Fault tolerance techniques for high-performance computing
Part 4

Anne Benoit
ENS Lyon
Anne.Benoit@ens-lyon.fr
http://graal.ens-lyon.fr/~abenoit

CR02 - 2016/2017
Outline

1. In-memory checkpointing

2. Probabilistic models for advanced methods
   - Failure prediction
   - Replication

3. Forward-recovery techniques
   - Introduction: Matrix-Matrix Multiplication
   - ABFT for Linear Algebra applications
   - Composite approach: ABFT & Checkpointing

4. Conclusion
Outline

1. In-memory checkpointing
2. Probabilistic models for advanced methods
3. Forward-recovery techniques
4. Conclusion
Outline

1. In-memory checkpointing
2. Probabilistic models for advanced methods
   - Failure prediction
   - Replication
3. Forward-recovery techniques
4. Conclusion
Outline

1. In-memory checkpointing

2. Probabilistic models for advanced methods
   - Failure prediction
   - Replication

3. Forward-recovery techniques

4. Conclusion
Outline

1. In-memory checkpointing

2. Probabilistic models for advanced methods
   - Failure prediction
   - Replication

3. Forward-recovery techniques

4. Conclusion
Replication

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

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

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

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

\[\uparrow\]

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

\(N\) bins, red and blue balls

Mean Number of Failures to Interruption (bring down application)

\(MNFTI = \) expected number of balls to throw until one bin gets one ball of each color
Theorem: $MNFTI = \mathbb{E}(NFTI|0)$ where

\[
\mathbb{E}(NFTI|n_f) = \begin{cases} 
2 & \text{if } n_f = N, \\
\frac{2N}{2N-n_f} + \frac{2N-2n_f}{2N-n_f} \mathbb{E}(NFTI|n_f + 1) & \text{otherwise.}
\end{cases}
\]

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

- application is still running
- failures have already hit $n_f$ different replica-groups

How do we obtain this result?
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
  \[ \text{Throughput}_{\text{Rep}} = N \left(1 - \sqrt{\frac{2C}{\mu_{\text{rep}}}}\right) \]
  \[ \mu_{\text{rep}} = MNFTI \times \mu_{2N} = MNFTI \times \frac{\mu}{2N} \]

- 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 $\mu_{ind} = 125$ 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. In-memory checkpointing
2. Probabilistic models for advanced methods
3. Forward-recovery techniques
   - Introduction: Matrix-Matrix Multiplication
   - ABFT for Linear Algebra applications
   - Composite approach: ABFT & Checkpointing
4. Conclusion
Outline

1. In-memory checkpointing
2. Probabilistic models for advanced methods
3. **Forward-recovery techniques**
   - Introduction: Matrix-Matrix Multiplication
   - ABFT for Linear Algebra applications
   - Composite approach: ABFT & Checkpointing
4. Conclusion
Generic vs. Application specific approaches

**Generic solutions**
- Universal
- Very low prerequisite
- One size fits all (pros and cons)

**Application specific solutions**
- Requires (deep) study of the application/algorithm
- Tailored solution: higher efficiency
Backward Recovery vs. 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, ...
Algorithm Based Fault Tolerance (ABFT)

**Principle**

- Limited to Linear Algebra computations
- Matrices are extended with rows and/or columns of checksums

\[
M = \begin{pmatrix}
5 & 1 & 7 & 13 \\
4 & 3 & 5 & 12 \\
4 & 6 & 9 & 19
\end{pmatrix}
\]
Algorithm Based Fault Tolerance (ABFT)

Principle

- Limited to Linear Algebra computations
- Matrices are extended with rows and/or columns of checksums

\[ M = \begin{pmatrix}
  5 & 1 & 7 & 13 \\
  4 & 3 & 5 & 12 \\
  4 & 6 & 9 & 19
\end{pmatrix} \]
Algorithm Based Fault Tolerance (ABFT)

Principle

- Limited to Linear Algebra computations
- Matrices are extended with rows and/or columns of checksums

\[ M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 4 & 3 & 5 & 12 \\ 4 & 6 & 9 & 19 \end{pmatrix} \]
ABFT and fail-stop errors

Missing checksum data

\[ M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 4 & 3 & 5 \\ 4 & 6 & 9 & 19 \end{pmatrix} \]

Simple recomputation: \( 4+3+5 = 12 \).

Missing original data

\[ M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 4 & 5 & 12 \\ 4 & 6 & 9 & 19 \end{pmatrix} \]

Simple recomputation: \( 12-(4+5) = 3 \).
ABFT and fail-stop errors

Missing checksum data

\[ M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 4 & 3 & 5 \\ 4 & 6 & 9 & 19 \end{pmatrix} \]

Simple recomputation: \( 4 + 3 + 5 = 12 \).

Missing original data

\[ M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 4 & 5 & 12 \\ 4 & 6 & 9 & 19 \end{pmatrix} \]

Simple recomputation: \( 12 - (4 + 5) = 3 \).
ABFT and fail-stop errors

Missing checksum data

\[
M = \begin{pmatrix}
5 & 1 & 7 & 13 \\
4 & 3 & 5 & \\
4 & 6 & 9 & 19
\end{pmatrix}
\]

Simple recomputation: \(4+3+5 = 12\).

Missing original data

\[
M = \begin{pmatrix}
5 & 1 & 7 & 13 \\
4 & 5 & 12 & \\
4 & 6 & 9 & 19
\end{pmatrix}
\]

Simple recomputation: \(12-(4+5) = 3\).
ABFT and fail-stop errors

Missing checksum data

\[
M = \begin{pmatrix}
5 & 1 & 7 & 13 \\
4 & 3 & 5 & \\
4 & 6 & 9 & 19
\end{pmatrix}
\]

Simple recomputation: \(4+3+5 = 12\).

Missing original data

\[
M = \begin{pmatrix}
5 & 1 & 7 & 13 \\
4 & 5 & 12 & \\
4 & 6 & 9 & 19
\end{pmatrix}
\]

Simple recomputation: \(12-(4+5) = 3\).
### ABFT and silent data corruption

$$M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 4 & 3 & 5 & 13 \\ 4 & 6 & 9 & 19 \end{pmatrix}$$

Error detection: $4 + 3 + 5 \neq 13$

#### Limitations

- The following matrix would have successfully passed the sanity check:

$$M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 5 & 3 & 5 & 13 \\ 4 & 6 & 9 & 19 \end{pmatrix}$$

- Can detect **one** error and correct **zero**.
ABFT and silent data corruption

\[ M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 4 & 3 & 5 & 13 \\ 4 & 6 & 9 & 19 \end{pmatrix} \]

Error detection: \( 4 + 3 + 5 \neq 13 \)

Limitations

- The following matrix would have successfully passed the sanity check:

\[ M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 5 & 3 & 5 & 13 \\ 4 & 6 & 9 & 19 \end{pmatrix} \]

- Can detect one error and correct zero.
ABFT and silent data corruption

One row and one column of checksums

\[
M = \begin{pmatrix}
5 & 1 & 7 & 13 \\
4 & 3 & 5 & 11 \\
4 & 6 & 9 & 19 \\
13 & 9 & 21 & 43 \\
\end{pmatrix}
\]

Checksum recomputation to look for silent data corruptions:

\[
\begin{align*}
5 + 1 + 7 &= 13 \\
4 + 3 + 5 &= 12 \\
4 + 6 + 9 &= 19 \\
13 + 10 + 21 &= 44 \\
\end{align*}
\]

Checksums do not match!
ABFT and silent data corruption

One row and one column of checksums

\[ M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 4 & 3 & 5 & 11 \\ 4 & 6 & 9 & 19 \\ 13 & 9 & 21 & 43 \end{pmatrix} \]

Checksum recomputation to look for silent data corruptions:

\[
\begin{align*}
5 + 1 + 7 &= 13 \\
4 + 3 + 5 &= 12 \\
4 + 6 + 9 &= 19 \\
13 + 10 + 21 &= 44 
\end{align*}
\]

Checksums do not match!
ABFT and silent data corruption

\[ M = \begin{pmatrix}
5 & 1 & 7 & 13 \\
4 & 3 & 5 & 11 \\
4 & 6 & 9 & 19 \\
13 & 9 & 21 & 43
\end{pmatrix} \begin{pmatrix}
5 + 1 + 7 = 13 \\
4 + 3 + 5 = 12 \\
4 + 6 + 9 = 19 \\
13 + 10 + 21 = 44
\end{pmatrix} \]

Both checksums are affected, giving out the location of the error.
We solve:

\begin{align*}
4 + x + 5 &= 11 \\
1 + x + 6 &= 9
\end{align*}

Recomputing the checksums we find that:

\begin{align*}
5 + 1 + 7 &= 13 \\
4 + 2 + 5 &= 11 \\
4 + 6 + 9 &= 19 \\
13 + 9 + 21 &= 43
\end{align*}

Checksums match 😊

Can detect two errors and correct one
ABFT and silent data corruption

\[ M = \begin{pmatrix}
5 & 1 & 7 & 13 \\
4 & 3 & 5 & 11 \\
4 & 6 & 9 & 19 \\
13 & 9 & 21 & 43
\end{pmatrix} \quad \begin{pmatrix}
5 + 1 + 7 = 13 \\
4 + 3 + 5 = 12 \\
4 + 6 + 9 = 19 \\
13 + 10 + 21 = 44
\end{pmatrix} \]

Both checksums are affected, giving out the location of the error.
We solve:

\[
4 + x + 5 = 11 \\
1 + x + 6 = 9
\]

Recomputing the checksums we find that:
\[
\begin{pmatrix}
5 + 1 + 7 = 13 \\
4 + 2 + 5 = 11 \\
4 + 6 + 9 = 19 \\
13 + 9 + 21 = 43
\end{pmatrix}
\]

Checksums match 😊
Can detect two errors and correct one
ABFT and silent data corruption

\[ M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 4 & 3 & 5 & 11 \\ 4 & 6 & 9 & 19 \\ 13 & 9 & 21 & 43 \end{pmatrix} \begin{pmatrix} 5 + 1 + 7 = 13 \\ 4 + 3 + 5 = 12 \\ 4 + 6 + 9 = 19 \\ 13 + 10 + 21 = 44 \end{pmatrix} \]

Both checksums are affected, giving out the location of the error.

We solve:

\[ 4 + x + 5 = 11 \quad 1 + x + 6 = 9 \]

Recomputing the checksums we find that:

\[ \begin{pmatrix} 5 + 1 + 7 = 13 \\ 4 + 2 + 5 = 11 \\ 4 + 6 + 9 = 19 \\ 13 + 9 + 21 = 43 \end{pmatrix} \]

Checksums match 😊

Can detect two errors and correct one
ABFT for Matrix-Matrix multiplication

**Aim:** Computation of $C = A \times B$

Let $e^T = [1, 1, \cdots, 1]$, we define

$$A^c := \begin{pmatrix} A \\ e^T A \end{pmatrix}, \quad B^r := \begin{pmatrix} B & Be \end{pmatrix}, \quad C^f := \begin{pmatrix} C & Ce \\ e^T C & e^T Ce \end{pmatrix}.$$ 

Where $A^c$ is the **column checksum matrix**, $B^r$ is the **row checksum matrix** and $C^f$ is the **full checksum matrix**.

$$A^c \times B^r = \begin{pmatrix} A \\ e^T A \end{pmatrix} \times \begin{pmatrix} B & Be \end{pmatrix} = \begin{pmatrix} AB & ABe \\ e^T AB & e^T ABe \end{pmatrix} = \begin{pmatrix} C & Ce \\ e^T C & e^T Ce \end{pmatrix} = C^f$$
ABFT for Matrix-Matrix multiplication

**Aim:** Computation of $C = A \times B$

Let $e^T = [1, 1, \cdots, 1]$, we define

$$A^c := \begin{pmatrix} A \\ e^T A \end{pmatrix}, \quad B^r := (B \quad Be), \quad C^f := \begin{pmatrix} C \\ e^T C \\ e^T Ce \end{pmatrix}.$$ 

Where $A^c$ is the *column checksum matrix*, $B^r$ is the *row checksum matrix* and $C^f$ is the *full checksum matrix*.

$$A^c \times B^r = \begin{pmatrix} A \\ e^T A \end{pmatrix} \times (B \quad Be) = \begin{pmatrix} AB \\ e^T AB \\ e^T ABe \end{pmatrix} = \begin{pmatrix} C \\ e^T C \\ e^T Ce \end{pmatrix} = C^f$$
In practice... things are more complicated!

- When do errors strike? Are all data always protected?
- Computations are approximate because of floating-point rounding
- Error detection and error correction capabilities depend on the number of checksum rows and columns
Outline

1. In-memory checkpointing
2. Probabilistic models for advanced methods
3. Forward-recovery techniques
   - Introduction: Matrix-Matrix Multiplication
   - ABFT for Linear Algebra applications
   - Composite approach: ABFT & Checkpointing
4. Conclusion
Example: block LU factorization

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

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

Failure of rank 2
Algorithm Based Fault Tolerant LU decomposition

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

- **Checksum**: invertible operation on the data of the row / column
- Checksum replication can be avoided by dedicating computing resources to checksum storage
Algorithm Based Fault Tolerant LU decomposition

- Checkpoint the next set of Q-Panels to be able to return to it in case of failures
Algorithm Based Fault Tolerant LU decomposition

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

For the part of the data that is not updated this way, the checksum must be re-calculated.
To avoid slowing down all processors and panel operation, group checksum updates every $Q$ block columns.
Algorithm Based Fault Tolerant LU decomposition

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

To avoid slowing down all processors and panel operation, group checksum updates every $Q$ block columns
Then, update the missing coverage. Keep checkpoint block column to cover failures during that time.
Algorithm Based Fault Tolerant LU decomposition

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

In case of failure, conclude the operation, then

- Missing Checksum = Sum(Existing Data)
Failure inside a $Q$–panel factorization

- Failures may happen while inside a $Q$–panel factorization.
Valid Checksum Information allows to recover most of the missing data, but not all: the checksum for the current $Q-$panels are not valid.
We use the checkpoint to restore the $Q$–panel in its initial state.
Failure inside a $Q$–panel factorization

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

As supercomputers grow ever larger in scale, the Mean Time to Failure becomes shorter and shorter, making the complete and successful execution of complex applications more and more difficult. FT-LA delivers a new approach, utilizing Algorithm-Based Fault Tolerance (ABFT), to help factorization algorithms survive fail-stop failures. The FT-LA software package extends ScaLAPACK with ABFT routines, and in sharp contrast with legacy checkpoint-based approaches, ABFT does not incur I/O overhead, and promises a much more scalable protection scheme.

**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

**OVERHEADS**

- Process grid: $P \times Q$
  - $F$: simultaneous failures tolerated
  - Protection against 2 faults on $192 \times 192$ processes => 1% overhead
- Usually $F \ll Q$;
  - Overheads in $F/\sqrt{Q}$
- Protection cost is inversely proportional to machine scale!

**COMPUTATION**

- Flops for the checksum update
- Matrix is extended with $2F$ columns every $Q$ columns

**MEMORY**

**FIND OUT MORE AT**

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

**PERFORMANCE ON KRAKEN**

<table>
<thead>
<tr>
<th>#Processors (P x Q grid); Matrix size (N)</th>
<th>Performance (TFlop/s)</th>
<th>Relative Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6x6; 20k</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12x12; 40k</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24x24; 80k</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48x48; 160k</td>
<td></td>
<td></td>
</tr>
<tr>
<td>96x96; 320k</td>
<td></td>
<td></td>
</tr>
<tr>
<td>192x192; 640k</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MPI-Next ULFM Performance**

- Open MPI with ULFM; Kraken supercomputer;

Anne.Benoit@ens-lyon.fr  CR02  Fault tolerance (3)
ABFT QR decomposition: performance

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

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

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

5.3 Checkpoint-on-Failure QR Performance

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

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

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

1. In-memory checkpointing
2. Probabilistic models for advanced methods
3. Forward-recovery techniques
   - Introduction: Matrix-Matrix Multiplication
   - ABFT for Linear Algebra applications
   - Composite approach: ABFT & Checkpointing
4. Conclusion
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
**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

Anne.Benoit@ens-lyon.fr

CR02

Fault tolerance (3)
Typical Application

```c
for ( aninsanenumbe r ) { 
    /* 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

Anne.Benoit@ens-lyon.fr
Problem Statement

```c
for ( a in aSet ) {
    /* Extract data from simulation, fill up matrix */
    sim2mat();
    /* Factorize matrix, solve */
   dgeqrf();
    dsolve();
    /* 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
ABFT & Periodic Ckpt: no failure
ABFT & Periodic Checkpoint: failure during Library phase

- Process 0
  - Application
  - Library

- Process 1
  - Application
  - Library

- Process 2
  - Application
  - Library

Failure (during Library)

Rollback (partial)

Recovery

ABFT Recovery
ABFT\&PeriodicCkpt: failure during General phase

Process 0
Process 1
Process 2
Application
Library
Application
Library
Application
Library

Failure (during General)
Rollback (full)
Recovery
If the duration of the **GENERAL** phase is too small: don’t add checkpoints

If the duration of the **LIBRARY** phase is too small: don’t do ABFT recovery, remain in **GENERAL** mode

- this assumes a performance model for the library call
ABFT&PeriodicCkpt: Optimizations

- If the duration of the **GENERAL** phase is too small: don’t add checkpoints
- If the duration of the **LIBRARY** phase is too small: don’t do ABFT recovery, remain in **GENERAL** mode
  - this assumes a performance model for the library call
A few notations

Process 0
Process 1
Process 2

Application
Library
Application
Library
Application
Library

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
\( T^{\text{ff}}, T^{\text{ff}}, T^{\text{ff}} \): “Fault Free” times
\( t^{\text{lost}}_G, t^{\text{lost}}_L \): Lost time (recovery overloads)
\( T^{\text{final}}_G, T^{\text{final}}_L \): Total times (with faults)
A few notations

Costs

\[ C_L = \rho C: \text{time to take a checkpoint of the Library data set} \]

\[ C_L = (1 - \rho)C: \text{time to take a checkpoint of the General data set} \]

\[ R, R_L: \text{time to load a full General data set checkpoint} \]

\[ D: \text{down time (time to allocate a new machine / reboot)} \]

\[ \text{Recons}_{ABFT}: \text{time to apply the ABFT recovery} \]

\[ \phi: \text{Slowdown factor on the Library phase, when applying ABFT} \]
Time (with overheads) of \textit{Library} phase is constant (in $P_G$):

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

Time (with overheads) of \textit{General} phase accepts two cases:

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

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

\[ WASTE = 1 - \frac{T_0}{T_{G_{\text{final}}} + T_{L_{\text{final}}}} \]
Toward Exascale, and beyond!

Let's think at scale

- Number of components $\uparrow \Rightarrow$ 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 #1

Weak Scale Scenario #1

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

- $\mu$ at $n = 10^5$: 1 day, is in $O(\frac{1}{n})$
- $C (= R)$ at $n = 10^5$, is 1 minute, is in $O(n)$
- $\alpha$ 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 (3)
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

Anne.Benoit@ens-lyon.fr CR02  Fault tolerance (3)
Weak Scale Scenario #3

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

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

![Graph showing faulty nodes over different scales with legend for PeriodicCkpt, Bi-PeriodicCkpt, and ABFT PeriodicCkpt.]

Waste Nodes

- PeriodicCkpt
- Bi-PeriodicCkpt
- ABFT PeriodicCkpt

Conclusions and References

Anne.Benoit@ens-lyon.fr  CR02  Fault tolerance (3)
Outline

1. In-memory checkpointing
2. Probabilistic models for advanced methods
3. Forward-recovery techniques
4. Conclusion
Resilient research on resilience

Models needed to assess techniques at scale
without bias 😊
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 😊
Exascale
- Toward Exascale Resilience, Cappello F. et al., IJHPCA 23, 4 (2009)
- The International Exascale Software Roadmap, Dongarra, J., Beckman, P. et al., IJHPCA 25, 1 (2011)

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


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

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

New Monograph, Springer Verlag 2015

Thanks Yves Robert, Thomas Hérault, George Bosilca, Aurélien Bouteiller and Frédéric Vivien for the slides (SC'15 tutorial, JLESC’16 summer school)