An overview of fault-tolerant techniques for HPC

Thomas Hérault[1] & Yves Robert[1,2]

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

herault@icl.utk.edu | yves.robert@ens-lyon.fr

http://graal.ens-lyon.fr/~yrobert/sc13tutorial.pdf

SC'2013 Tutorial

Thanks

INRIA & ENS Lyon

- Anne Benoit
- Frédéric Vivien
- PhD students (Guillaume Aupy, Dounia Zaidouni)

UT Knoxville

- George Bosilca
- Aurélien Bouteiller
- Jack Dongarra

Others

- Franck Cappello, Argonne and UIUC-Inria joint lab
- Henri Casanova, Univ. Hawai'i
- Amina Guermouche, UIUC-Inria joint lab

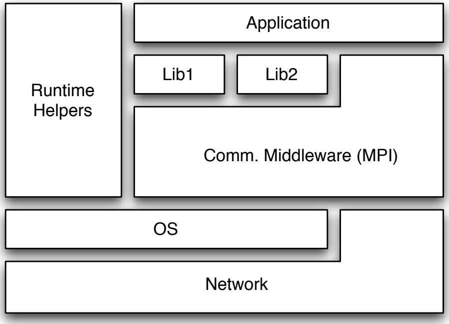
Outline

1. Application-specific fault-tolerance techniques (45mn)
   - Fault-Tolerant Middleware
   - Bags of tasks
   - Iterative algorithms and fixed-point convergence
   - ABFT for Linear Algebra applications
   - Composite approach: ABFT & Checkpointing

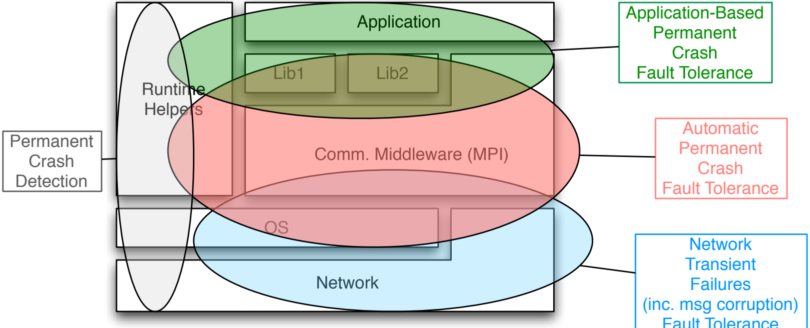
Outline

1. Application-specific fault-tolerance techniques (45mn)
   - Fault-Tolerant Middleware
   - Bags of tasks
   - Iterative algorithms and fixed-point convergence
   - ABFT for Linear Algebra applications
   - Composite approach: ABFT & Checkpointing

Fault Tolerance Software Stack



Fault Tolerance Software Stack



Motivation

Motivation

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

Outline

HPC – MPI

HPC

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

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

HPC – MPI

HPC

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

  > *This document does not specify the state of a computation after an erroneous MPI call has occurred.*
  >
  > – MPI Standard 3.0, p. 21, l. 24:25

HPC – MPI

MPI Implementations

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

FT Middleware in HPC

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

MPI-Next-FT proposal: ULFM

Goal

Resume Communication Capability for MPI (and nothing more)

- Failure Reporting
- Failure notification propagation / Distributed State reconciliation

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

- Recovery
- Termination

MPI-Next-FT proposal: ULFM

Goal

Resume Communication Capability for MPI (and nothing more)

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

New APIs

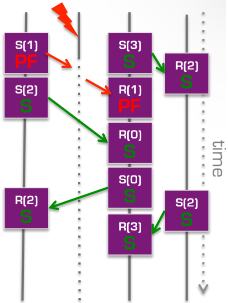
- REVOKE allows to resolve non-uniform error status
- SHRINK allows to rebuild error-free communicators
- AGREE allows to quit a communication pattern knowing it is fully complete

MPI-Next-FT proposal: ULFM

Errors are visible only for operations that cannot complete

Error Reporting

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

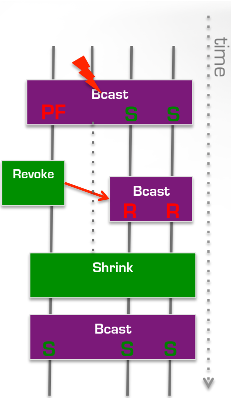


MPI-Next-FT proposal: ULFM

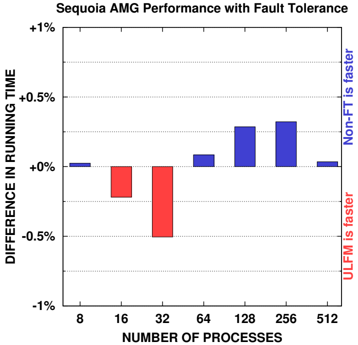
Inconsistent Global State and Resolution

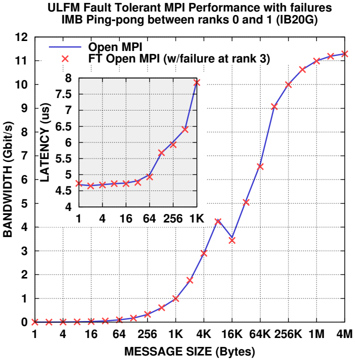
Error Reporting

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



MPI-Next-FT proposal: ULFM



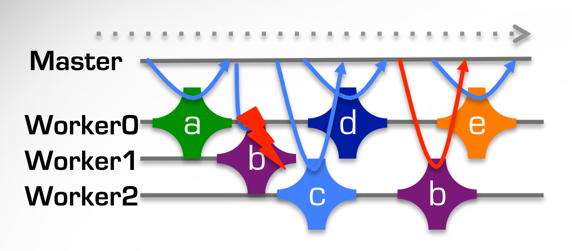


Open MPI - ULFM support

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

Outline

Master/Worker



Worker

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

Master/Worker

Master

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

FT Master

Fault Tolerant Master

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

FT Master

Fault Tolerant Master

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

FT Master

Fault Tolerant Master

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

FT Master

Fault Tolerant Master

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

FT Master

Fault Tolerant Master

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

Outline

Iterative Algorithm

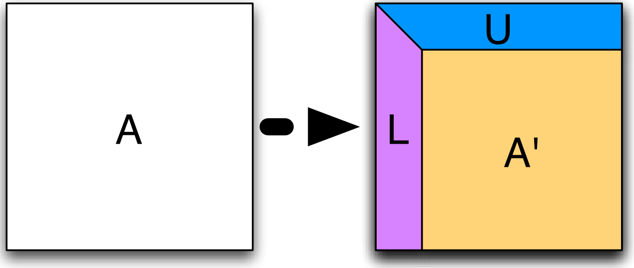
```
while( gnorm > epsilon ) {
     iterate();
     compute_norm(&lnorm);
     rc = MPI_Allreduce( &lnorm, &gnorm, 1,
                          MPI_DOUBLE, MPI_MAX, comm);
     if( (MPI_ERR_PROC_FAILED == rc) ||
         (MPI_ERR_COMM_REVOKED == rc) ||
         (gnorm <= epsilon) ) {

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

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

Iterative Algorithm

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

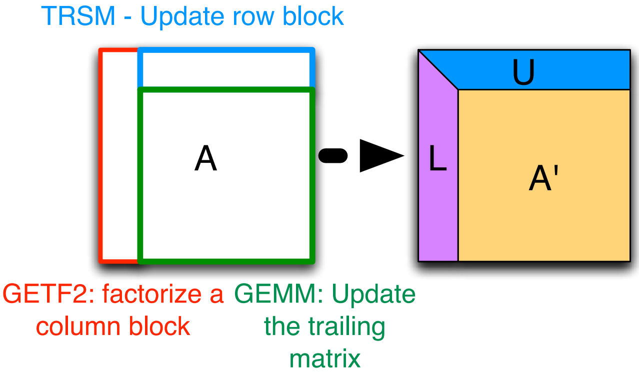
Outline

Example: block LU/QR factorization



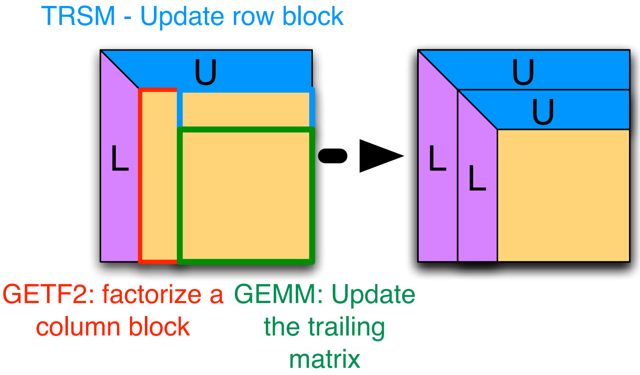
- Solve $A \cdot x = b$ (hard)
- Transform A into a LU factorization
- Solve $L \cdot y = b$, then $U \cdot x = y$

Example: block LU/QR factorization



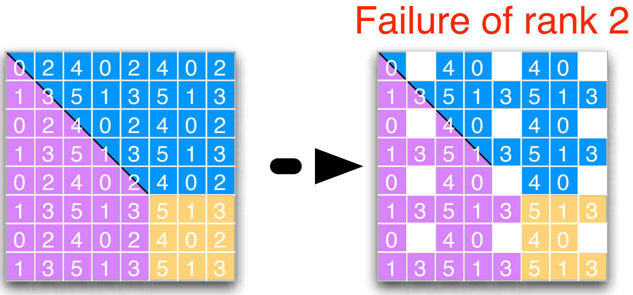
- Solve $A \cdot x = b$ (hard)
- Transform A into a LU factorization
- Solve $L \cdot y = b$, then $U \cdot x = y$

Example: block LU/QR factorization



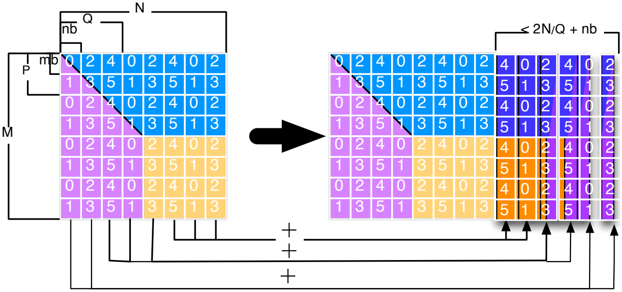
- Solve $A \cdot x = b$ (hard)
- Transform A into a LU factorization
- Solve $L \cdot y = b$, then $U \cdot x = y$

Example: block LU/QR factorization



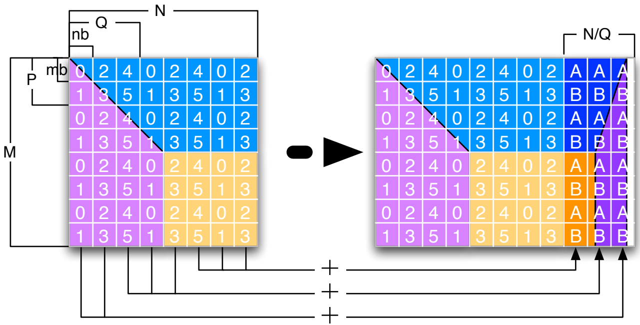
- 2D Block Cyclic Distribution (here 2×3)
- A single failure $\Rightarrow$ many data lost

Algorithm Based Fault Tolerant LU decomposition



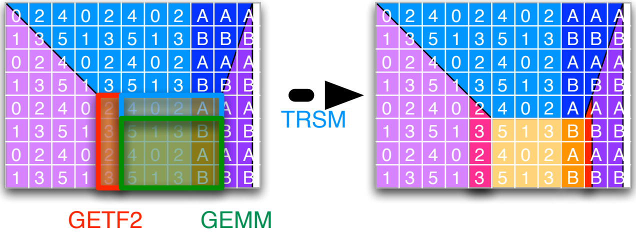
- Checksum: invertible operation on the data of the row / column
  - Checksum blocks are doubled, to allow recovery when data and checksum are lost together

Algorithm Based Fault Tolerant LU decomposition



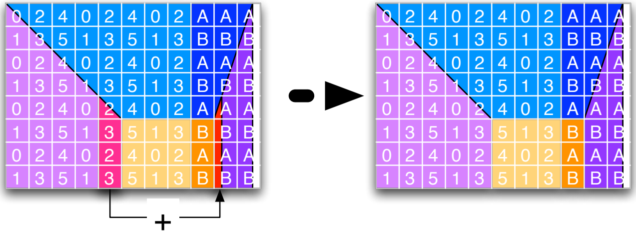
- Checksum: invertible operation on the data of the row / column
  - Checksum replication can be avoided by dedicating computing resources to checksum storage

Algorithm Based Fault Tolerant LU decomposition



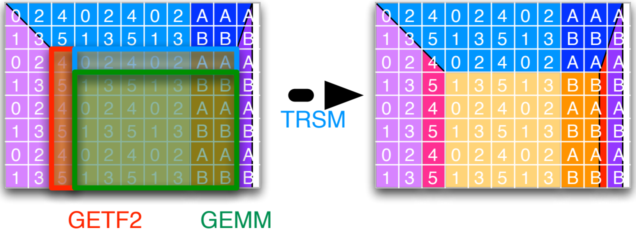
- Checksum: invertible operation on the data of the row / column
    - Idea of ABFT: applying the operation on data and checksum preserves the checksum properties

Algorithm Based Fault Tolerant LU decomposition



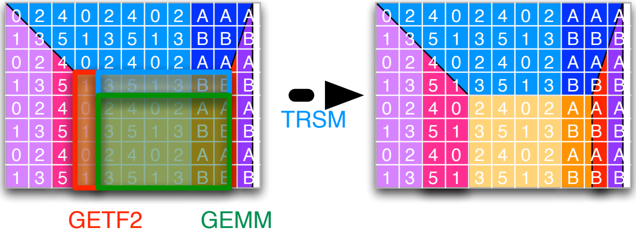
- Checksum: invertible operation on the data of the row / column
  - For the part of the data that is not updated this way, the checksum must be re-calculated

Algorithm Based Fault Tolerant LU decomposition



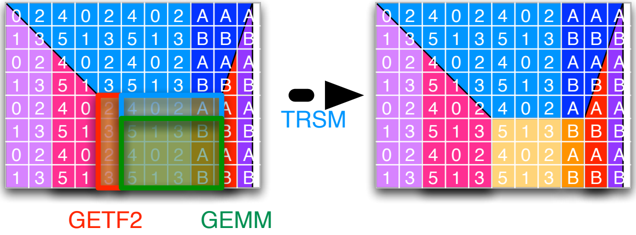
- Checksum: invertible operation on the data of the row / column
  - To avoid slowing down all processors and panel operation, group checksum updates every q block columns

Algorithm Based Fault Tolerant LU decomposition



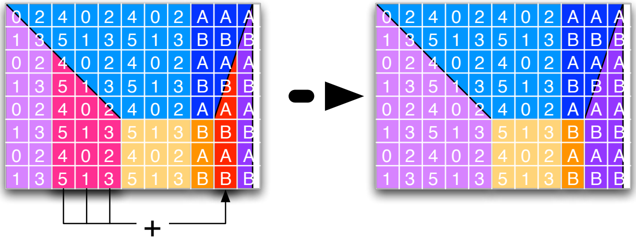
- Checksum: invertible operation on the data of the row / column
  - To avoid slowing down all processors and panel operation, group checksum updates every q block columns

Algorithm Based Fault Tolerant LU decomposition



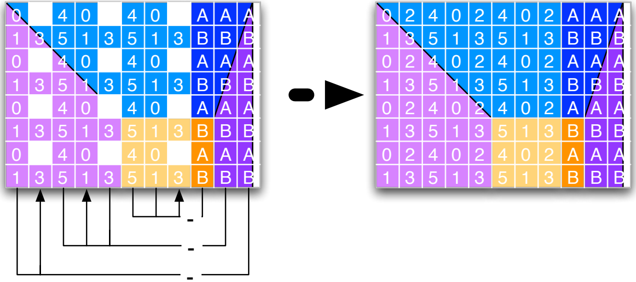
- Checksum: invertible operation on the data of the row / column
  - To avoid slowing down all processors and panel operation, group checksum updates every q block columns

Algorithm Based Fault Tolerant LU decomposition



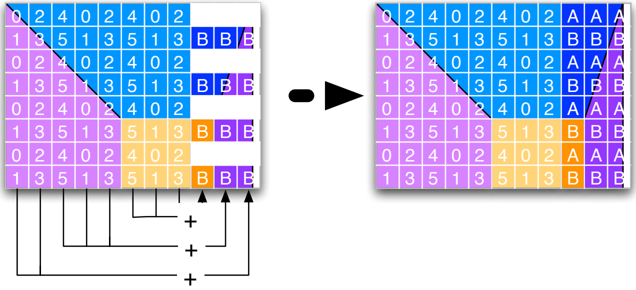
- Checksum: invertible operation on the data of the row / column
    - Then, update the missing coverage. Keep checkpoint block column to cover failures during that time

Algorithm Based Fault Tolerant LU decomposition



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

Algorithm Based Fault Tolerant LU decomposition



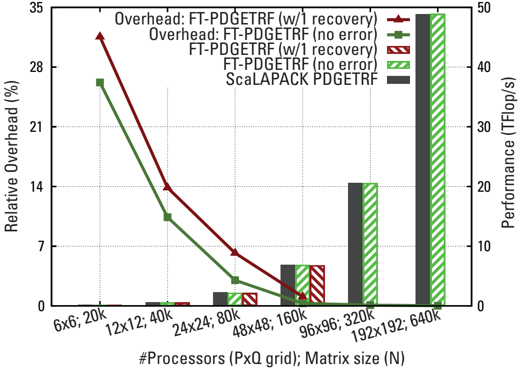
- In case of failure, conclude the operation, then
  - Missing Checksum = Sum(Existing Data)s

ABFT LU decomposition: implementation

MPI Implementation

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

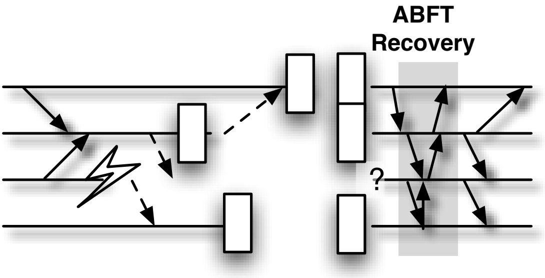
ABFT LU decomposition: performance



MPI-Next ULFM Performance

- Open MPI with ULFM; Kraken supercomputer;

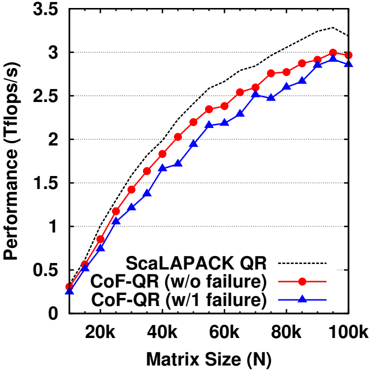
ABFT LU decomposition: implementation



Checkpoint on Failure - MPI Implementation

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

ABFT QR decomposition: performance



Checkpoint on Failure - MPI Performance

- Open MPI; Kraken supercomputer;

Outline

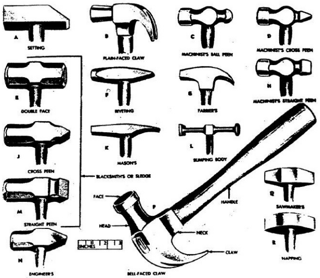
Fault Tolerance Techniques

General Techniques

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

Application-Specific Techniques

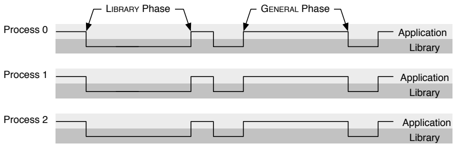
- Algorithm Based Fault Tolerance (ABFT)
- Iterative Convergence
- Approximated Computation



Application

Typical Application

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

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

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



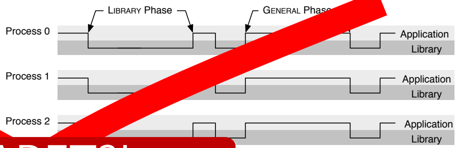
Characteristics

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

Application

Typical Applicatio

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

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

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



Goodbye ABFT?!

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

Application

Problem Statement

How to use fault tolerant operations$^{(*)}$ within a non-fault tolerant$^{(**)}$ application?$^{(***)}$

(*) ABFT, or other application-specific FT
(**) Or within an application that does not have the same kind of FT
(***) And keep the application globally fault tolerant...

Typica

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

  /* I
   * S
  dge
  dsol

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

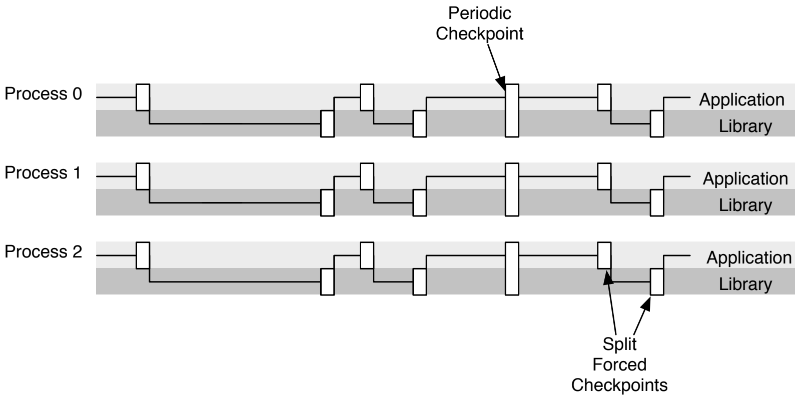
- Application
  Library
- Application
  Library
- Application
  Library

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

☹ modify data not covered by ABFT algorithms

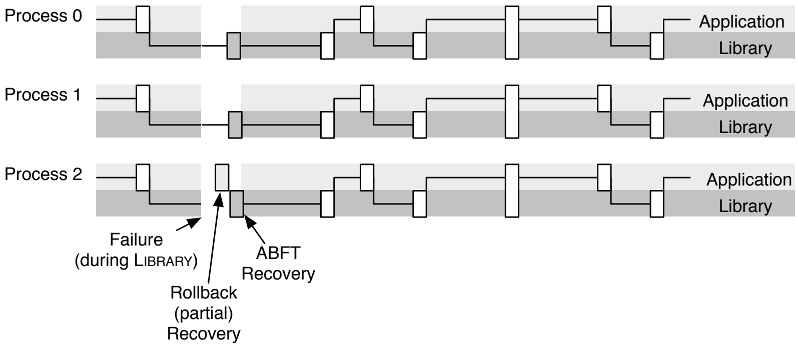
ABFT&PeriodicCkpt

ABFT&PeriodicCkpt: no failure



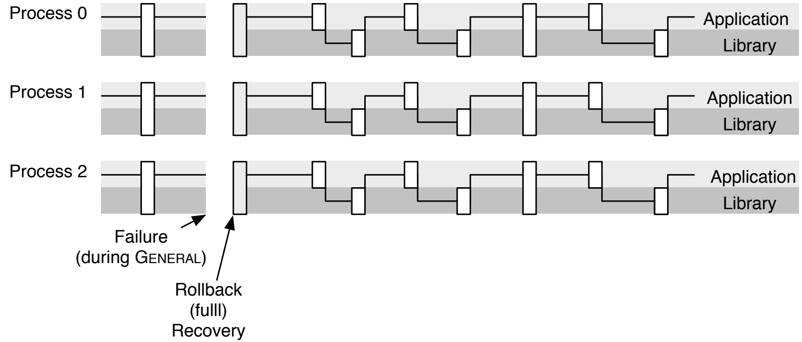
ABFT&PeriodicCkpt

ABFT&PeriodicCkpt: failure during Library phase

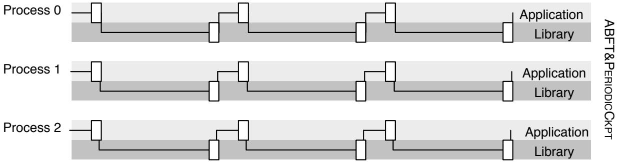


ABFT&PeriodicCkpt

ABFT&PeriodicCkpt: failure during General phase



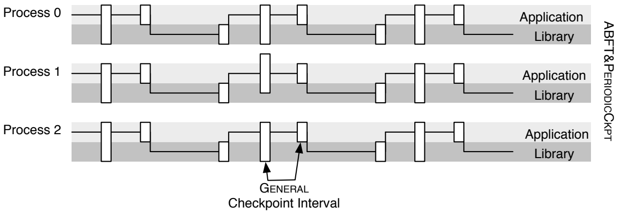
ABFT&PeriodicCkpt: Optimizations



ABFT&PeriodicCkpt: Optimizations

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

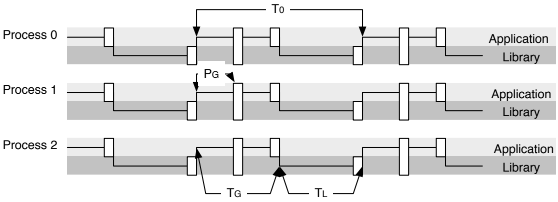
ABFT&PeriodicCkpt: Optimizations



ABFT&PeriodicCkpt: Optimizations

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

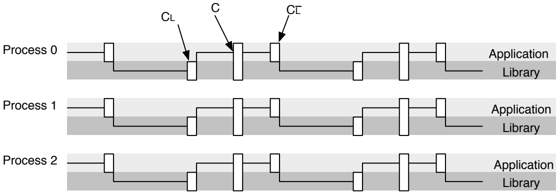
A few notations



Times, Periods

T_0: Duration of an Epoch (without FT)
$T_L = \alpha T_0$: Time spent in the LIBRARY phase
$T_G = (1 - \alpha) T_0$: Time spent in the GENERAL phase
P_G: Periodic Checkpointing Period
$T^{\text{ff}}, T_G^{\text{ff}}, T_L^{\text{ff}}$: "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

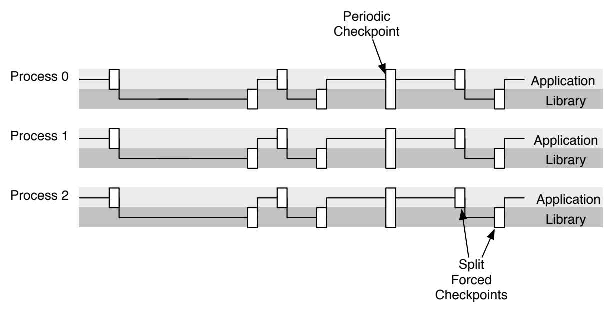


Costs

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

GENERAL phase, fault free waste

GENERAL phase

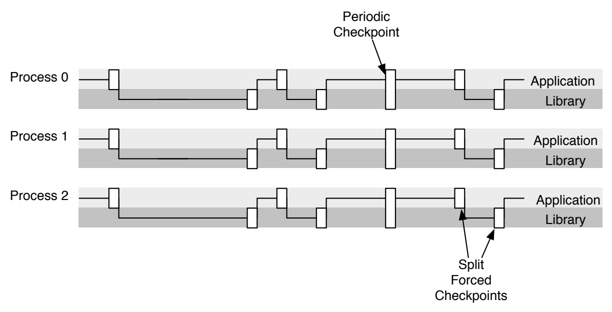


Without Failures

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

LIBRARY phase, fault free waste

LIBRARY phase

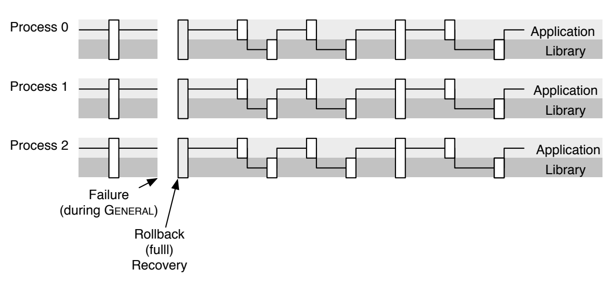


Without Failures

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

GENERAL phase, failure overhead

GENERAL phase

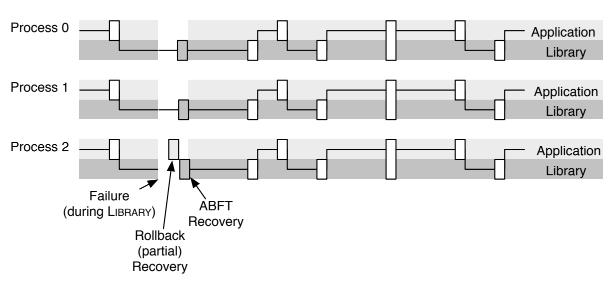


Failure Overhead

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

Library phase, failure overhead

Library phase



Failure Overhead

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

Overall

Overall

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

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

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

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

Which is minimal in the second case, if

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

Waste

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

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

Toward Exascale, and Beyond!

Let's think at scale

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

☺ ABFT&PERIODICCKPT should perform better with scale

? By how much?

Competitors

FT algorithms compared

PeriodicCkpt Basic periodic checkpointing

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

ABFT&PeriodicCkpt The algorithm described above

Weak Scale #1

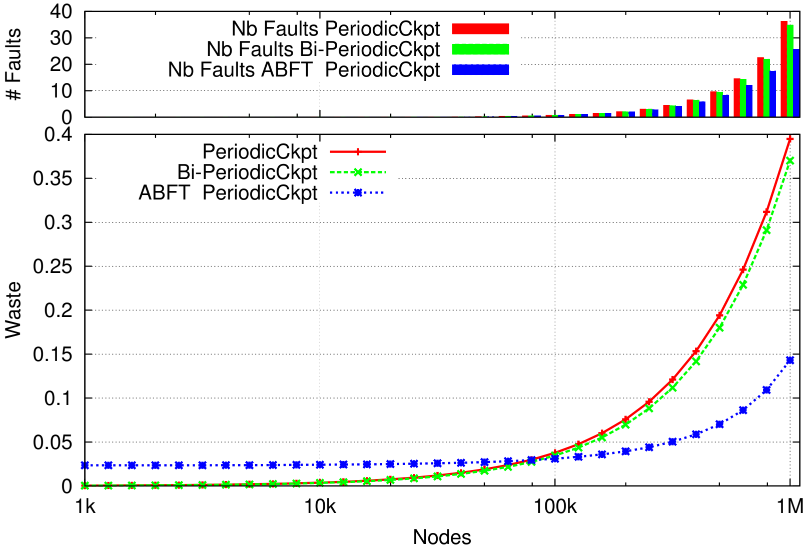
Weak Scale Scenario #1

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

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

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

Weak Scale #1



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

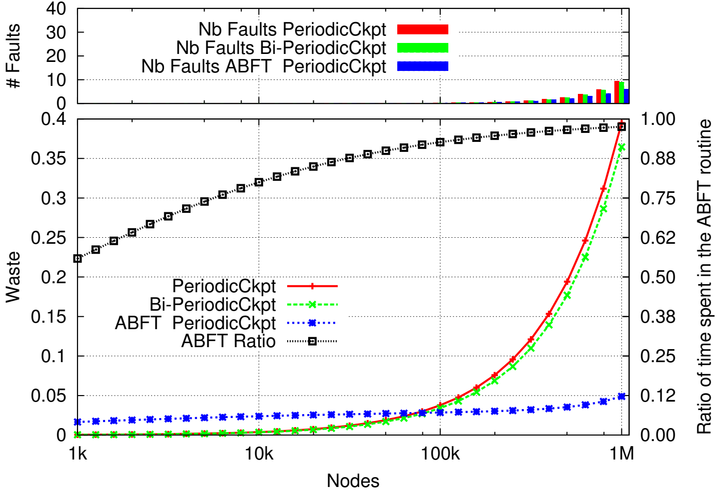
Weak Scale #2

Weak Scale Scenario #2

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

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

Weak Scale #2


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

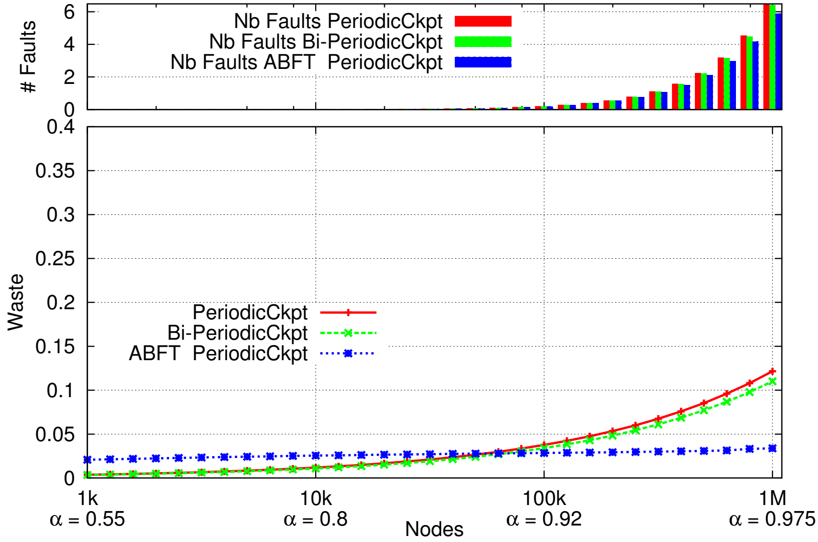
Weak Scale #3

Weak Scale Scenario #3

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

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

Weak Scale #3



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