

Fault-tolerant Techniques for HPC: Theory and Practice

George Bosilca¹, Aurélien Bouteiller¹,
Thomas Hérault¹ & Yves Robert^{1,2}

1 – University of Tennessee Knoxville

2 – Ecole Normale Supérieure de Lyon

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

<http://fault-tolerance.org/downloads/sc17-tutorial.pdf>

<http://fault-tolerance.org/sc17/>



Outline

- 1 Introduction (20mn)
- 2 Checkpointing: Protocols (40mn)
- 3 Checkpointing: Probabilistic models (30mn)
- 4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)
- 5 Hierarchical checkpointing (20mn)
- 6 Forward-recovery techniques (20mn)
- 7 Silent errors (35mn)
- 8 Conclusion (15mn)
- 9 Advanced Models

Outline

1 Introduction (20mn)

- Large-scale computing platforms
- Faults and failures

2 Checkpointing: Protocols (40mn)

3 Checkpointing: Probabilistic models (30mn)

4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)

5 Hierarchical checkpointing (20mn)

6 Forward-recovery techniques (20mn)

7 Silent errors (35mn)

8 Conclusion (15mn)

9 Advanced Models

Outline

1 Introduction (20mn)

- Large-scale computing platforms
- Faults and failures

2 Checkpointing: Protocols (40mn)

3 Checkpointing: Probabilistic models (30mn)

4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)

5 Hierarchical checkpointing (20mn)

6 Forward-recovery techniques (20mn)

7 Silent errors (35mn)

8 Conclusion (15mn)

9 Advanced Models

Exascale platforms (courtesy Jack Dongarra)

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

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

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

Toward Exascale Computing (My Roadmap)

Based on proposed DOE roadmap with MTTI adjusted to scale linearly

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

Exascale platforms

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

MTBF – one node	1 year	10 years	120 years
MTBF – platform of 10^6 nodes	30sec	5mn	1h

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

Exascale platforms

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

MTBF – one node	1 year	10 years	120 years
MTBF – platform of 10^6 nodes	30sec	5mn	1h

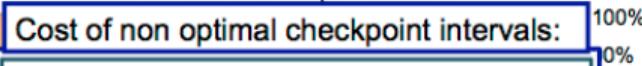
Exascale
More nodes ≠ Petascale × 1000 failures)

Even for today's platforms (courtesy F. Cappello)



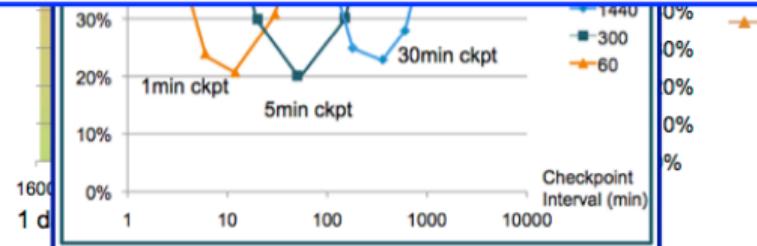
Fault tolerance becomes critical at Petascale (MTTI <= 1day)
Poor fault tolerance design may lead to huge overhead

Overhead of checkpoint/restart



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

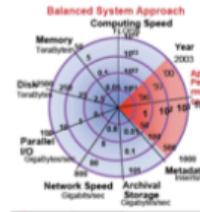
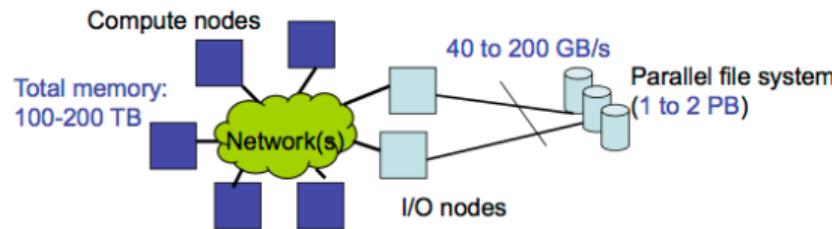
Dr. E.N. (Mootaz) Elnozahy et al. *System Resilience at Extreme Scale*,
DARPA



Even for today's platforms (courtesy F. Cappello)

Classic approach for FT: Checkpoint-Restart

Typical "Balanced Architecture" for PetaScale Computers



RoadRunner



TACC Ranger

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

Systems	Perf.	Ckpt time	Source
RoadRunner	1PF	~20 min.	Panasas
LLNL BG/L	500 TF	>20 min.	LLNL
LLNL Zeus	11TF	26 min.	LLNL
YYY BG/P	100 TF	~30 min.	YYY

Outline

1

Introduction (20mn)

- Large-scale computing platforms
- Faults and failures

2

Checkpointing: Protocols (40mn)

3

Checkpointing: Probabilistic models (30mn)

4

Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)

5

Hierarchical checkpointing (20mn)

6

Forward-recovery techniques (20mn)

7

Silent errors (35mn)

8

Conclusion (15mn)

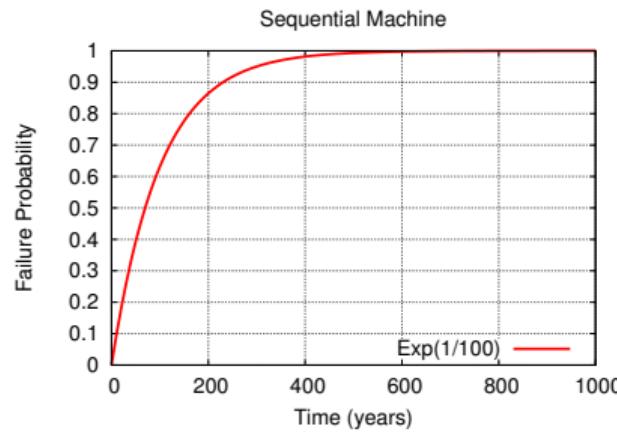
9

Advanced Models

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 (Silent Data Corruptions) addressed later in the tutorial

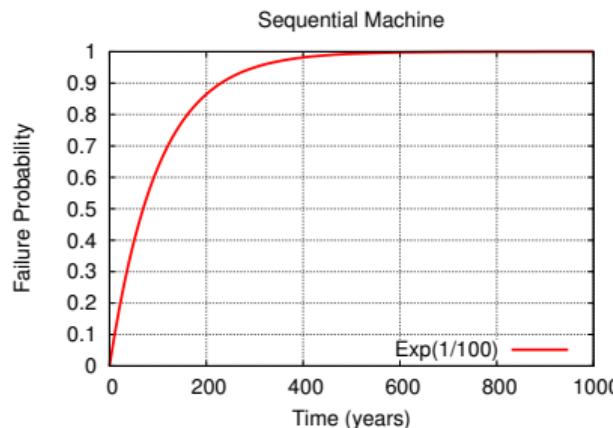
Failure distributions: (1) Exponential



Exp(λ): Exponential distribution law of parameter λ :

- Pdf: $f(t) = \lambda e^{-\lambda t} dt$ for $t \geq 0$
- Cdf: $F(t) = 1 - e^{-\lambda t}$
- Mean = $\frac{1}{\lambda}$

Failure distributions: (1) Exponential



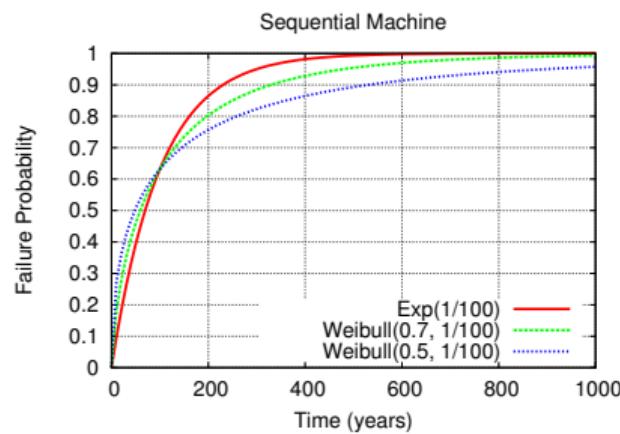
X random variable for $\text{Exp}(\lambda)$ failure inter-arrival times:

- $\mathbb{P}(X \leq t) = 1 - e^{-\lambda t} dt$ (by definition)
- **Memoryless property:** $\mathbb{P}(X \geq t + s | X \geq s) = \mathbb{P}(X \geq t)$

at any instant, time to next failure does not depend upon time elapsed since last failure

- Mean Time Between Failures (MTBF) $\mu = \mathbb{E}(X) = \frac{1}{\lambda}$

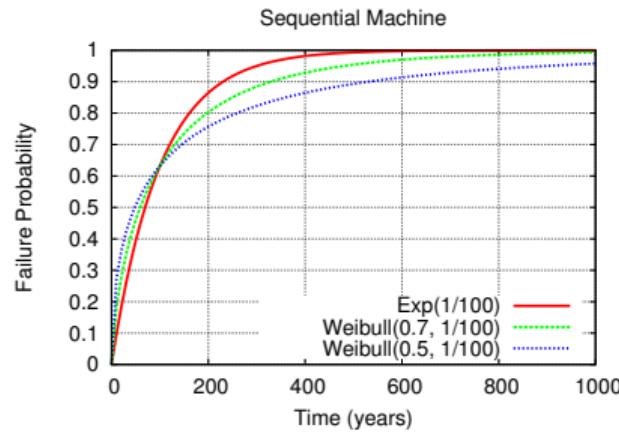
Failure distributions: (2) Weibull



Weibull(k, λ): Weibull distribution law of shape parameter k and scale parameter λ :

- Pdf: $f(t) = k\lambda(t\lambda)^{k-1}e^{-(\lambda t)^k} dt$ for $t \geq 0$
- Cdf: $F(t) = 1 - e^{-(\lambda t)^k}$
- Mean = $\frac{1}{\lambda}\Gamma(1 + \frac{1}{k})$

Failure distributions: (2) Weibull



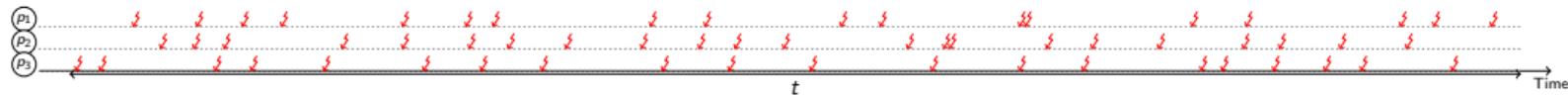
X random variable for $Weibull(k, \lambda)$ failure inter-arrival times:

- 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

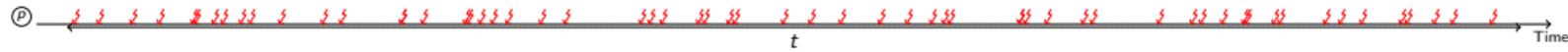
Failure distributions: with several processors

- Processor (or node): any entity subject to failures
⇒ approach **agnostic to granularity**
- If the MTBF is μ with one processor,
what is its value with p processors?

Intuition



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 ($\mu_p = \frac{t}{60}$)

Platform MTBF

- Rebooting only faulty processor
- Platform failure distribution
 - ⇒ superposition of p IID processor distributions
 - ⇒ IID only for Exponential
- Define μ_p by

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

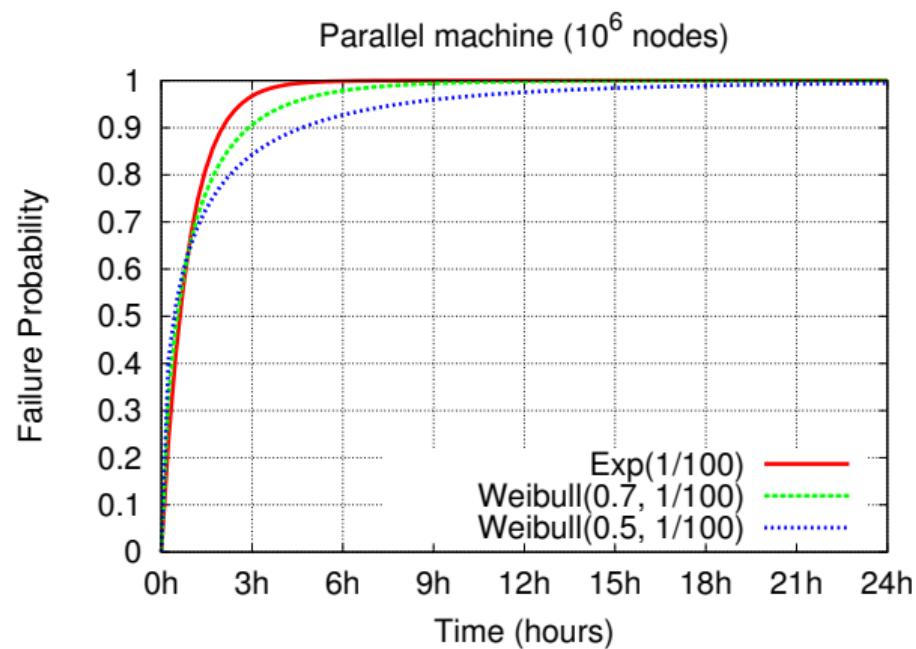
$n(F)$ = number of platform failures until time F is exceeded

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?



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 (20mn)

2

Checkpointing: Protocols (40mn)

- Process Checkpointing
- Coordinated Checkpointing
- Application-Level Checkpointing

3

Checkpointing: Probabilistic models (30mn)

4

Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)

5

Hierarchical checkpointing (20mn)

6

Forward-recovery techniques (20mn)

7

Silent errors (35mn)

8

Conclusion (15mn)

9

Advanced Models

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 (20mn)

2 Checkpointing: Protocols (40mn)

- Process Checkpointing
- Coordinated Checkpointing
- Application-Level Checkpointing

3 Checkpointing: Probabilistic models (30mn)

4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)

5 Hierarchical checkpointing (20mn)

6 Forward-recovery techniques (20mn)

7 Silent errors (35mn)

8 Conclusion (15mn)

9 Advanced Models

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(er 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 time) in the size of memory and in the size of modified files

Threads must be synchronized, or each thread must checkpoint

Asynchronous Checkpointing

System-level approach: make use of copy on write of `fork` syscall

User-level approach: critical sections, when needed

Staging Checkpointing

Alternative to asynchronous checkpointing.

Use memory hierarchy to reduce checkpoint time.

Storage

Remote Reliable Storage

Intuitive. I/O intensive. Disk usage.

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 (20mn)

2 Checkpointing: Protocols (40mn)

- Process Checkpointing
- **Coordinated Checkpointing**
- Application-Level Checkpointing

3 Checkpointing: Probabilistic models (30mn)

4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)

5 Hierarchical checkpointing (20mn)

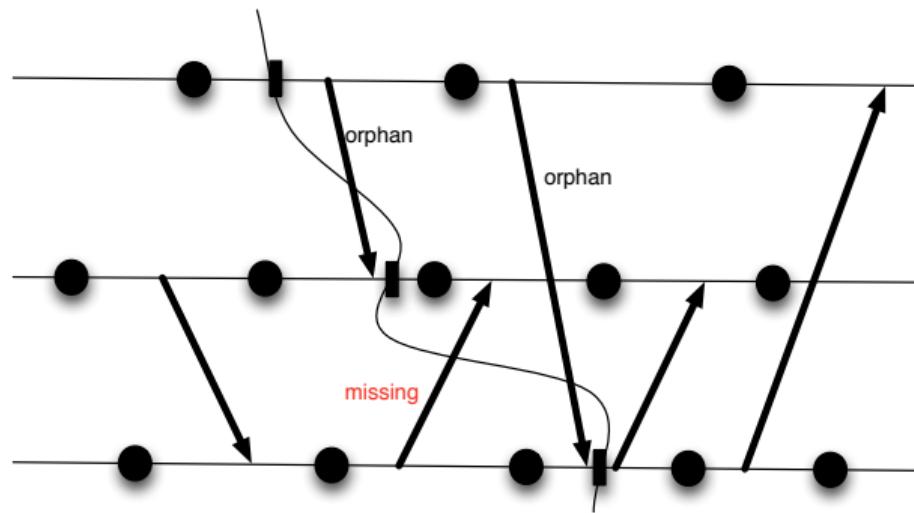
6 Forward-recovery techniques (20mn)

7 Silent errors (35mn)

8 Conclusion (15mn)

9 Advanced Models

Coordinated checkpointing

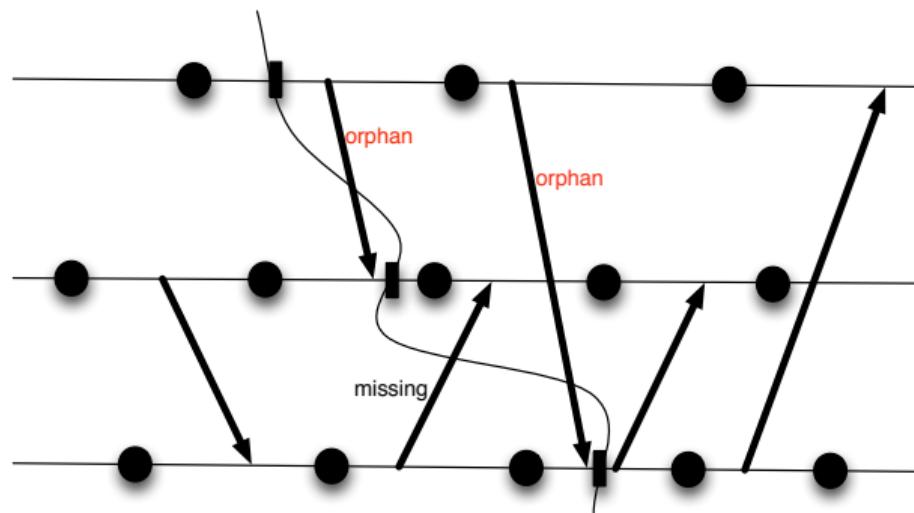


Definition (Missing Message)

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



Coordinated checkpointing

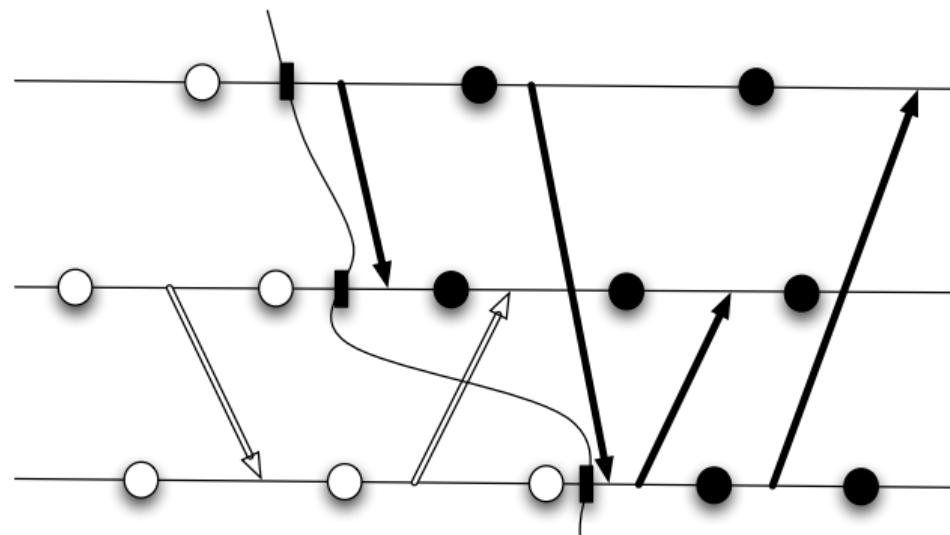


Definition (Orphan Message)

A message is orphan if in the current configuration, the receiver received it, while the sender did not send it



Coordinated Checkpointing: Main Idea

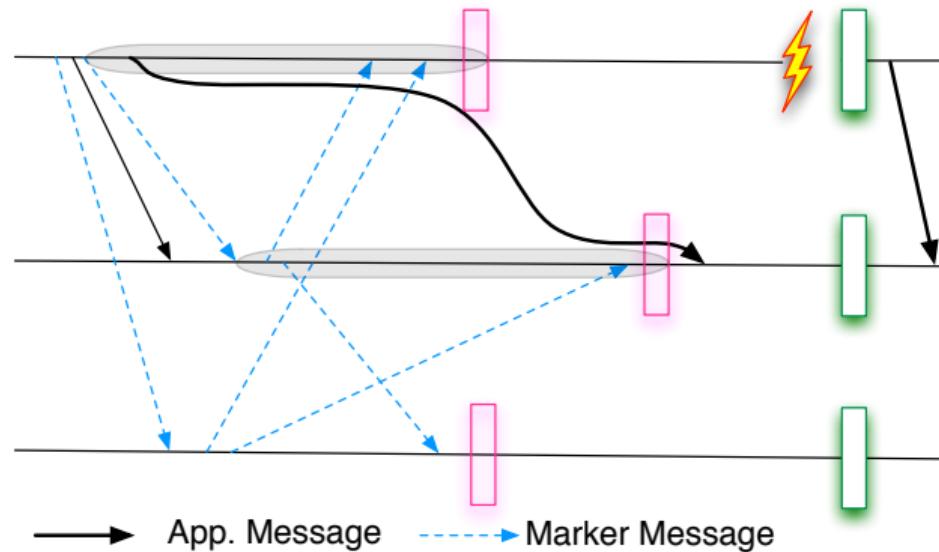


Create a consistent view of the application

- Every message belongs to a single checkpoint wave
- All communication channels must be flushed (all2all)

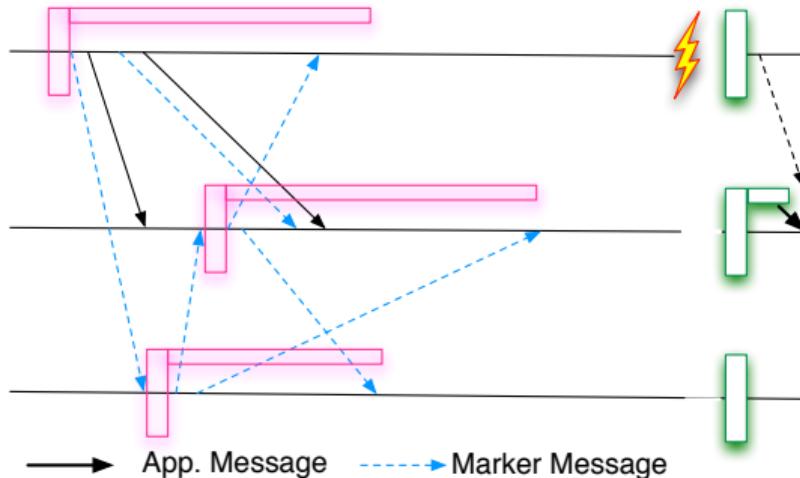


Blocking Coordinated Checkpointing



- Silences the network during the checkpoint

Non-Blocking Coordinated Checkpointing



- 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

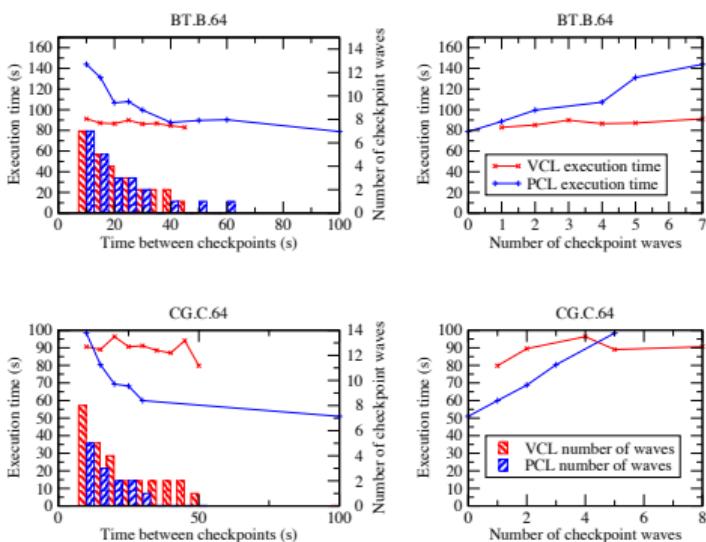
Communication Library

- 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



Coordinated Protocol Performance

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

Outline

1 Introduction (20mn)

2 Checkpointing: Protocols (40mn)

- Process Checkpointing
- Coordinated Checkpointing
- Application-Level Checkpointing

3 Checkpointing: Probabilistic models (30mn)

4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)

5 Hierarchical checkpointing (20mn)

6 Forward-recovery techniques (20mn)

7 Silent errors (35mn)

8 Conclusion (15mn)

9 Advanced Models

Application-Level Checkpointing

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

MPI 1D Stencil

```
1 int main (int argc, char *argv[])
2 {
3     double locals[NBLOCALS],           /* The local values */
4         *globals,                   /* all values, defined only for 0 */
5         local_error, global_error; /* Estimates of the error */
6     int taskid, numtasks;             /* rank and world size */
7     MPI_Init(&argc,&argv);
8     MPI_Comm_size(MPI_COMM_WORLD ,&numtasks);
9     MPI_Comm_rank(MPI_COMM_WORLD ,&taskid);
10    /** Read the local domain from an input file */
11    if( taskid == 0 ) globals = ReadFile("input");
12    /** And distribute it on all nodes */
13    MPI_Scatter(globals, NBLOCALS, MPI_DOUBLE, locals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
14    do {
15        /** Update the domain, exchanging information with neighbors */
16        UpdateLocals(locals, NBLOCALS, taskid, numtasks);
17        /** Compute the local error */
18        local_error = LocalError(locals, NBLOCALS);
19        /** Compute the global error */
20        MPI_AllReduce(&local_error, &global_error, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
21    } while( global_error > THRESHOLD );
22    /** Output result to output file */
23    MPI_Gather(locals, NBLOCALS, MPI_DOUBLE, globals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
24    if( taskid == 0 ) SaveFile("Result", globals);
25    MPI_Finalize();
26    return 0;
27 }
```



Example: MPI-1D Stencil

MPI 1D Stencil

```
1 int main (int argc, char *argv[])
2 {
3     double locals[NBLOCALS],           /* The local values */
4         *globals,                   /* all values, defined only for 0 */
5         local_error, global_error; /* Estimates of the error */
6     int taskid, numtasks;             /* rank and world size */
7     MPI_Init(&argc,&argv);
8     MPI_Comm_size(MPI_COMM_WORLD ,&numtasks);
9     MPI_Comm_rank(MPI_COMM_WORLD ,&taskid);
10    /** Read the local domain from an input file */
11    if( taskid == 0 ) globals = ReadFile("input");
12    /** And distribute it on all nodes */
13    MPI_Scatter(globals, NBLOCALS, MPI_DOUBLE, locals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
14    do {
15        /** Update the domain, exchanging information with neighbors */
16        UpdateLocals(locals, NBLOCALS, taskid, numtasks);
17        /** Compute the local error */
18        local_error = LocalError(locals, NBLOCALS);
19        /** Compute the global error */
20        MPI_AllReduce(&local_error, &global_error, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD); Natural Synchronization Point
21    } while( global_error > THRESHOLD );
22    /** Output result to output file */
23    MPI_Gather(locals, NBLOCALS, MPI_DOUBLE, globals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
24    if( taskid == 0 ) SaveFile("Result", globals);
25    MPI_Finalize();
26    return 0;
27 }
```



Example: MPI-1D Stencil

User-Level Checkpointing

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

Example: MPI-1D Stencil

User-Level Checkpointing

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

Example: MPI-1D Stencil

User-Level Rollback

```
1 int main (int argc, char *argv[])
2 {
3     double locals[NBLOCALS],           /* The local values */
4         *globals,                   /* all values, defined only for 0 */
5         local_error, global_error; /* Estimates of the error */
6     int taskid, numtasks;             /* rank and world size */
7     MPI_Init(&argc,&argv);
8     MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
9     MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
10    /** Read the local domain from an input file */
11    if( taskid == 0 ) globals = ReadFile(argv[1]); Read Checkpoint or Input
12    /** And distribute it on all nodes */
13    MPI_Scatter(globals, NBLOCALS, MPI_DOUBLE, locals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
14    do {
```

Application-Level Checkpointing – Gather Approach

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

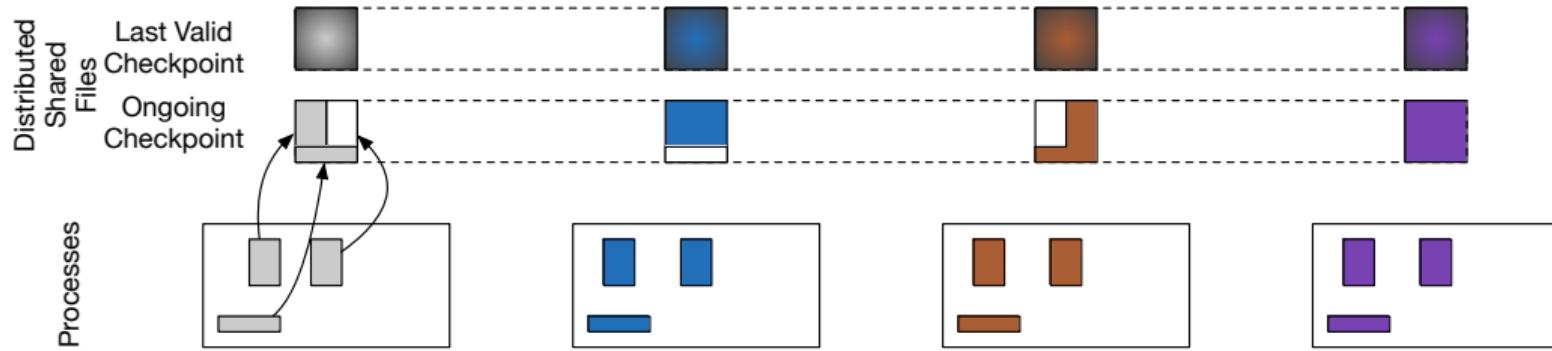
User-Level Rollback

- 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

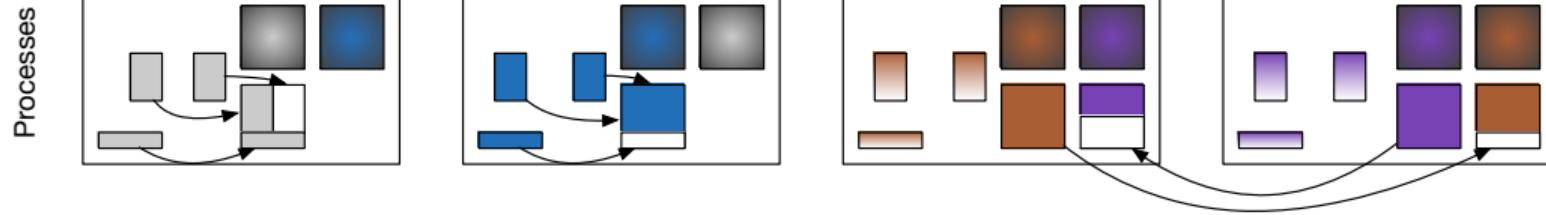
Application-Level Checkpointing – Distributed Checkpointing Approach



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)

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)

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 \Rightarrow Fault Tolerant storage (Replication, or XOR)

Helping Libraries – SCR

SCR Example – Init

```
1 int main (int argc, char *argv[])
2 {
3     double locals[NBLOCALS],           /* The local values */
4         *globals,                   /* all values, defined only for 0 */
5         local_error, global_error; /* Estimates of the error */
6     int taskid, numtasks;            /* rank and world size */
7     char name[256], scr_file_name[SCR_MAX_FILENAME];
8     FILE *f;
9     size_t n;
10    int rc, scr_want_to_checkpoint;
11
12    MPI_Init(&argc,&argv);
13    SCR_Init();
14    MPI_Comm_size(MPI_COMM_WORLD ,&numtasks);
15    MPI_Comm_rank(MPI_COMM_WORLD ,&taskid);
16
17    sprintf(name, "Checkpoint-%d", taskid);
18    if( SCR_Route_file("MyCheckpoint", scr_file_name) != SCR_SUCCESS ) {
19        fprintf(stderr, "Checkpoint disabled--aborting\n");
20        MPI_Abort(MPI_COMM_WORLD);
21    }
```

SCR Example – Fini

```
1 if( taskid == 0 ) SaveFile("Result", globals);
2 SCR_Finalize();
3 MRT_Finalize();
```



Helping Libraries – SCR

SCR Example – Checkpoint

```
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     SCR_Need_checkpoint(&scr_want_to_checkpoint);
10    if( global_error > THRESHOLD && scr_want_to_checkpoint ) {
11        SCR_Start_checkpoint();
12        f = fopen(scr_file_name, "w");
13        if( NULL != f ) {
14            n = fwrite(f, locals, NBLOCALS * sizeof(double));
15            rc = fclose(f);
16        }
17        SCR_Complete_checkpoint(f != NULL &&
18                                n == NBLOCALS * sizeof(double) &&
19                                rc == 0 );
20    }
21 } while( global_error > THRESHOLD );
```

Helping Libraries – SCR

```
1      if( argc > 1 && (strcmp(argv[1], "-restart") == 0) ) {
2          f = fopen(scr_file_name, "r");
3          if( NULL != f ) {
4              n = fread(f, locals, NBLOCALS * sizeof(double));
5              rc = fclose(f);
6          }
7          if( f == NULL ||
8              n != NBLOCALS * sizeof(double) ||
9              rc != 0 ) {
10              fprintf(stderr, "Unable to read checkpoint file\n");
11              MPI_Abort(MPI_COMM_WORLD);
12          }
13      } else {
14          /** Read the local domain from an input file */
15          if( taskid == 0 ) globals = ReadFile(argv[1]);
16          /** And distribute it on all nodes */
17          MPI_Scatter(globals, NBLOCALS, MPI_DOUBLE,
18                      locals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
19      }
```

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

Helping Libraries – FTI

FTI Example – Init

```
1 int main (int argc, char *argv[])
2 {
3     double locals[NBLOCALS],           /* The local values */
4         *globals,                   /* all values, defined only for 0 */
5         local_error, global_error; /* Estimates of the error */
6     int taskid, numtasks;             /* rank and world size */
7
8     MPI_Init(&argc,&argv);
9     FTI_Init("conf.fti", MPI_COMM_WORLD);
10    MPI_Comm_size(MPI_COMM_WORLD ,&numtasks);
11    MPI_Comm_rank(MPI_COMM_WORLD ,&taskid);
```

SCR Example – Fini

```
1 if( taskid == 0 ) SaveFile("Result", globals);
2 FTI_Finalize();
3 MPI_Finalize();
4 return 0;
5 }
```

Helping Libraries – FTI

FTI Example – Checkpoint

```
1  /** Read the local domain from an input file */
2  if( taskid == 0 ) globals = ReadFile(argv[1]);
3  /** And distribute it on all nodes */
4  MPI_Scatter(globals, NBLOCALS, MPI_DOUBLE,
5               locals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
6
7  FTI_Protect(0, locals, NBLOCALS * sizeof(double), FTI_DFLT);
8
9  do {
10     /** Update the domain, exchanging information with neighbors */
11     UpdateLocals(locals, NBLOCALS, taskid, numtasks);
12     /** Compute the local error */
13     local_error = LocalError(locals, NBLOCALS);
14     /** Compute the global error */
15     MPI_AllReduce(&local_error, &global_error, 1, MPI_DOUBLE,
16                   MPI_MAX, MPI_COMM_WORLD);
17     FTI_Snapshot();
18 } 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

```
1     FTI_Protect(0, locals, NBLOCALS * sizeof(double), FTI_DFLT);
2
3     do {
4         /** Update the domain, exchanging information with neighbors */
5         UpdateLocals(locals, NBLOCALS, taskid, numtasks);
6         /** Compute the local error */
7         local_error = LocalError(locals, NBLOCALS);
8         /** Compute the global error */
9         MPI_AllReduce(&local_error, &global_error, 1, MPI_DOUBLE,
10                       MPI_MAX, MPI_COMM_WORLD);
11         FTI_Snapshot();
12     } 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



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.

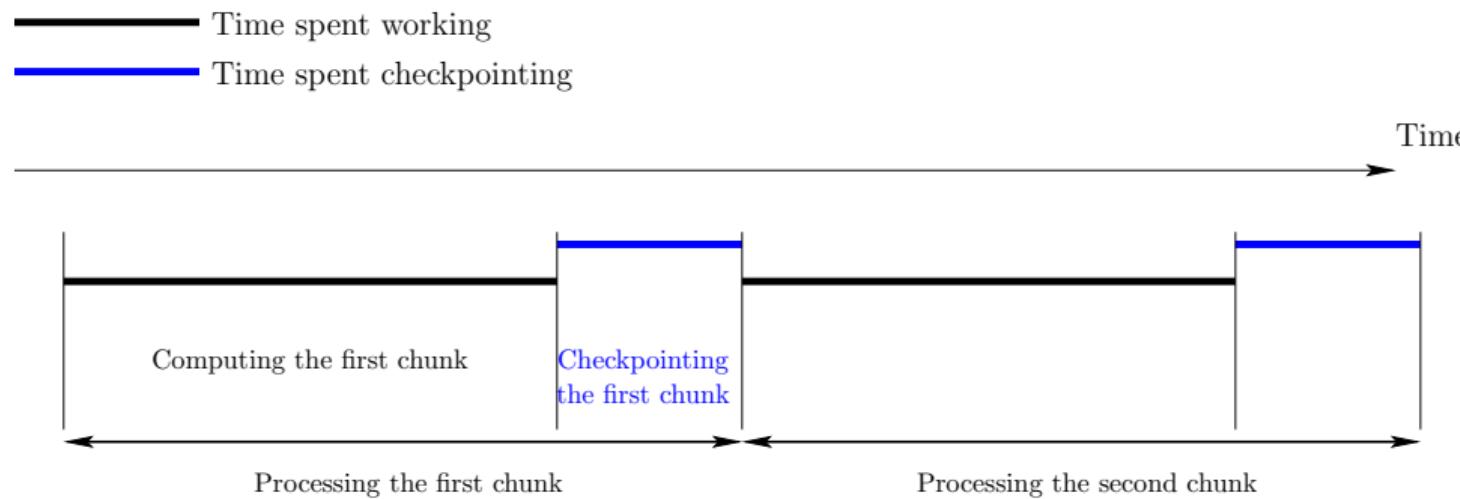
Outline

- 1 Introduction (20mn)
- 2 Checkpointing: Protocols (40mn)
- 3 Checkpointing: Probabilistic models (30mn)
 - Young/Daly's approximation
 - Exponential distributions
 - In-memory checkpointing
 - Multi-level checkpointing
- 4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)
- 5 Hierarchical checkpointing (20mn)
- 6 Forward-recovery techniques (20mn)
- 7 Silent errors (35mn)
- 8 Conclusion (15mn)
- 9 Advanced Models

Outline

- 1 Introduction (20mn)
- 2 Checkpointing: Protocols (40mn)
- 3 Checkpointing: Probabilistic models (30mn)
 - Young/Daly's approximation
 - Exponential distributions
 - In-memory checkpointing
 - Multi-level checkpointing
- 4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)
- 5 Hierarchical checkpointing (20mn)
- 6 Forward-recovery techniques (20mn)
- 7 Silent errors (35mn)
- 8 Conclusion (15mn)
- 9 Advanced Models

Periodic checkpointing



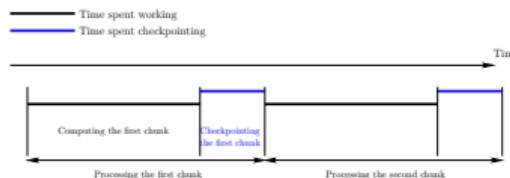
Blocking model: while a checkpoint is taken, no computation can be performed

Framework

- Periodic checkpointing policy of period $T = W + C$
 - Independent and identically distributed failures
 - Applies to a single processor with MTBF $\mu = \mu_{ind}$
 - Applies to a platform with p processors and MTBF $\mu = \frac{\mu_{ind}}{p}$
 - coordinated checkpointing
 - tightly-coupled application
 - progress \Leftrightarrow all processors available
- ⇒ 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
- TIME_{FF} : with periodic checkpoints but failure-free

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

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

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

Waste due to failures

- $\text{TIME}_{\text{base}}$: application base time
- TIME_{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_{faults} number of failures during execution

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

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

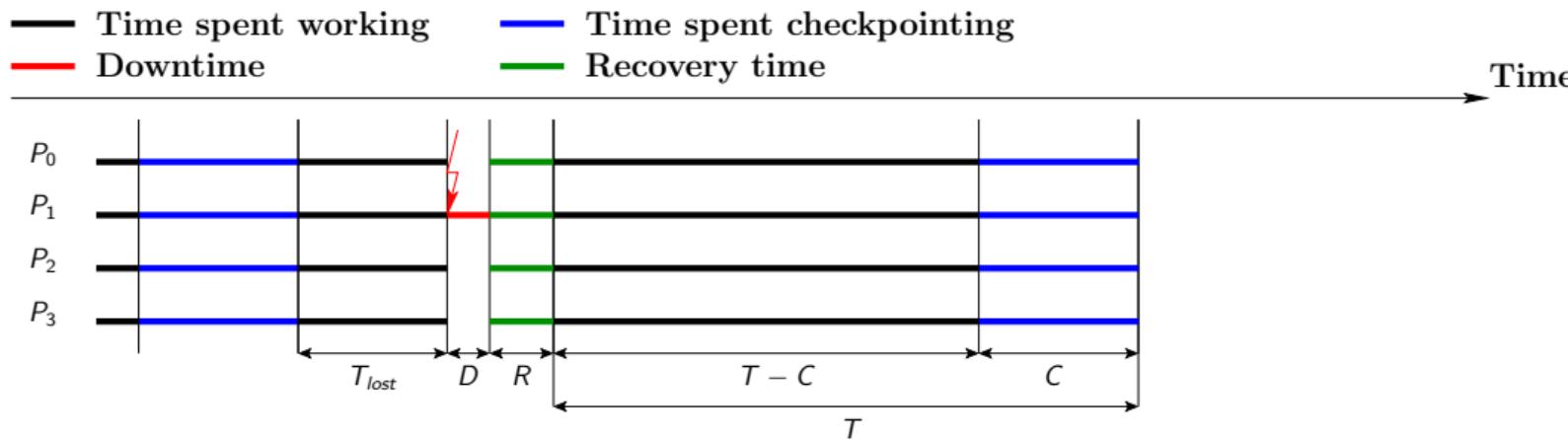
N_{faults} number of failures during execution

T_{lost} : average time lost per failure

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

T_{lost} ?

Computing T_{lost}



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

Rationale

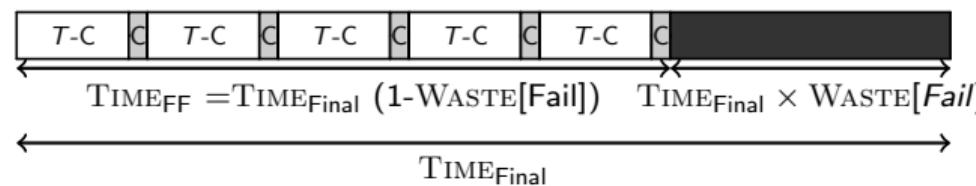
- ⇒ Instants when periods begin and failures strike are independent
- ⇒ Approximation used for all distribution laws
- ⇒ Exact for Exponential and uniform distributions

Waste due to failures

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

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

Total waste

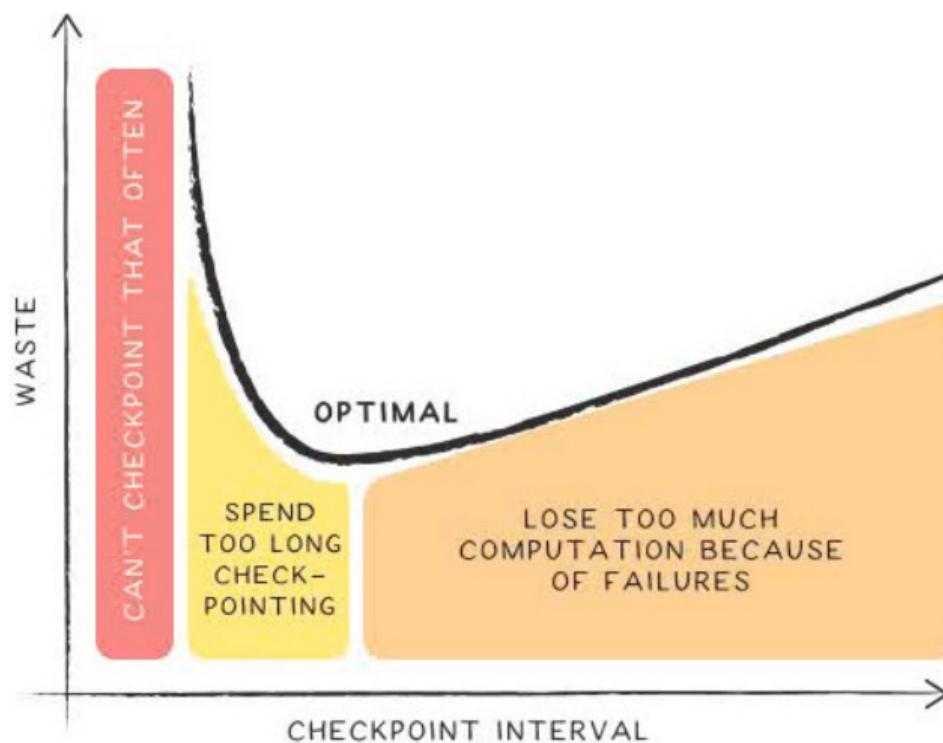


$$\text{WASTE} = \frac{\text{TIME}_{\text{final}} - \text{TIME}_{\text{base}}}{\text{TIME}_{\text{final}}}$$

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

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

Optimal checkpointing interval



Waste minimization

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

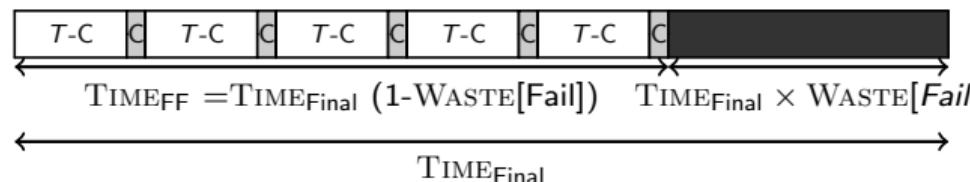
$$\text{WASTE} = \frac{u}{T} + v + wT$$

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

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 $D = R = 0$

$$\Rightarrow T = \sqrt{2\mu C} + C$$

Validity of the approach (1/3)

Technicalities

- $\mathbb{E}(N_{\text{faults}}) = \frac{\text{TIME}_{\text{final}}}{\mu}$ and $\mathbb{E}(T_{\text{lost}}) = D + R + \frac{T}{2}$
but expectation of product is not product of expectations
(not independent RVs here)
- 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 $\mu = \frac{\mu_{\text{ind}}}{p}$ too small for large p , regardless of μ_{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 γ is some tuning parameter
 - Poisson process of parameter $\theta = \frac{T}{\mu}$
 - Probability of having $k \geq 0$ failures : $P(X = k) = \frac{\theta^k}{k!} e^{-\theta}$
 - Probability of having two or more failures:
$$\pi = P(X \geq 2) = 1 - (P(X = 0) + P(X = 1)) = 1 - (1 + \theta)e^{-\theta}$$
 - $\gamma = 0.27 \Rightarrow \pi \leq 0.03$
$$\Rightarrow \text{overlapping faults for only } 3\% \text{ of checkpointing segments}$$

Validity of the approach (3/3)

- Enforce $T \leq \gamma\mu$, $C \leq \gamma\mu$, and $D + R \leq \gamma\mu$
- Optimal period $\sqrt{2(\mu - (D + R))C}$ may not belong to admissible interval $[C, \gamma\mu]$
- Waste is then minimized for one of the bounds of this admissible interval (by convexity)

Wrap up

- Capping periods, and enforcing a lower bound on MTBF
⇒ 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

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

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

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

Scale by 100: $C = 20$ min $\mu = 0.24$ hrs $\Rightarrow \text{WASTE}[opt] = 100\%$

Lesson learnt for fail-stop failures

(Not) Secret data

- Tsubame 2: ~2 failures during last 18 months $\mu = 13$ hrs
- Blue Waters: 2-3 node failures per day
- Titan: a few failures per day
- Tianhe 2:

Exascale \neq Petascale $\times 1000$

Need more reliable components

Need to checkpoint faster

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

Lesson learnt for fail-stop failures

(Not so) Secret data

- Tsubame 2: 962 failures during last 18 months so $\mu = 13$ hrs
- Blue Waters: 2-3 node failures per day
- Titan: a few failures per day
- T

Silent errors:
detection latency \Rightarrow additional problems

• Detection latency is the time between failure and detection.

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 (20mn)
- 2 Checkpointing: Protocols (40mn)
- 3 Checkpointing: Probabilistic models (30mn)
 - Young/Daly's approximation
 - **Exponential distributions**
 - In-memory checkpointing
 - Multi-level checkpointing
- 4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)
- 5 Hierarchical checkpointing (20mn)
- 6 Forward-recovery techniques (20mn)
- 7 Silent errors (35mn)
- 8 Conclusion (15mn)
- 9 Advanced Models

Expected execution time for a single chunk

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

Recursive Approach

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

Expected execution time for a single chunk

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

Recursive Approach

Probability
of success

$$\overbrace{\mathcal{P}_{\text{succ}}(W + C)}^{(W + C)}$$

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

Expected execution time for a single chunk

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

Recursive Approach

Time needed
to compute
the work W and
checkpoint it

$$\mathcal{P}_{\text{succ}}(W + C) \overbrace{(W + C)}$$

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

Expected execution time for a single chunk

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

Recursive Approach

$$\mathcal{P}_{\text{succ}}(W + C)(W + C)$$

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

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

Probability of failure

Expected execution time for a single chunk

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

Recursive Approach

$$\mathcal{P}_{\text{succ}}(W + C)(W + C)$$

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

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

Time elapsed
before failure
stroke

Expected execution time for a single chunk

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

Recursive Approach

$$\mathcal{P}_{\text{succ}}(W + C)(W + C)$$

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

$$(1 - \mathcal{P}_{\text{succ}}(W + C)) (\mathbb{E}(T_{\text{lost}}(W + C)) + \underbrace{\mathbb{E}(T_{\text{rec}}) + \mathbb{E}(W)}_{\text{Time needed to perform downtime}})$$

Time needed
to perform
downtime

Expected execution time for a single chunk

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

Recursive Approach

$$\mathcal{P}_{\text{succ}}(W + C)(W + C)$$

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

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

Time needed
to compute W

anew

Computation of $\mathbb{E}(W)$

$$\mathcal{P}_{\text{succ}}(W + C)(W + C)$$

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

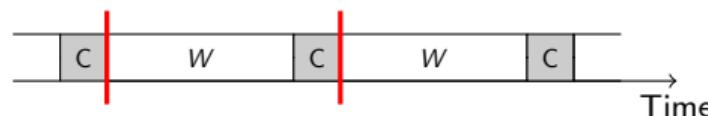
$$(1 - \mathcal{P}_{\text{succ}}(W + C)) (\mathbb{E}(T_{\text{lost}}(W + C)) + \mathbb{E}(T_{\text{rec}}) + \mathbb{E}(W))$$

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

$$\mathbb{E}(W) = e^{\lambda R} \left(\frac{1}{\lambda} + D \right) (e^{\lambda(W+C)} - 1)$$

Optimal checkpointing interval

Minimize expected execution overhead $H(W) = \frac{\mathbb{E}(W)}{W} - 1$



- Exact solution:

$$H(W) = \frac{e^{\lambda R} \left(\frac{1}{\lambda} + D \right) e^{\lambda(W+C)}}{W} - 1, \text{ use Lambert function}$$

- First-order approximation [Young/Daly]:

$$W_{\text{opt}} = \sqrt{\frac{2C}{\lambda}} = \sqrt{2C\mu}$$

$$H_{\text{opt}} = \sqrt{2\lambda C} + \Theta(\lambda)$$

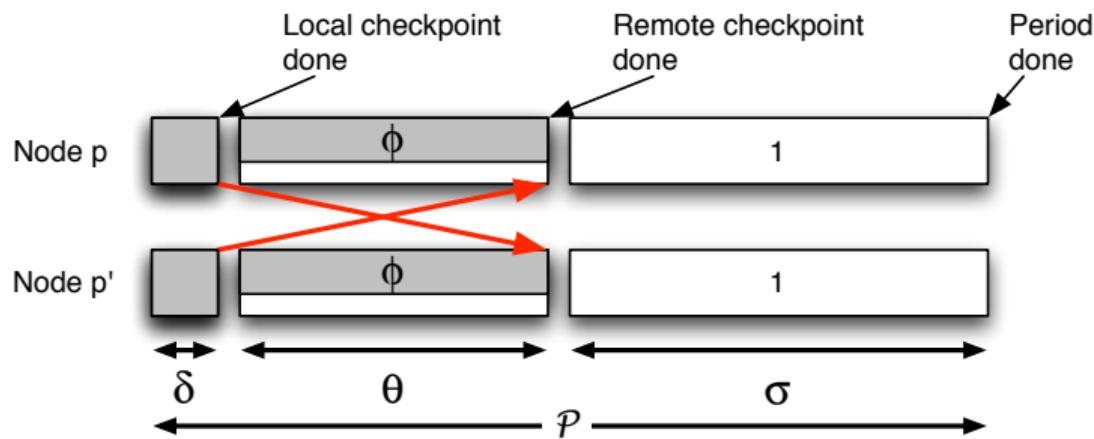
Outline

- 1 Introduction (20mn)
- 2 Checkpointing: Protocols (40mn)
- 3 Checkpointing: Probabilistic models (30mn)
 - Young/Daly's approximation
 - Exponential distributions
 - In-memory checkpointing
 - Multi-level checkpointing
- 4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)
- 5 Hierarchical checkpointing (20mn)
- 6 Forward-recovery techniques (20mn)
- 7 Silent errors (35mn)
- 8 Conclusion (15mn)
- 9 Advanced Models

Motivation

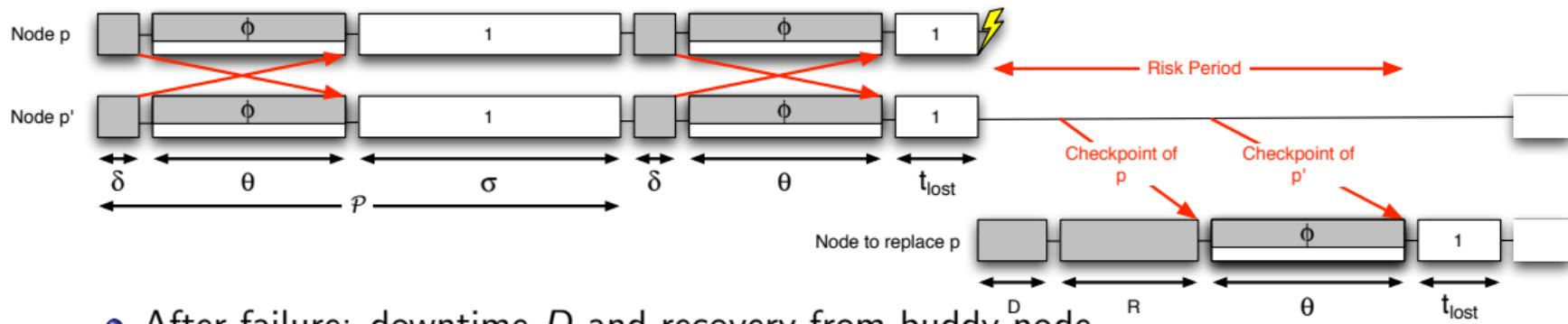
- Checkpoint transfer and storage
⇒ critical issues of rollback/recovery protocols
- Stable storage: high cost
- Distributed in-memory storage:
 - Store checkpoints in local memory ⇒ no centralized storage
 Much better scalability
 - Replicate checkpoints ⇒ application survives single failure
 Still, 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

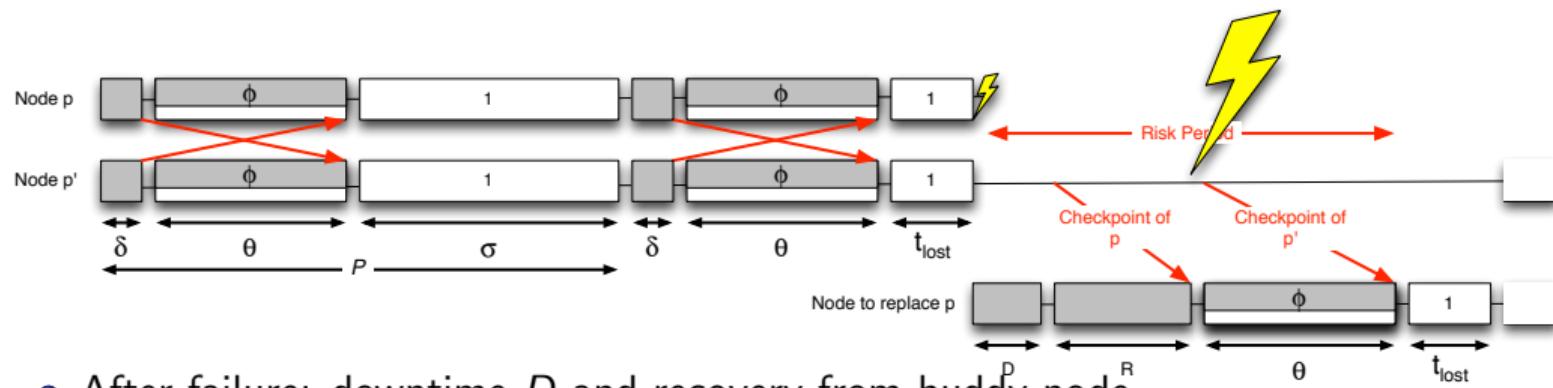
Failures



- After failure: downtime D and recovery from buddy node
- Two checkpoint files lost, must be re-sent to faulty processor

Best trade-off between performance and risk?

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 (20mn)

2 Checkpointing: Protocols (40mn)

3 Checkpointing: Probabilistic models (30mn)

- Young/Daly's approximation
- Exponential distributions
- In-memory checkpointing
- Multi-level checkpointing

4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)

5 Hierarchical checkpointing (20mn)

6 Forward-recovery techniques (20mn)

7 Silent errors (35mn)

8 Conclusion (15mn)

9 Advanced Models

Multi-level checkpointing

Coordinated checkpointing

⇒ **Scalability problem for large-scale platforms**

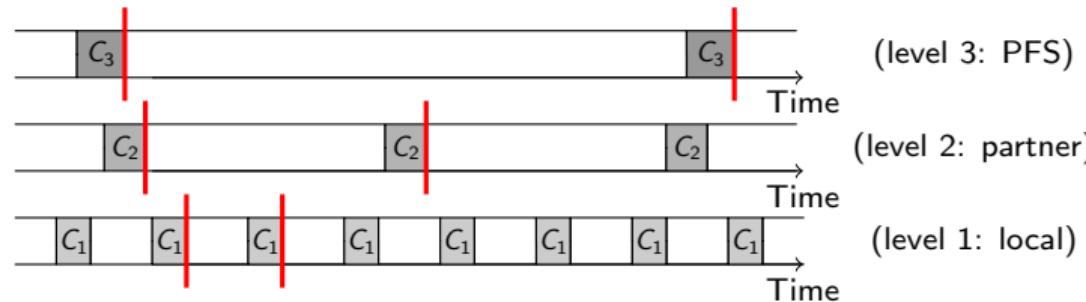
Multiple technologies to cope with different failure types:

- Local memory/SSD
- Partner copy/XOR
- Reed-Solomon coding
- Parallel file system

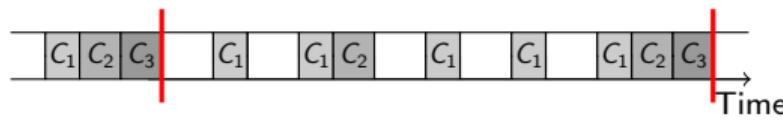
Scalable Checkpoint/Restart (SCR) library
Fault Tolerance Interface (FTI)

Simplified model

- Independent checkpointing:

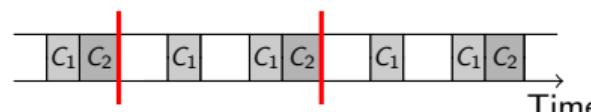


- Synchronized checkpointing:



Two Levels

Easier because pattern repeats (memoryless property)



- Exact solution: very complicated (which error type occurs first?), equal-length chunks, see [1]
- First-order approximation:

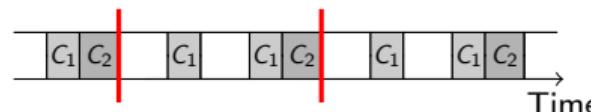
$$H_{\text{opt}} = \sqrt{2\lambda_1 C_1} + \sqrt{2\lambda_2 C_2} + \Theta(\lambda)$$

(obtained for some optimal pattern)

[1] S. Di, Y. Robert, F. Vivien, F. Cappello. Toward an optimal online checkpoint solution under a two-level HPC checkpoint model, *IEEE TPDS*, 2017.

Two Levels

Easier because pattern repeats (memoryless property)



- Exact solution: very complicated (which error type occurs first?), equal-length chunks, see [1]
- First-order approximation:

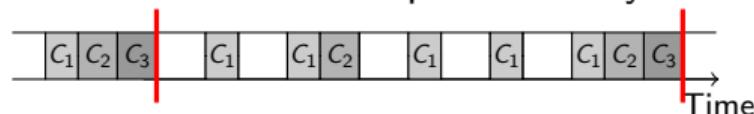
$$H_{\text{opt}} = \sqrt{2\lambda_1 C_1} + \sqrt{2\lambda_2 C_2} + \Theta(\lambda)$$

(obtained for some optimal pattern)

[1] S. Di, Y. Robert, F. Vivien, F. Cappello. Toward an optimal online checkpoint solution under a two-level HPC checkpoint model, *IEEE TPDS*, 2017.

Three Levels

Difficult because sub-patterns may differ



- Exact solution: unknown
- First-order approximation:

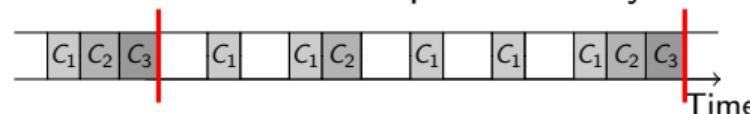
$$H_{\text{opt}} = \sqrt{2\lambda_1 C_1} + \sqrt{2\lambda_2 C_2} + \sqrt{2\lambda_3 C_3} + \Theta(\lambda)$$

- Choose optimal set of levels:

Level	Overhead
1, 2, 3	$\sqrt{2C_1\lambda_1} + \sqrt{2C_2\lambda_2} + \sqrt{2C_3\lambda_3}$
1, 3	$\sqrt{2C_1\lambda_1} + \sqrt{2C_3(\lambda_2 + \lambda_3)}$
2, 3	$\sqrt{2C_2(\lambda_1 + \lambda_2)} + \sqrt{2C_3\lambda_3}$
3	$\sqrt{2C_3(\lambda_1 + \lambda_2 + \lambda_3)}$

Three Levels

Difficult because sub-patterns may differ



- Exact solution: unknown
- First-order approximation:

$$H_{\text{opt}} = \sqrt{2\lambda_1 C_1} + \sqrt{2\lambda_2 C_2} + \sqrt{2\lambda_3 C_3} + \Theta(\lambda)$$

- Choose optimal set of levels:

Level	Overhead
1, 2, 3	$\sqrt{2C_1\lambda_1} + \sqrt{2C_2\lambda_2} + \sqrt{2C_3\lambda_3}$
1, 3	$\sqrt{2C_1\lambda_1} + \sqrt{2C_3(\lambda_2 + \lambda_3)}$
2, 3	$\sqrt{2C_2(\lambda_1 + \lambda_2)} + \sqrt{2C_3\lambda_3}$
3	$\sqrt{2C_3(\lambda_1 + \lambda_2 + \lambda_3)}$

k Levels

Theorem

The optimal k -level pattern, under the first-order approximation, has equal-length chunks at all levels:

$$\text{Optimal pattern length: } W^{\text{opt}} = \sqrt{\frac{\sum_{\ell=1}^k N_{\ell}^{\text{opt}} C_{\ell}}{\frac{1}{2} \sum_{\ell=1}^k \frac{\lambda_{\ell}}{N_{\ell}^{\text{opt}}}}}$$

$$\text{Optimal \#chkpts at level } \ell: N_{\ell}^{\text{opt}} = \sqrt{\frac{\lambda_{\ell}}{C_{\ell}} \cdot \frac{C_k}{\lambda_k}}, \quad \forall \ell = 1, \dots, k$$

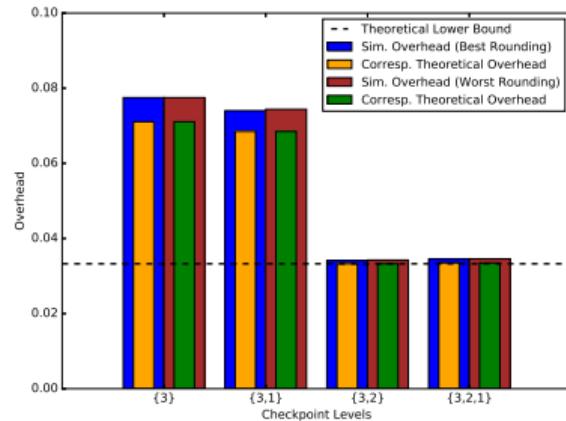
$$\text{Optimal pattern overhead: } H_{\text{opt}} = \sum_{\ell=1}^k \sqrt{2\lambda_{\ell} C_{\ell}} + \Theta(\lambda)$$

- Dynamic programming algorithm to choose set of levels

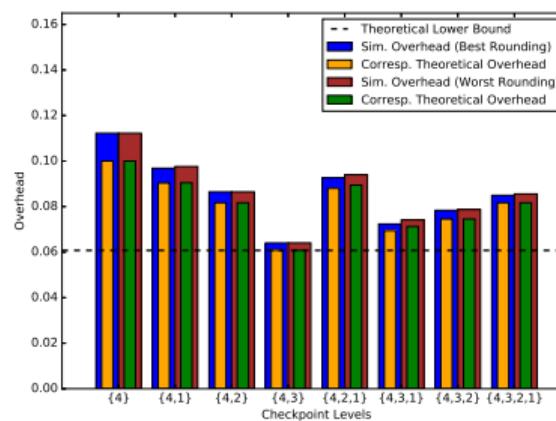
- Rounding for integer solution: $n_{\ell}^{\text{opt}} = \frac{N_{\ell}^{\text{opt}}}{N_{\ell+1}^{\text{opt}}} = \sqrt{\frac{\lambda_{\ell}}{\lambda_{\ell+1}} \cdot \frac{C_{\ell+1}}{C_{\ell}}}$

Simulations

Set	Source	Level	1	2	3	4
(A)	Moody et al. [1]	C (s)	0.5	4.5	1051	-
		MTBF (s)	5.00e6	5.56e5	2.50e6	-
(B)	Balaprakash et al. [2]	C (s)	10	20	20	100
		MTBF (s)	3.60e4	7.20e4	1.44e5	7.20e5



(A)



(B)

[1] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. *Supercomputing*, 2010.

[2] P. Balaprakash, L. A. Bautista-Gomez, M.-S. Bouguerra, S. M. Wild, F. Cappello, and P. D. Hovland. Analysis of the tradeoffs between fault tolerance and performance. *Fault-tolerance for HPC*

Conclusion

Explicit formulas for (almost) optimal multi-level checkpointing

$$H_{\text{opt}} = \sum_{\ell=1}^k \sqrt{2\lambda_\ell C_\ell} + \Theta(\lambda)$$

Limitations:

- First-order accurate for platform MTBF in hours
 \iff 10,000s of nodes. **Beyond?**
- Independent errors 😞
Correlated failures across levels?

Outline

- 1 Introduction (20mn)
- 2 Checkpointing: Protocols (40mn)
- 3 Checkpointing: Probabilistic models (30mn)
- 4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)
- 5 Hierarchical checkpointing (20mn)
- 6 Forward-recovery techniques (20mn)
- 7 Silent errors (35mn)
- 8 Conclusion (15mn)
- 9 Advanced Models

Hands-on

Hands-On

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

It is available online:

<http://fault-tolerance.org/sc17>

If you have docker already installed: docker pull abouteiller/mpi-ft-ulfm

Outline

1 Introduction (20mn)

2 Checkpointing: Protocols (40mn)

3 Checkpointing: Probabilistic models (30mn)

4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)

5 Hierarchical checkpointing (20mn)

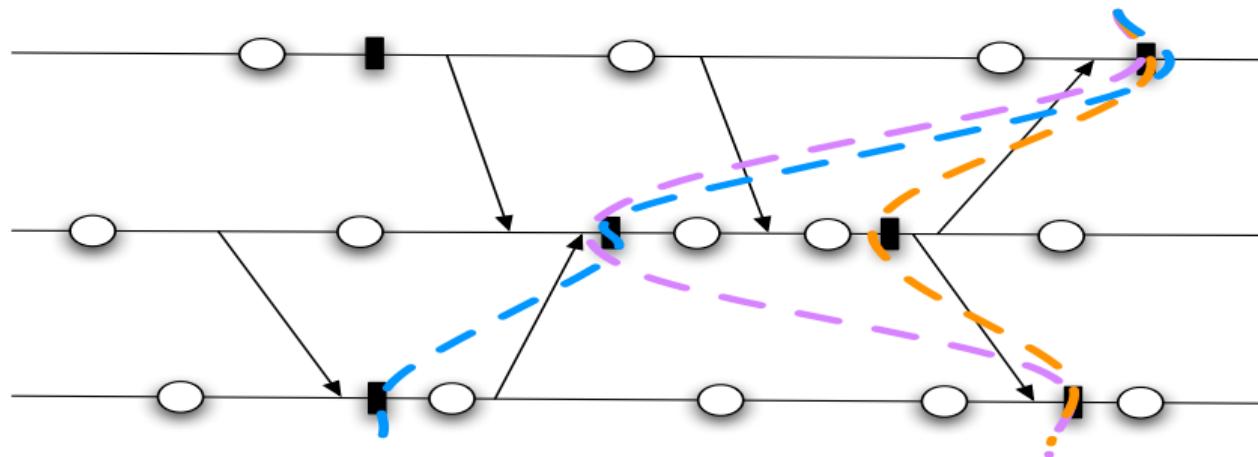
6 Forward-recovery techniques (20mn)

7 Silent errors (35mn)

8 Conclusion (15mn)

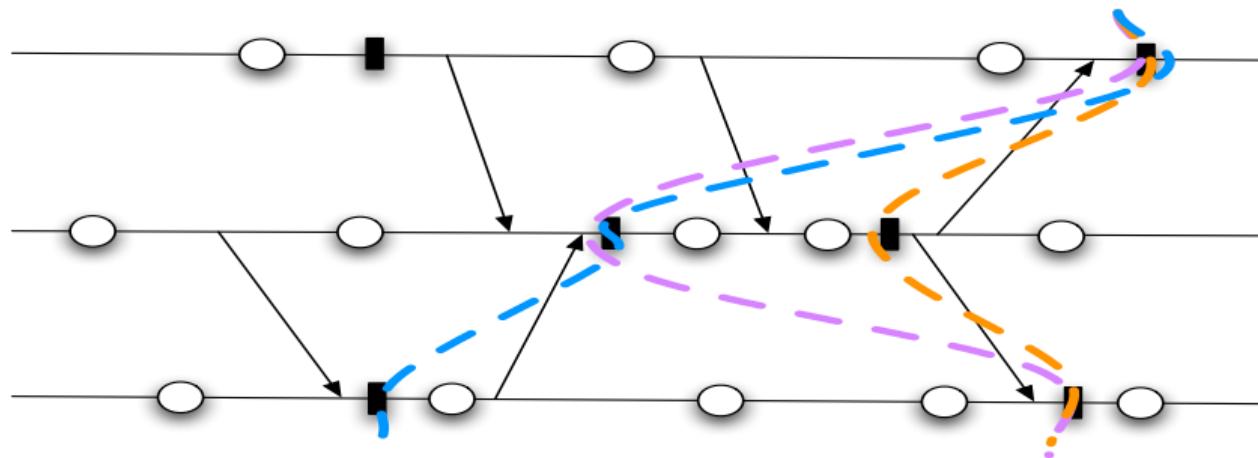
9 Advanced Models

Uncoordinated Checkpointing: Main Idea



Processes checkpoint independently

Uncoordinated Checkpointing: Main Idea

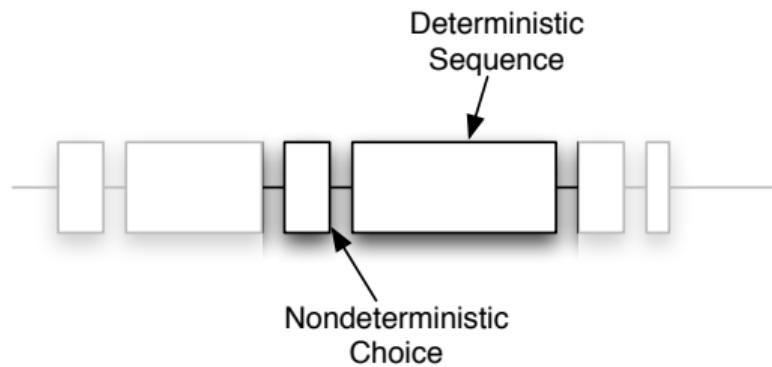


Optimistic Protocol

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



Piece-wise Deterministic Assumption

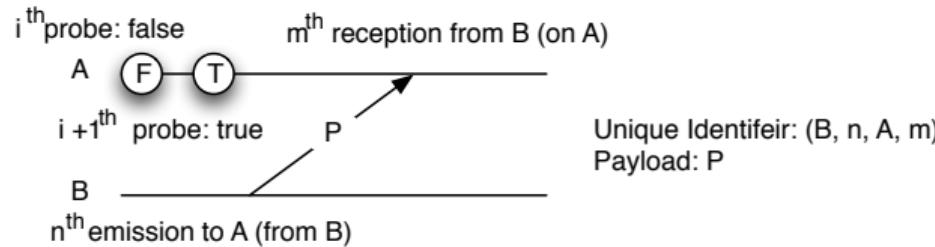


Piece-wise Deterministic Assumption

- Process: alternate sequence of non-deterministic choice and deterministic steps
- Translated in Message Passing:
 - Receptions / Progress test are non-deterministic (`MPI_Wait(ANY_SOURCE), if(MPI_Test())<...>; else <...>`)
 - Emissions / others are deterministic



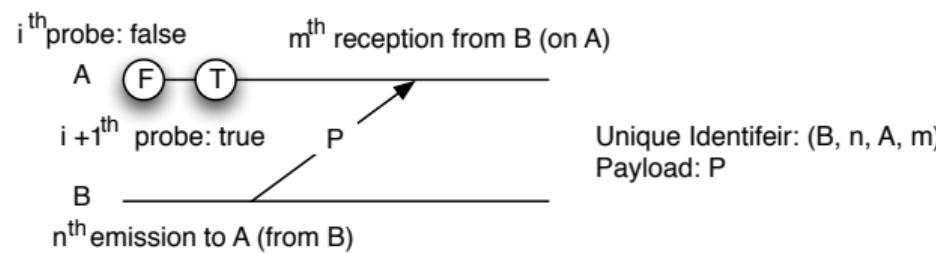
Message Logging



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

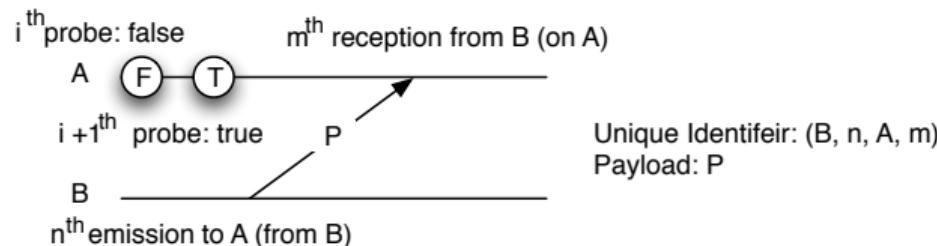
Message Logging



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

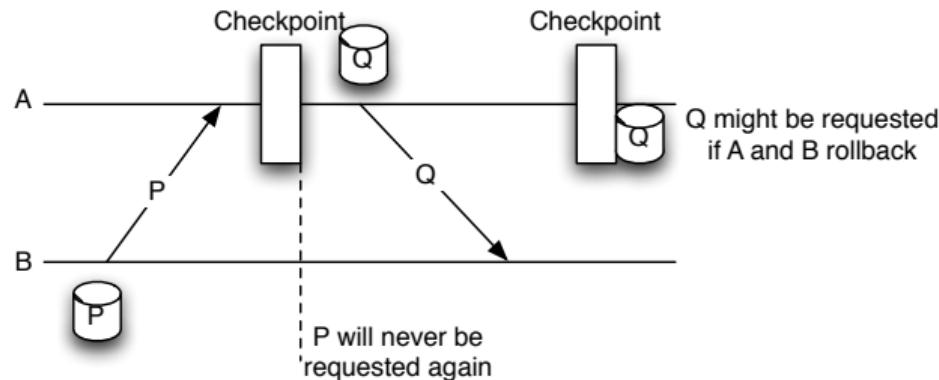
Message Logging



Message / Events

- 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

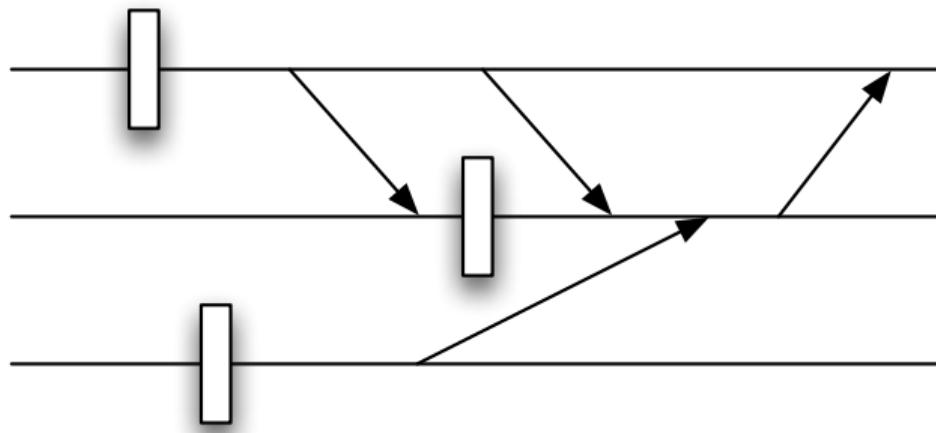
Message Logging



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

Message Logging

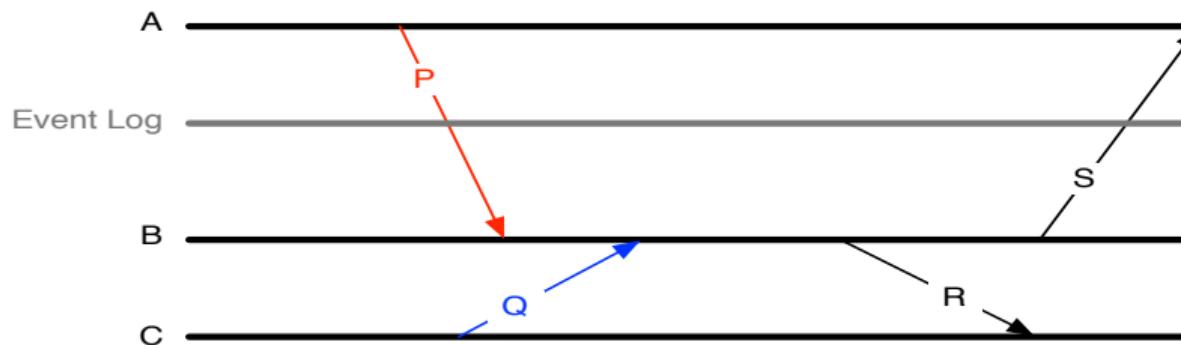


Where to save the Events?

- Events must be saved on a reliable space
- Must avoid: loss of events ordering information, for all events that can impact the outgoing communications
- Two (three) approaches: pessimistic + reliable system or causal (or optimistic)



Optimistic Message Logging

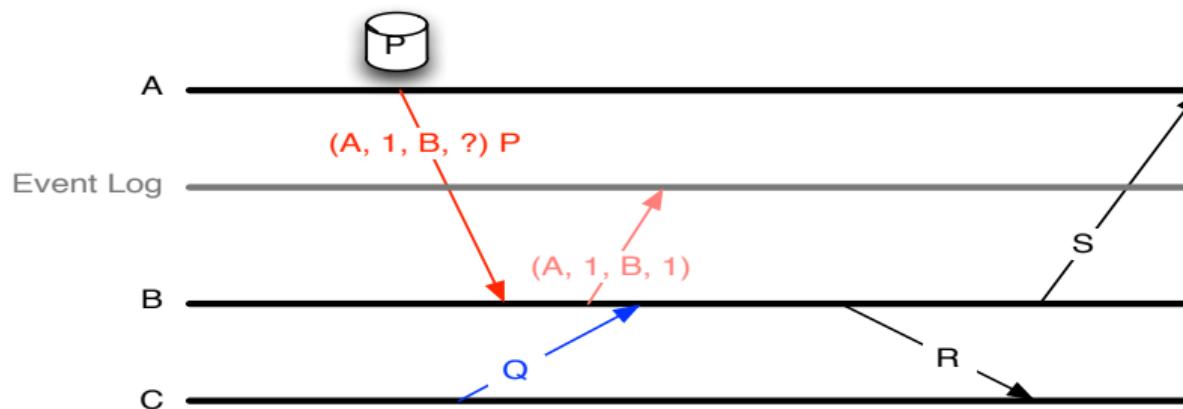


Where to save the Events?

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



Optimistic Message Logging

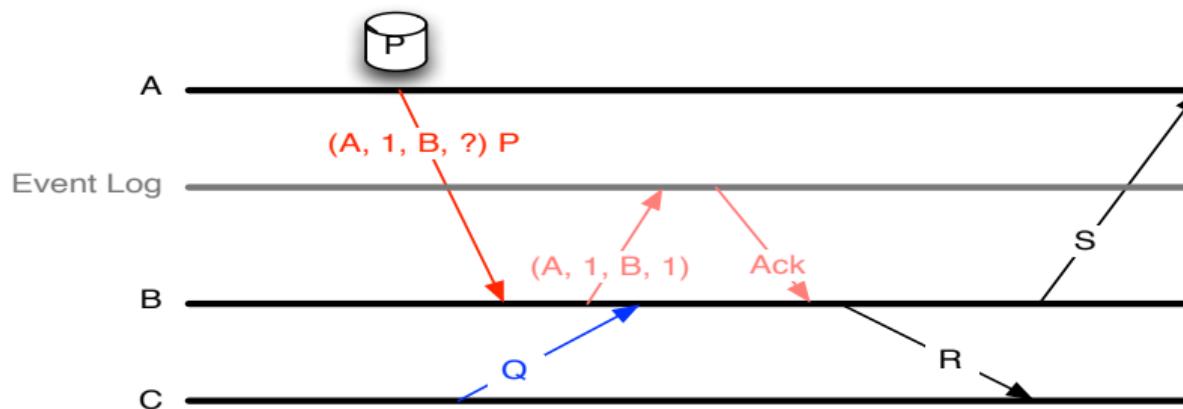


Where to save the Events?

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



Optimistic Message Logging

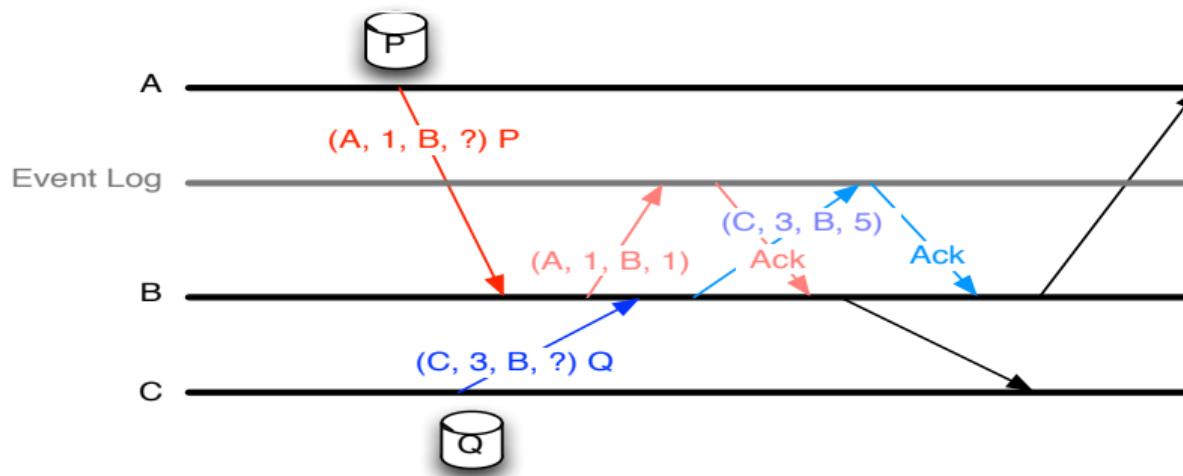


Where to save the Events?

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



Optimistic Message Logging

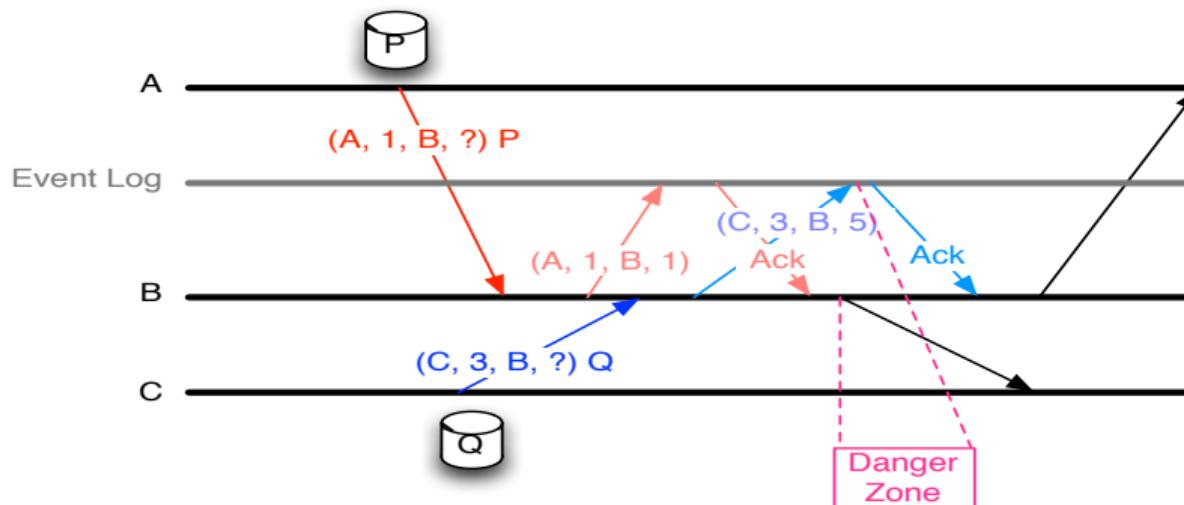


Where to save the Events?

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



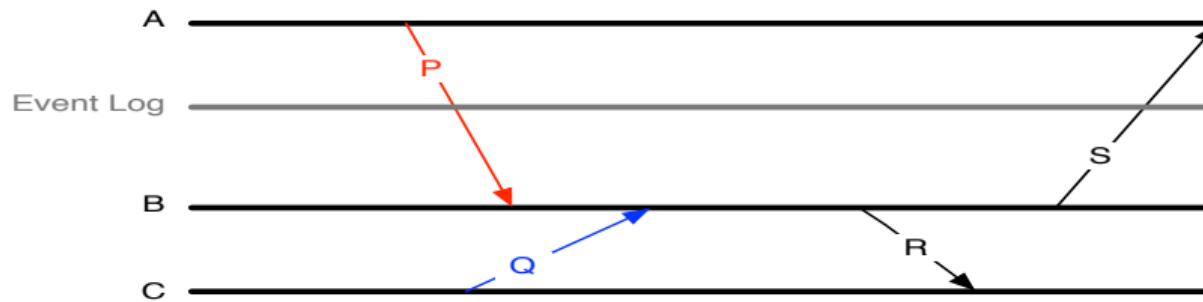
Optimistic Message Logging



Where to save the Events?

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

Pessimistic Message Logging

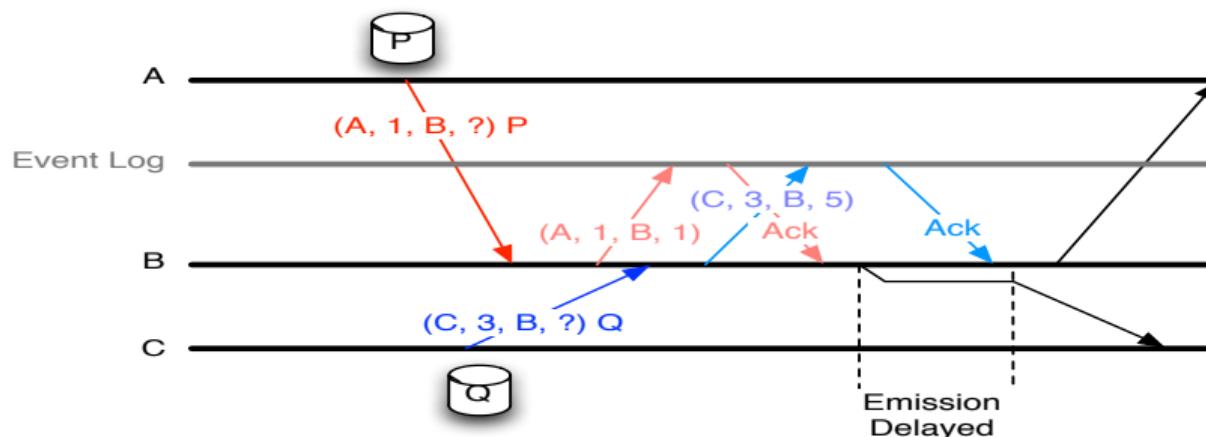


Where to save the Events?

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



Pessimistic Message Logging

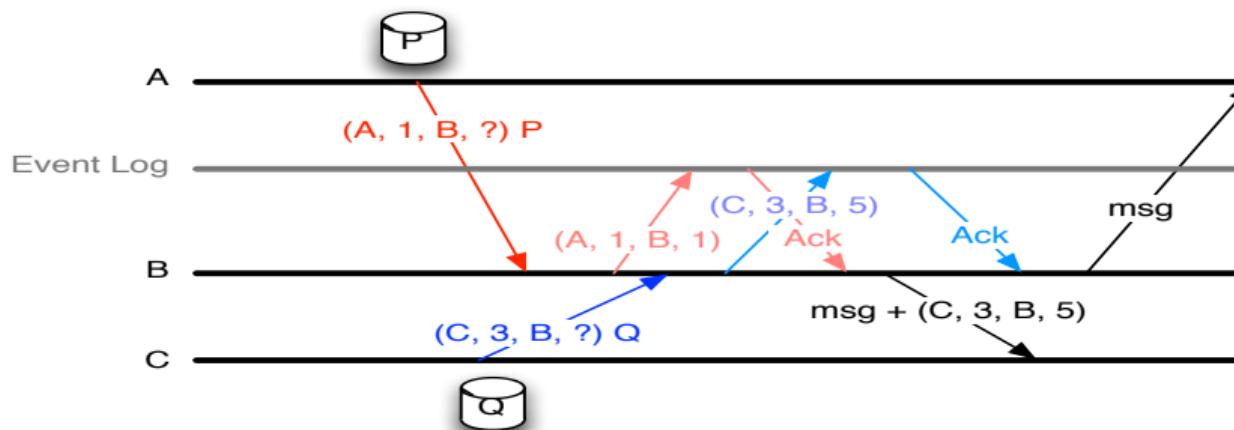


Where to save the Events?

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



Causal Message Logging

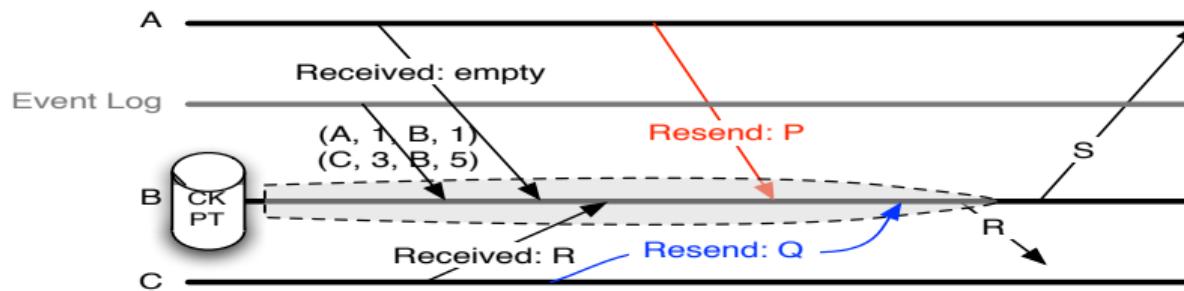


Where to save the Events?

- Any message carries with it (piggybacked) the whole history of non-deterministic events that precede
- Garbage collection using checkpointing, detection of cycles
- Can be coupled with asynchronous storage on reliable media to help garbage



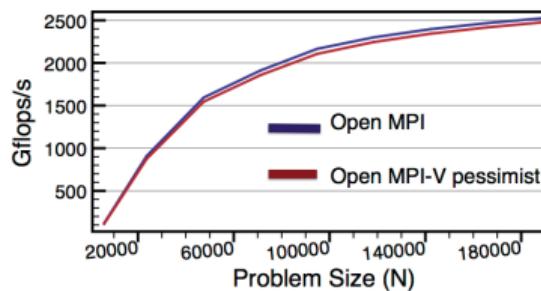
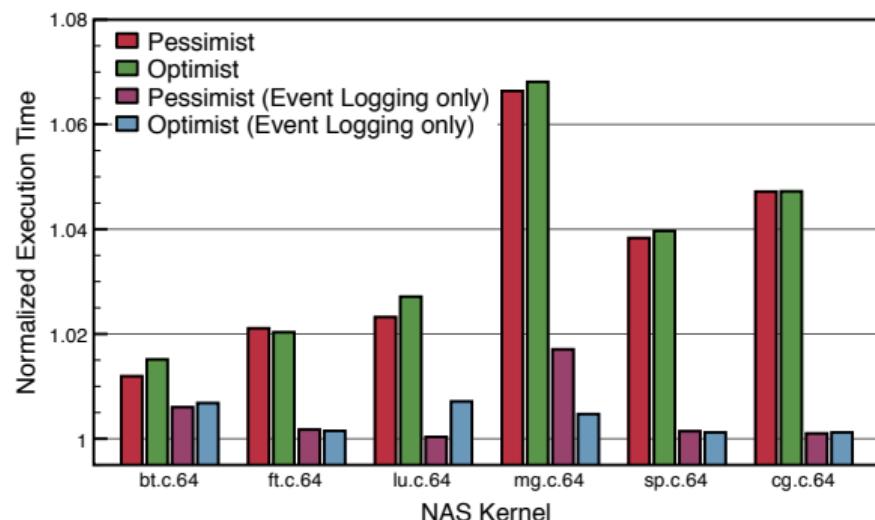
Recovery in Message Logging



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



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

Uncoordinated Protocol Performance

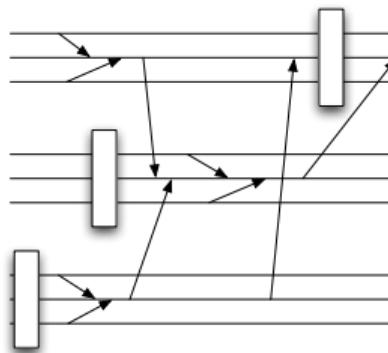
- NAS Parallel Benchmarks – 64 nodes
- High Performance Linpack

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

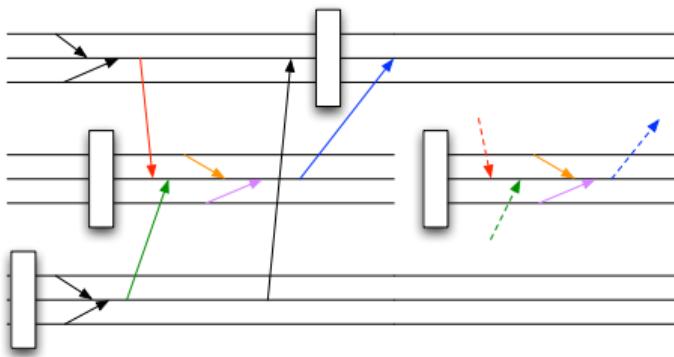
Hierarchical Protocols



Hierarchical Protocol

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

Hierarchical Protocols



Hierarchical Protocol

- Coordinated Checkpointing: the processes can behave as a non-deterministic entity (interactions between processes)
- Need to log the non-deterministic events: Hierarchical Protocols are uncoordinated protocols + event logging
- No need to log the payload

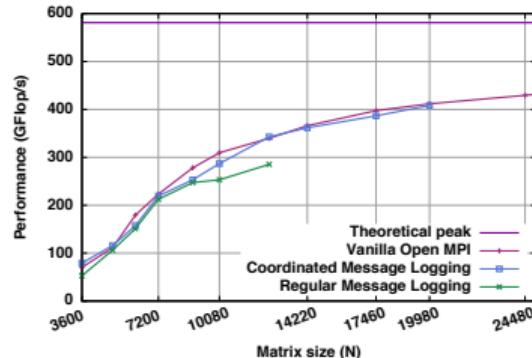
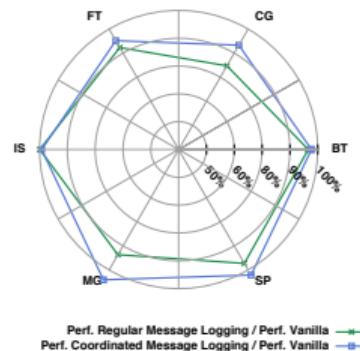


Event Log Reduction

Strategies to reduce the amount of event log

- Few HPC applications use message ordering / timing information to take decisions
- Many receptions (in MPI) are in fact deterministic: do not need to be logged
- For others, although the reception is non-deterministic, the order does not influence the interactions of the process with the rest (send-determinism). No need to log either
- Reduction of the amount of log to a few applications, for a few messages: event logging can be overlapped

Hierarchical Protocol Performance



Hierarchical Protocol Performance

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

Outline

- 1 Introduction (20mn)
- 2 Checkpointing: Protocols (40mn)
- 3 Checkpointing: Probabilistic models (30mn)
- 4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)
- 5 Hierarchical checkpointing (20mn)
- 6 Forward-recovery techniques (20mn)
 - ABFT for Linear Algebra applications
 - Composite approach: ABFT & Checkpointing
- 7 Silent errors (35mn)
- 8 Conclusion (15mn)
- 9 Advanced Models

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

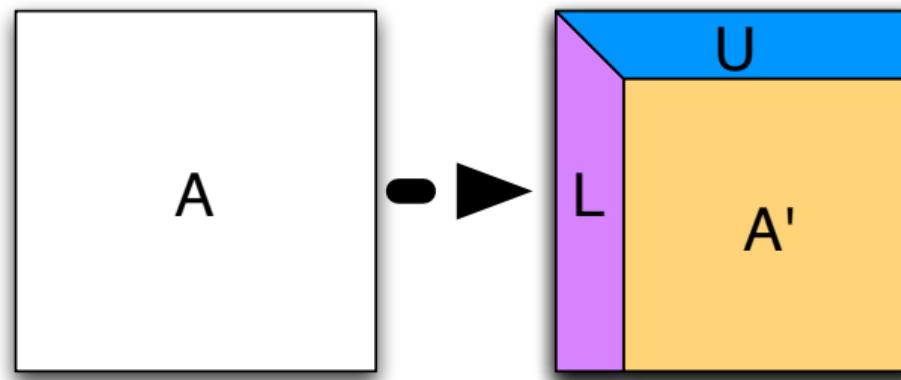
Forward Recovery

- 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 (20mn)
- 2 Checkpointing: Protocols (40mn)
- 3 Checkpointing: Probabilistic models (30mn)
- 4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)
- 5 Hierarchical checkpointing (20mn)
- 6 Forward-recovery techniques (20mn)
 - ABFT for Linear Algebra applications
 - Composite approach: ABFT & Checkpointing
- 7 Silent errors (35mn)
- 8 Conclusion (15mn)
- 9 Advanced Models

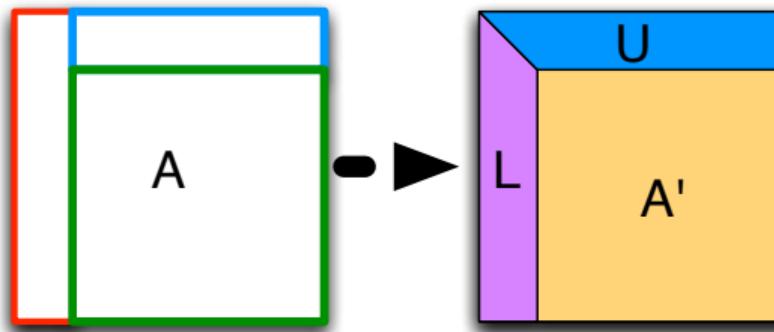
Example: block LU factorization



- 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 factorization

TRSM - Update row block

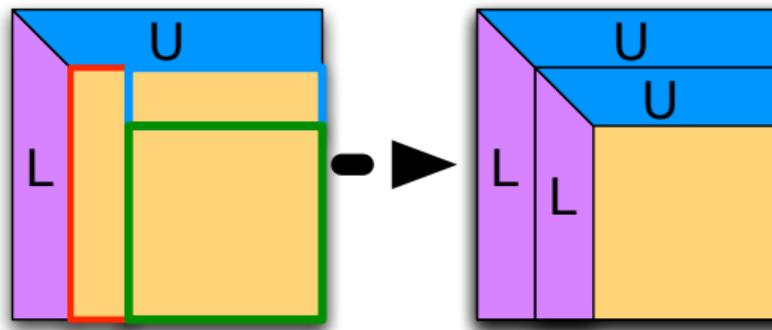


GETF2: factorize a column block
GEMM: Update the trailing matrix

- 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 factorization

TRSM - Update row block



GETF2: factorize a GEMM: Update
column block 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 factorization

Failure of rank 2

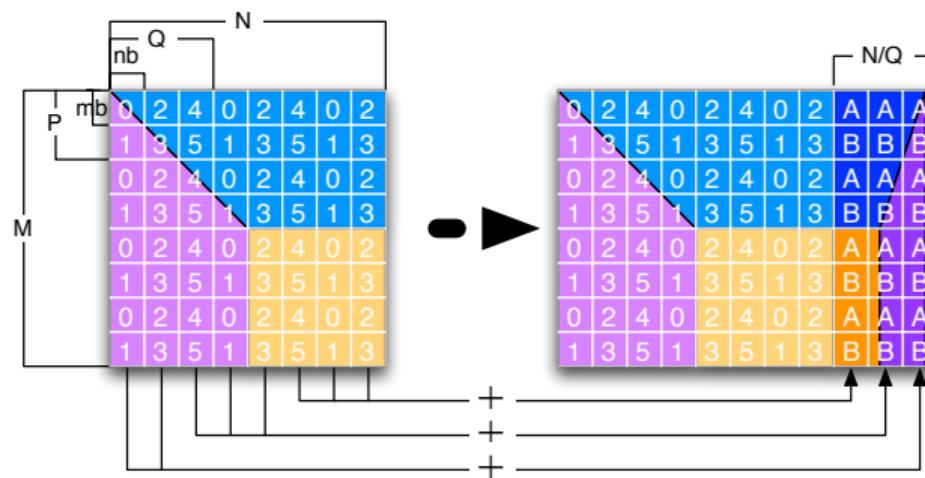
0	2	4	0	2	4	0	2
1	3	5	1	3	5	1	3
0	2	4	0	2	4	0	2
1	3	5	1	3	5	1	3
0	2	4	0	2	4	0	2
1	3	5	1	3	5	1	3
0	2	4	0	2	4	0	2
1	3	5	1	3	5	1	3



0		4	0		4	0	
1	3	5	1	3	5	1	3
0		4	0		4	0	
1	3	5	1	3	5	1	3
0		4	0		4	0	
1	3	5	1	3	5	1	3
0		4	0		4	0	
1	3	5	1	3	5	1	3

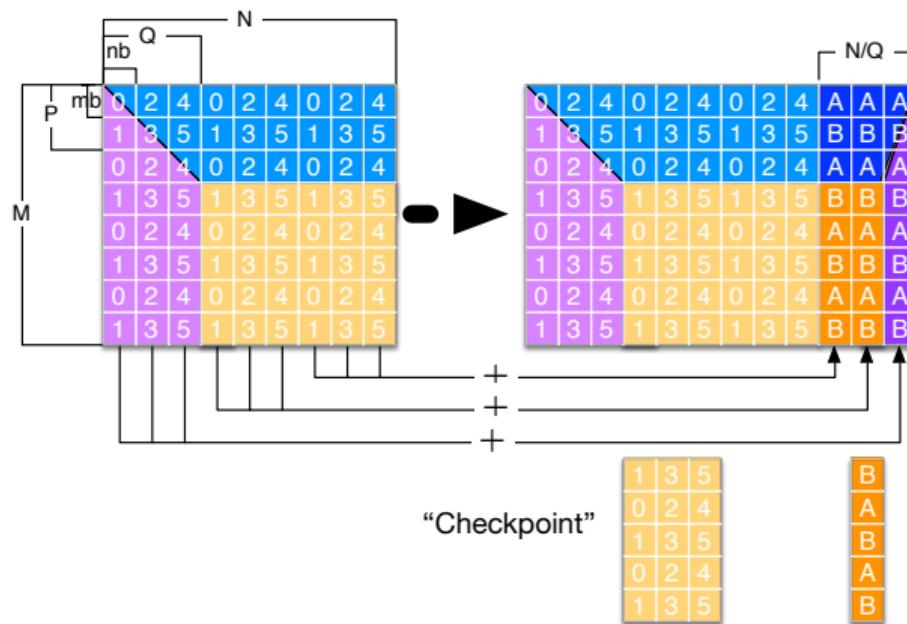
- 2D Block Cyclic Distribution (here 2×3)
- A single failure \Rightarrow many data lost

Algorithm Based Fault Tolerant LU decomposition



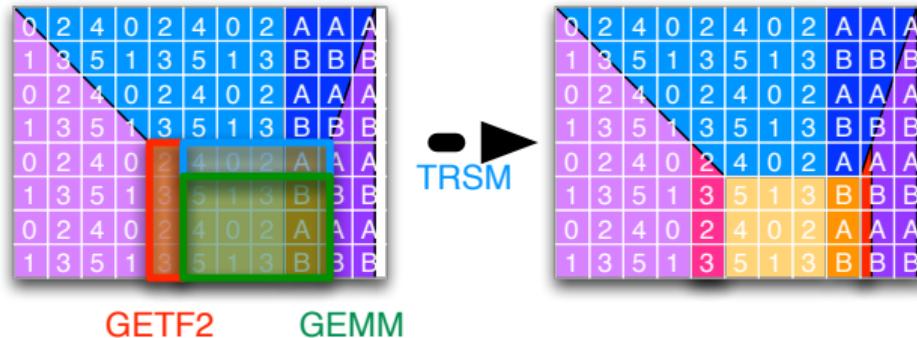
- Checksum: invertible operation on the data of the row / column
 - Checksum replication can be avoided by dedicating computing resources to checksum storage

Algorithm Based Fault Tolerant LU decomposition



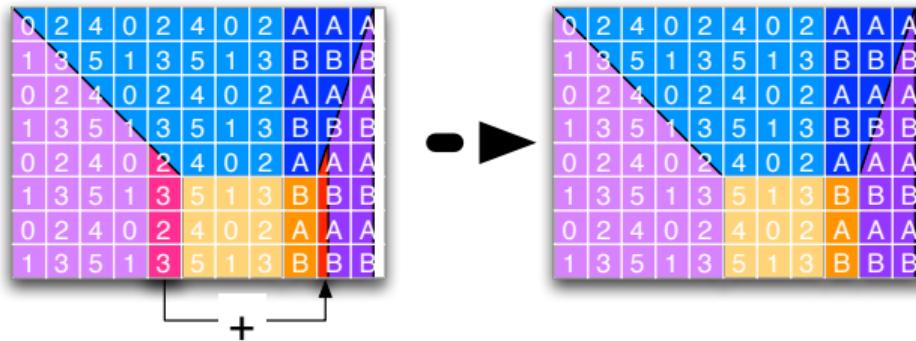
- Checkpoint the next set of Q-Panels to be able to return to it in case of failures

Algorithm Based Fault Tolerant LU decomposition



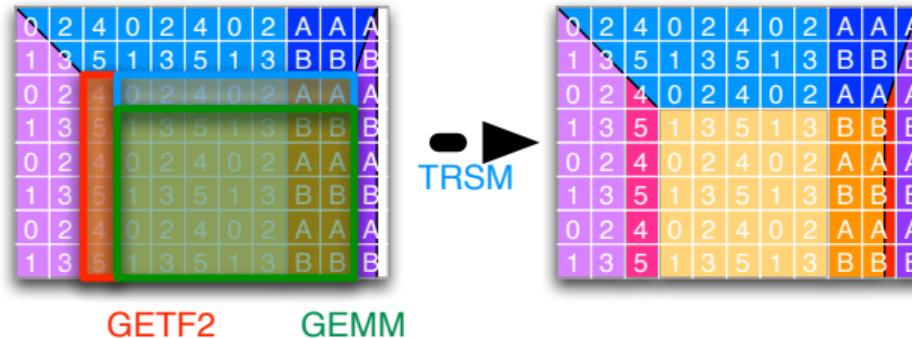
- Idea of ABFT: applying the operation on data and checksum preserves the checksum properties

Algorithm Based Fault Tolerant LU decomposition



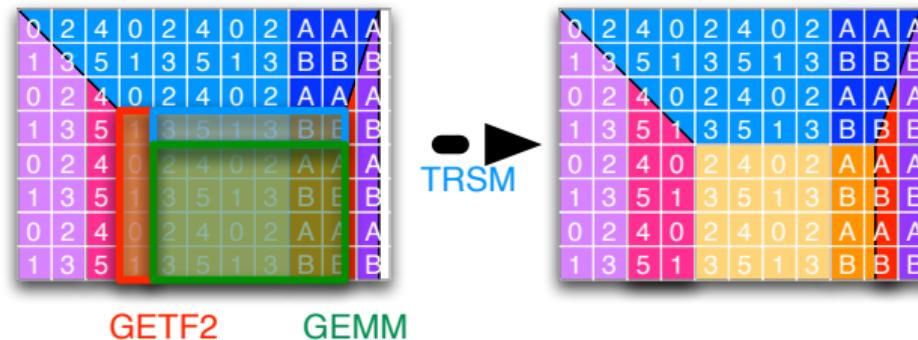
- For the part of the data that is not updated this way, the checksum must be re-calculated

Algorithm Based Fault Tolerant LU decomposition



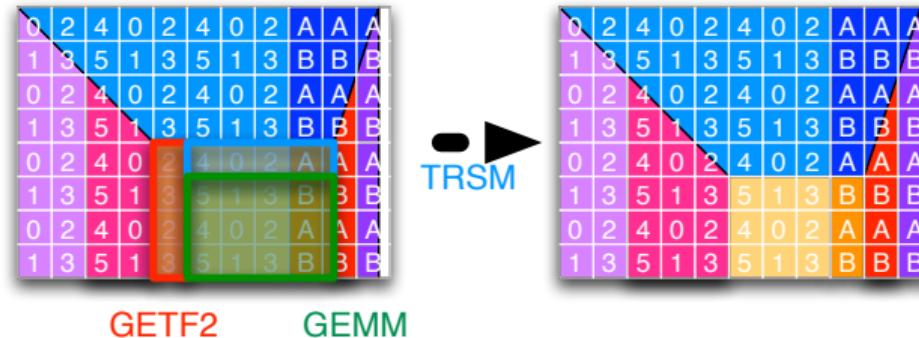
- To avoid slowing down all processors and panel operation, group checksum updates every Q block columns

Algorithm Based Fault Tolerant LU decomposition



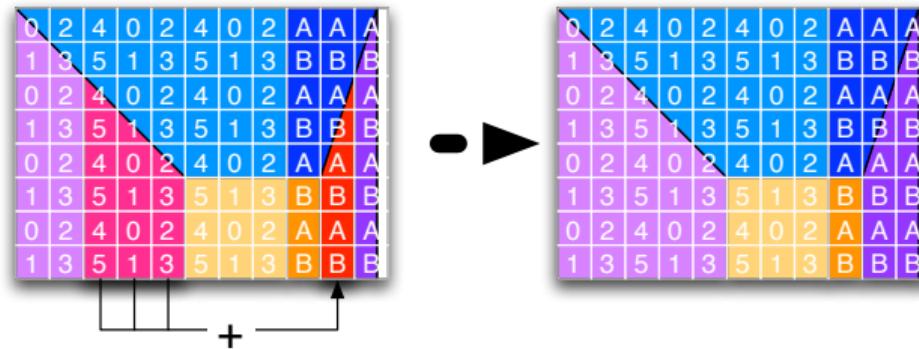
- To avoid slowing down all processors and panel operation, group checksum updates every Q block columns

Algorithm Based Fault Tolerant LU decomposition



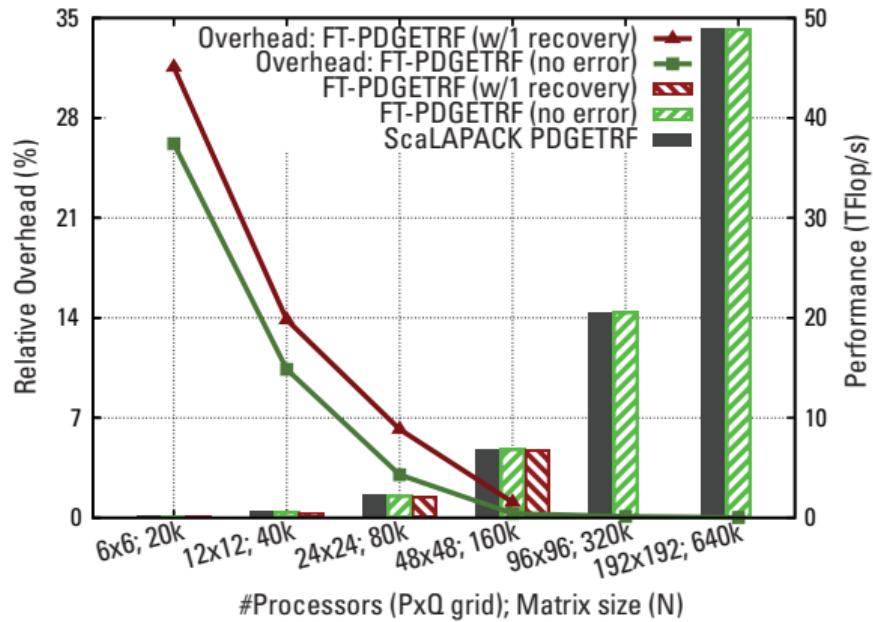
- To avoid slowing down all processors and panel operation, group checksum updates every Q block columns

Algorithm Based Fault Tolerant LU decomposition



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

ABFT LU decomposition: performance



MPI-Next ULFM Performance

- Open MPI with ULMF; Kraken supercomputer;



Outline

- 1 Introduction (20mn)
- 2 Checkpointing: Protocols (40mn)
- 3 Checkpointing: Probabilistic models (30mn)
- 4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)
- 5 Hierarchical checkpointing (20mn)
- 6 Forward-recovery techniques (20mn)
 - ABFT for Linear Algebra applications
 - Composite approach: ABFT & Checkpointing
- 7 Silent errors (35mn)
- 8 Conclusion (15mn)
- 9 Advanced Models

Fault Tolerance Techniques

General Techniques

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

Application-Specific Techniques

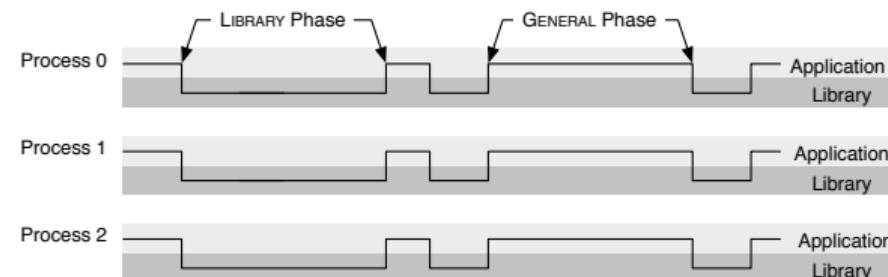
- Algorithm Based Fault Tolerance (ABFT)
- Iterative Convergence
- Approximated Computation



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

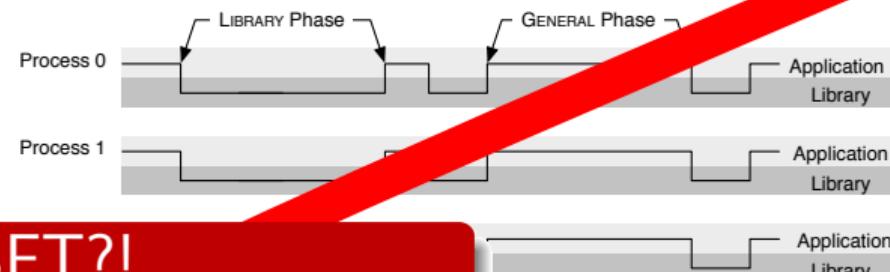
- 😊 Large part of (total) computation spent in factorization/solve
- Between LA operations:
 - 😢 use resulting vector / matrix with operations that do not preserve the checksums on the data
 - 😢 modify data not covered by ABFT algorithms

Application

Typical Application

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

Goodbye ABFT?!



- 😊 Large part ('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 Application

```
for( aninsanenumber ) {  
    /* Ex:  
     * si:  
     * m:  
     * sim2m  
     */  
}
```

Problem Statement

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

```
/* Up:  
 * wi:  
 * vec2s  
 }  
 }
```

(*) ABFT, or other application-specific FT

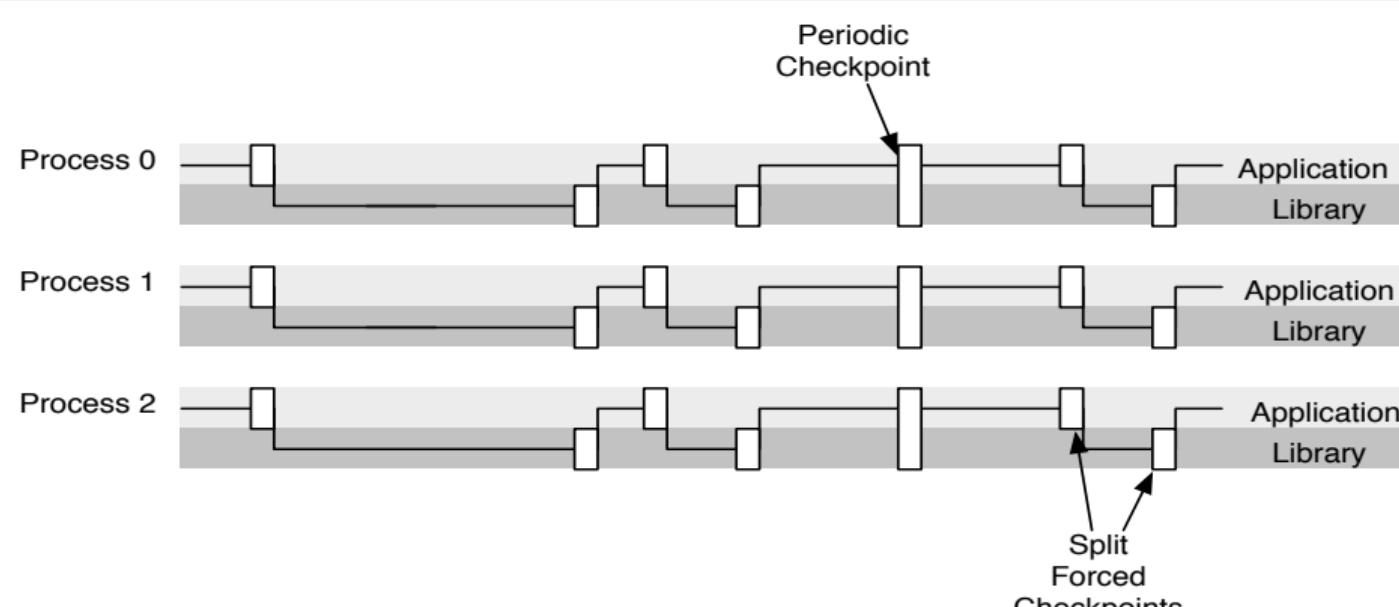
(**) Or within an application that does not have the same kind of FT

(***) And keep the application globally fault tolerant...

algorithms

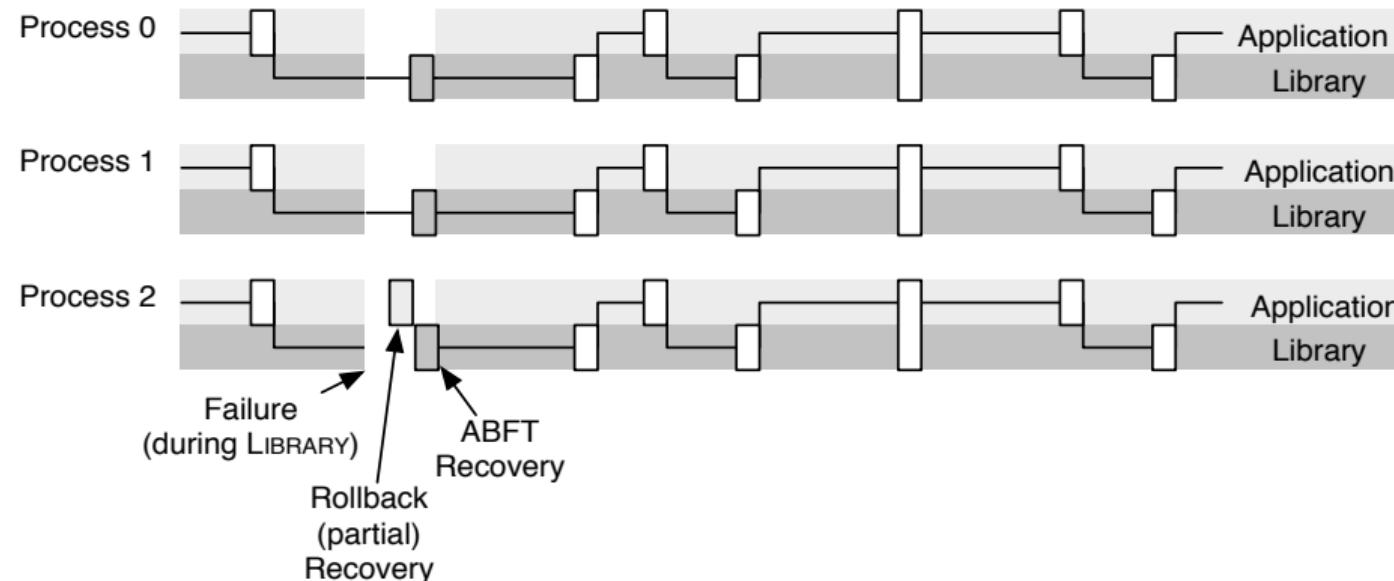
ABFT&PERIODICCKPT

ABFT&PERIODICCKPT: no failure



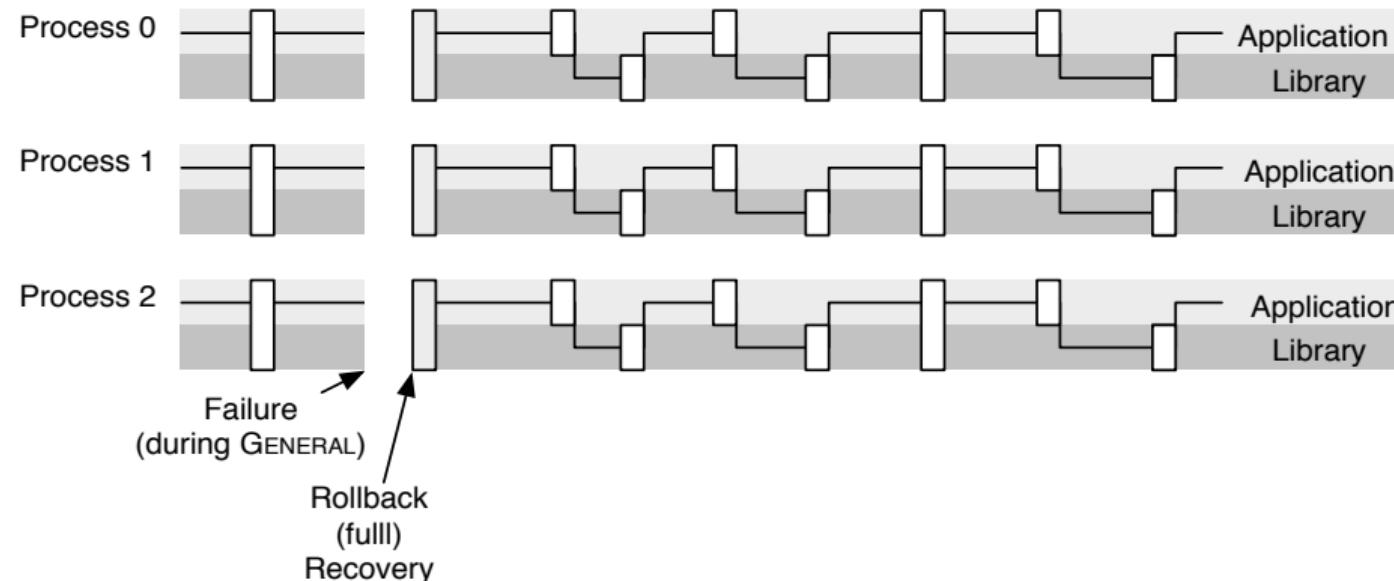
ABFT&PERIODICCKPT

ABFT&PERIODICCKPT: failure during LIBRARY phase

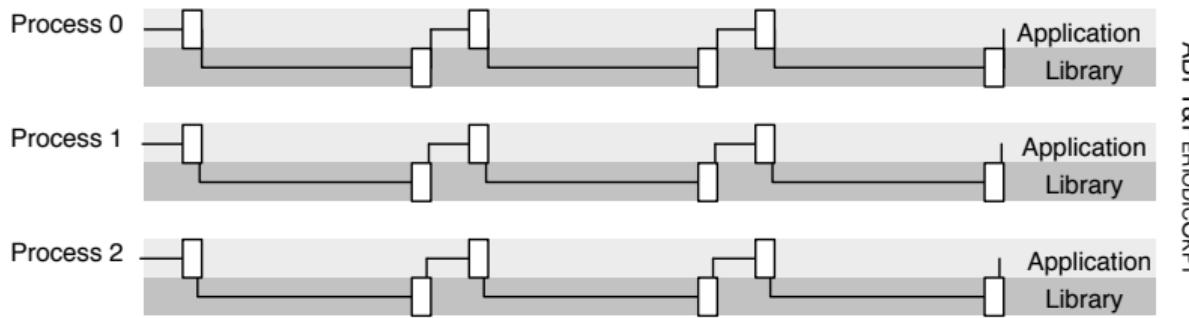


ABFT&PERIODICCKPT

ABFT&PERIODICCKPT: failure during GENERAL phase



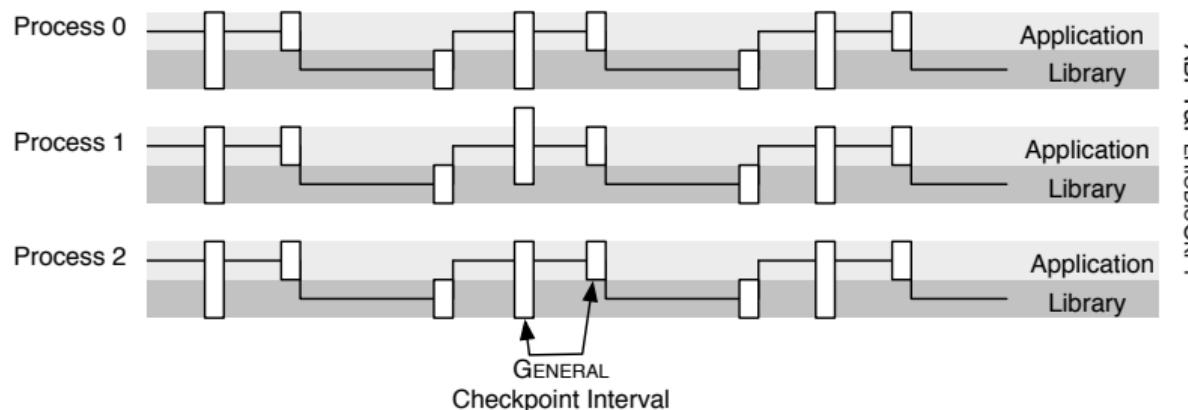
ABFT&PERIODICCKPT: Optimizations



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

Toward Exascale, and Beyond!

Let's think at scale

- Number of components ↗⇒ MTBF ↓
 - Number of components ↗⇒ Problem Size ↗
 - Problem Size ↗⇒
 Computation Time spent in LIBRARY phase ↗
-
- 😊 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.

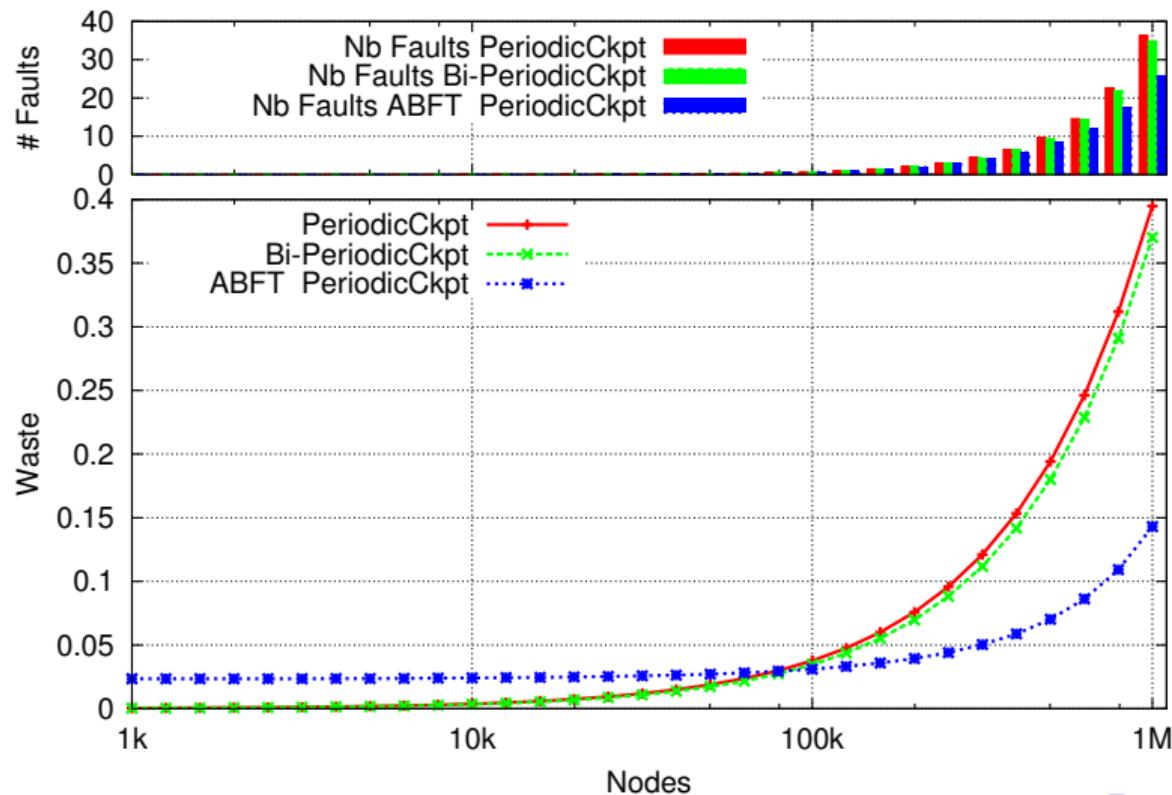
ABFT&PeriodicCkpt The algorithm described above

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

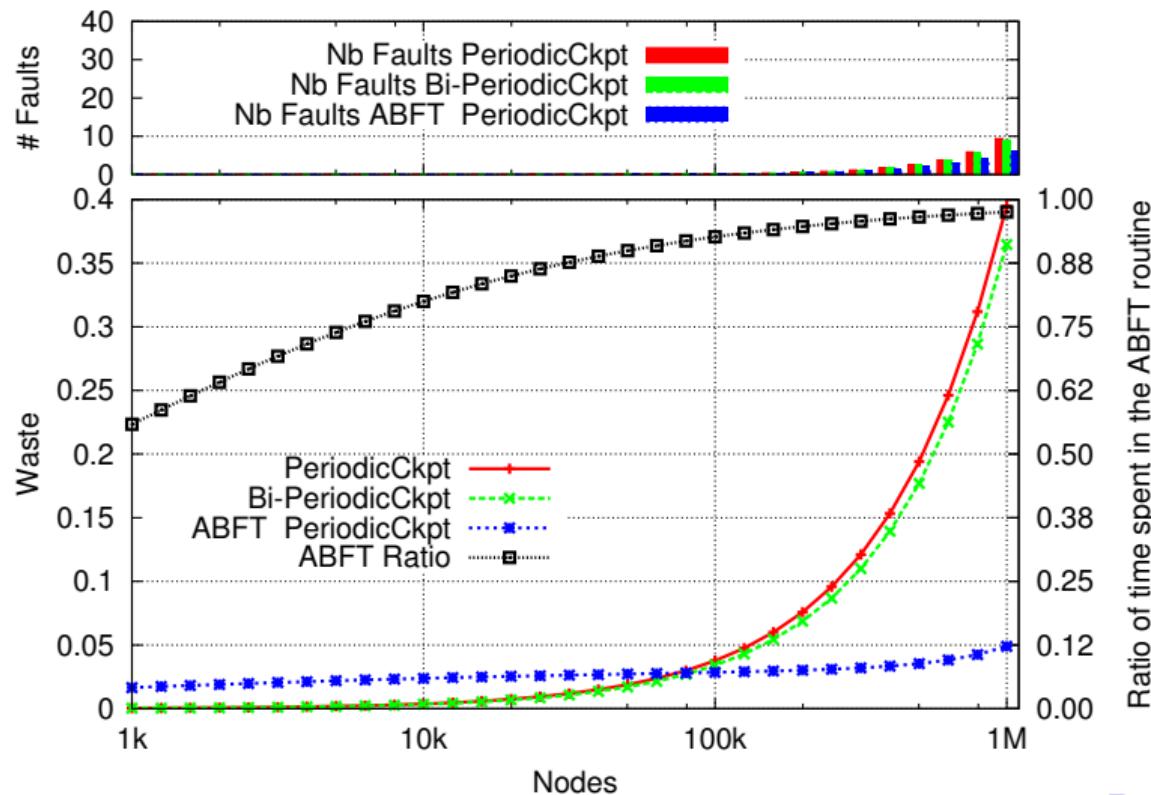


Weak Scale #2

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

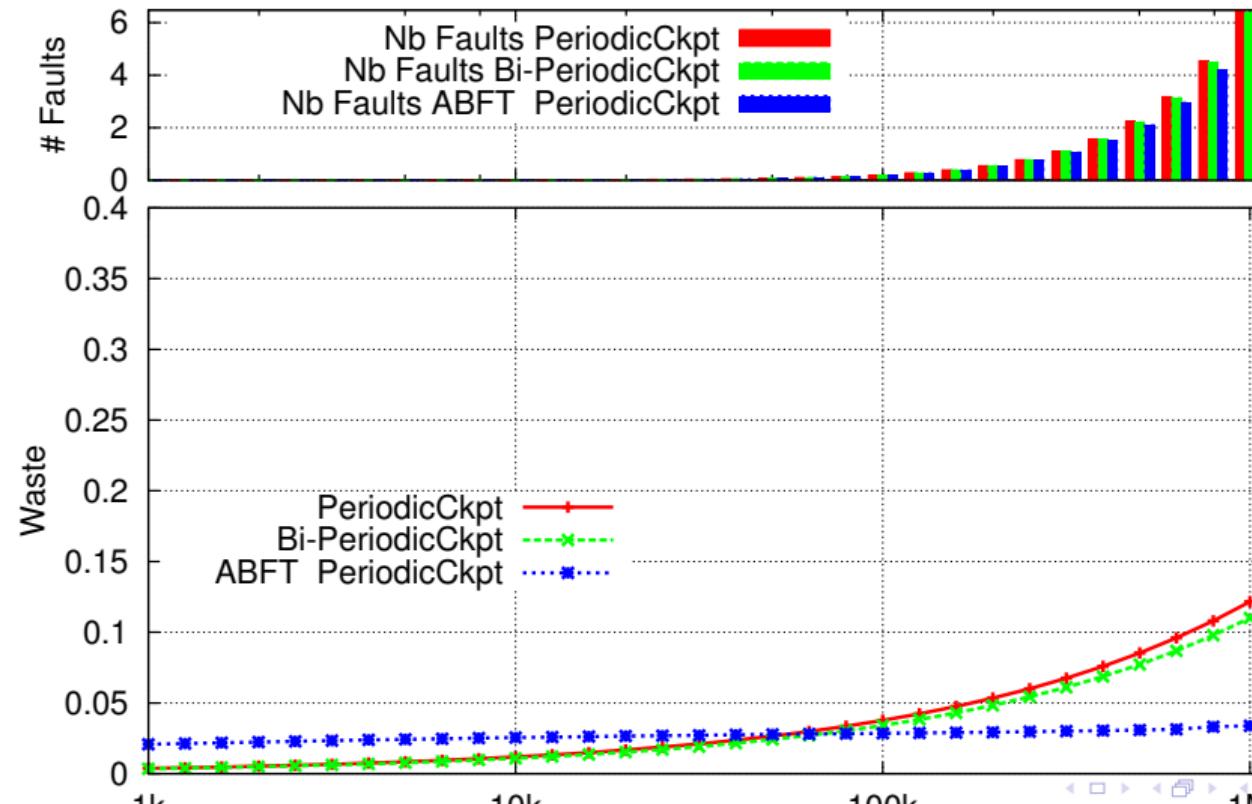


Weak Scale #3

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



Conclusion

- Application Specific Techniques are **harder** to design
- But they are **much more efficient**
- They are often **not sufficient**
 - Because not all the application is amenable to a technique
 - Because the technique might not tolerate all kind of failures
- **Composition** of approaches is often necessary

Outline

- 1 Introduction (20mn)
- 2 Checkpointing: Protocols (40mn)
- 3 Checkpointing: Probabilistic models (30mn)
- 4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)
- 5 Hierarchical checkpointing (20mn)
- 6 Forward-recovery techniques (20mn)
- 7 Silent errors (35mn)
 - Coupling checkpointing and verification
 - Application-specific methods
- 8 Conclusion (15mn)
- 9 Advanced Models

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 AI 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

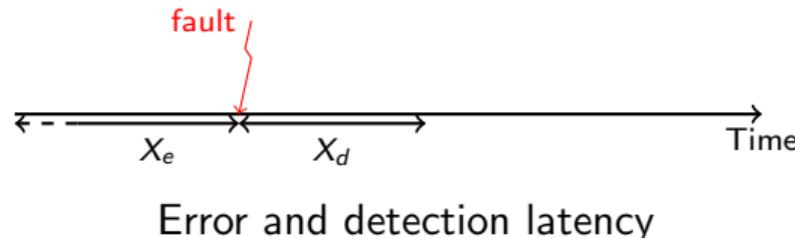


Theorem: $\mu_p = \frac{\mu_{\text{ind}}}{p}$ for arbitrary distributions

Outline

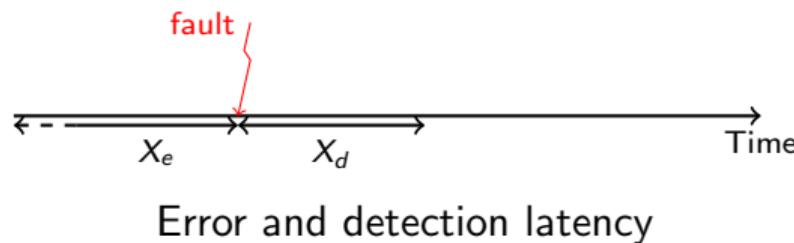
- 1 Introduction (20mn)
- 2 Checkpointing: Protocols (40mn)
- 3 Checkpointing: Probabilistic models (30mn)
- 4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)
- 5 Hierarchical checkpointing (20mn)
- 6 Forward-recovery techniques (20mn)
- 7 Silent errors (35mn)
 - Coupling checkpointing and verification
 - Application-specific methods
- 8 Conclusion (15mn)
- 9 Advanced Models

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
 - 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

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

```

1 : Compute  $r^{(0)} = b - Ax^{(0)}$ ,  $z^{(0)} = M^{-1}r^{(0)}$ ,  $p^{(0)} = z^{(0)}$ ,
   and  $\rho_0 = r^{(0)T}z^{(0)}$  for some initial guess  $x^{(0)}$ 
2 : checkpoint:  $A$ ,  $M$ , and  $b$ 
3 : for  $i = 0, 1, \dots$ 
4 :   if (  $(i > 0)$  and  $(i \% d = 0)$  )
5 :     if (  $\frac{p^{(i+1)T}q^{(i)}}{\|p^{(i+1)}\| \cdot \|q^{(i)}\|} > 10^{-10}$ 
        or  $\frac{\|r^{(i+1)} + Ax^{(i+1)} - b\|}{\|b\| \cdot \|A\|} > 10^{-10}$  )
6 :       recover:  $A$ ,  $M$ ,  $b$ ,  $i$ ,  $\rho_i$ ,
            $p^{(i)}$ ,  $x^{(i)}$ , and  $r^{(i)}$ .
7 :     else if (  $i \% (cd) = 0$  )
8 :       checkpoint:  $i$ ,  $\rho_i$ ,  $p^{(i)}$ , and  $x^{(i)}$ 
9 :   endif
10:  endif
11:   $q^{(i)} = Ap^{(i)}$ 
12:   $\alpha_i = \rho_i / p^{(i)T}q^{(i)}$ 
13:   $x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$ 
14:   $r^{(i+1)} = r^{(i)} - \alpha_i q^{(i)}$ 
15:  solve  $Mz^{(i+1)} = r^{(i+1)}$ , where  $M = M^T$ 
16:   $\rho_{i+1} = r^{(i+1)T}z^{(i+1)}$ 
17:   $\beta_i = \rho_{i+1} / \rho_i$ 
18:   $p^{(i+1)} = z^{(i+1)} + \beta_i p^{(i)}$ 
19:  check convergence; continue if necessary
20: end

```

Zizhong Chen, PPoPP'13

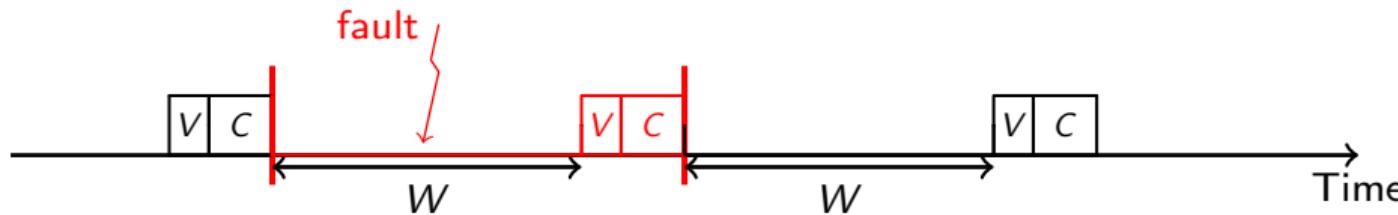
- Iterate PCG

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

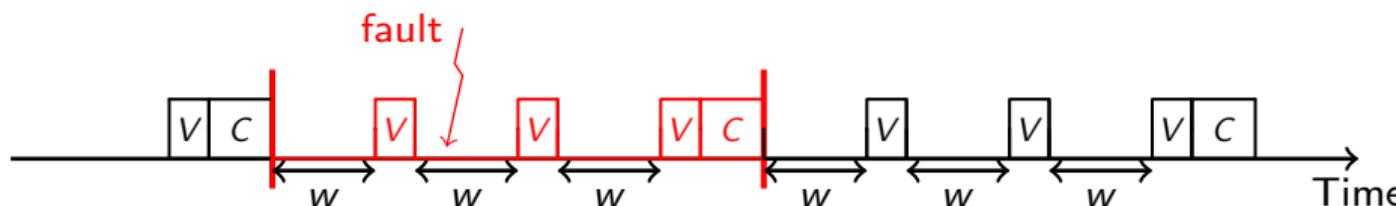
Cost: three vectors, or two vectors + SpMV at recovery
- Experimental method to choose c and d

Base pattern (and revisiting Young/Daly)



	Fail-stop (classical)	Silent errors
Pattern	$T = W + C$	$S = W + V + C$
WASTE[FF]	$\frac{C}{T}$	$\frac{V+C}{S}$
WASTE[fail]	$\frac{1}{\mu}(D + R + \frac{W}{2})$	$\frac{1}{\mu}(R + W + V)$
Optimal	$T_{\text{opt}} = \sqrt{2C\mu}$	$S_{\text{opt}} = \sqrt{(C + V)\mu}$
WASTE[opt]	$\sqrt{\frac{2C}{\mu}}$	$2\sqrt{\frac{C+V}{\mu}}$

With $p = 1$ checkpoint and $q = 3$ verifications



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

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

Outline

- 1 Introduction (20mn)
- 2 Checkpointing: Protocols (40mn)
- 3 Checkpointing: Probabilistic models (30mn)
- 4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)
- 5 Hierarchical checkpointing (20mn)
- 6 Forward-recovery techniques (20mn)
- 7 Silent errors (35mn)
 - ➊ Coupling checkpointing and verification
 - ➋ Application-specific methods
- 8 Conclusion (15mn)
- 9 Advanced Models

Literature (1/2)

- 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)
- Algorithm-based focused recovery: use application data-flow to identify potential error source and corrupted nodes (Fang and Chien 2014)

Literature (2/2)

- Dynamic monitoring of datasets based on physical laws (e.g., temperature/speed limit) and space or temporal proximity (Bautista-Gomez and Cappello)
- Time-series prediction, spatial multivariate interpolation (Di et al.)
- Offline training, online detection based on SDC signature for convergent iterative applications (Liu and Agrawal)
- Spatial regression based on support vector machines (Subasi et al.)
- Many others at-a-analytics/machine learning approaches

Application-specific detectors

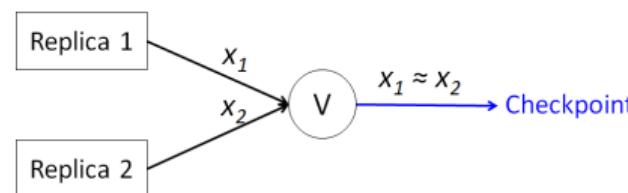
Do you believe it?

- Detectors are not perfect
- High recall is expensive if at all achievable
- With higher error rates, it would be good to correct a few errors

Replication mandatory at scale? 😞

Why Is Replication Useful?

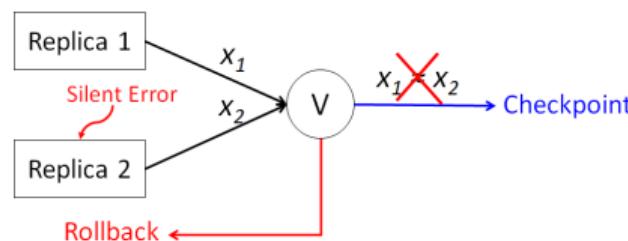
- Error detection (duplication):



- Error correction (triplication):

Why Is Replication Useful?

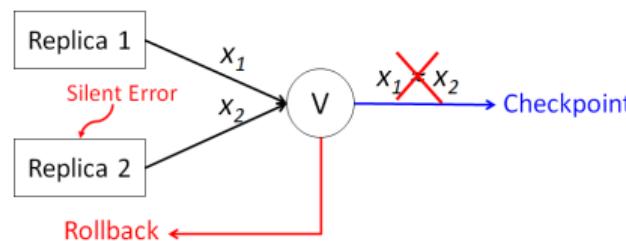
- Error detection (duplication):



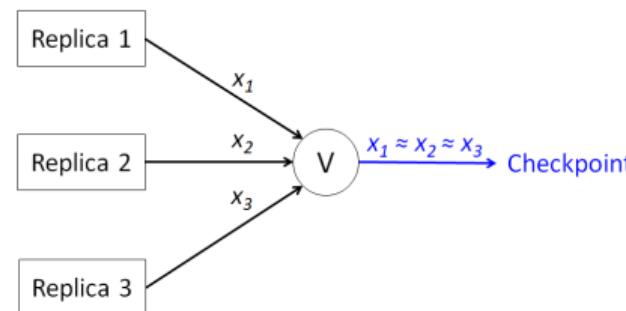
- Error correction (triplication):

Why Is Replication Useful?

- Error detection (duplication):

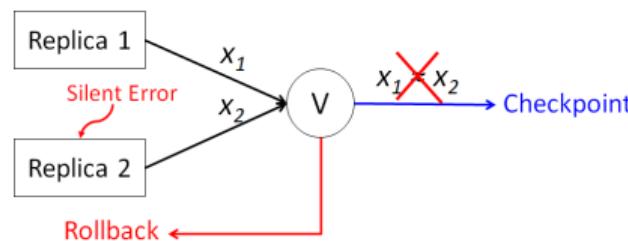


- Error correction (triplication):

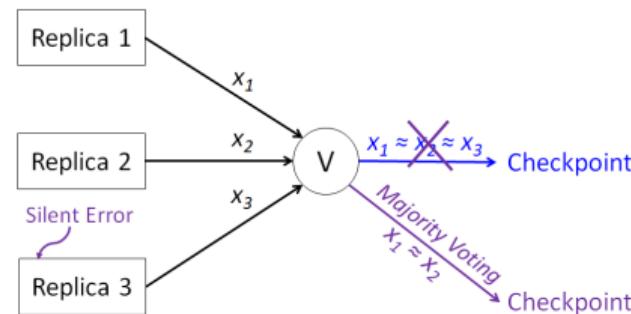


Why Is Replication Useful?

- Error detection (duplication):

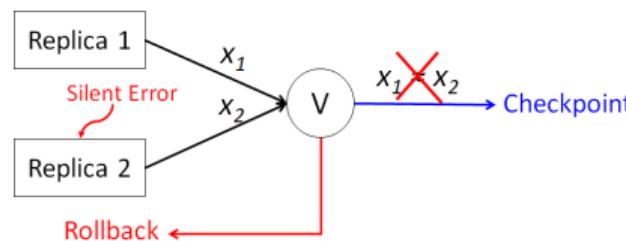


- Error correction (triplication):

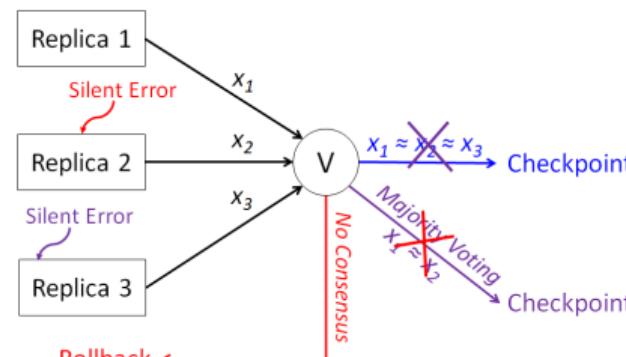


Why Is Replication Useful?

- Error detection (duplication):

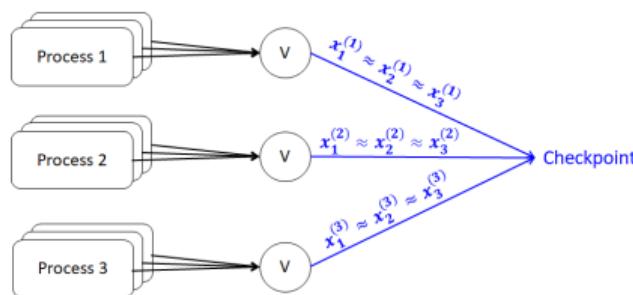


- Error correction (triplication):

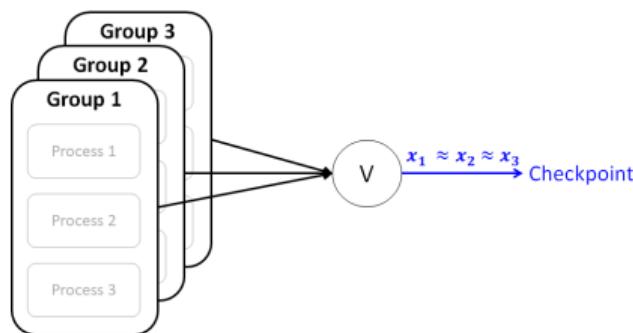


Two Replication Modes

- **Process Replication:**

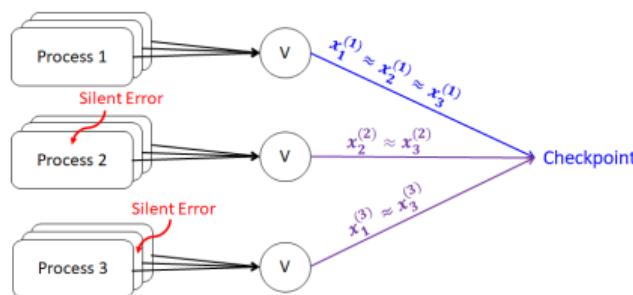


- **Group Replication:**

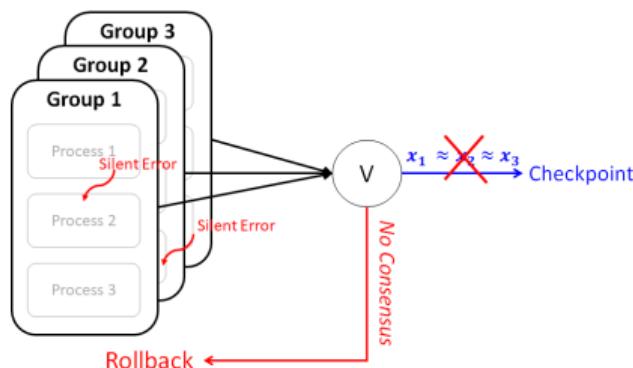


Two Replication Modes

- **Process Replication:**



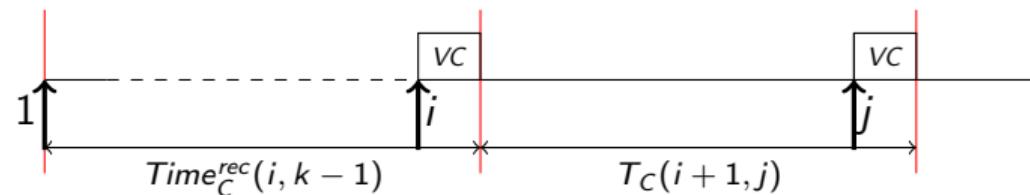
- **Group Replication:**



Dynamic programming for linear chains of tasks

- $\{T_1, T_2, \dots, T_n\}$: linear chain of n tasks
- Each task T_i fully parametrized:
 - w_i computational weight
 - C_i, R_i, V_i : checkpoint, recovery, verification
- Error rates:
 - λ^F rate of fail-stop errors
 - λ^S rate of silent errors

VC-ONLY

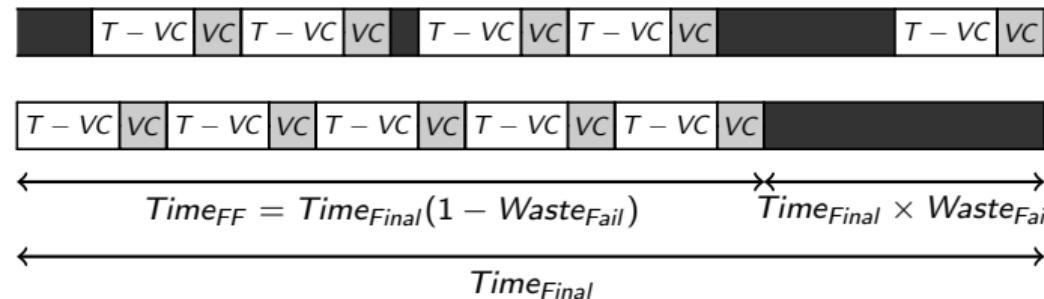


$$\min_{0 \leq k < n} Time_C^{rec}(n, k)$$

$$Time_C^{rec}(j, k) = \min_{k \leq i < j} \{ Time_C^{rec}(i, k-1) + T_C^{SF}(i+1, j) \}$$

$$T_C^{SF}(i, j) = p_{i,j}^F (T_{lost_{i,j}} + R_{i-1} + T_C^{SF}(i, j)) \\ + (1 - p_{i,j}^F) \left(\sum_{\ell=i}^j w_\ell + V_j + p_{i,j}^S (R_{i-1} + T_C^{SF}(i, j)) + (1 - p_{i,j}^S) C_j \right)$$

Young/Daly



$$\text{Waste} = \text{Waste}_{ef} + \text{Waste}_{fail}$$

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

$$T_{\text{opt}} = \sqrt{\frac{2(V + C)}{\lambda^F(s) + 2\lambda^S(s)}}$$

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

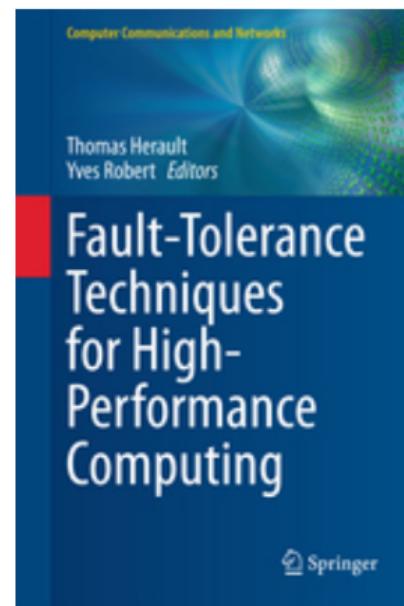
Silent errors

- 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 😊**

Bibliography



First chapter = extensive survey, freely available as LAWN 289 (LApack Working Note)

Outline

1 Introduction (20mn)

2 Checkpointing: Protocols (40mn)

3 Checkpointing: Probabilistic models (30mn)

4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)

5 Hierarchical checkpointing (20mn)

6 Forward-recovery techniques (20mn)

7 Silent errors (35mn)

8 Conclusion (15mn)

9 Advanced Models

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?

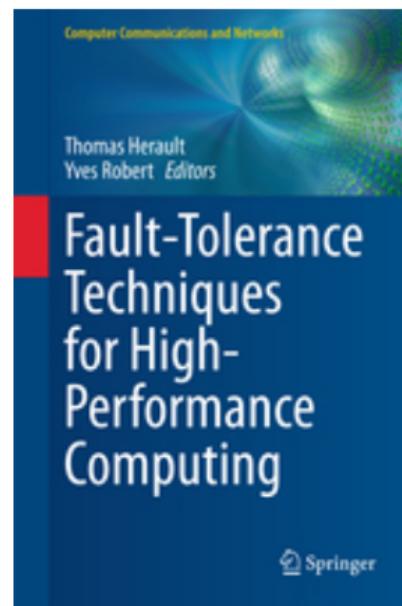
Conclusion

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



First chapter = extensive survey, freely available as LAWN 289 (LApack Working Note)

Your opinion matters!

File the SC17 tutorial evaluation form

<http://bit.ly/sc17-eval>



Outline

1 Introduction (20mn)

2 Checkpointing: Protocols (40mn)

3 Checkpointing: Probabilistic models (30mn)

4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)

5 Hierarchical checkpointing (20mn)

6 Forward-recovery techniques (20mn)

7 Silent errors (35mn)

8 Conclusion (15mn)

9 Advanced Models

- Hierarchical checkpointing
- Failure Prediction
- Replication

Outline

- 1 Introduction (20mn)
- 2 Checkpointing: Protocols (40mn)
- 3 Checkpointing: Probabilistic models (30mn)
- 4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)
- 5 Hierarchical checkpointing (20mn)
- 6 Forward-recovery techniques (20mn)
- 7 Silent errors (35mn)
- 8 Conclusion (15mn)
- 9 Advanced Models
 - Hierarchical checkpointing
 - Failure Prediction
 - Replication

Which checkpointing protocol to use?

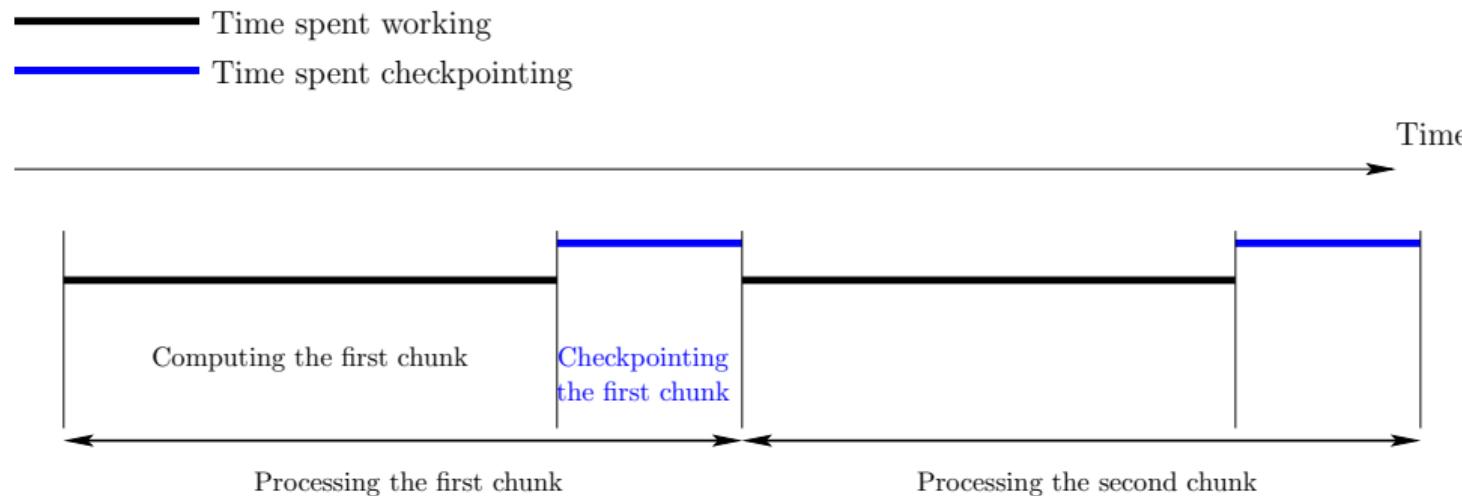
Coordinated checkpointing

- 😊 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

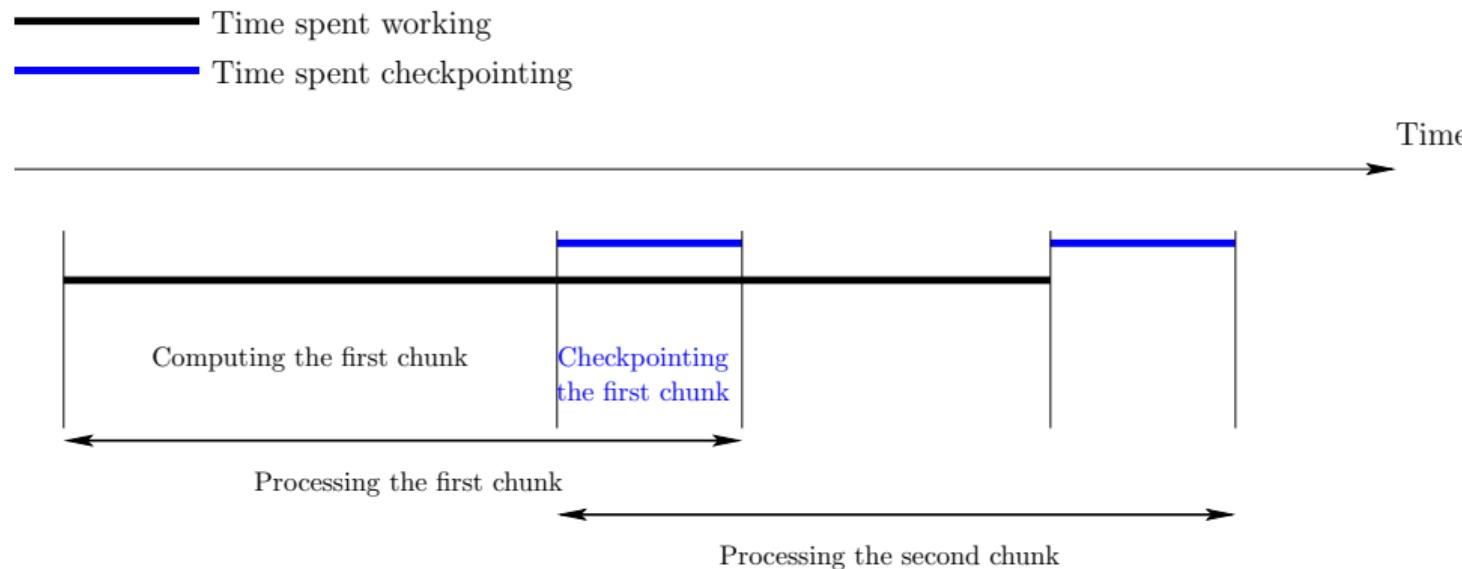
- 😢 Need to log inter-groups messages
 - Slowdowns failure-free execution
 - Increases checkpoint size/time
- 😊 Only processors from failed group need to roll back
- 😊 Faster re-execution with logged messages
- 😊 Rumor: Should scale to very large platforms

Blocking vs. non-blocking



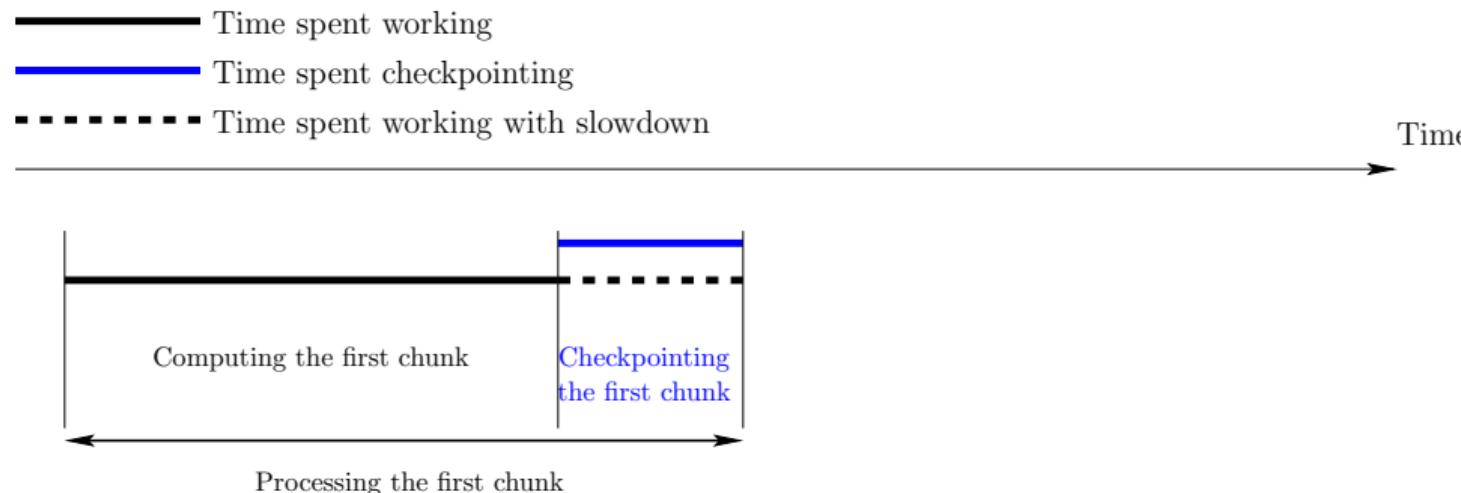
Blocking model: checkpointing blocks all computations

Blocking vs. non-blocking



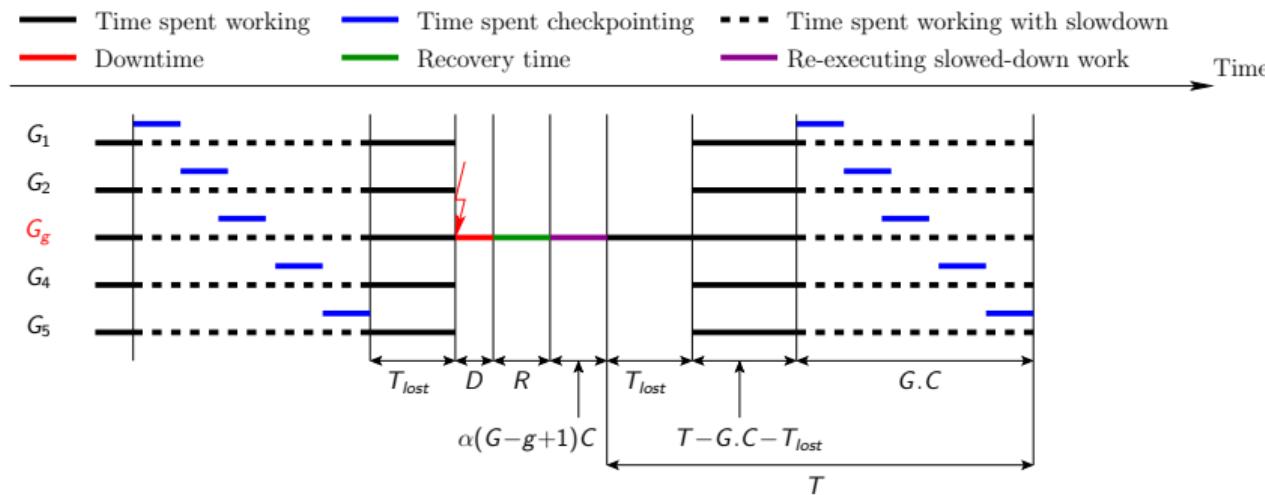
Non-blocking model: checkpointing has no impact on computations (e.g., first copy state to RAM, then copy RAM to disk)

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 αC without checkpointing ($0 \leq \alpha \leq 1$)

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

Four platforms: basic characteristics

Name	Number of cores	Number of processors P_{total}	Number of cores per processor	Memory per processor	I/O Network Read	Bandwidth (b_{io}) Write	I/O Bandwidth (b_{port}) Read/Write per processor
Titan	299,008	16,688	16	32GB	300GB/s	300GB/s	20GB/s
K-Computer	705,024	88,128	8	16GB	150GB/s	96GB/s	20GB/s
Exascale-Slim	1,000,000,000	1,000,000	1,000	64GB	1TB/s	1TB/s	200GB/s
Exascale-Fat	1,000,000,000	100,000	10,000	640GB	1TB/s	1TB/s	400GB/s

Name	Scenario	$G(C(q))$	β for 2D-STENCIL	β for MATRIX-PRODUCT
Titan	COORD-IO	1 (2,048s)	/	/
	HIERARCH-IO	136 (15s)	0.0001098	0.0004280
	HIERARCH-PORT	1,246 (1.6s)	0.0002196	0.0008561
K-Computer	COORD-IO	1 (14,688s)	/	/
	HIERARCH-IO	296 (50s)	0.0002858	0.001113
	HIERARCH-PORT	17,626 (0.83s)	0.0005716	0.002227
Exascale-Slim	COORD-IO	1 (64,000s)	/	/
	HIERARCH-IO	1,000 (64s)	0.0002599	0.001013
	HIERARCH-PORT	200,0000 (0.32s)	0.0005199	0.002026
Exascale-Fat	COORD-IO	1 (64,000s)	/	/
	HIERARCH-IO	316 (217s)	0.00008220	0.0003203
	HIERARCH-PORT	33,3333 (1.92s)	0.00016440	0.0006407

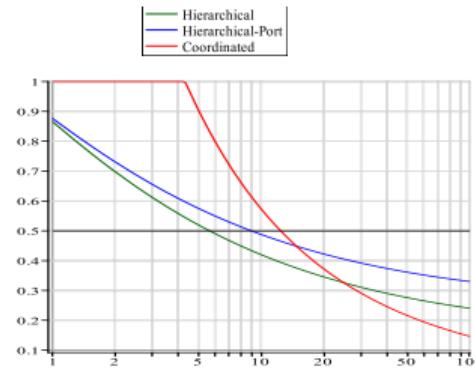
Checkpoint time

Name	C
K-Computer	14,688s
Exascale-Slim	64,000
Exascale-Fat	64,000

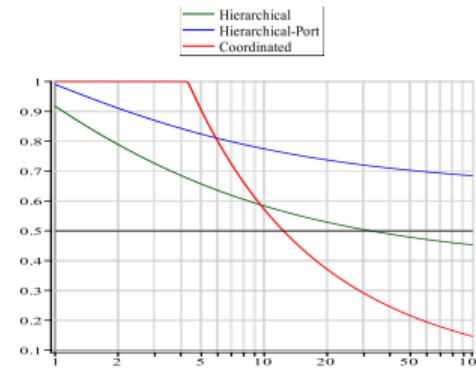
- Large time to dump the memory
- Using $1\%C$
- Comparing with $0.1\%C$ for exascale platforms
- $\alpha = 0.3$, $\lambda = 0.98$ and $\rho = 1.5$

Plotting formulas – Platform: Titan

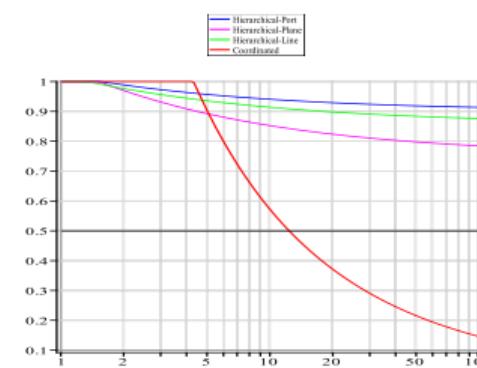
Stencil 2D



Matrix product



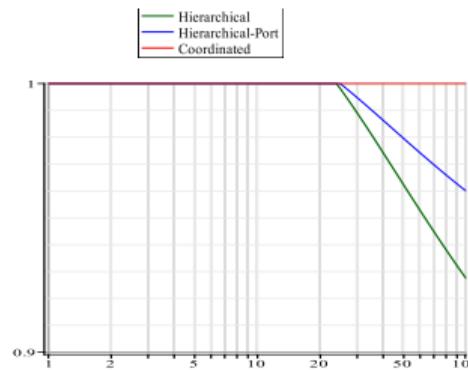
Stencil 3D



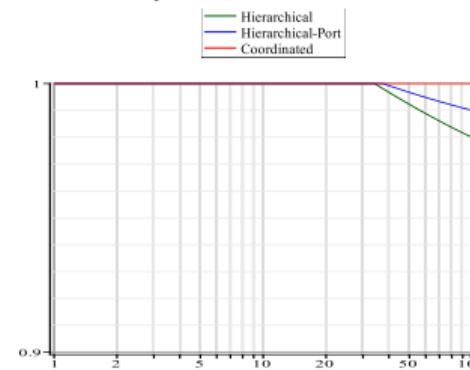
Waste as a function of processor MTBF μ_{ind}

Platform: K-Computer

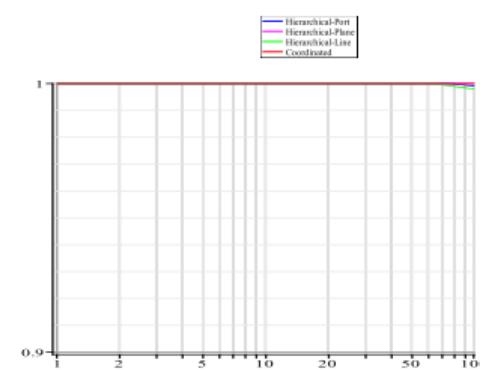
Stencil 2D



Matrix product



Stencil 3D



Waste as a function of processor MTBF μ_{ind}

Plotting formulas – Platform: Exascale

WASTE = 1 for all scenarios!!!

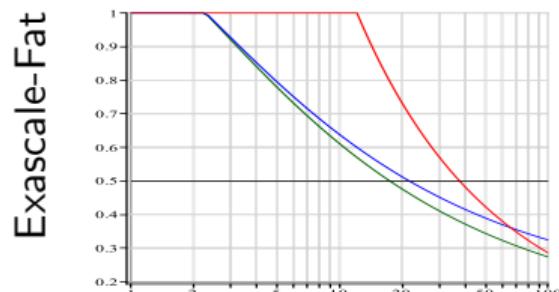
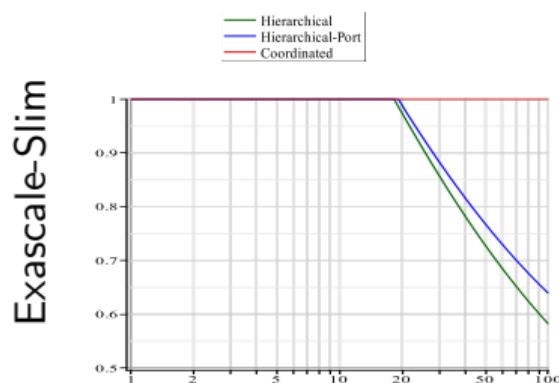
Plotting formulas – Platform: Exascale

RASTE = ... for all scenarios!!!

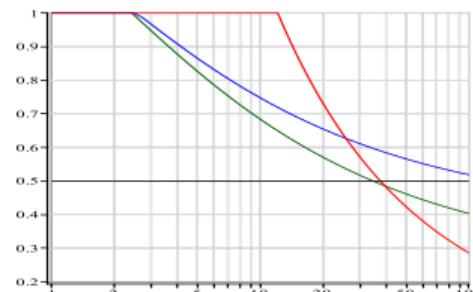
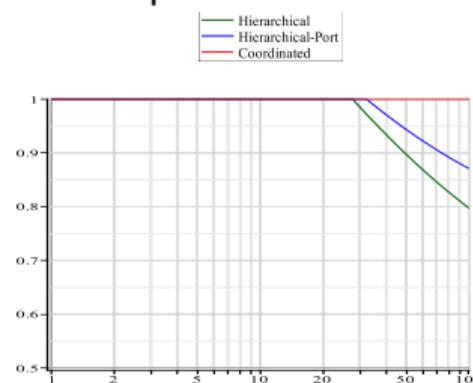
Goodbye Exascale?!

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

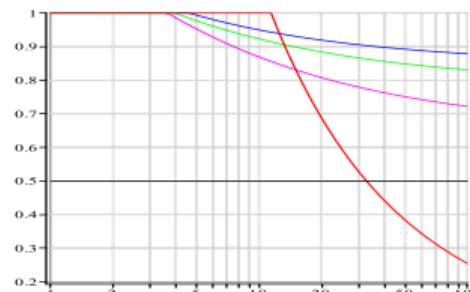
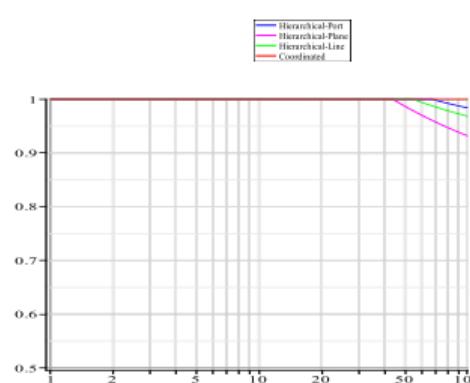
Stencil 2D



Matrix product



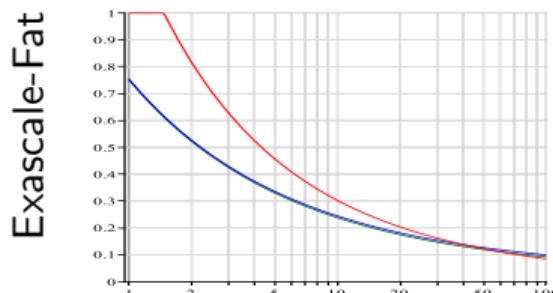
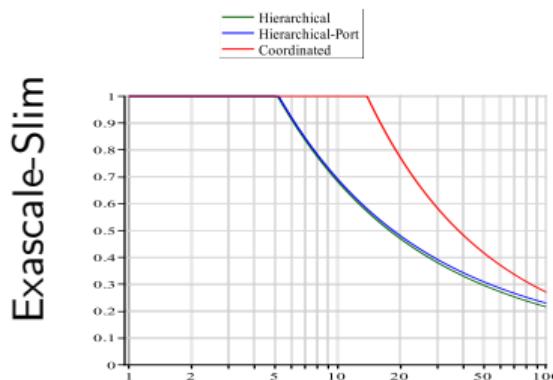
Stencil 3D



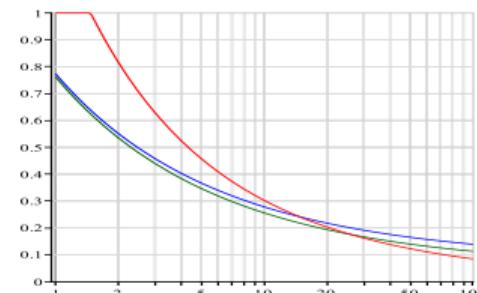
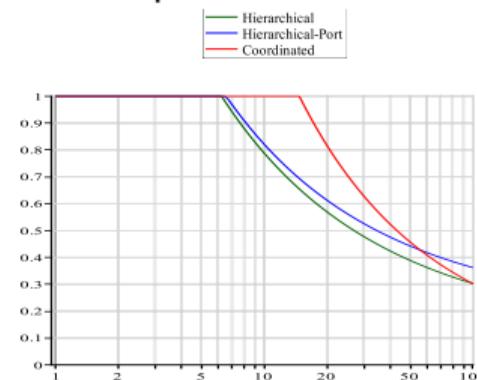
Waste as a function of processor MTBF μ_{ind} , $C = 1,000$

Plotting formulas – Platform: Exascale with $C = 100$

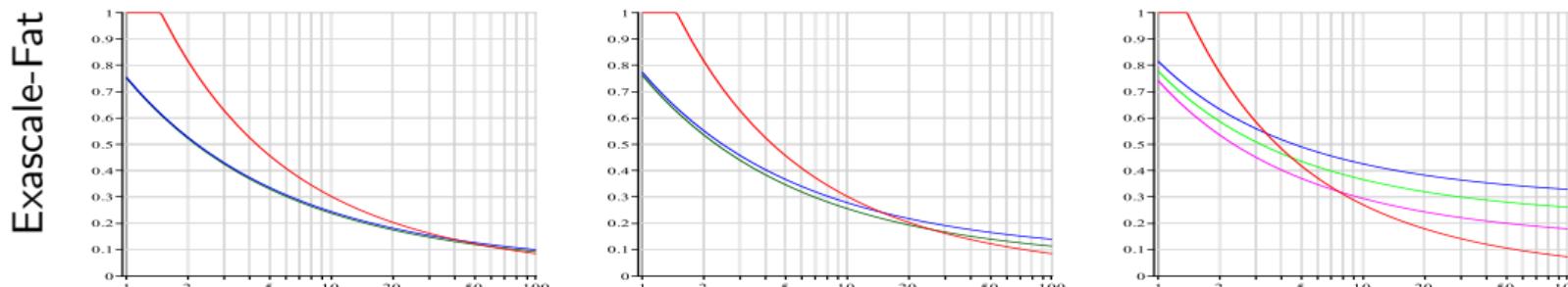
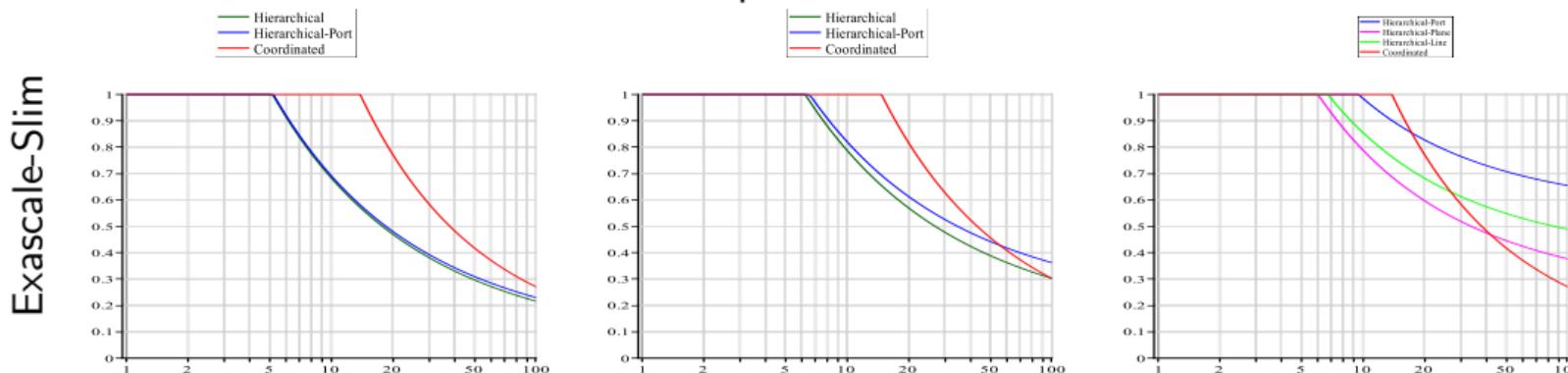
Stencil 2D



Matrix product



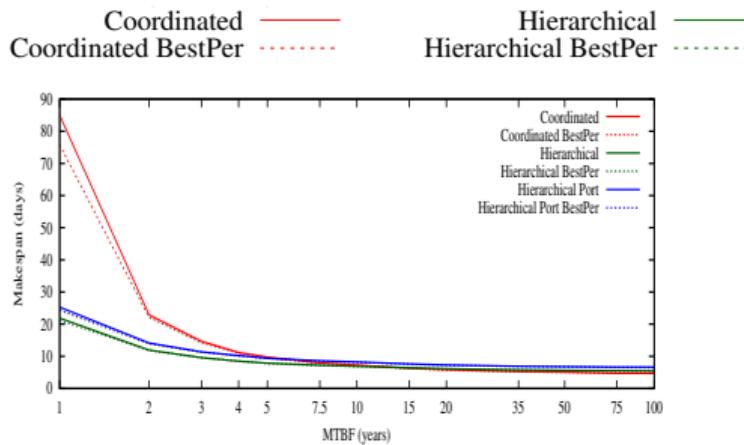
Stencil 3D



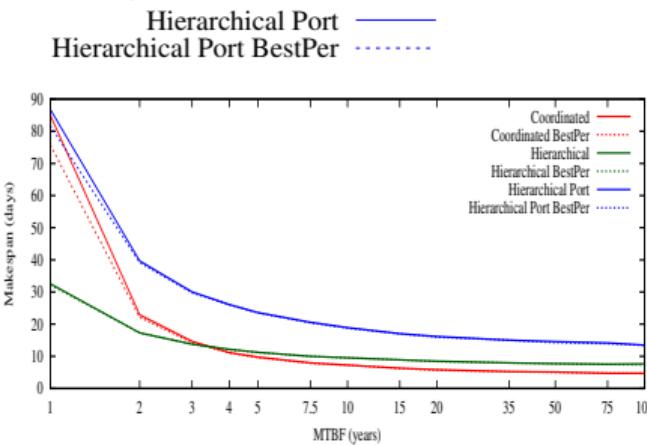
Waste as a function of processor MTBF μ_{ind} , $C = 100$

Simulations – Platform: Titan

Stencil 2D



Matrix product

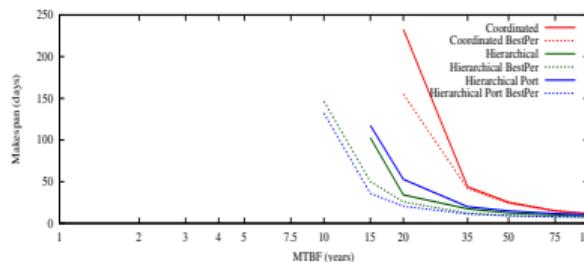


Makespan (in days) as a function of processor MTBF μ_{ind}

Simulations – Platform: Exascale with $C = 1,000$

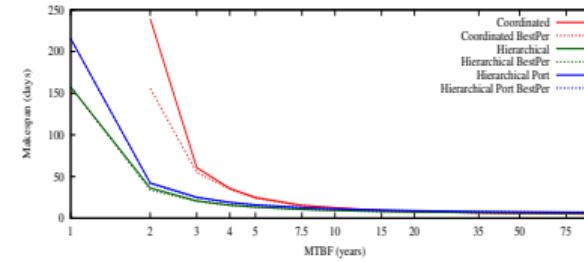
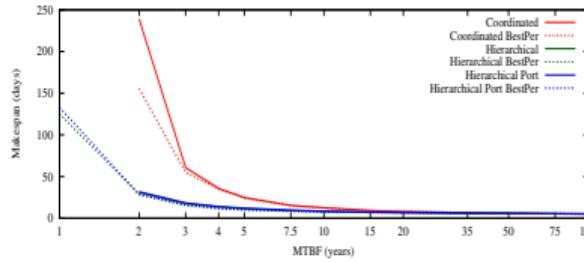
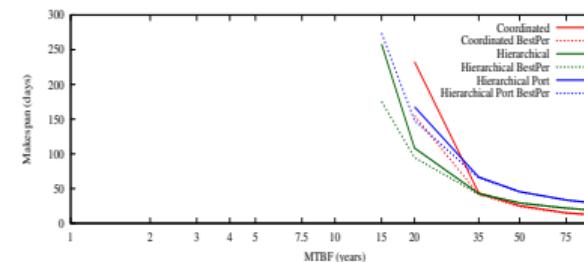
Stencil 2D

Exascale-Slim



Matrix product

Exascale-Fat



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

Outline

- 1 Introduction (20mn)
- 2 Checkpointing: Protocols (40mn)
- 3 Checkpointing: Probabilistic models (30mn)
- 4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)
- 5 Hierarchical checkpointing (20mn)
- 6 Forward-recovery techniques (20mn)
- 7 Silent errors (35mn)
- 8 Conclusion (15mn)
- 9 Advanced Models
 - Hierarchical checkpointing
 - Failure Prediction
 - Replication

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

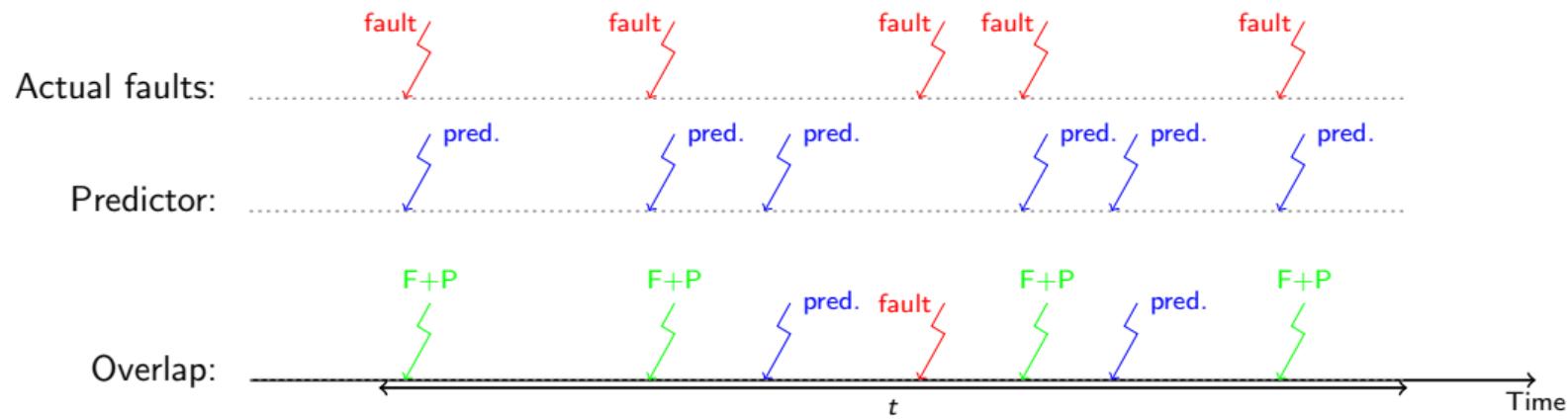
- μ : mean time between failures (MTBF)
- μ_P mean time between predicted events (both true positive and false positive)
- μ_{NP} mean time between unpredicted faults (false negative).
- μ_e : mean time between events (including three event types)

$$r = \frac{\text{True}_P}{\text{True}_P + \text{False}_N} \quad \text{and} \quad p = \frac{\text{True}_P}{\text{True}_P + \text{False}_P}$$

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

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

Example



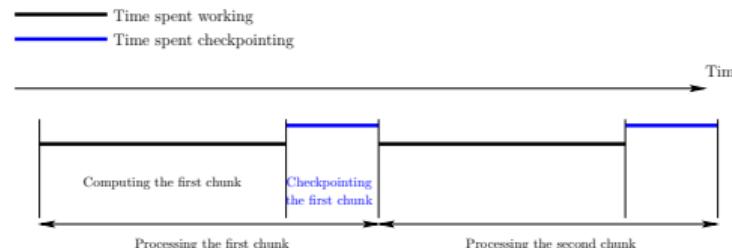
- Predictor predicts six faults in time t
- Five actual faults. One fault not predicted
- $\mu = \frac{t}{5}$, $\mu_P = \frac{t}{6}$, and $\mu_{NP} = t$
- Recall $r = \frac{4}{5}$ (green arrows over red arrows)
- Precision $p = \frac{4}{6}$ (green arrows over blue arrows)

Algorithm

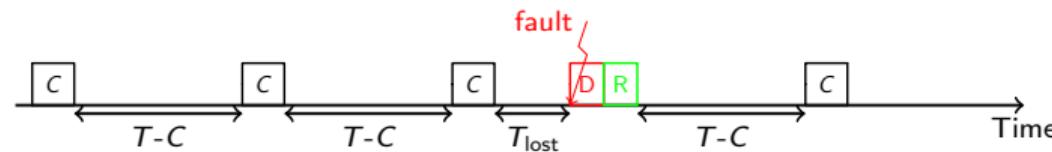
- ➊ While no fault prediction is available:
 - checkpoints taken periodically with period T
- ➋ When a fault is predicted at time t :
 - take a checkpoint ALAP (completion right at time t)
 - after the checkpoint, complete the execution of the period

Computing the waste

① **Fault-free execution:** $\text{WASTE}[FF] = \frac{C}{T}$

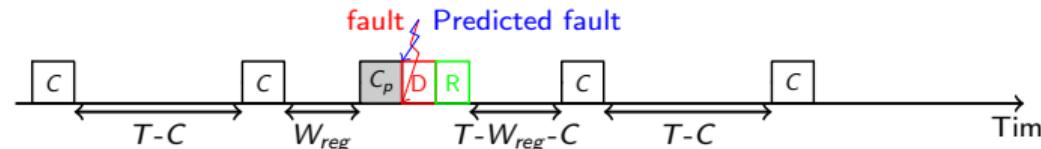


② **Unpredicted faults:** $\frac{1}{\mu_{NP}} [D + R + \frac{T}{2}]$

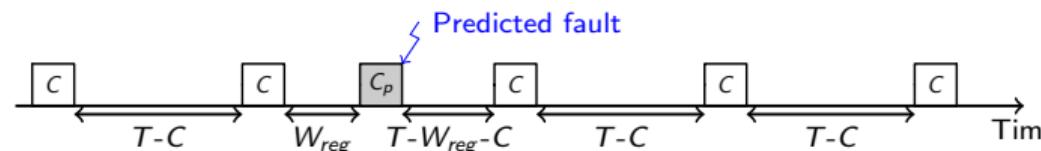


Computing the waste

③ **Predictions:** $\frac{1}{\mu_P} [p(C + D + R) + (1 - p)C]$



with actual fault (true positive)



no actual fault (false negative)

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

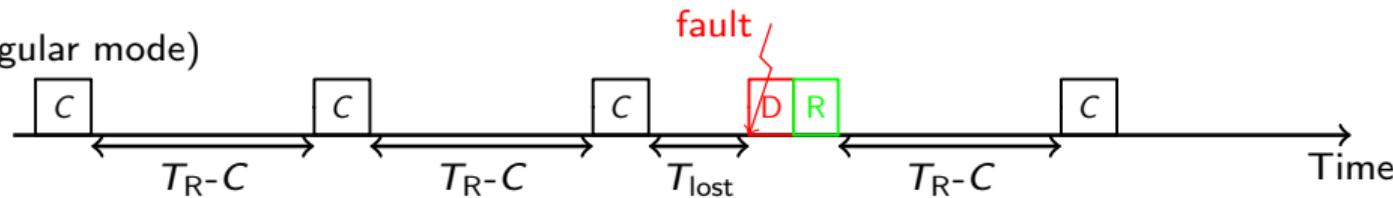
Refinements

- Use different value C_p for proactive checkpoints
- Avoid checkpointing too frequently for false negatives
 - ⇒ Only trust predictions with some fixed probability q
 - ⇒ Ignore predictions with probability $1 - q$

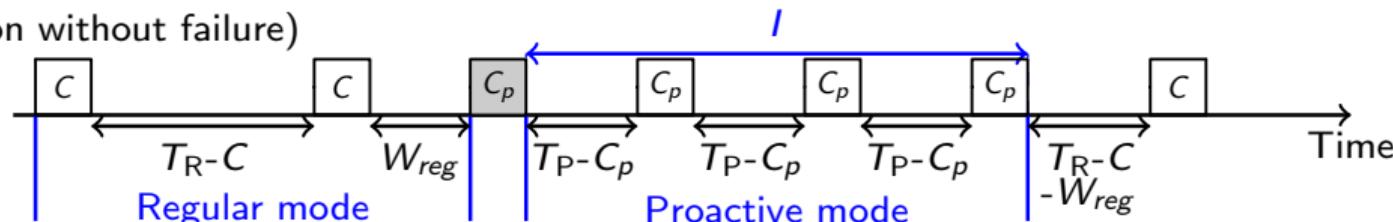
Conclusion: trust predictor always or never ($q = 0$ or $q = 1$)
- Trust prediction depending upon position in current period
 - ⇒ Increase q when progressing
 - ⇒ Break-even point $\frac{C_p}{p}$

With prediction windows

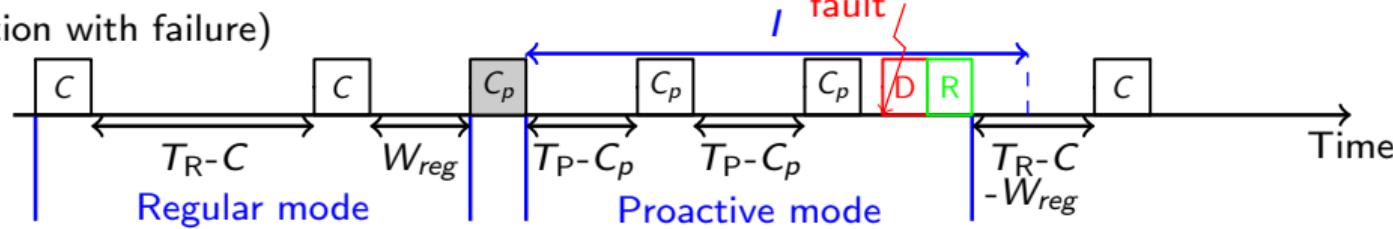
(Regular mode)



(Prediction without failure)



(Prediction with failure)



Gets too complicated! 😐

Outline

1 Introduction (20mn)

2 Checkpointing: Protocols (40mn)

3 Checkpointing: Probabilistic models (30mn)

4 Hands-on: User Level Failure Mitigation (MPI) (2 x 90mn)

5 Hierarchical checkpointing (20mn)

6 Forward-recovery techniques (20mn)

7 Silent errors (35mn)

8 Conclusion (15mn)

9 Advanced Models

- Hierarchical checkpointing
- Failure Prediction
- Replication

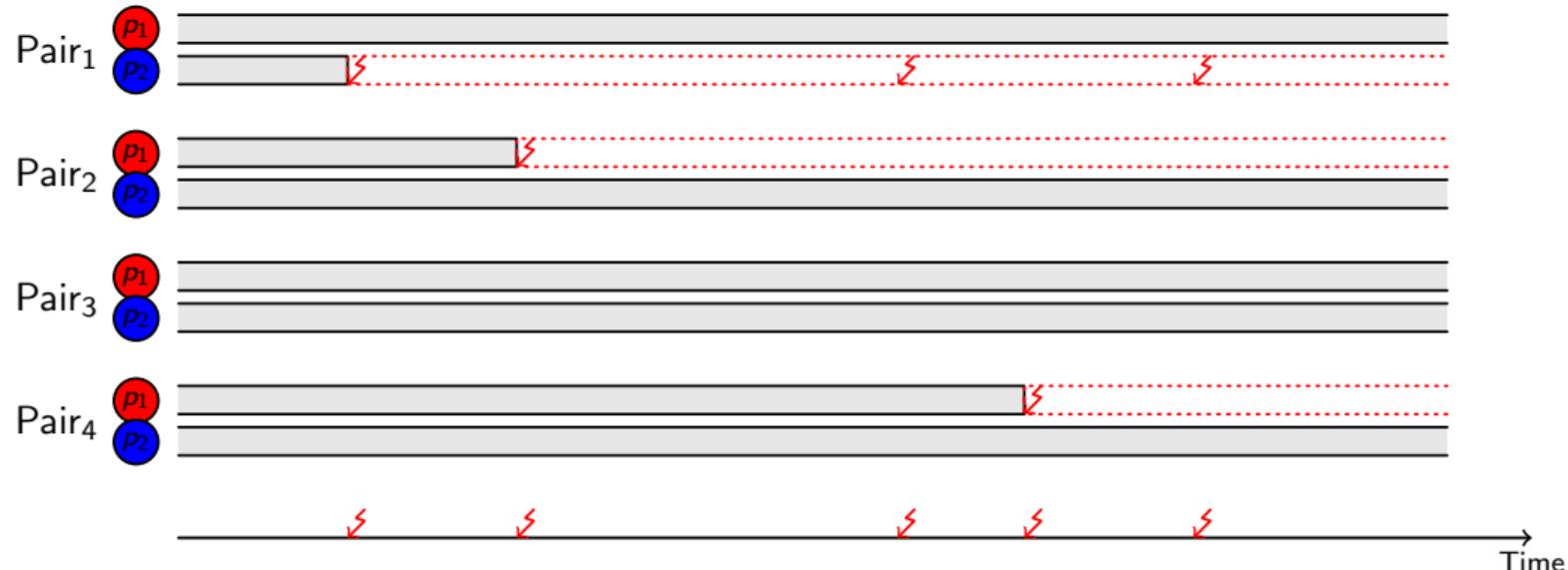
Replication

- 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

Example



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?

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

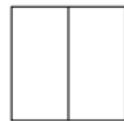
The analogy

Two people with same birthday

≡

Two failures hitting same replica-group

Differences with birthday problem



1



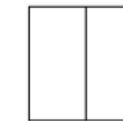
2

...



i

...



N

- $2N$ 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

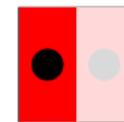


1



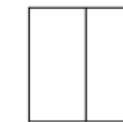
2

...



i

...



N

- $2N$ 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

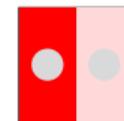


1



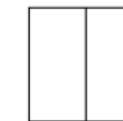
2

...



i

...



N

- $2N$ 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



1



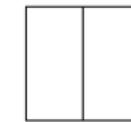
2

...



i

...



N

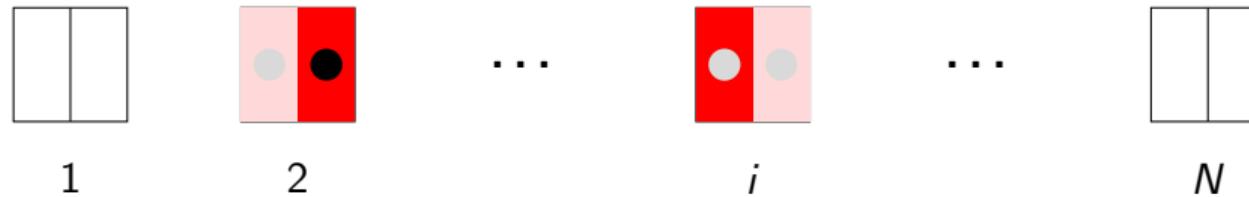
- $2N$ processors but N processes, each replicated twice
- Uniform distribution of failures
- First failure: each replica-group has probability $1/N$ to be hit
- Second failure **cannot** hit failed PE
 - Failure uniformly distributed over $2N - 1$ PEs
 - Probability that replica-group i is hit by failure: $1/(2N - 1)$
 - 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 $\neq i$ is hit by failure: $2/(2N - 1)$
 - Failure **not** uniformly distributed over replica-groups:
this is **not** the birthday problem

Differences with birthday problem

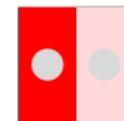


1



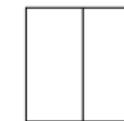
2

...



i

...



N

- $2N$ 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

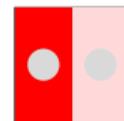


1



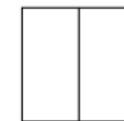
2

...



i

...



N

- $2N$ 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

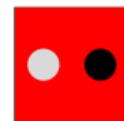


1



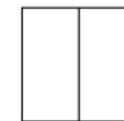
2

...



i

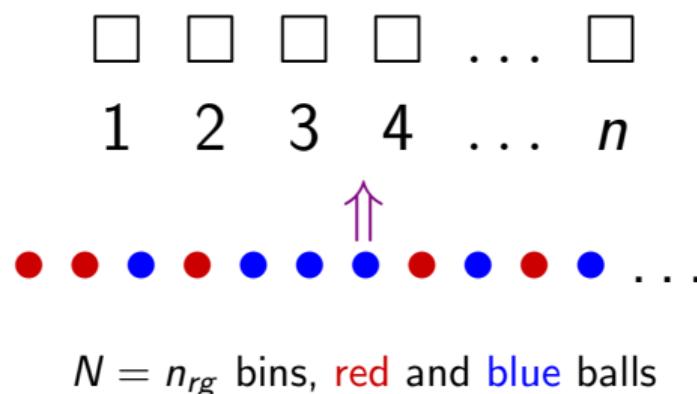
...



N

- $2N$ 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



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

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.

$$MNFTI^{ah} = 1 + MNFTI^{rp}$$

Exponential failures

Theorem $MNFTI^{\text{ah}} = \mathbb{E}(NFTI^{\text{ah}}|0)$ where

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

$\mathbb{E}(NFTI^{\text{ah}}|n_f)$: 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

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

$$\begin{aligned} \mathbb{E} \left(NFTI^{\text{ah}} | n_f \right) &= \frac{2n_{rg} - 2n_f}{2n_{rg}} \times \left(1 + \mathbb{E} \left(NFTI^{\text{ah}} | n_f + 1 \right) \right) \\ &\quad + \frac{2n_f}{2n_{rg}} \times \left(\frac{1}{2} \times 1 + \frac{1}{2} \left(1 + \mathbb{E} \left(NFTI^{\text{ah}} | n_f \right) \right) \right). \end{aligned}$$

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

Exponential failures (cont'd)

Proof

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

$$\begin{aligned} \mathbb{E} \left(NFTI^{\text{ah}} | n_f \right) &= \frac{2n_{rg} - 2n_f}{2n_{rg}} \times \left(1 + \mathbb{E} \left(NFTI^{\text{ah}} | n_f + 1 \right) \right) \\ &\quad + \frac{2n_f}{2n_{rg}} \times \left(\frac{1}{2} \times 1 + \frac{1}{2} \left(1 + \mathbb{E} \left(NFTI^{\text{ah}} | n_f \right) \right) \right). \end{aligned}$$

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

Exponential failures (cont'd)

Proof

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

$$\begin{aligned} \mathbb{E} \left(NFTI^{\text{ah}} | n_f \right) &= \frac{2n_{rg} - 2n_f}{2n_{rg}} \times \left(1 + \mathbb{E} \left(NFTI^{\text{ah}} | n_f + 1 \right) \right) \\ &\quad + \frac{2n_f}{2n_{rg}} \times \left(\frac{1}{2} \times 1 + \frac{1}{2} \left(1 + \mathbb{E} \left(NFTI^{\text{ah}} | n_f \right) \right) \right). \end{aligned}$$

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

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

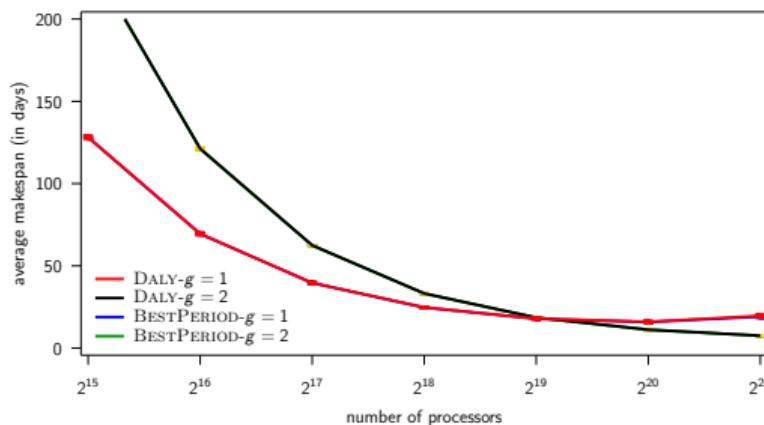
$$\text{THROUGHPUT}_{\text{Rep}} = N \left(1 - \sqrt{\frac{2C}{\mu_{\text{rep}}}}\right)$$

$$\mu_{\text{rep}} = MNFTI \times \mu_{2N} = MNFTI \times \frac{\mu}{2N}$$

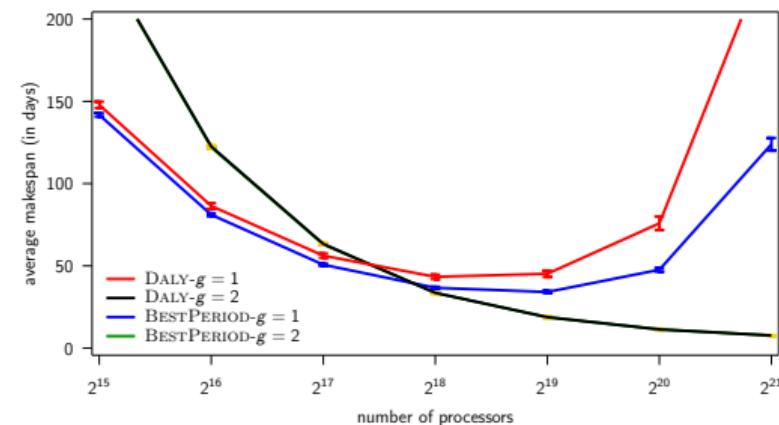
- Platform with $2N = 2^{20}$ processors $\Rightarrow MNFTI = 1284.4$

$\mu = 10$ years \Rightarrow better if C shorter than 6 minutes

Failure distribution



(a) Exponential

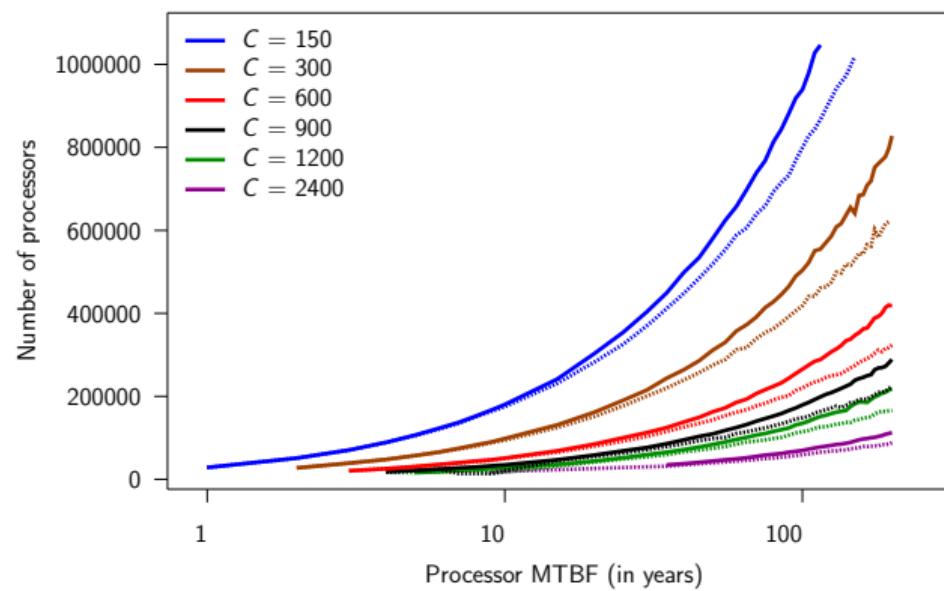
(b) Weibull, $k = 0.7$

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 μ + large C + big p_{total}