I/O Bandwidth (b$_{port}$)</th>
<th>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>
<td></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>
<td></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>
<td></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>
<td></td>
</tr>
</tbody>
</table>

### Fault-tolerance for HPC

<table>
<thead>
<tr>
<th>Name</th>
<th>Scenario</th>
<th>$G \left(C(q)\right)$</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></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></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>0.0002859</td>
<td>0.002227</td>
</tr>
<tr>
<td></td>
<td>HIERARCH-IO</td>
<td>1,000 (64s)</td>
<td>0.0005199</td>
<td></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>0.0003203</td>
</tr>
<tr>
<td></td>
<td>HIERARCH-IO</td>
<td>316 (217s)</td>
<td>0.00016440</td>
<td></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

Waste as a function of processor MTBF $\mu_{\text{ind}}$
Platform: K-Computer

Stencil 2D

Matrix product

Stencil 3D

Waste as a function of processor MTBF $\mu_{ind}$
WASTE = 1 for all scenarios!!!
Plotting formulas – Platform: Exascale

WASTE = 1 for all scenarios!!!

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

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

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

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

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
Simulations – Platform: Titan

Stencil 2D

Matrix product

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

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

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

Exascale-Slim

Exascale-Fat

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
Outline

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
   - Young/Daly’s approximation
   - Exponential distributions
   - Assessing protocols at scale
   - 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
Failures

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

Best trade-off between performance and risk?
Outline

1. Introduction (15mn)

2. Checkpointing: Protocols (30mn)

3. Checkpointing: Probabilistic models (45mn)
   - Young/Daly’s approximation
   - Exponential distributions
   - Assessing protocols at scale
   - 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)
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
Fault rates

- $\mu$: mean time between failures (MTBF)
- $\mu_P$ mean time between predicted events (both true positive and false positive)
- $\mu_{NP}$ mean time between unpredicted faults (false negative).
- $\mu_e$: mean time between events (including three event types)

$$r = \frac{True_P}{True_P + False_N} \quad \text{and} \quad p = \frac{True_P}{True_P + False_P}$$

$$\frac{1 - r}{\mu} = \frac{1}{\mu_{NP}} \quad \text{and} \quad \frac{r}{\mu} = \frac{p}{\mu_P}$$

$$\frac{1}{\mu_e} = \frac{1}{\mu_P} + \frac{1}{\mu_{NP}}$$
Example

- Predictor predicts six faults in time $t$
- Five actual faults. One fault not predicted
- $\mu = \frac{t}{5}$, $\mu_P = \frac{t}{6}$, and $\mu_{NP} = t$
- Recall $r = \frac{4}{5}$ (green arrows over red arrows)
- Precision $p = \frac{4}{6}$ (green arrows over blue arrows)
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 of fault-free execution]

2. **Unpredicted faults:**

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

   ![Diagram of unpredicted faults]

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

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

(Row 1: Regular mode)

(Row 2: Proactive mode)

(Row 3: Proactive mode)

(Row 4: Regular mode)

Gets too complicated! 😞
Outline

1. Introduction (15mn)
2. Checkpointing: Protocols (30mn)
3. Checkpointing: Probabilistic models (45mn)
   - Young/Daly’s approximation
   - Exponential distributions
   - Assessing protocols at scale
   - 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
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
Example

Pair_1  \( p_1 \)  \( p_2 \)
---
\( \vdash \)

Pair_2  \( p_1 \)  \( p_2 \)
\( \vdash \)
\( \vdash \)

Pair_3  \( p_1 \)  \( p_2 \)

Pair_4  \( p_1 \)  \( p_2 \)
\( \vdash \)
\( \vdash \)
\( \vdash \)
\( \vdash \)
\( \vdash \)
\( \vdash \)
\( \vdash \)

Time
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?

$$Birthday(m) = 1 + \int_0^{+\infty} e^{-x}(1 + x/m)^{m-1} dx = \frac{2}{3} + \sqrt{\frac{\pi m}{2}} + \sqrt{\frac{\pi}{288m}} - \frac{4}{135m} + \ldots$$

The analogy

Two people with same birthday

≡

Two failures hitting same replica-group
### Differences with birthday problem

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>...</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>...</td>
<td>i</td>
<td>N</td>
</tr>
</tbody>
</table>

- 2\(N\) processors but \(N\) 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

- 2N processors but 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

- 2N processors but 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

- $2N$ processors but $N$ 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

- 2N processors but 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

- $2N$ processors but $N$ 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

- 2N processors but 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 ≠ 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

- $2N$ processors but $N$ 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

- $2N$ processors but $N$ 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

- $2N$ processors but $N$ 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

- $2N$ processors but $N$ 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

- \(2N\) processors but \(N\) 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

\[
\begin{array}{cccccccc}
\text{□ □ □ □ □} & \ldots & \text{□} \\
1 & 2 & 3 & 4 & \ldots & n \\
\end{array}
\]

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

Mean Number of Failures to Interruption (bring down application)
\[MNFTI = \text{expected number of balls to throw until one bin gets one ball of each color}\]
Number of failures to bring down application

- $MNFTI_{ah}$ Count each failure hitting any of the initial processors, including those already hit by a failure
- $MNFTI_{rp}$ Count failures that hit running processors, and thus effectively kill replicas.

\[ MNFTI_{ah} = 1 + MNFTI_{rp} \]
Number of failures to bring down application

- $MNFTI^{ah}$ Count each failure hitting any of the initial processors, including those *already hit* by a failure
- $MNFTI^{rp}$ Count failures that hit *running processors*, and thus effectively kill replicas.

$$MNFTI^{ah} = 1 + MNFTI^{rp}$$
**Theorem** $MNFTI^{ah} = \mathbb{E}(NFTI^{ah}|0)$ where

$$\mathbb{E}(NFTI^{ah}|n_f) = \begin{cases} 2 & \frac{2n_{rg}}{2n_{rg} - n_f} + \frac{2n_{rg} - 2n_f}{2n_{rg} - n_f} \mathbb{E}(NFTI^{ah}|n_f + 1) \\ \text{otherwise.} & \text{if } n_f = n_{rg}, \end{cases}$$

$\mathbb{E}(NFTI^{ah}|n_f)$: expectation of number of failures to kill application, knowing that

- application is still running
- failures have already hit $n_f$ different replica-groups
Exponential failures (cont’d)

Proof

\[
\mathbb{E} \left( NFTI^{\text{ah}} | n_{rg} \right) = \frac{1}{2} \times 1 + \frac{1}{2} \times \left( 1 + \mathbb{E} \left( NFTI^{\text{ah}} | n_{rg} \right) \right).
\]

\[
\mathbb{E} \left( NFTI^{\text{ah}} | n_f \right) = \frac{2n_{rg} - 2n_f}{2n_{rg}} \times \left( 1 + \mathbb{E} \left( NFTI^{\text{ah}} | n_f + 1 \right) \right) \\
+ \frac{2n_f}{2n_{rg}} \times \left( \frac{1}{2} \times 1 + \frac{1}{2} \left( 1 + \mathbb{E} \left( NFTI^{\text{ah}} | n_f \right) \right) \right).
\]

\[MTTI = \text{systemMTBF}(2n_{rg}) \times MNFTI^{\text{ah}}\]
Exponential failures (cont’d)

Proof

\[
\mathbb{E} \left( NFTI^{ah} | n_{rg} \right) = \frac{1}{2} \times 1 + \frac{1}{2} \times \left( 1 + \mathbb{E} \left( NFTI^{ah} | n_{rg} \right) \right).
\]

\[
\mathbb{E} \left( NFTI^{ah} | n_f \right) = \frac{2n_{rg} - 2n_f}{2n_{rg}} \times \left( 1 + \mathbb{E} \left( NFTI^{ah} | n_f + 1 \right) \right) + \frac{2n_f}{2n_{rg}} \times \left( \frac{1}{2} \times 1 + \frac{1}{2} \left( 1 + \mathbb{E} \left( NFTI^{ah} | n_f \right) \right) \right).
\]

MTTI = systemMTBF(2n_{rg}) \times MNFTI^{ah}
Exponential failures (cont’d)

Proof

\[ \mathbb{E} \left( NFTI^{ah} \mid n_{rg} \right) = \frac{1}{2} \times 1 + \frac{1}{2} \times \left( 1 + \mathbb{E} \left( NFTI^{ah} \mid n_{rg} \right) \right). \]

\[ \mathbb{E} \left( NFTI^{ah} \mid n_f \right) = \frac{2n_{rg} - 2n_f}{2n_{rg}} \times \left( 1 + \mathbb{E} \left( NFTI^{ah} \mid n_f + 1 \right) \right) \]

\[ + \frac{2n_f}{2n_{rg}} \times \left( \frac{1}{2} \times 1 + \frac{1}{2} \left( 1 + \mathbb{E} \left( NFTI^{ah} \mid n_f \right) \right) \right). \]

\[ \text{MTTI} = \text{systemMTBF}(2n_{rg}) \times MNFTI^{ah} \]
Comparison

- 2N processors, no replication
  \[ \text{Throughput}_{\text{Std}} = 2N(1 - \text{Waste}) = 2N \left(1 - \sqrt{\frac{2C}{\mu_{2N}}} \right) \]

- N replica-pairs
  \[ \text{Throughput}_{\text{Rep}} = N \left(1 - \sqrt{\frac{2C}{\mu_{\text{rep}}}} \right) \]
  \[ \mu_{\text{rep}} = MNFTI \times \mu_{2N} = MNFTI \times \frac{\mu}{2^N} \]

- Platform with 2N = 2^{20} processors \(\Rightarrow MNFTI = 1284.4\)
  \(\mu = 10\) years \(\Rightarrow\) better if \(C\) shorter than 6 minutes
Failure distribution

(a) Exponential

(b) Weibull, $k = 0.7$

Crossover point for replication when $\mu_{\text{ind}} = 125$ years
Weibull distribution with $k = 0.7$

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

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

Contact Information:
- bosilca,bouteiller,herault@icl.utk.edu
- yves.robert@inria.fr
Fault Tolerance Software Stack

- Application
- Lib1
- Lib2
- Comm. Middleware (MPI)
- OS
- Network
- Runtime Helpers
- Network
- Transient Failures (inc. msg corruption)

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

Permanent Crash Detection

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
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)
  - On failure detection, the runtime system kills all processes
  - trunk: error is never reported to the MPI processes.
  - ft-branch: the error is reported, MPI might be partly usable.

- MPICH (http://www.mcs.anl.gov/mpi/mpich/)
  - Default: on failure detection, the runtime kills all processes.
    Can be de-activated by a runtime switch
  - Errors might be reported to MPI processes in that case. MPI might be partly usable.
FT Middleware in HPC

- Not MPI. Sockets, PVM... CCI?
  http://www.olcf.ornl.gov/center-projects/common-communication-interface/UCCS?


- 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
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
Incoherent global state and resolution

- 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) {
    worker[i++] = worker[workers--];
    resubmit_work( worker[i], i )
}
```

```c
Worker

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

```c
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);
}
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();
```
Fault Tolerant Master

```c
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/sc15
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
   - 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
   - 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)
Material to support this part of the tutorial includes code skeletons.

It is available online:
http://fault-tolerance.org/sc15
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
   - 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)
Material to support this part of the tutorial includes code skeletons.

It is available online:
http://fault-tolerance.org/sc15
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
   - 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)
Material to support this part of the tutorial includes code skeletons.

It is available online:
http://fault-tolerance.org/sc15
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
   - 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/sc15
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)
Forward-Recovery

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

- 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

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

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

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

<p>| | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
</tbody>
</table>

Then, update the missing coverage. Keep checkpoint block column to cover failures during that time.
Algorithm Based Fault Tolerant QR decomposition

- In case of failure, conclude the operation, then
  - Missing Data = Checksum - Sum(Existing Data) s
Algorithm Based Fault Tolerant QR decomposition

In case of failure, conclude the operation, then

- Missing Checksum = Sum(Existing Data)
Algorithm Based Fault Tolerant QR decomposition

 Failures may happen while inside a $Q$–panel factorization

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

Fault-tolerance for HPC 172/ 235
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.
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

![Graph showing performance and relative overhead of ScaLAPACK PDGEQRF and fault-tolerant PDGEQRF (FT-PDGEQRF) for different matrix sizes and processor grids. The graph compares the performance (TFlop/s) and relative overhead (%) for the following scenarios:

- **ScaLAPACK PDGEQRF**
- **FT-PDGEQRF (no errors)**
- **FT-PDGEQRF (one error)**

The graph displays the performance and overhead for matrix sizes of 6x6; 20k, 12x12; 40k, 24x24; 80k, and 48x48; 160k, with different processor grids (P x Q). The performance decreases as the number of processors increases, indicating a scalability issue. The overhead also increases with the number of processors, but the fault-tolerant approach shows a smaller overhead compared to the baseline.

**Legend:**
- ScaLAPACK PDGEQRF
- FT-PDGEQRF (no errors)
- FT-PDGEQRF (one error)

**Key Observations:**
- For smaller matrices (6x6; 20k), the overhead is minimal and remains consistent across different processor configurations.
- As the matrix size and number of processors increase (e.g., 48x48; 160k), the overhead increases significantly, but the fault-tolerant approach shows a smaller overhead compared to the baseline.

**Implications:**
- The fault-tolerant approach (FT-PDGEQRF) is more efficient in handling failures, especially for larger matrices and processor grids.
- The scalability issue is evident as the number of processors increases, indicating the need for more sophisticated fault-tolerant algorithms.

---

**References:**
- A. Bouteiller, T. Herault, G. Bosilca, P. Du, and J. Dongarra

---

**Footnote:**
- MPI-Next ULFM Performance

---

**Contact:**

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

---

**Presentation Title:** Fault-tolerance for HPC

---

**Page Number:** 174/235
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 $P \times Q$ 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

**FIND OUT MORE AT**

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

**MPI-Next ULFM Performance**

- Open MPI with ULFM; Kraken supercomputer;

### Performance On Kraken

<table>
<thead>
<tr>
<th>#Processors (PxQ grid); Matrix size (N)</th>
<th>Relative Overhead (%)</th>
<th>Performance (TFlop/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6x6; 20k</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>12x12; 40k</td>
<td>0</td>
<td>40</td>
</tr>
<tr>
<td>24x24; 80k</td>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td>48x48; 160k</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>96x96; 320k</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>192x192; 640k</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>192x192; 640k</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Flops for the checksum update**

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

**Protection cost is inversely proportional to machine scale!**

**Computation Memory**

<table>
<thead>
<tr>
<th>F</th>
<th>simultaneous failures tolerated</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>$P \times Q$</td>
</tr>
</tbody>
</table>

Overheads in $F / Q$
ABFT LU decomposition: implementation

Checkpoint on Failure - MPI Implementation

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

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

![Graph showing performance comparison between ScaLAPACK QR, CoF-QR (w/o failure), and CoF-QR (w/1 failure).]

Checkpoint on Failure - MPI Performance

Open MPI; Kraken supercomputer;
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

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

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

**Typical Application**

```c
for( aninsanenumber ) {
    /* Extract data */
    * simulation ,
    * matrix */
    sim2mat();
    /* Factorize matrix */
    * Solve */
    dgeqrf();
    dsolve();
    /* Update simulation */
    * with result vector */
    vec2sim();
}
```

Goodbye 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

```csharp
for ( ; ; ) {
    /∗ Extra data from simulation, fill up matrix ∗/ sim2mat();
    /∗ Factorize matrix, Solve ∗/ dgeqr2(); dsolv();
    /∗ Update simulation with result vector */ vec2sim();
}
```

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...

- 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
ABFT & Periodic Checkpoint: no failure

- Process 0
- Process 1
- Process 2

Periodic Checkpoint

Split Forced Checkpoints

Application

Library
ABFT & Periodic Checkpoint: failure during Library phase

Process 0
Process 1
Process 2

Application
Library
Application
Library
Application
Library

Failure (during Library)
Rollback (partial) Recovery
ABFT Recovery

{bosilca, bouteiller, herault}@icl.utk.edu | yves.robert@inria.fr
ABFT & Periodic Checkpoint: failure during General phase
ABFT & Periodic Ckpt: 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 & Periodic Ckpt: 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
\( T^{\text{ff}}, T_G^{\text{ff}}, T_L^{\text{ff}} \): “Fault Free” times
\( t_G^{\text{lost}}, t_L^{\text{lost}} \): Lost time (recovery overhreads)
\( T_G^{\text{final}}, T_L^{\text{final}} \): Total times (with faults)
A few notations

Costs

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

### GENERAL phase

![Diagram showing the process phases and checkpoints]

- **Process 0**
  - Application
  - Library
  - Periodic Checkpoint

- **Process 1**
  - Application
  - Library
  - Split Forced Checkpoints

- **Process 2**
  - Application
  - Library

### Without Failures

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

---

**Without Failures**

\[ T_{ff}^L = \phi \times T_L + C_L \]

---

**Library phase**

- **Periodic Checkpoint**
- **Split Forced Checkpoints**

---

**Without Failures**

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

**GENERAL phase**

- Process 0
- Process 1
- Process 2

Failures (during GENERAL)

Rollback (full)

Recovery

**Failure Overhead**

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

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

**Library phase**

- Process 0
- Process 1
- Process 2

**Application**

**Library**

- Failure (during LIBRARY)
- Rollback (partial)
- Recovery

- ABFT Recovery

**Failure Overhead**

\[ t_{L}^{\text{lost}} = D + R_{L} + \text{Recons}_{ABFT} \]
Overall

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

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

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

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

Which is minimal in the second case, if

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

\[
\text{WASTE} = 1 - \frac{T_0}{T_{final}^G + T_{final}^L}
\]
Toward Exascale, and Beyond!

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?
Competitors

FT algorithms compared

**PeriodicCkpt**  Basic periodic checkpointing

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

**ABFT&PeriodicCkpt**  The algorithm described above
Weak Scale 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 $O(\frac{1}{n})$
- $C (\equiv R)$ 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

Fault-tolerance for HPC
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 #2

- Nb Faults PeriodicCkpt
- Nb Faults Bi-PeriodicCkpt
- Nb Faults ABFT PeriodicCkpt

Waste Ratio of time spent in the ABFT routine

Nodes
- PeriodicCkpt
- Bi-PeriodicCkpt
- ABFT PeriodicCkpt
- ABFT Ratio

{bosilca,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
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 $O\left(\frac{1}{n}\right)$
- $C (=R)$ 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

### Table

<table>
<thead>
<tr>
<th># Faults</th>
<th>Nb Faults PeriodicCkpt</th>
<th>Nb Faults Bi-PeriodicCkpt</th>
<th>Nb Faults ABFT PeriodicCkpt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.05</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.35</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Graphs

1. **Graph 1**
   - Title: Nb Faults PeriodicCkpt, Nb Faults Bi-PeriodicCkpt, Nb Faults ABFT PeriodicCkpt
   - X-axis: Nodes
   - Y-axis: # Faults

2. **Graph 2**
   - Title: Waste Nodes PeriodicCkpt, Bi-PeriodicCkpt, ABFT PeriodicCkpt
   - X-axis: Nodes
   - Y-axis: Waste

### Notes
- \( \alpha = 0.55 \)
- \( \alpha = 0.8 \)
- \( \alpha = 0.92 \)
- \( \alpha = 0.975 \)

---

{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
Quotes

• **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

Theorem: \( \mu_p = \frac{\mu_{\text{ind}}}{p} \) 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)
General-purpose approach

![Diagram showing error and detection latency with a fault occurring before detection]

- 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

- 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
Optimal period?

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)

{bosisla,bouteiller,herault}@icl.utk.edu | yves.robert@inria.fr
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!
Coupling checkpointing and verification

- 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)

#### Diagram:
- **fault**
- **Detection**

#### Time
- $W$
- $V$
- $C$

#### Fail-stop (classical)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$WASTE[FF]$</td>
<td>$T = W + C$</td>
</tr>
<tr>
<td>$WASTE[fail]$</td>
<td>$\frac{C}{T}$</td>
</tr>
<tr>
<td>Optimal</td>
<td>$T_{opt} = \sqrt{2C\mu}$</td>
</tr>
<tr>
<td>$WASTE[opt]$</td>
<td>$\sqrt{\frac{2C}{\mu}}$</td>
</tr>
</tbody>
</table>

#### Silent errors

<table>
<thead>
<tr>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S = W + V + C$</td>
</tr>
<tr>
<td>$\frac{V+C}{S}$</td>
</tr>
<tr>
<td>$\frac{1}{\mu}(R + W + V)$</td>
</tr>
<tr>
<td>$S_{opt} = \sqrt{(C + V)\mu}$</td>
</tr>
<tr>
<td>$2\sqrt{\frac{C + V}{\mu}}$</td>
</tr>
</tbody>
</table>
With $p = 1$ checkpoint and $q = 3$ verifications

Base Pattern: $p = 1, q = 1$

New Pattern: $p = 1, q = 3$

WASTE[opt] = $2\sqrt{\frac{C+V}{\mu}}$

WASTE[opt] = $2\sqrt{\frac{4(C+3V)}{6\mu}}$
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

1. (proba $2w/W$) $T_{\text{lost}} = R + 2w + V$
2. (proba $2w/W$) $T_{\text{lost}} = R + 4w + 2V$
3. (proba $w/W$) $T_{\text{lost}} = 2R + 6w + C + 4V$
4. (proba $w/W$) $T_{\text{lost}} = R + w + 2V$
5. (proba $2w/W$) $T_{\text{lost}} = R + 3w + 2V$
6. (proba $2w/W$) $T_{\text{lost}} = R + 5w + 3V$

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

- $o_{ff}$ failure-free overhead per pattern
- $f_{re}$ fraction of work that is re-executed

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

$\text{WASTE}[opt] = 2\sqrt{\frac{o_{ff}f_{re}}{\mu}} + o\left(\sqrt{\frac{1}{\mu}}\right)$
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 $p \geq 1$

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

- Assess gain due to the $p-1$ intermediate checkpoints
- $f_{re}^{(1)} - f_{re}^{(p)} = \sum_{i=1}^{p} (\alpha_i \sum_{j=1}^{i-1} \alpha_j)$
- 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
- In that case, $f_{re}^{(1)} = \frac{q+1}{2q}$ 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}}
Summary

- **Balanced Algorithm** optimal when $C, R, V \ll \mu$
- Keep only 2 checkpoints in memory/storage
- Closed-form formula for $\text{WASTE}[\text{opt}]$
- 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)

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

Fault-tolerance for HPC
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

- $\{T_1, T_2, \ldots, T_n\} : \text{linear chain of } n \text{ tasks}$
- Each task $T_i$ fully parametrized:
  - $w_i$ computational weight
  - $C_i, R_i, V_i : \text{checkpoint, recovery, verification}$
- Error rates:
  - $\lambda^F$ rate of fail-stop errors
  - $\lambda^S$ rate of silent errors
\[
\begin{align*}
\min_{0 \leq k < n} & \quad Time_{C}^{\text{rec}}(n, k) \\
Time_{C}^{\text{rec}}(j, k) &= \min_{k \leq i < j} \{ Time_{C}^{\text{rec}}(i, k - 1) + T_{C}^{\text{SF}}(i + 1, j) \} \\
T_{C}^{\text{SF}}(i, j) &= p_{i,j}^{F} (T_{\text{lost}}_{i,j} + R_{i-1} + T_{C}^{\text{SF}}(i, j)) \\
&\quad+ (1 - p_{i,j}^{F}) \left( \sum_{\ell=i}^{j} w_{\ell} + V_{j} + p_{i,j}^{S} (R_{i-1} + T_{C}^{\text{SF}}(i, j)) + (1 - p_{i,j}^{S}) C_{j} \right)
\end{align*}
\]
\[ T_{\text{opt}} = \sqrt{\frac{2(V + C)}{\lambda^F(s) + 2\lambda^S(s)}} \]

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

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

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

\[ T_{\text{Final}} \times \text{Waste}_{\text{Fail}} \]
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, ...
- Multi-level patterns for both fail-stop and silent errors

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, ...
- Multi-level patterns for both fail-stop and silent errors

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, ...
- Multi-level patterns for both fail-stop and silent errors

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, ...
- Multi-level patterns for both fail-stop and silent errors

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)
Conclusion

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

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

Application-Specific Fault Tolerance

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

General Purpose Fault Tolerance

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

Several challenging algorithmic/scheduling problems 😊
Bibliography

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

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


**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., CCPE 26(17), pp 2772-2791 (2014)
Bibliography

New Monograph, Springer Verlag 2015

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