Algorithm-Based Fault Tolerance

Thomas Hérault\textsuperscript{1}, Yves Robert\textsuperscript{1,2} & Frédéric Vivien\textsuperscript{2}

\textsuperscript{1} – University of Tennessee Knoxville, USA
\textsuperscript{2} – ENS Lyon & INRIA, France

frederic.vivien@inria.fr

3\textsuperscript{rd} JLESC Summer School
Outline

1. Introduction: Matrix-Matrix Multiplication
2. ABFT for block LU factorization
3. Composite approach: ABFT & Checkpointing
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction: Matrix-Matrix Multiplication</td>
</tr>
<tr>
<td>2</td>
<td>ABFT for block LU factorization</td>
</tr>
<tr>
<td>3</td>
<td>Composite approach: ABFT &amp; Checkpointing</td>
</tr>
</tbody>
</table>
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

*KEEP CALM AND DO IT AGAIN*
Backward Recovery vs. Forward Recovery

**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
Algorithm Based Fault Tolerance (ABFT)

Principle

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

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

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

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

\[ M = \begin{pmatrix} 5 & 1 & 7 & 13 \\ 4 & 3 & 5 & 13 \\ 4 & 6 & 9 & 19 \end{pmatrix} \]
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 \textbf{one} error and correct \textbf{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:

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

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.
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:

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

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

Where \( A^c \) is the *column checksum matrix*, \( B^r \) is the *row checksum matrix* and \( C^f \) is the *full checksum matrix*. 
Aim: Computation of $C = A \times B$

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

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

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

$$A^c \times B^r = \begin{pmatrix} A \\ e^T A \end{pmatrix} \times (B \quad Be)$$

$$= \begin{pmatrix} AB & ABe \\ e^T AB & e^T ABe \end{pmatrix} = \begin{pmatrix} C & Ce \\ e^T C & e^T Ce \end{pmatrix} = C^f$$
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. Introduction: Matrix-Matrix Multiplication
2. ABFT for block LU factorization
3. Composite approach: ABFT & Checkpointing
Block LU factorization

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

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

- 2D Block Cyclic Distribution (here $2 \times 3$)
- A single failure $\Rightarrow$ many data lost
Algorithm Based Fault Tolerant LU decomposition

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

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

- 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.
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
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

+ Then, update the missing coverage.
  Keep checkpoint block column to cover failures during that time
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

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

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

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

Thomas Hérault, Yves Robert, and Frédéric Vivien
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.

Thomas Hérault, Yves Robert, and Frédéric Vivien
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
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

**PERFORMANCE ON KRAKEN**

Open MPI with ULFM; Kraken supercomputer.
Fig. 11. Weak scalability of FT-QR: run time overhead on Kraken when failures strike.

Local snapshots have to be used along with re-factorization to recover the lost data and restore the matrix state. This is referred to as the “failure within Q panels.” Figure 10 shows the overhead from these two cases for the LU factorization, along with the no-error overhead as a reference. In the “border” case, the failure is simulated to strike when the 96th panel (which is a multiple of grid columns, 6, 12, 24, ..., 48) has just finished. In the “non-border” case, failure occurs during the \((Q+2)\)th panel factorization. For example, when \(Q = 12\), the failure is injected when the trailing update for the step with panel \((1301,1301)\) finishes. From the result in Figure 10, the recovery procedure in both cases adds a small overhead that also decreases when scaled to large problem size and process grid. For largest setups, only 2-3 percent of the execution time is spent recovering from a failure.

7.4. Extension to Other factorization

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

Checkpoint on Failure - MPI Implementation

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

5.3 Checkpoint-on-Failure QR Performance

Supercomputer Performance:

Figure 2 presents the performance on the Kraken supercomputer. The process grid is 24 \times 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 \times 16 process grid. Although a smaller test platform, the Dancer cluster features local storage on nodes and a variety of performance analysis tools unavailable on Kraken. As expected (see [12]), the ABFT method has a higher relative cost on this smaller machine. Compared to the Kraken platform, the relative cost of CoF failure recovery is smaller on Dancer. The CoF protocol incurs disk accesses to store and load checkpoints when a failure hits, hence the recovery overhead depends on I/O performance. By breaking down the relative cost of each recovery step in CoF, Figure 4 shows that checkpoint saving and loading only take a small percentage of the total run-time, thanks to the availability of solid state disks on every node. Since checkpoint reloading immediately follows checkpointing, the OS cache satisfy most disk access, resulting in high I/O performance. For matrices larger than...
Outline

1. Introduction: Matrix-Matrix Multiplication
2. ABFT for block LU factorization
3. Composite approach: ABFT & Checkpointing
Fault Tolerance Techniques

General Techniques
- Replication
- Rollback Recovery
  - Coordinated Checkpointing
  - Uncoordinated Checkpointing & Message Logging
  - Hierarchical Checkpointing
  - Multilevel 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
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
**Problem Statement**

Typical Application

```c
for ( int i = 0; i < n; ++i ) {
    /* Extract data from simulation, fill up matrix */
    sim2mat();
    /* Factorize matrix, Solve */
    dgeqrf();
    dgsolve();
    /* 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 Checkpoint

ABFT & Periodic Checkpoint: no failure
ABFT\&PeriodicCkpt: failure during Library phase

Process 0

Process 1

Process 2

Failure (during Library)

Rollback (partial)

Recovery

ABFT Recovery

Application

Library

Application

Library

Application

Library

Thomas Héralt, Yves Robert, and Frédéric Vivien
ABFT&PeriodicCkpt: failure during General phase

Process 0
- Application
- Library

Process 1
- Application
- Library

Process 2
- Application
- Library

Failure (during General)
Rollback (full)
Recovery
ABFT&PeriodicCkpt: Optimizations

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

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

Times, Periods

- $T_0$: Duration of an Epoch (without FT)
- $T_L = \alpha T_0$: Time spent in the Library phase
- $T_G = (1 - \alpha) T_0$: Time spent in the General phase
- $P_G$: Periodic Checkpointing Period
- $T^{ff}, T^{ff}, T^{ff}$: “Fault Free” times
- $t^{lost}_G, t^{lost}_L$: Lost time (recovery overheads)
- $T^{final}_G, T^{final}_L$: Total times (with faults)
A few notations

Costs

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

**GENERAL phase**

Without Failures

\[
T^{\text{ff}}_G = \begin{cases} 
T_G + \frac{C_L}{P_G - C} \times P_G & \text{if } T_G < P_G \\
T_G + C_L \times P_G & \text{if } T_G \geq P_G 
\end{cases}
\]
Without Failures

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

**GENERAL phase**

- Process 0
- Process 1
- Process 2

Failure (during GENERAL)

Rollback (full)

Recovery

---

**Failure Overhead**

\[
\begin{align*}
    t_{G}^{\text{lost}} &= \begin{cases} 
    D + R + \frac{T_{G}^{\text{ff}}}{2} & \text{if } T_{G} < P_{G} \\
    D + R + \frac{P_{G}}{2} & \text{if } T_{G} \geq P_{G}
    \end{cases}
\end{align*}
\]
**Library phase, failure overhead**

### Library phase

- Process 0
- Process 1
- Process 2

**Failure** (during LIBRARY)

**Rollback** (partial)

**Recovery**

**ABFT**

**Recovery**

**Failure Overhead**

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

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

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

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

Which is minimal in the second case, if

\[
P_G = \sqrt{2C(\mu - D - R)}
\]
Waste

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

$$\text{WASTE} = 1 - \frac{T_0}{T_G^{\text{final}} + T_L^{\text{final}}}$$
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 & Periodic Ckpt 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 $O(1/\sqrt{n})$
- $C$ ($=R$) 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

<table>
<thead>
<tr>
<th># Faults</th>
<th>PeriodicCkpt</th>
<th>Bi-PeriodicCkpt</th>
<th>ABFT PeriodicCkpt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Chart

- **Waste**
  - PeriodicCkpt
  - Bi-PeriodicCkpt
  - ABFT PeriodicCkpt

- **Nodes**
  - X-axis: 1k, 10k, 100k, 1M

---

Thomas Hérault, Yves Robert, and Frédéric Vivien
Weak Scale #2

**Weak Scale Scenario #2**

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

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

![Graph showing the relationship between the number of faults and the waste ratio for different checkpointing and ABFT methods. The x-axis represents the number of nodes, and the y-axis represents the waste ratio. The graph compares PeriodicCheckpoint, Bi-PeriodicCheckpoint, and ABFT PeriodicCheckpoint methods.]
Weak Scale #3

<table>
<thead>
<tr>
<th>Weak Scale Scenario #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of components, $n$, increase</td>
</tr>
<tr>
<td>Memory per component remains constant</td>
</tr>
<tr>
<td>Problem Size increases in $O(\sqrt{n})$ (e.g. matrix operation)</td>
</tr>
</tbody>
</table>

- $\mu$ 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

<table>
<thead>
<tr>
<th>Nb Faults</th>
<th>Periodic Ckpt</th>
<th>Bi-Periodic Ckpt</th>
<th>ABFT Periodic Ckpt</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>

Waste Nodes:
- Periodic Ckpt
- Bi-Periodic Ckpt
- ABFT Periodic Ckpt

Thomas Hérault, Yves Robert, and Frédéric Vivien
Conclusion

Algorithm-Based Fault Tolerance

- Application-specific solution for linear algebra kernels
- Low-overhead forward-recovery solution
- Used alone or in conjunction with backward-recovery solutions

Going further

- **Algorithm-Based Fault Tolerance for Dense Matrix Factorizations, Multiple Failures and Accuracy.** A. Bouteiller, Th. Herault, G. Bosilca, P. Du, J. Dongarra. ACM Transactions on Parallel Computing 1(2), 2015.

http://www.netlib.org/lapack/lawnspdf/lawn289.pdf