Fault-tolerant Techniques for HPC: Theory and Practice

George Bosilca[1], Aurélien Bouteiller[1], Thomas Hérault[1] & Yves Robert[1,2]

1 – University of Tennessee Knoxville
2 – Ecole Normale Supérieure de Lyon

ttp://bit.ly/sc19-eval

{bosilca,bouteiller,herault,yrobert}@icl.utk.edu

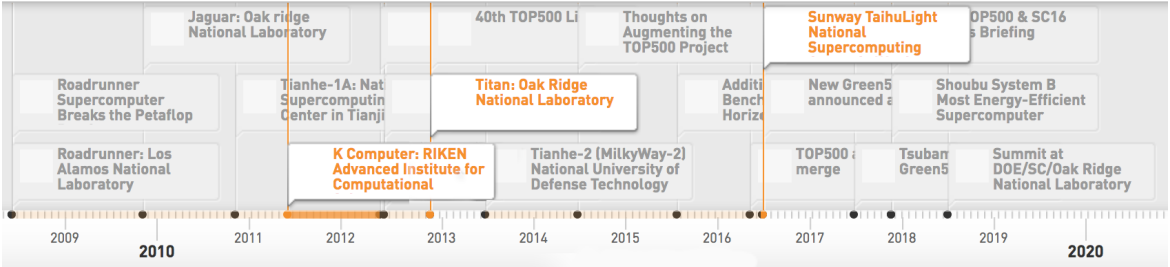
http://fault-tolerance.org/downloads/sc19-tutorial.pdf

http://fault-tolerance.org/sc19/

Outline

Outline

Large-scale Computing Platforms



Not many traces and studies of Large-scale Computing Platforms reliability
Not all studies look at simple characteristics (*e.g.*, MTBF):

- Only *Applications* failures
- Only *Hardware* failures
- *Logged events*

Petascale Platforms – K-Computer

#20 Top500 (June'19, Rmax = 10.510 PFlop/s); 06/11 — 08/19
864 cabinets; 88,128 nodes; 705,024 cores; 1.34PBytes; 12.6 MW;

Fumiyoshi Shoji, Shuji Matsui, Mitsuo Okamoto, Fumichika Sueyasu, Toshiyuki Tsukamoto, Atsuya Uno and Keiji Yamamoto. **Long term failure analysis of 10 petascale supercomputer**. Poster at ISC'15

MFR = Monthly Failure Rate
$(= \frac{\#\text{failures during month}}{\#\text{unit total}})$

CPU MFR	0.004%	⇌	0.017%
DIMM MFR	0.0017%	↘	0.001%
Sys. Board MFR	0.16%	↘	0.05%
Sys. Board MTBF	1.1 day	↗	3 day

AFR = Annualized Failure Rate
$(= 1 - e^{\frac{-1year}{MTBF}})$
FIT = Failure In Time = #fail. / 10^9 h

	# Parts	AFR	MTBF	FIT
CPU	82,944	0.06%	7.33 days	72.00
DIMM	663,552	0.0016%	34.4 days	18.02

(Numbers in red are derived from reported numbers in black)

Petascale Platforms – Titan

#12 Top500 (June'19, Rmax = 17.590 PFlop/s); 10/12 — 08/19
200 cabinets; 18,688 nodes; 299,008 cores; 18,688 K20X; 693.6TBytes; 8.2 MW;

R. A. Ashraf and C. Engelmann, **Analyzing the Impact of System Reliability Events on Applications in the Titan Supercomputer**, 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS), 2018

Event log from Reliability, Availability and Serviceability (RAS) System

68k apps have 1+ events (over $2 \cdot 10^6$)

95% of apps with 11k+ nodes have 1+ events
91% of apps with 125- nodes have 0 events
85% apps under 30min have 0 events
80% apps above 24h have 1+ events

Nature of first event hitting an application:

Parallel Filesystem	73.7%
Processor Failure	15.7 %
Machine Check Exception	6.5%
GPU Failure	1.5%
OOM / SEGFAULT	1.9%
Interconnect	0.8 %

Petascale Platforms – Sunway TaihuLight

#3 Top500 (June'19, Rmax = 93.014 PFlop/s); 06/16 —
40 cabinets; 20,480 nodes; 10,649,600 cores; 1.32PBytes; 15 MW;

Liu RT, Chen ZN. **A large-scale study of failures on petascale supercomputers.** Journal of Computer Science and Technology 33(1): 24-41, Jan. 2018

Nature of faults:

	All Failures	Fatal Failures
CPU	40%	68%
Memory	48%	21%
Power	9%	5%
MD$^{(*)}$	2%	5%

((∗) MD: Maintenance and Diagnosis system)

Effect of Application (1 cabinet, #faults/month):

	Mem Faults	CPU Faults
Comp. only	$7 \rightleftharpoons 93$	$5 \rightleftharpoons 33$
Comp. & Mem	$527 \rightleftharpoons 995$	$136 \rightleftharpoons 867$
Comm. & Mem	$115 \rightleftharpoons 284$	$20 \rightleftharpoons 85$

3 time spans; 1 CPU; 2 Computing Cards
Weibull Distribution best fits CDF

	P_1	P_2	P_3
CPU_1 MTBF	8 days	4 days	10 days
$Card_1$ MTBF	9.5 days		
$Card_2$ MTBF	10 days		

MTBF computed from η and m parameters given in the article:
PDF: $f(x) = \frac{m}{\eta}(\frac{x}{\eta})^{m-1}e^{-(x/\eta)^m}$;
MTBF $= \eta\,\Gamma(1+1/m)$

Exascale platforms (courtesy C. Engelmann & S. Scott), 2008–2011

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 (courtesy C. Engelmann & S. Scott), 2008–2011

Toward Exascale Computing (My Roadmap)
(in 2008–2011)

Based on proposed DOE roadmap with MTTI adjusted to scale linearly

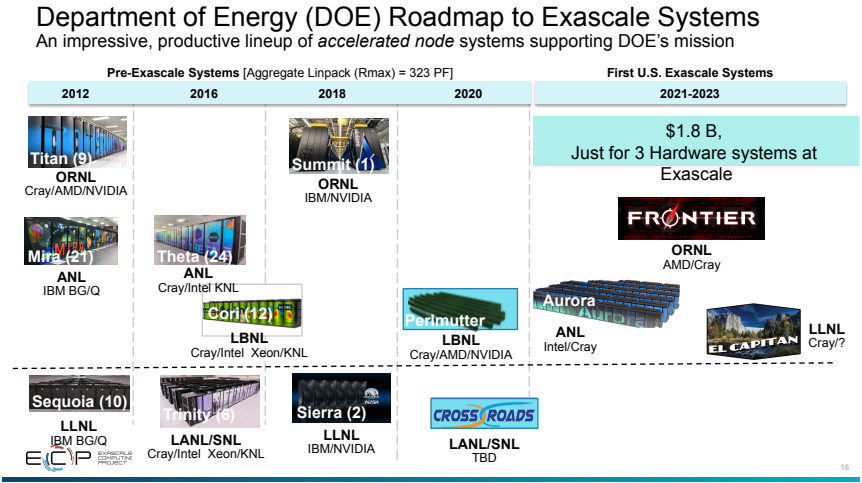
Systems	2009	2011	2015	~~2018~~
System peak	2 Peta	20 Peta	100-200 Peta	1 Exa
System memory	0.3 PB	1.6 PB	5 PB	10 PB
Node performance	125 GF	200GF	200-400 GF	1-10TF
Node memory BW	25 GB/s	40 GB/s	100 GB/s	200-400 GB/s
Node concurrency	12	32	O(100)	O(1000)
Interconnect BW	1.5 GB/s	22 GB/s	25 GB/s	50 GB/s
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

2020? 2021?

O(10k)? O(100k)?

???

Exascale platforms (courtesy Jack Dongarra)



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 ⇒ Shorter MTBF (Mean Time Between Failures)

Exascale platforms

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

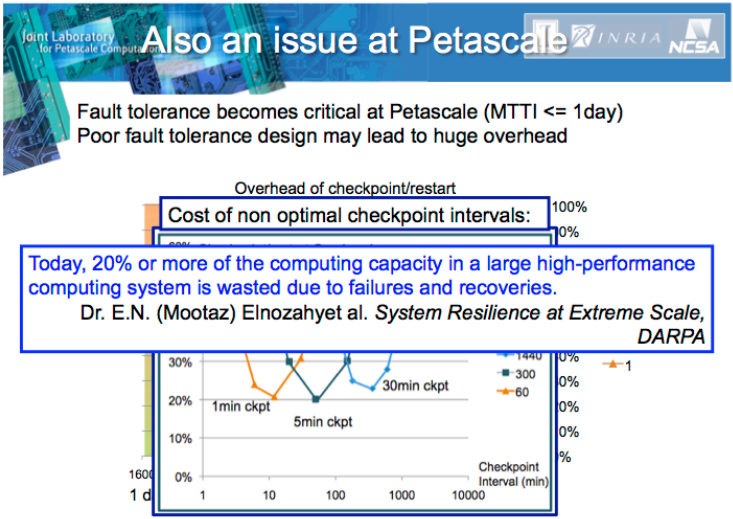
MTBF – one node	1 year	10 years	120 years
MTBF – platform of 10^6 nodes	30sec	5mn	1h

More no [illegible] ailures)

Exascale

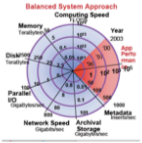
$\neq$ Petascale $\times 1000$

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

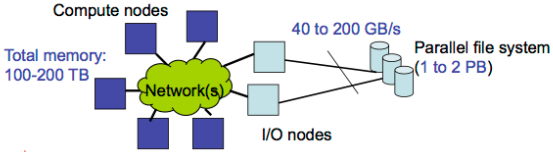


Even for today's platforms (courtesy F. Cappello)

Classic approach for FT: Checkpoint-Restart



Typical "Balanced Architecture" for PetaScale Computers



RoadRunner

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

LLNL BG/L

A few definitions

- Many types of faults: software error, hardware malfunction, memory corruption
- Many possible behaviors: silent, transient, unrecoverable
- Fail-stop errors
  - Lead to application failures (crashes)
  - Includes all hardware faults, and some software ones
  - Use terms *fault* and *failure* interchangeably
- Silent errors
  - Silent Data Corruptions (SDC)
  - Examples: bit flips in memory/cache/registers, arithmetic errors
  - Undetected, manifest themselves after some (unknown) latency

Outline

Outline

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

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

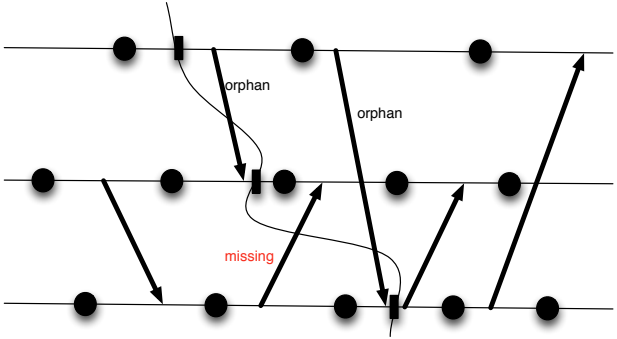
- local memory
- local disk (SSD, HDD)
- remote disk
  - Scalable Checkpoint Restart (SCR): `https://computing.llnl.gov/projects/scalable-checkpoint-restart-for-mpi`
  - VeloC (merge of SCR and FTI): `https://github.com/ECP-VeloC/VELOC`

Checkpoint is valid when finished on reliable storage

Distributed Memory Storage

- In-memory checkpointing

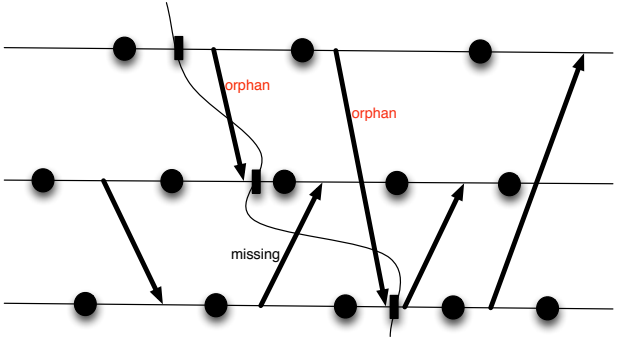
Coordinated checkpointing



Definition (Missing Message)

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

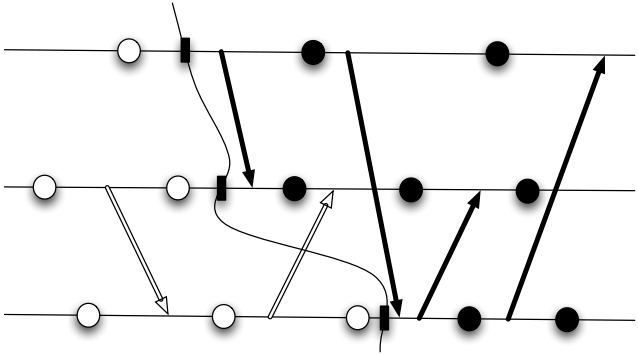
Coordinated checkpointing



Definition (Orphan Message)

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

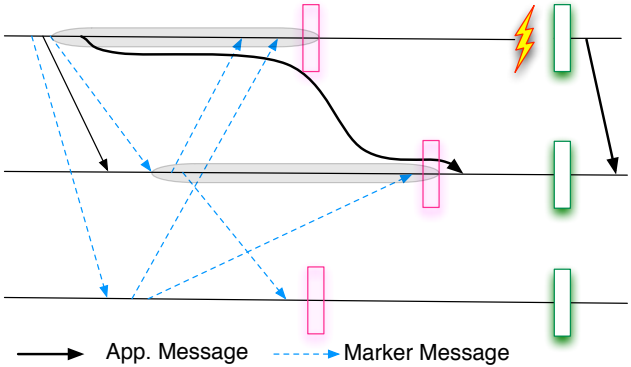
Coordinated Checkpointing: Main Idea



Create a consistent view of the application

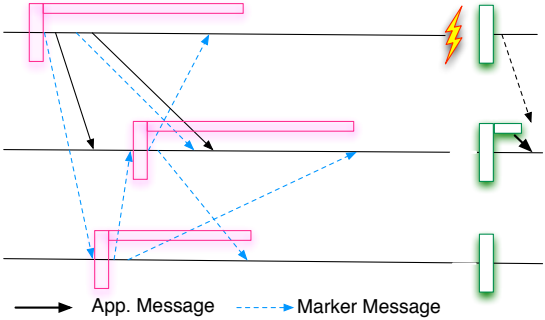
- Every message belongs to a single checkpoint wave
- All communication channels must be flushed (all2all)

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

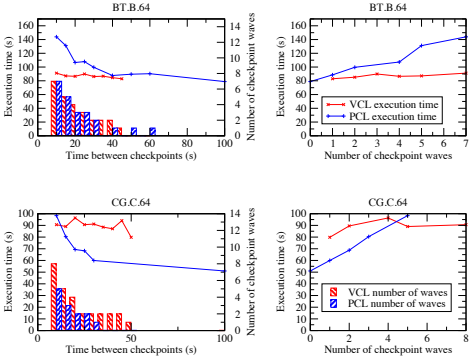
Communication Library

- 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

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 – VeloC, DMTCP, CRIU
- Store the Checkpoint
  - In files (Some frameworks can help – VeloC, CRIU)
  - In memory (Some frameworks can help – VeloC)
- 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

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

Example: MPI-1D Stencil

MPI 1D Stencil

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

Natural Synchronization Point

Example: MPI-1D Stencil

User-Level Checkpointing

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

Example: MPI-1D Stencil

User-Level Checkpointing

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

Atomic Commit of the Valid Checkpoint

Example: MPI-1D Stencil

User-Level Rollback

```
int main (int argc, char *argv[])
{
    double locals[NBLOCALS],                /* The local values */
           *globals,                        /* all values, defined only for 0 */
            local_error, global_error;      /* Estimates of the error */
    int     taskid, numtasks;               /* rank and world size */
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
    /** Read the local domain from an input file */
    if( taskid == 0 ) globals = ReadFile(argv[1]);
    /** And distribute it on all nodes */
    MPI_Scatter(globals, NBLOCALS, MPI_DOUBLE, locals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    do {
```

Read Checkpoint or Input

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

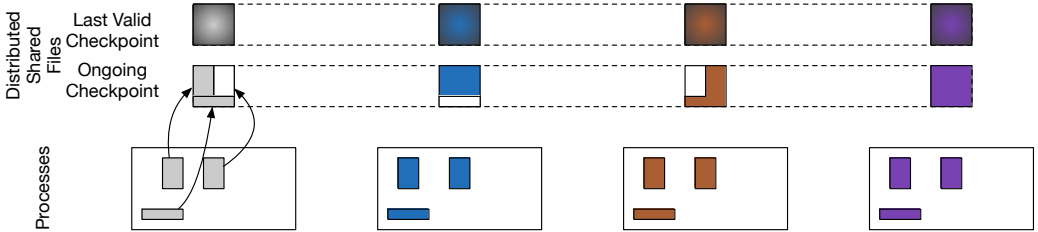
User-Level Rollback

- 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

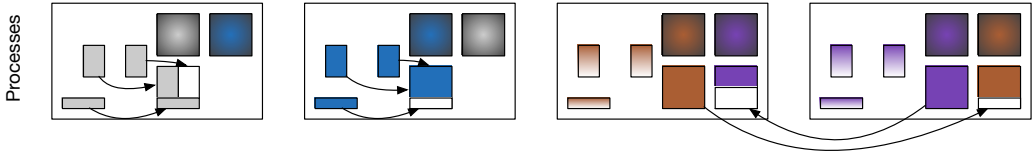
Application-Level Checkpointing – Distributed Checkpointing Approach



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 – VeloC

VeloC – multi-level checkpoint-restart runtime

- Manages Reliability of Storage for the user
- Manages Atomic Commit of Checkpoints
- Exascale Computing Project
- Combines efforts of FTI and SCR
- Checkpoint on files and in the memory hierarchy
- Use local storage, as much as possible
  - Efficiency of local I/O
  - Risk of loosing data $\Longrightarrow$ Fault Tolerant storage (Replication, or XOR)

Helping Libraries – VeloC

VeloC

- Dual API for *checkpointing*:
  - Memory-Based API:
    - Declarative approach: programmer declares what regions of memory need to be checkpointed, then calls (regularly) the checkpointing routine
  - File-Based API:
    - Programmer obtains a (unique and UNIX) file name in which each process serializes their state to checkpoint.
- File-Based API only for restart.

Helping Libraries – VeloC/File API

VeloC Example – Init

```
int main (int argc, char *argv[])
{
    double locals[NBLOCALS],                /* The local values */
           *globals,                        /* all values, defined only for 0 */
            local_error, global_error;      /* Estimates of the error */
    int     i = 0, taskid, numtasks;        /* rank and world size */
    FILE*   fd;                             /* UNIX File pointer to checkpoint */

    MPI_Init(&argc,&argv);
    VELOC_Init(MPI_COMM_WORLD, "conf.cfg");
    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
```

VeloC Example – Fini

```
    } while( global_error > THRESHOLD );

    /** Output result to output file */
    MPI_Gather(locals, NBLOCALS, MPI_DOUBLE,
               globals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    if( taskid == 0 ) SaveFile("Result", globals);
    VELOC_Finalize();
    MPI_Finalize();
    return 0;
```

Helping Libraries – VeloC

VeloC File API Example – Checkpoint

```
        /** Compute the global error */
        MPI_AllReduce(&local_error, &global_error, 1, MPI_DOUBLE,
                      MPI_MAX, MPI_COMM_WORLD);
        if( (++i) % N == 0 ) {
            int valid = 1;
            char veloc_file[VELOC_MAX_NAME];
            /** Regularly, checkpoint regions */
            /** Wait that all MPI ranks are ready to checkpoint */
            VELOC_Checkpoint_wait();
            /** Signal the beginning of checkpoint at iteration i */
            VELOC_Checkpoint_begin("ckpt", i);
            /** Get the filename to use */
            VELOC_Route_file(veloc_file);
            /** Open the UNIX file in write mode */
            FILE* fd = fopen(veloc_file, "wb");
            /** Populate the file with the data to checkpoint */
            if( fwrite(locals, sizeof(double), NBLOCALS, fd) != NBLOCALS )
                valid = 0;
            fclose(fd);
            /** Tell VeloC that the checkpoint is complete and valid */
            VELOC_Checkpoint_end(valid);
        }
    } while( global_error > THRESHOLD );
```

Helping Libraries – VeloC

VeloC File API Example – Restart

```
/** Check if this is a restart */
int v = VELOC_Restart_test("ckpt", 0);
if (v <= 0) {
    int valid = 1;
    char veloc_file[VELOC_MAX_NAME];
    /** Restart a checkpoint */
    VELOC_Restart_begin("ckpt", v);
    /** Get the UNIX filename in which the checkpoint is stored */
    VELOC_Route_file(veloc_file);
    /** Open and unserialize the file into memory */
    fd = fopen(veloc_file, "rb");
    if (fd != NULL) {
        if (fread(locals, sizeof(double), NBLOCALS, fd) != NBLOCALS)
            valid = 0;
    } else {
        valid = 0;
    }
    /** Signal the checkpoint was successfully loaded */
    VELOC_Restart_end(valid);
} else {
    /** Read the local domain from an input file */
    if( taskid == 0 ) globals = ReadFile(argv[1]);
    /** And distribute it on all nodes */
    MPI_Scatter(globals, NBLOCALS, MPI_DOUBLE,
                locals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Helping Libraries – VeloC

VeloC Memory API Example – Declare Critical Memory Regions

```
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);

/** Declare that locals must be protected */
VELOC_Mem_protect(0, locals, NBLOCALS, sizeof(double));
```

VeloC Memory API Example – Checkpoint

```
do {
    /** Update the domain, exchanging information with neighbors */
    UpdateLocals(locals, NBLOCALS, taskid, numtasks);
    /** Compute the local error */
    local_error = LocalError(locals, NBLOCALS);
    /** Compute the global error */
    MPI_AllReduce(&local_error, &global_error, 1, MPI_DOUBLE,
                  MPI_MAX, MPI_COMM_WORLD);
    if( (++i) % N == 0 ) {
        /** Regularly, checkpoint regions */
        VELOC_Checkpoint("ckpt", i);
    }
} while( global_error > THRESHOLD );
```

Helping Libraries – VeloC

VeloC Memory API Example – Restart

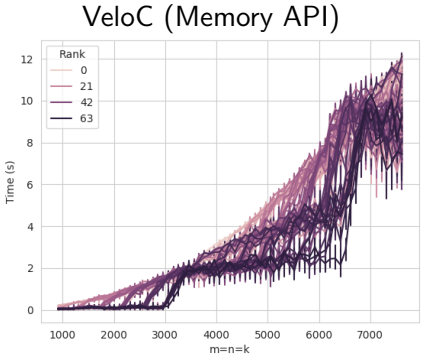
```
/** Check if this is a restart */
v = VELOC_Restart_test("ckpt", 0);
if (v > 0) {
    /** And reload protected regions in that case */
    VELOC_Restart("ckpt", v);
} else {
    /** Read the local domain from an input file */
    if( taskid == 0 ) globals = ReadFile(argv[1]);
    /** And distribute it on all nodes */
    MPI_Scatter(globals, NBLOCALS, MPI_DOUBLE,
                locals, NBLOCALS, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

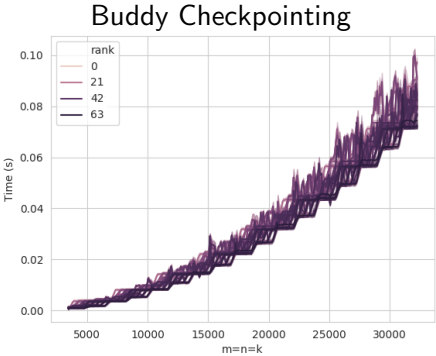
Helping Libraries – VeloC

VeloC Runtime

- VeloC Runtime: two modes
  - Synchronous mode: all operations are implemeted by blocking calls in the library
    - `VELOC_Checkpoint_end` is a costly operation that terminates only when the data is replicated according to the configuration file
  - Asynchronous mode: an MPI helper application is spawned on the same nodes, ensures replication in the background

VeloC vs Buddy Checkpointing





64 Nodes, Local SSD + shared FS, Matrix-Matrix Multiply (PDGEMM) – block size of 432^2 doubles

Implemented by an undergraduate student: 2 days to implement and test with VeloC, 1 month to implement and test with Buddy.

12s checkpoint time at size 8000 for VeloC; 0.12s checkpoint time at 8000 for Buddy.

Helping Libraries – DMTCP

DMTCP

- Distributed MultiThreaded Checkpointing
- `http://dmtcp.sourceforge.net/index.html`
- System-Level checkpointing library/runtime
  - 1st generation: single process, single node
  - 2nd generation: distributed processes, single computer cluster
    - TCP (sockets) and Infiniband (OFED verbs)
  - 3rd generation: distributed applications interacting with the external world
    - Use plugins to synchronize checkpoints and restarts with the external world

Helping Libraries – DMTCP

How does it work

- Use `LD_PRELOAD` to load the DMTCP library before the application, create a checkpoint thread per process
- Saves the entire processes memory map to files
- Flushes communication channels (TCP or IB), creating a consistent cut
- Follows `fork`, `exec` system calls

When does it checkpoint

- Fixed checkpoint interval
- Or checkpoint on demande based on signal interruptions

How to restart

- Command line tool to restart a set of checkpoints

Helping Libraries – CRIU

CRIU

- Checkpoint/Restore In Userspace
- System-level checkpointer for Linux, process-level
  - Limitations on what processes can be checkpointed:
    - With special options for "external" resources (e.g. pipes not part of the process tree, files on remote filesystems, ...)
    - Not possible for non TCP/UDP or other specialized sockets; processes that map devices
- API & Command-line based tools

CRIU Behavior

- Synchronous library: calls are blocking and complete the operation
- No technique to protect the checkpoints from corruption
- Checkpoints are hosted on the filesystem
- Incremental checkpointing is the default behavior

Helping Libraries – CRIU

CRIU – Command line tools

- Tools to check the compatibility between checkpoints and hardware (`check`) to check the capability of the linux kernel (`cpuinfo`)
- Tools to consolidate incremental checkpoints (`dedup`)
- Tools to take a chekpoint (`dump`) and restore a process tree (`restore`).

CRIU – API

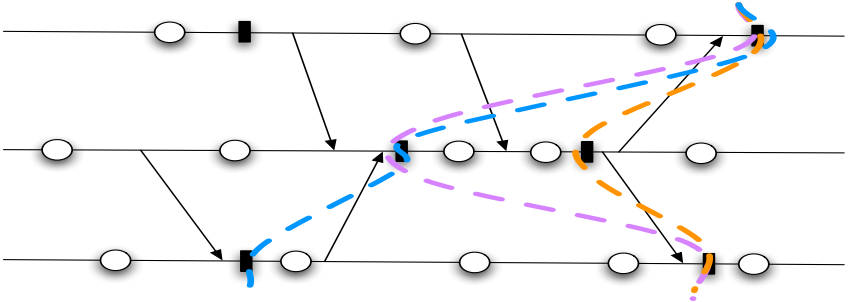
- Set of functions to define checkpoint storage directory, process identifier, options (should the process terminate after the checkpoint, should the sockets be closed, special options for dumping, etc.)
- Three main calls: `criu_check();`, `criu_dump()`, and `criu_restore();` equivalent to corresponding command line tools

Helping Libraries – GVR

Global View Resilience

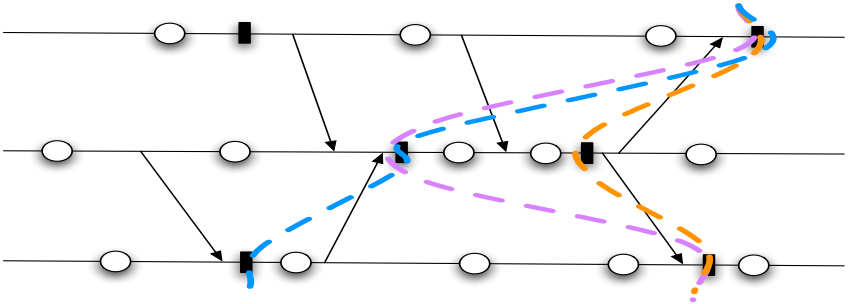
- Manages Reliability of Storage for the user
- Global View Resilience provides a reliable tuple-space for users to store persistent data. E.g., checkpoints
- Storage is entirely in memory, in independent processes accessible through the GVR API.
  - Spatial redundancy – coding at multiple levels
  - Temporal redundancy - Multi-version memory, integrated memory and NVRAM management
- Partitionned Global Address Space approach
- Data resides in the global GVR space, local values for specific versions are pulled for rollback, pushed for checkpoints
- Code is very different from the ones seen above, and outside the scope of this tutorial

Uncoordinated Checkpointing: Main Idea



Processes checkpoint independently

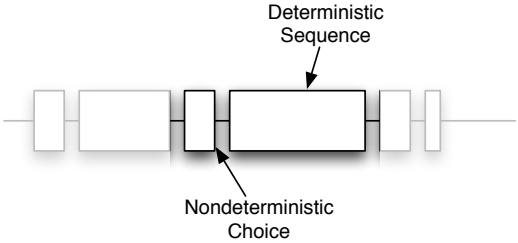
Uncoordinated Checkpointing: Main Idea



Optimistic Protocol

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

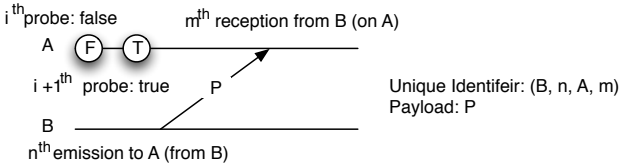
Piece-wise Deterministic Assumption



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

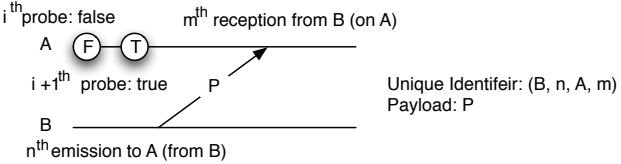
Message Logging



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

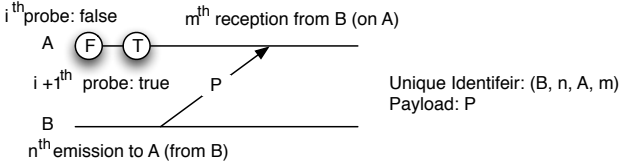
Message Logging



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

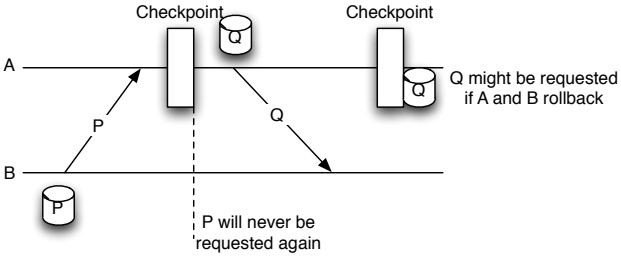
Message Logging



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

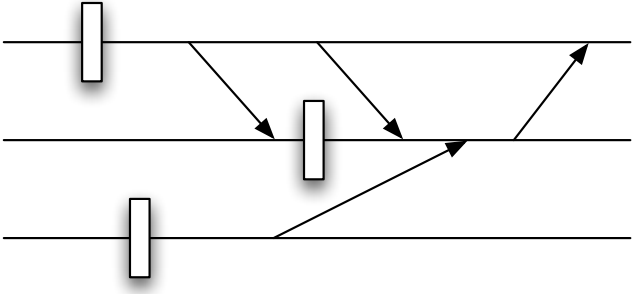
Message Logging



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

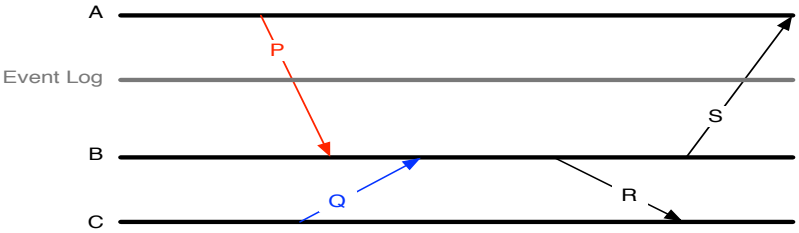
Message Logging



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)

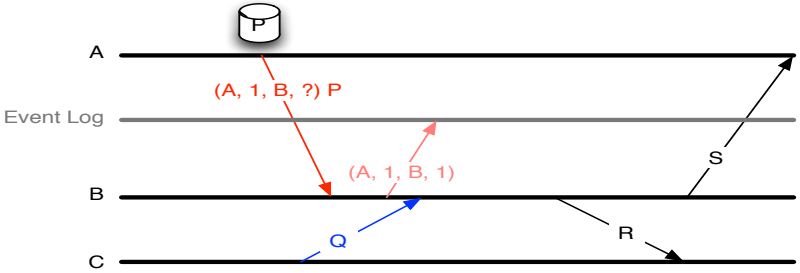
Optimistic Message Logging



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)

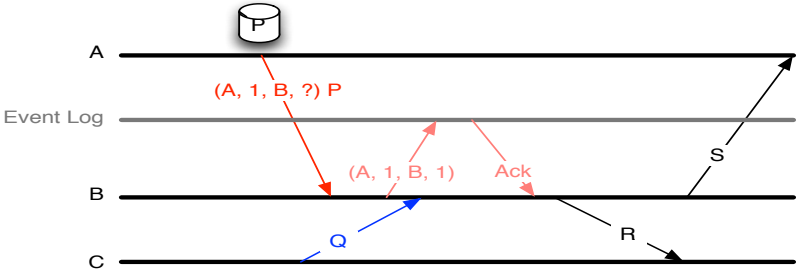
Optimistic Message Logging



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)

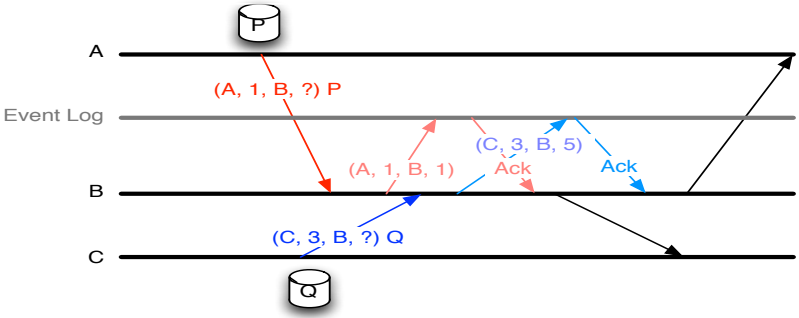
Optimistic Message Logging



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)

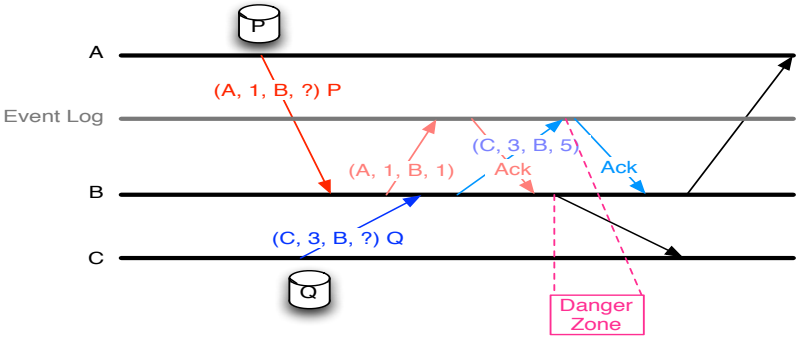
Optimistic Message Logging



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)

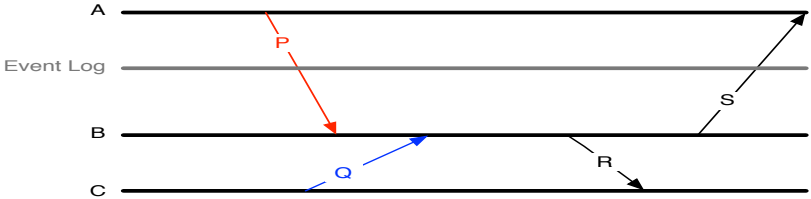
Optimistic Message Logging



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)

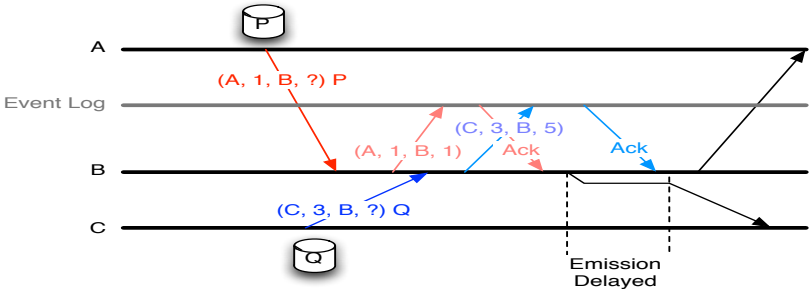
Pessimistic Message Logging



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

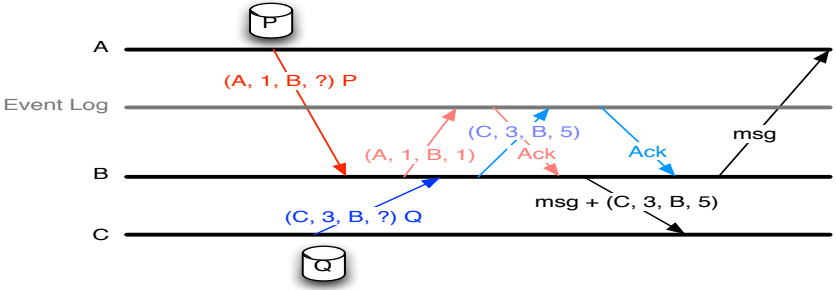
Pessimistic Message Logging



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

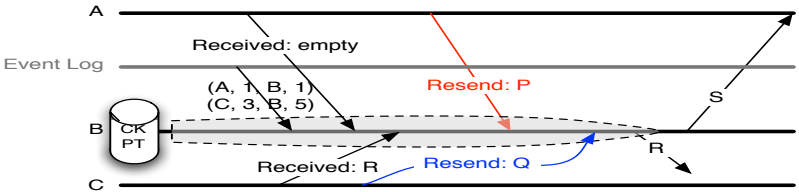
Causal Message Logging



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 collection

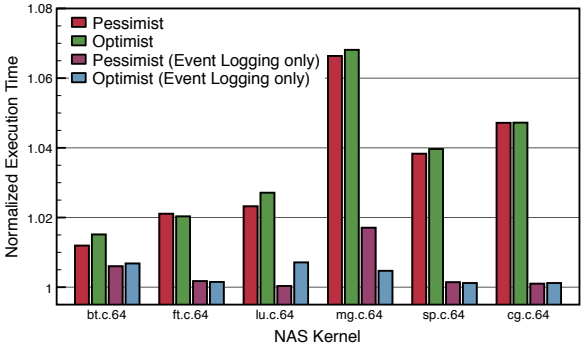
Recovery in Message Logging

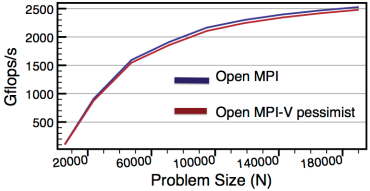


Recovery

- 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

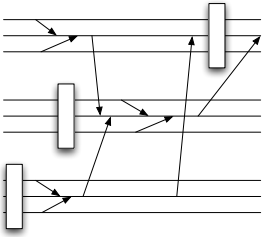
- NAS Parallel Benchmarks – 64 nodes
- High Performance Linpack

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

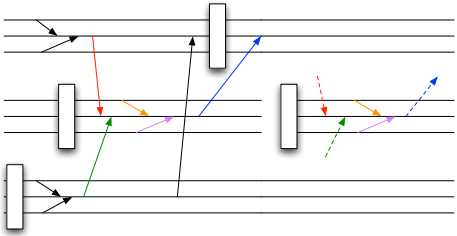
Hierarchical Protocols



Hierarchical Protocol

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

Hierarchical Protocols



Hierarchical Protocol

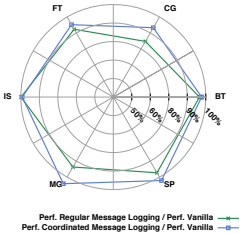
- Coordinated Checkpointing: the processes can behave as a non-deterministic entity (interactions between processes)
- Need to log the non-deterministic events: Hierarchical Protocols *are* uncoordinated protocols + event logging
- No need to log the payload

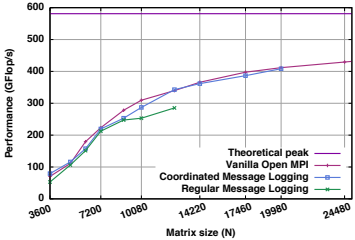
Event Log Reduction

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

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

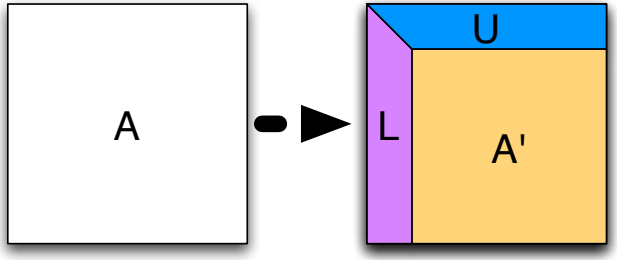
Forward-Recovery

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, ...

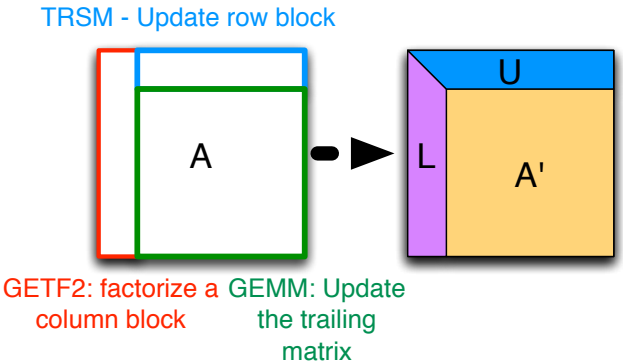
Outline

Example: block LU factorization



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

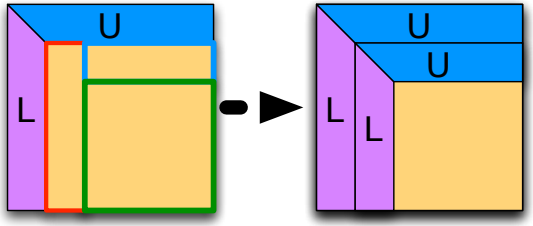
Example: block LU factorization



- 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



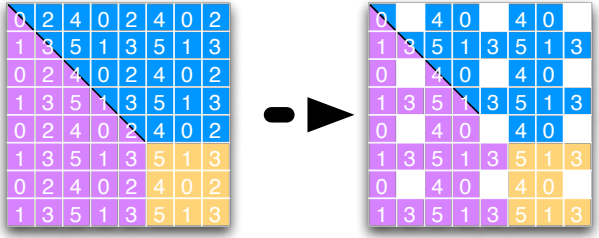
GETF2: factorize a column block

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 $U \cdot x = y$

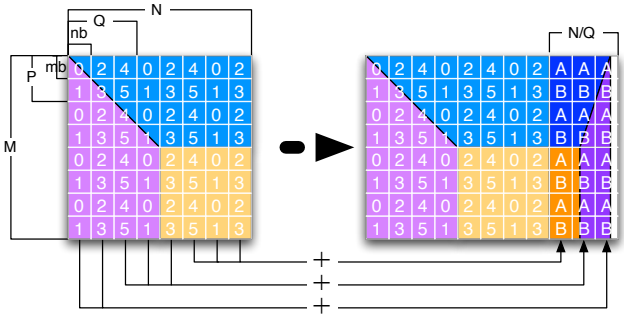
Example: block LU factorization

Failure of rank 2



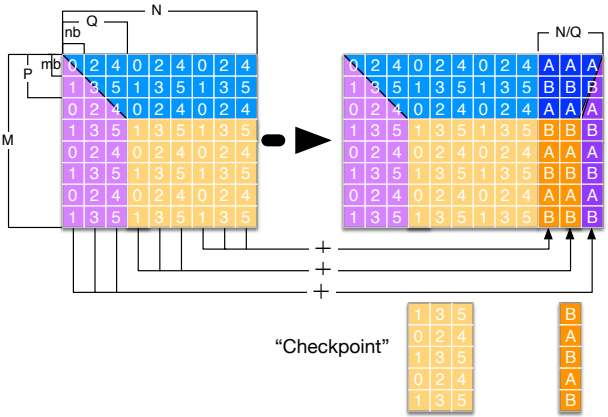
- 2D Block Cyclic Distribution (here 2×3)
- A single failure $\Rightarrow$ many data lost

Algorithm Based Fault Tolerant LU decomposition



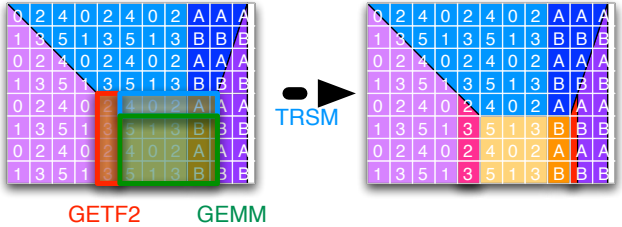
- Checksum: invertible operation on the data of the row / column
  - Checksum replication can be avoided by dedicating computing resources to checksum storage

Algorithm Based Fault Tolerant LU decomposition



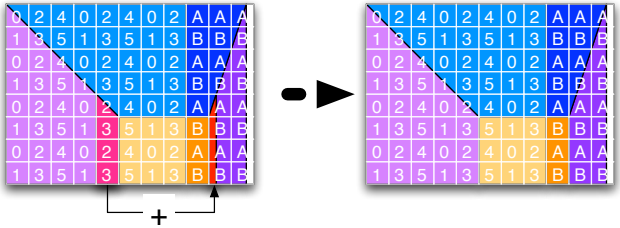
- Checkpoint the next set of Q-Panels to be able to return to it in case of failures

Algorithm Based Fault Tolerant LU decomposition



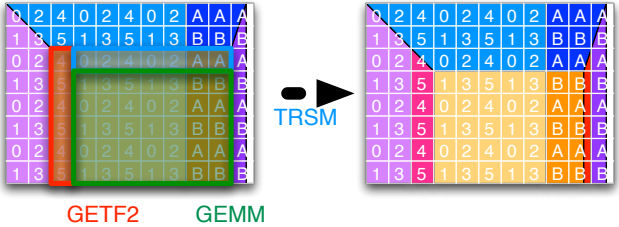
- Idea of ABFT: applying the operation on data and checksum preserves the checksum properties

Algorithm Based Fault Tolerant LU decomposition



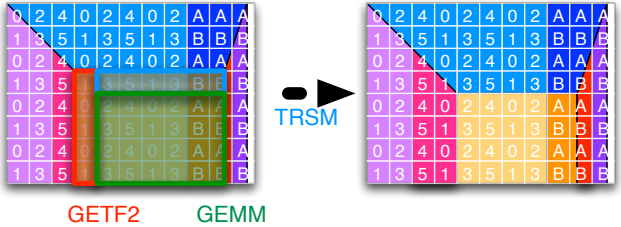
- For the part of the data that is not updated this way, the checksum must be re-calculated

Algorithm Based Fault Tolerant LU decomposition



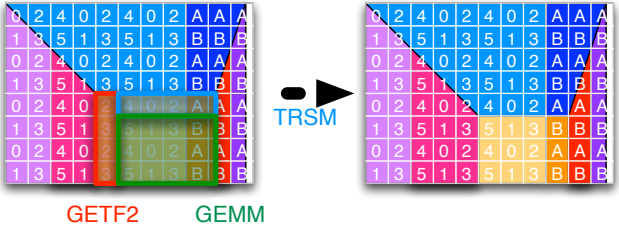
- To avoid slowing down all processors and panel operation, group checksum updates every Q block columns

Algorithm Based Fault Tolerant LU decomposition



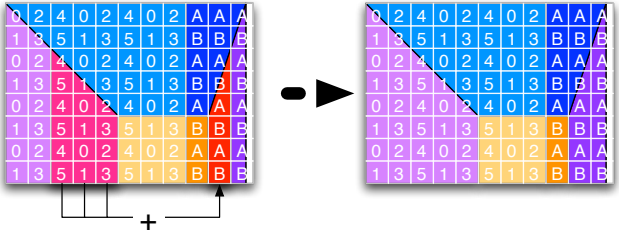
- To avoid slowing down all processors and panel operation, group checksum updates every Q block columns

Algorithm Based Fault Tolerant LU decomposition



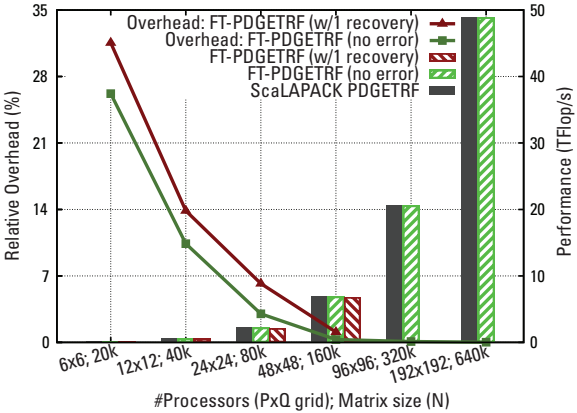
- To avoid slowing down all processors and panel operation, group checksum updates every Q block columns

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 ULFM; Kraken supercomputer;

Outline

Fault Tolerance Techniques

General Techniques

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

Application-Specific Techniques

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

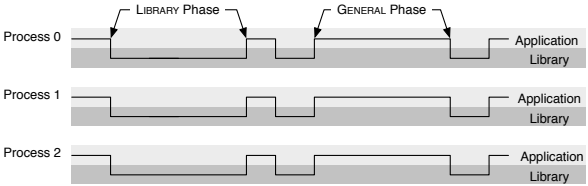
Application

Typical Application

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

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

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



Characteristics

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

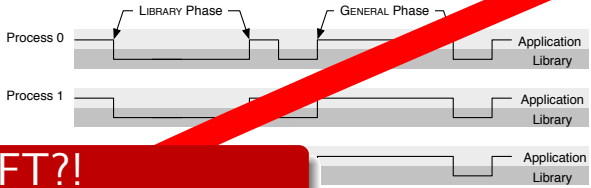
Application

Typical Application

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

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

  /* Update sim tion
   * with r t vector */
  vec2sim
}
```



Goodbye ABFT?!

- ☺ Large part (total) computation spent in factorization/so
- Between LA operatio
  - ☹ use resulting vector / m ix with operations that do not pres the checksums on the data
  - ☹ modify data not covered by ABFT algorithms

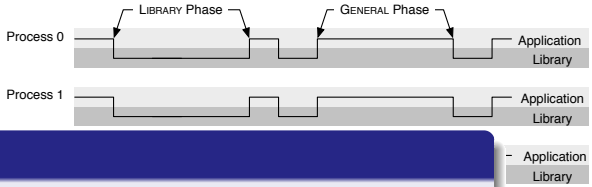
Application

Typical Application

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

  /* Fa
   * Sc
  dgeqr
  dsolv

  /* Up
   * wi
  vec2s
}
```



Problem Statement

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

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

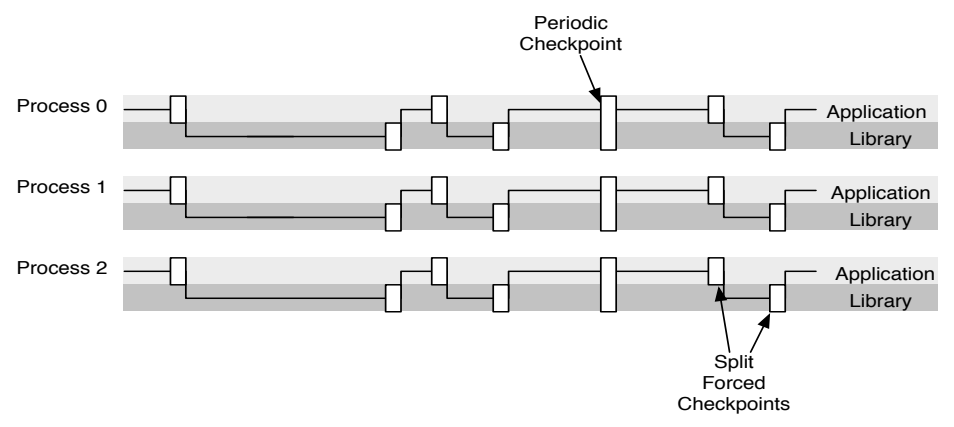
ent in

e

algorithms

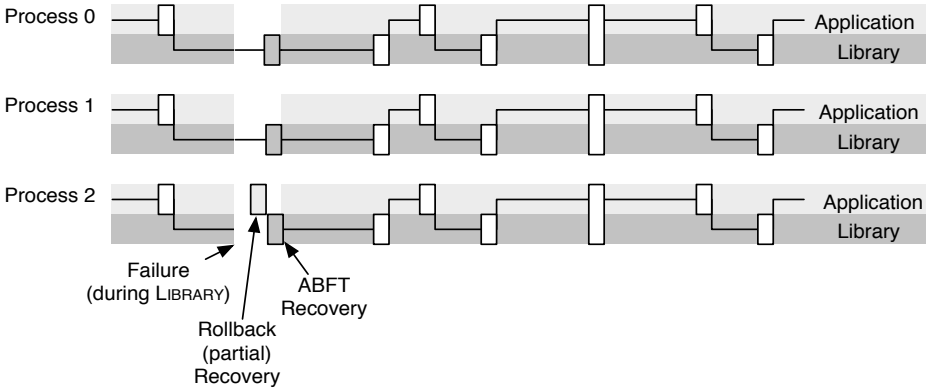
ABFT&PeriodicCkpt

ABFT&PeriodicCkpt: no failure



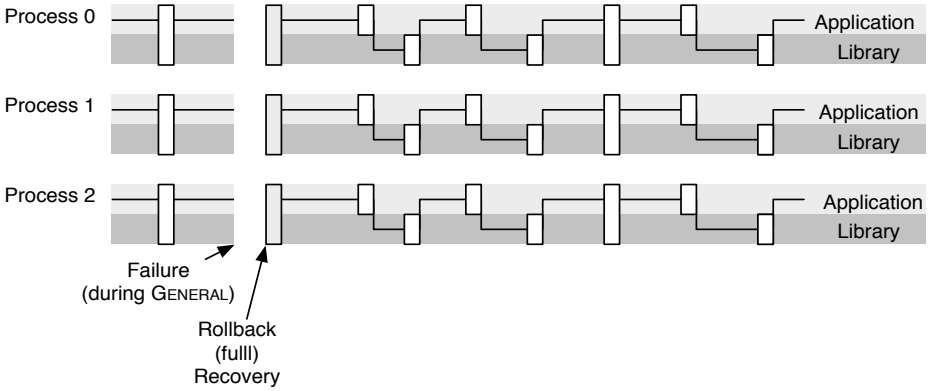
ABFT&PeriodicCkpt

ABFT&PeriodicCkpt: failure during Library phase

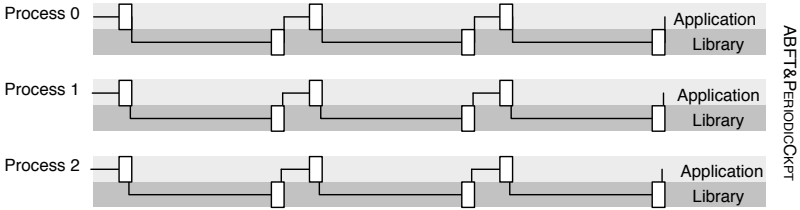


ABFT&PeriodicCkpt

ABFT&PeriodicCkpt: failure during General phase



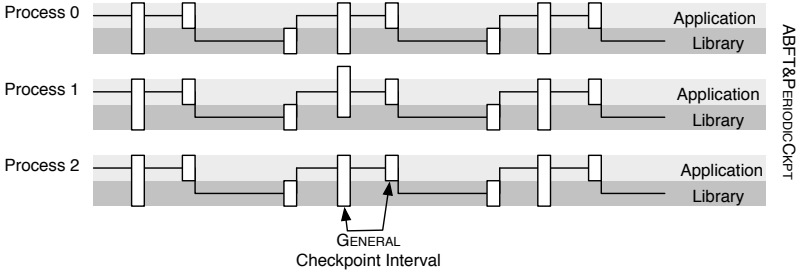
ABFT&PeriodicCkpt: Optimizations



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

☺ ABFT&PERIODICCKPT should perform better with scale

By how much?

Competitors

FT algorithms compared

PeriodicCkpt Basic periodic checkpointing

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

ABFT&PeriodicCkpt The algorithm described above

Weak Scale #1

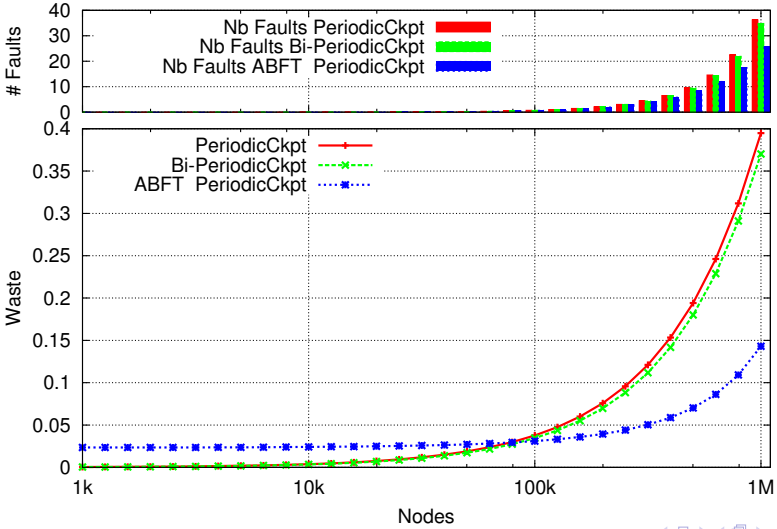
Weak Scale Scenario #1

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

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

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

Weak Scale #1


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

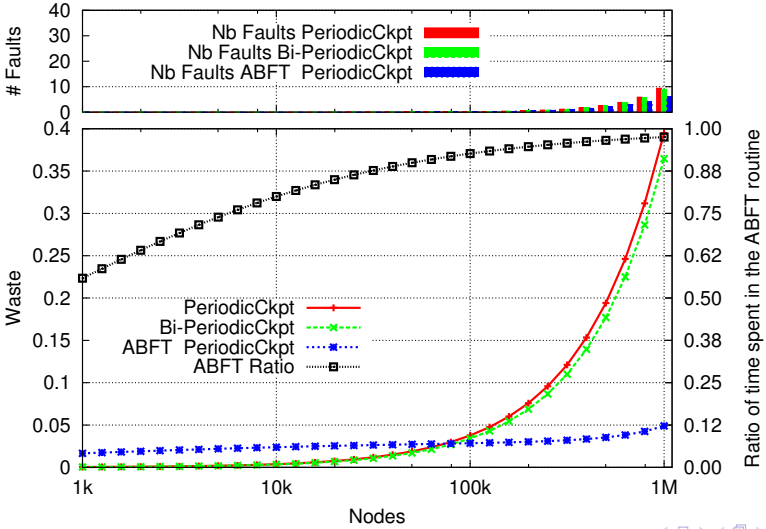
Weak Scale #2

Weak Scale Scenario #2

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

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

Weak Scale #2


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

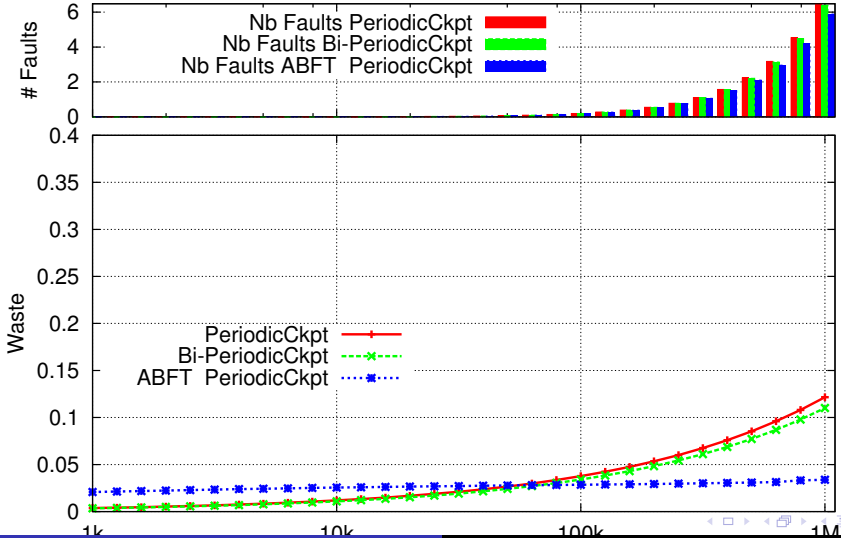
Weak Scale #3

Weak Scale Scenario #3

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

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

Weak Scale #3


Nb Faults PeriodicCkpt
Nb Faults Bi-PeriodicCkpt
Nb Faults ABFT PeriodicCkpt
Faults
6
4
2
0
Waste
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
PeriodicCkpt
Bi-PeriodicCkpt
ABFT PeriodicCkpt
1k
10k
100k
1M

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

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

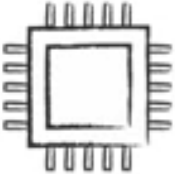
Coming Next

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

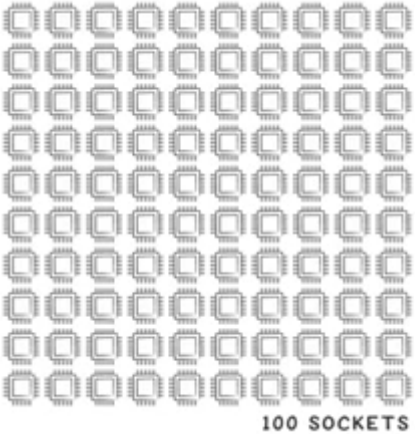
Outline

Outline

Scale is the enemy



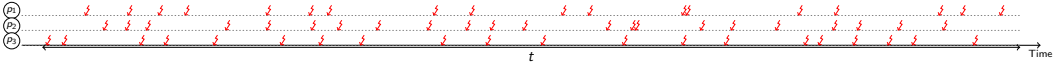
Scale is the enemy



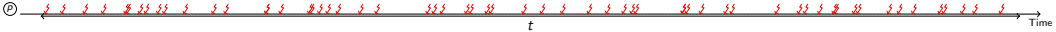
1
YEAR

MEAN TIME
BETWEEN FAILURES

Scale is the enemy

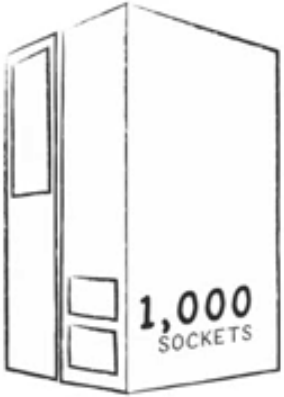


If three processors have around 20 faults during a time t ($\mu = \frac{t}{20}$)...



...during the same time, the platform has around 60 faults ($\mu_N = \frac{t}{60}$)

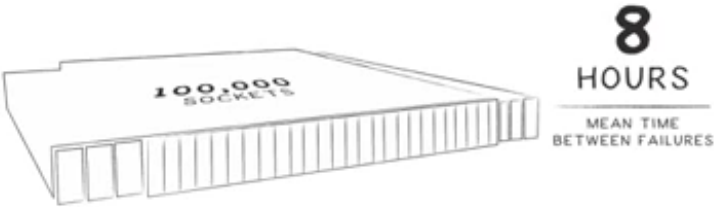
Scale is the enemy



36
DAYS

MEAN TIME
BETWEEN FAILURES

Scale is the enemy



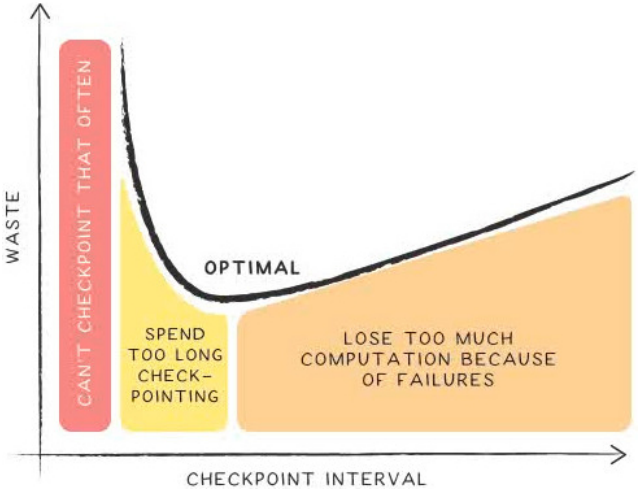
Scale is the enemy

Need to checkpoint!

But when?

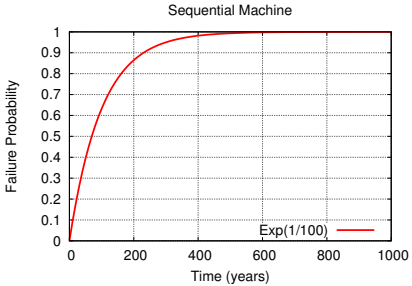
Scheduling matters ☺

Optimal checkpointing interval



Outline

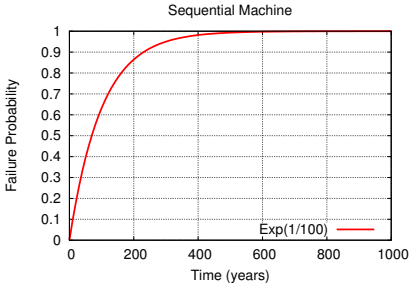
Failure distributions: (1) Exponential



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 $= \frac{1}{\lambda}$

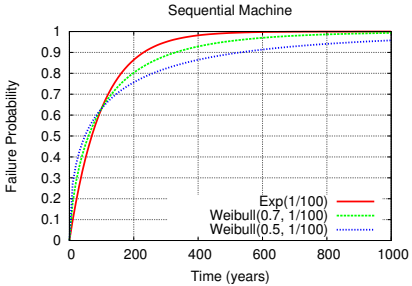
Failure distributions: (1) Exponential



X random variable for $Exp(\lambda)$ 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) $\mu = \mathbb{E}(X) = \frac{1}{\lambda}$

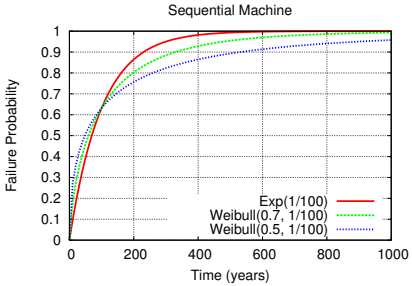
Failure distributions: (2) Weibull



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:

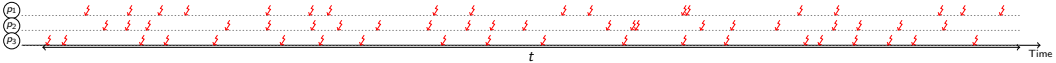
- If $k < 1$: failure rate decreases with time
  "infant mortality": defective items fail early
- If $k = 1$: $Weibull(1, \lambda) = Exp(\lambda)$ constant failure time

Failure distributions: with several processors

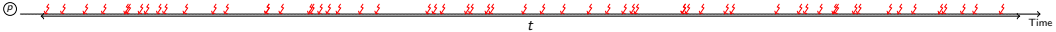
- Processor (or node): any entity subject to failures
  $\Rightarrow$ approach agnostic to granularity

- 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
  $\Rightarrow$ superposition of p IID processor distributions
  $\Rightarrow$ IID only for Exponential
- Define μ_p by

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

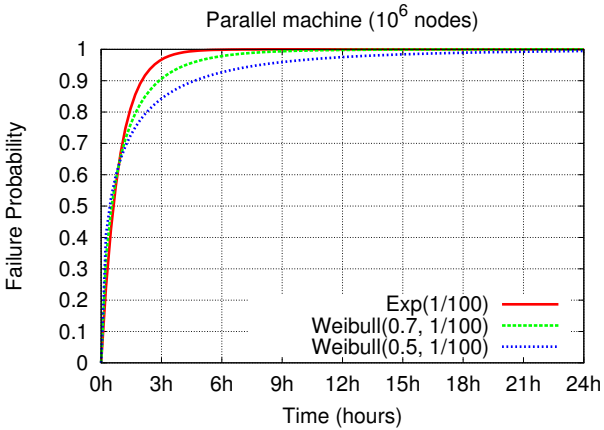
$n(F)$ = number of platform failures until time F is exceeded

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?



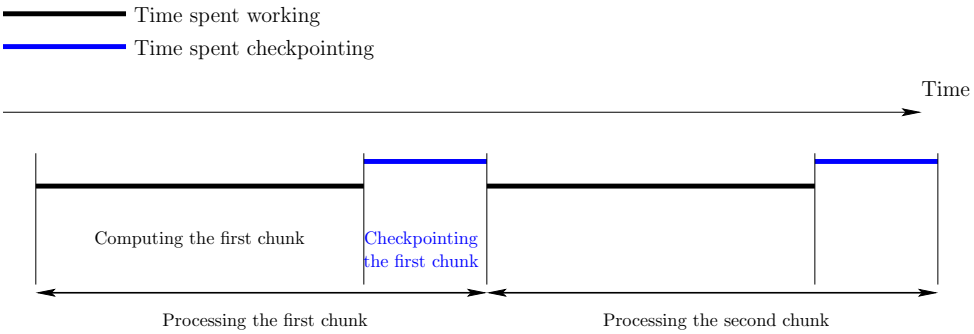
After infant mortality and before aging,
instantaneous failure rate of computer platforms is almost constant

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

Periodic checkpointing



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

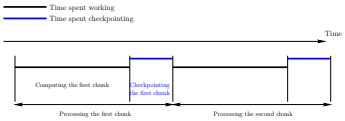
Framework

- 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

$\Rightarrow$ platform = single (powerful, unreliable) processor ☺

Waste: fraction of time not spent for useful computations

Waste in fault-free execution



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

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

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

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

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

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

N_{faults} number of failures during execution
T_{lost}: average time lost per failure

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

T_{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

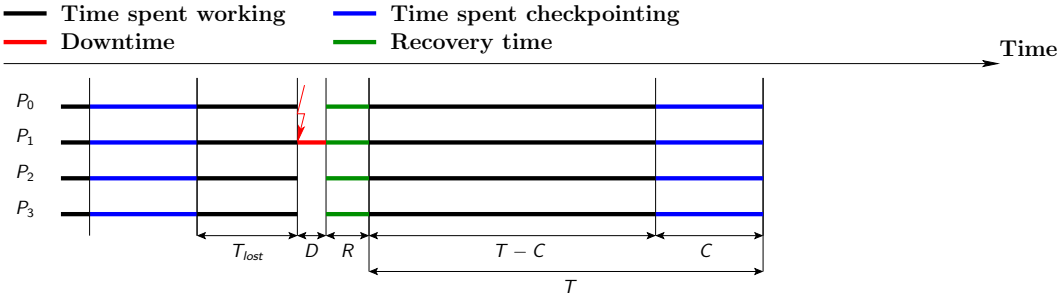
$$\text{TIME}_{\text{final}} = \text{TIME}_{\text{FF}} + N_{faults} \times T_{\text{lost}}$$

N_{faults} number of failures during execution
T_{lost}: average time lost per failure

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

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

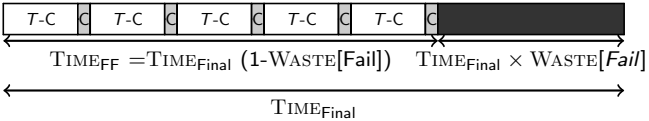
⇒ Exact for Exponential and uniform distributions

Waste due to failures

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

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

Total waste

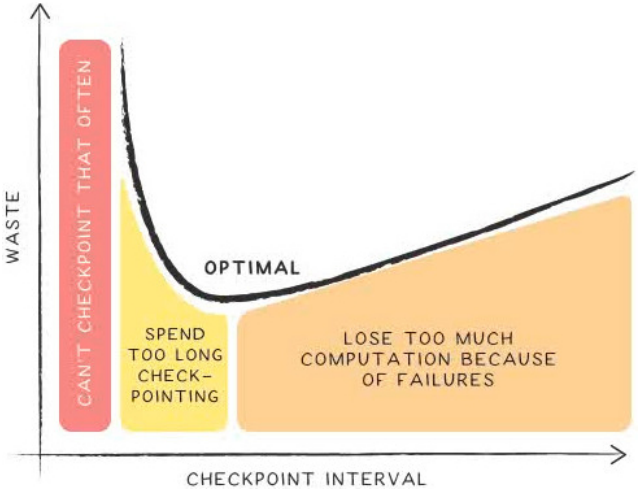


$$\text{WASTE} = \frac{\text{TIME}_{\text{final}} - \text{TIME}_{\text{base}}}{\text{TIME}_{\text{final}}}$$

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

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

Optimal checkpointing interval



Waste minimization

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

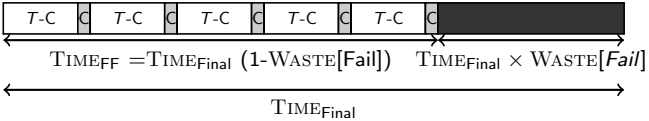
$$\text{WASTE} = \frac{u}{T} + v + wT$$

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

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

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

Comparison with Young/Daly



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

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

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

Validity of the approach (1/3)

Technicalities

- $\mathbb{E}(N_{faults}) = \frac{\text{TIME}_{\text{final}}}{\mu}$ 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 $\mu = \frac{\mu_{ind}}{p}$ 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:
    $\pi = P(X \geq 2) = 1 - (P(X = 0) + P(X = 1)) = 1 - (1 + \theta)e^{-\theta}$
  - $\gamma = 0.27 \Rightarrow \pi \leq 0.03$
    $\Rightarrow$ overlapping faults for only 3% of checkpointing segments

Validity of the approach (3/3)

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

Wrap up

- Capping periods, and enforcing a lower bound on MTBF
  $\Rightarrow$ 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

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

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

Lesson learnt for fail-stop failures

(Not [illegible] Secret data

- Tsubame 2: [illegible] failures during last 18 month[illegible] $\mu = 13$ hrs
- Blue Waters: 2-3 no[illegible] failures per day[illegible]
- Titan: a few failures per [illegible]
- Tianhe 2: [illegible]

Exascale $\neq$ Petascale $\times 1000$

Need more reliable components

Need to checkpoint faster

Petascale[illegible]	$C = 20$ min	$\mu = 24$ hrs	$\Rightarrow$ W[illegible]TE$[opt] = 17\%$
Scale [illegible]10:	$C = 20$ min	$\mu = 2.4$ hrs	$\Rightarrow$ WAS[illegible][opt] $= 53\%$
Sca[illegible] by 100:	$C = 20$ min	$\mu = 0.24$ hrs	$\Rightarrow$ WASTE[illegible]pt] $= 100\%$

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

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	$\Rightarrow \text{WASTE}[opt] = 100\%$

Outline

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

$$\mathbb{E}(W) = \overbrace{\mathcal{P}_{\text{succ}}(W+C)}^{\substack{\text{Probability}\\ \text{of success}}}(W+C)$$

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

$$\mathbb{E}(W) = \mathcal{P}_{\text{succ}}(W+C)\overbrace{(W+C)}^{\substack{\text{Time needed}\\ \text{to compute}\\ \text{the work } W \text{ and}\\ \text{checkpoint it}}}$$

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

$$\mathbb{E}(W) = \begin{array}{l} \mathcal{P}_{\text{succ}}(W+C)\,(W+C) \\ + \\ \underbrace{(1-\mathcal{P}_{\text{succ}}(W+C))}_{\text{Probability of failure}}\,(\mathbb{E}(T_{lost}(W+C)) + \mathbb{E}(T_{rec}) + \mathbb{E}(W)) \end{array}$$

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

$$\mathbb{E}(W) = \begin{array}{l} \mathcal{P}_{\text{succ}}(W+C)\,(W+C) \\ + \\ (1-\mathcal{P}_{\text{succ}}(W+C))\,(\underbrace{\mathbb{E}(T_{lost}(W+C))}_{\substack{\text{Time elapsed} \\ \text{before failure} \\ \text{stroke}}} + \mathbb{E}(T_{rec}) + \mathbb{E}(W)) \end{array}$$

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

$$\mathbb{E}((W) = \begin{array}{l} \mathcal{P}_{\text{succ}}(W+C)\,(W+C) \\ + \\ (1-\mathcal{P}_{\text{succ}}(W+C))\,(\mathbb{E}(T_{lost}(W+C)) + \underbrace{\mathbb{E}(T_{rec})}_{\text{Time needed to perform downtime and recovery}} + \mathbb{E}(W)) \end{array}$$

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

$$\mathbb{E}(W) = \begin{array}{l} \mathcal{P}_{\text{succ}}(W+C)\,(W+C) \\ + \\ (1-\mathcal{P}_{\text{succ}}(W+C))\,(\mathbb{E}(T_{lost}(W+C)) + \mathbb{E}(T_{rec}) + \underbrace{\mathbb{E}(W)}_{\text{Time needed to compute } W \text{ anew}}) \end{array}$$

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

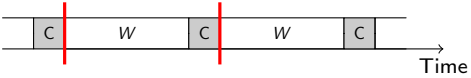
$$\mathbb{E}(W) = \begin{array}{l} \mathcal{P}_{\text{succ}}(W+C)\,(W+C) \\ + \\ (1-\mathcal{P}_{\text{succ}}(W+C))\,(\mathbb{E}(T_{lost}(W+C)) + \mathbb{E}(T_{rec}) + \mathbb{E}(W)) \end{array}$$

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

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

Optimal checkpointing interval

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



- Exact solution:

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

- 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

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

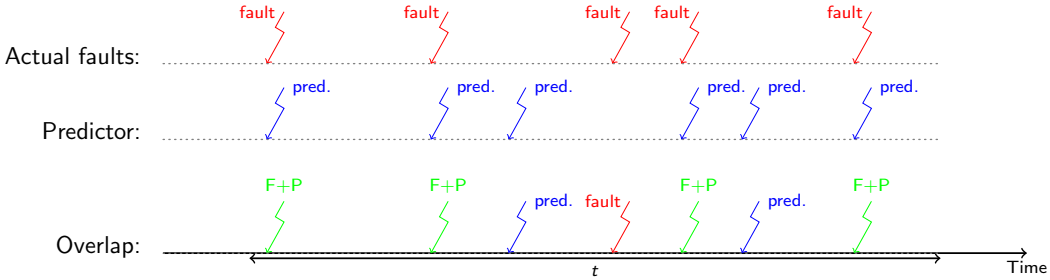
- μ: mean time between failures (MTBF)
- μ_P mean time between predicted events (both true positive and false positive)
- μ_{NP} mean time between unpredicted faults (false negative).
- μ_e: mean time between events (including three event types)

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

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

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

Example



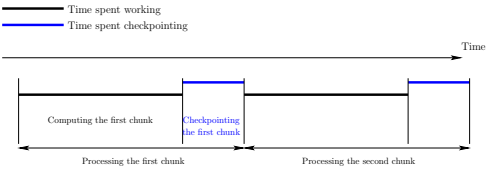
- Predictor predicts six faults in time t
- Five actual faults. One fault not predicted
- $\mu = \frac{t}{5}$, $\mu_P = \frac{t}{6}$, and $\mu_{NP} = t$
- Recall $r = \frac{4}{5}$ (green arrows over red arrows)
- Precision $p = \frac{4}{6}$ (green arrows over blue arrows)

Algorithm

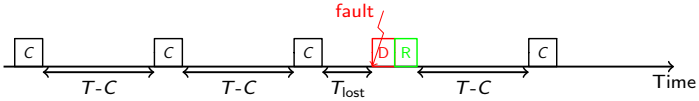
1. While no fault prediction is available:
   - checkpoints taken periodically with period T
2. When a fault is predicted at time t:
   - take a checkpoint ALAP (completion right at time t)
   - after the checkpoint, complete the execution of the period

Computing the waste

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

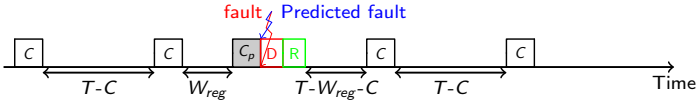


2. **Unpredicted faults:** $\frac{1}{\mu_{NP}}\left[D + R + \frac{T}{2}\right]$

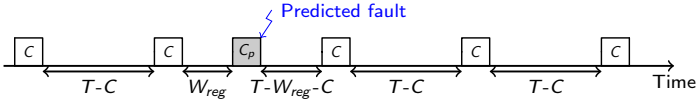


Computing the waste

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



with actual fault (true positive)



no actual fault (false negative)

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

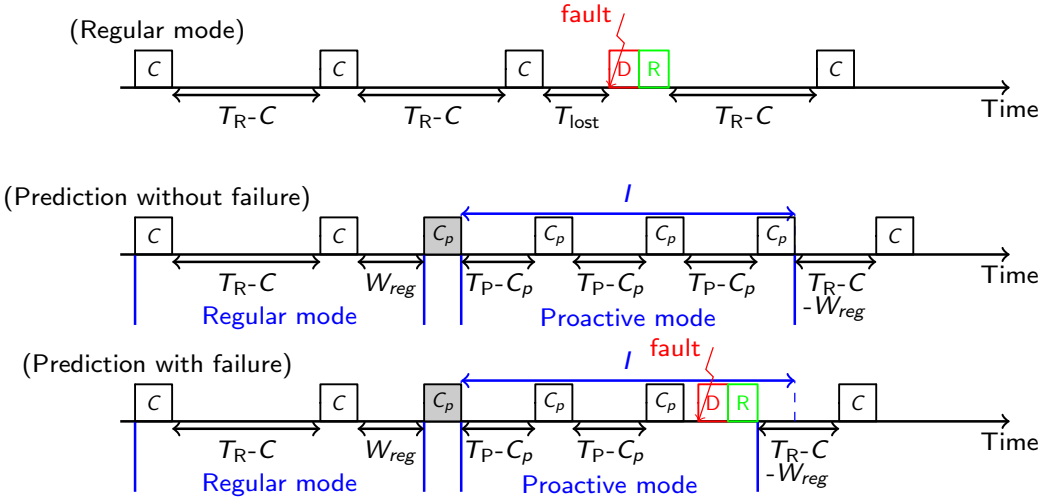
Refinements

- Use different value C_p for proactive checkpoints

- Avoid checkpointing too frequently for false negatives
  $\Rightarrow$ Only trust predictions with some fixed probability q
  $\Rightarrow$ Ignore predictions with probability $1 - q$
  Conclusion: trust predictor always or never ($q = 0$ or $q = 1$)

- Trust prediction depending upon position in current period
  $\Rightarrow$ Increase q when progressing
  $\Rightarrow$ Break-even point $\frac{C_p}{p}$

With prediction windows



Gets too complicated! ☹

Outline

Which checkpointing protocol to use?

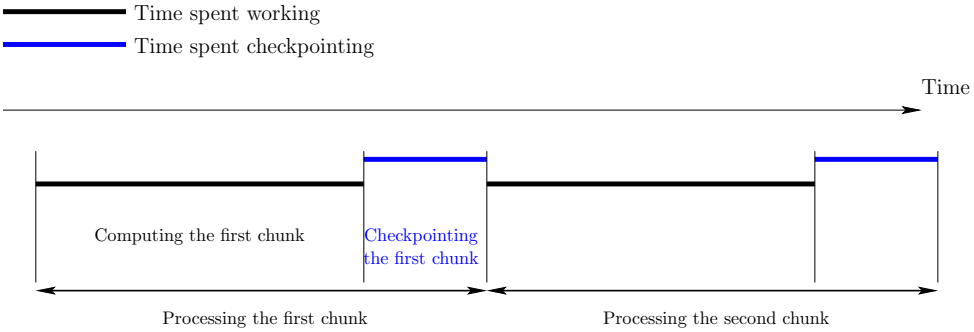
Coordinated checkpointing

- ☺ 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

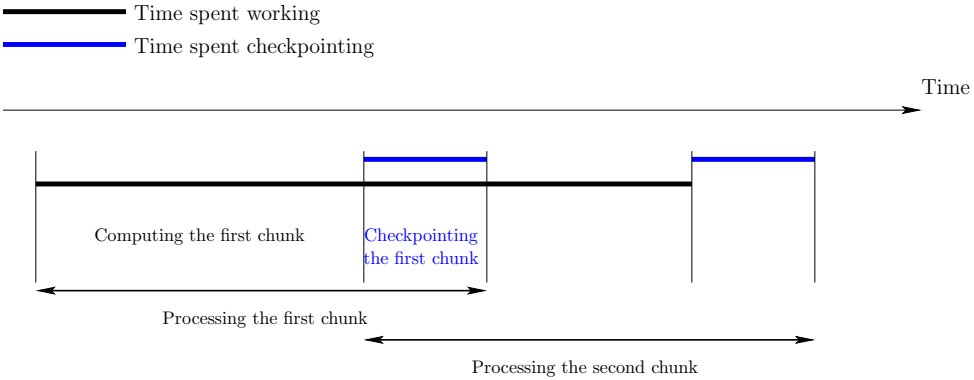
- ☹ Need to log inter-groups messages
  - Slowdowns failure-free execution
  - Increases checkpoint size/time
- ☺ Only processors from failed group need to roll back
- ☺ Faster re-execution with logged messages
- ☺ Rumor: Should scale to very large platforms

Blocking vs. non-blocking



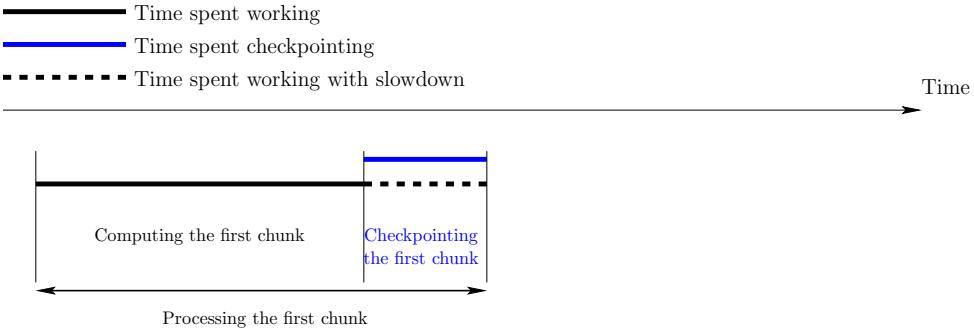
Blocking model: checkpointing blocks all computations

Blocking vs. non-blocking



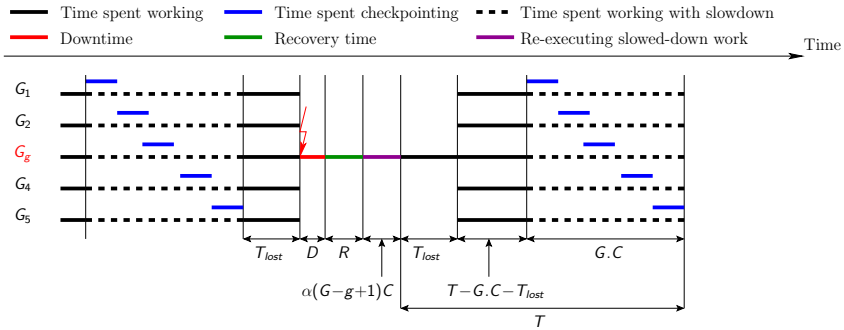
Non-blocking model: checkpointing has no impact on computations (e.g., first copy state to RAM, then copy RAM to disk)

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 Bandwidth (b_{io}) Read	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

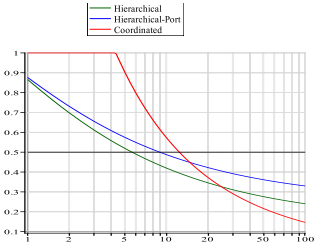
Checkpoint time

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

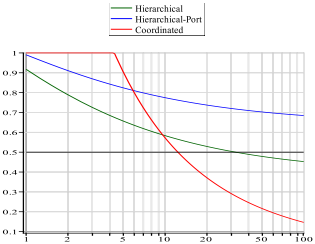
- Large time to dump the memory
- Using $1\%C$
- Comparing with $0.1\%C$ for exascale platforms
- $\alpha = 0.3$, $\lambda = 0.98$ and $\rho = 1.5$

Plotting formulas – Platform: Titan

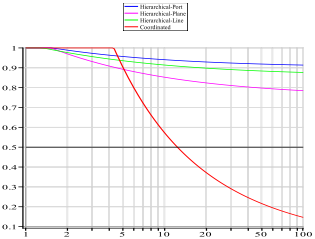
Stencil 2D



Matrix product



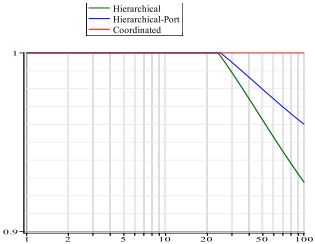
Stencil 3D



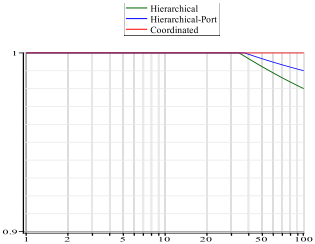
Waste as a function of processor MTBF μ_{ind}

Platform: K-Computer

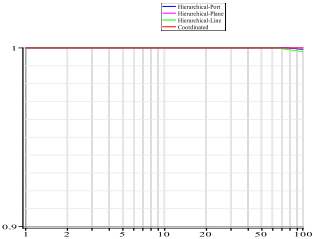
Stencil 2D



Matrix product



Stencil 3D



Waste as a function of processor MTBF μ_{ind}

Plotting formulas – Platform: Exascale

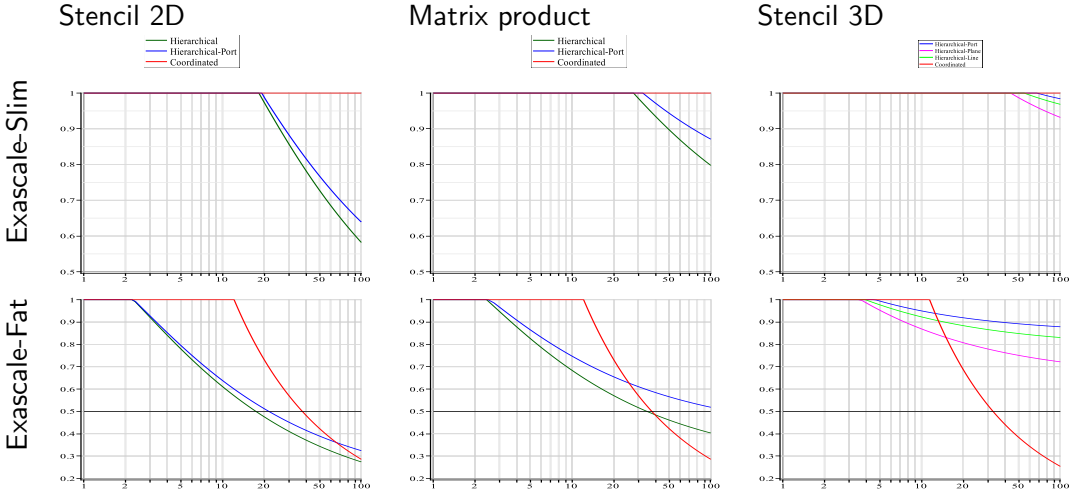
WASTE = 1 for all scenarios!!!

Plotting formulas – Platform: Exascale

ASTE = for all scenarios!!!

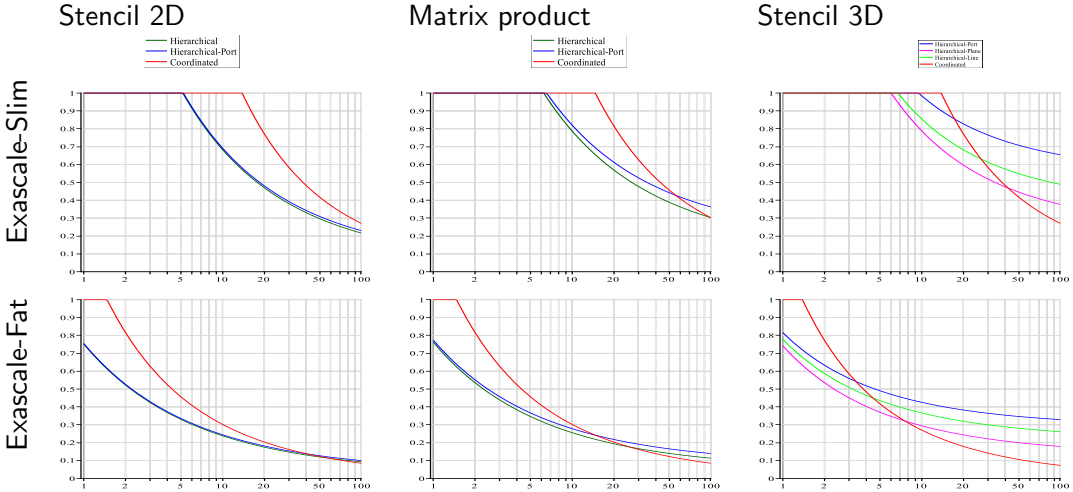
Goodbye Exascale?!

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



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

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



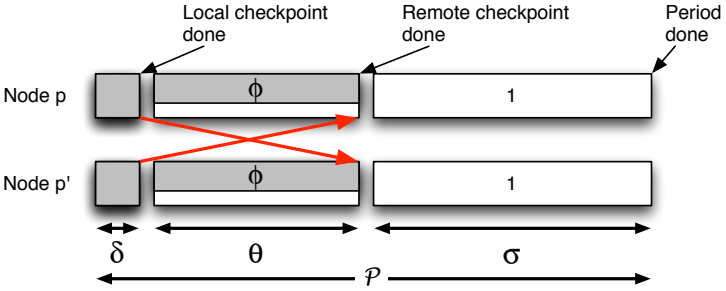
Waste as a function of processor MTBF μ_{ind}, $C = 100$

Outline

Motivation

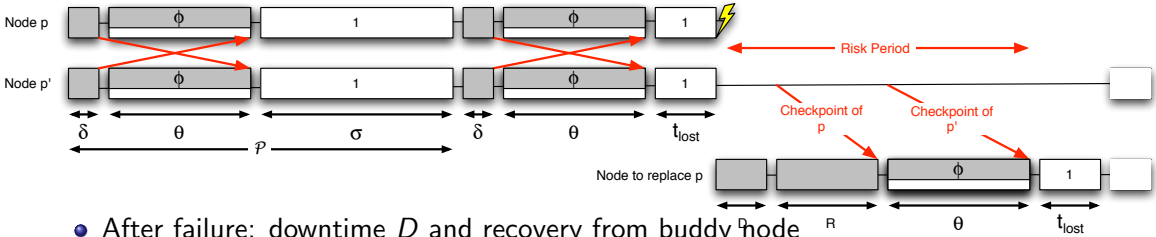
- Checkpoint transfer and storage
  $\Rightarrow$ critical issues of rollback/recovery protocols
- Stable storage: high cost
- Distributed in-memory storage:
  - Store checkpoints in local memory $\Rightarrow$ no centralized storage
    ☺ Much better scalability
  - Replicate checkpoints $\Rightarrow$ 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

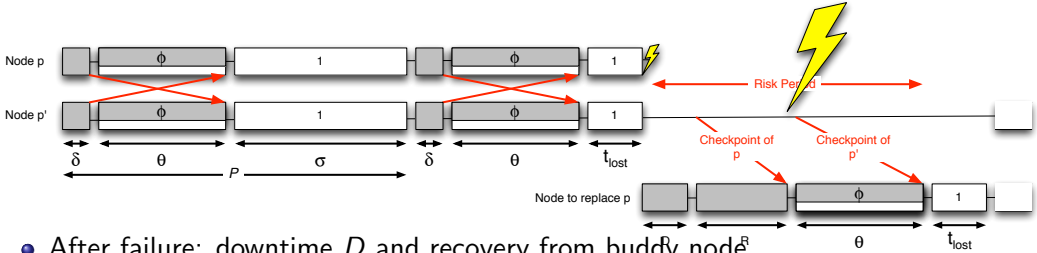
Failures



- 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

Multi-level checkpointing

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

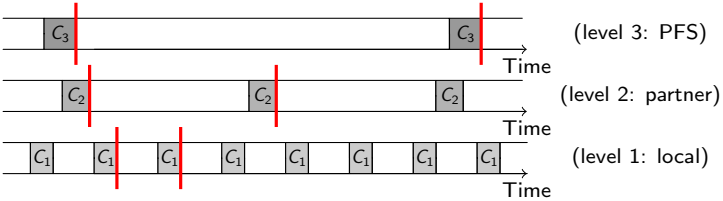
Multiple technologies to cope with different failure types:

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

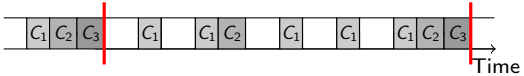
Scalable Checkpoint/Restart (SCR) library
Fault Tolerance Interface (FTI)

Simplified model

- Independent checkpointing:

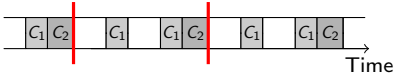


- Synchronized checkpointing:



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:

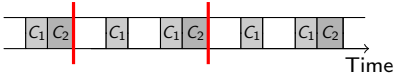
$$H_{\text{opt}} = \sqrt{2\lambda_1 C_1} + \sqrt{2\lambda_2 C_2} + \Theta(\lambda)$$

(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:

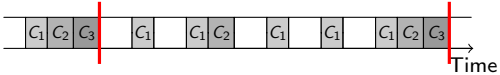
$$H_{\text{opt}} = \sqrt{2\lambda_1 C_1} + \sqrt{2\lambda_2 C_2} + \Theta(\lambda)$$

(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:

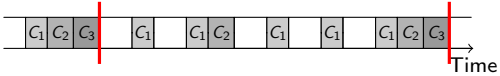
$$H_{\text{opt}} = \sqrt{2\lambda_1 C_1} + \sqrt{2\lambda_2 C_2} + \sqrt{2\lambda_3 C_3} + \Theta(\lambda)$$

- 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:

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

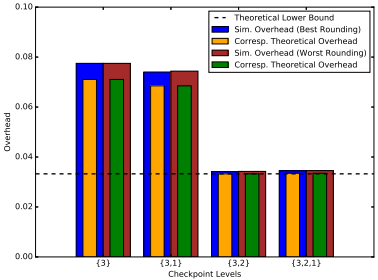
$$\text{Optimal \#chkpts at level } \ell\text{: } N_\ell^{opt} = \sqrt{\frac{\lambda_\ell}{C_\ell} \cdot \frac{C_k}{\lambda_k}}, \quad \forall \ell = 1, \ldots, k$$

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

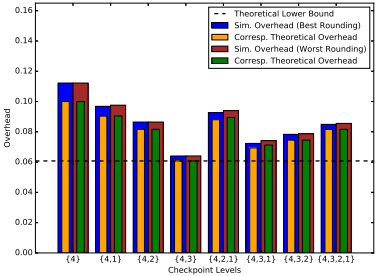
- Dynamic programming algorithm to choose set of levels
- Rounding for integer solution: $n_\ell^{\text{opt}} = \frac{N_\ell^{\text{opt}}}{N_{\ell+1}^{\text{opt}}} = \sqrt{\frac{\lambda_\ell}{\lambda_{\ell+1}} \cdot \frac{C_{\ell+1}}{C_\ell}}$

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 energy and run time for multilevel checkpointing. *PMBS*, 2014.

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
  $\Longleftrightarrow$ 10,000s of nodes. Beyond?
- Independent errors ☹
  Correlated failures across levels?

Outline

The Young/Daly formula

$$T_{\text{opt}} = \sqrt{\frac{2C}{\lambda_N}} = \sqrt{2\mu_N C} = \Theta(\lambda^{-\frac{1}{2}}) \quad (1)$$

$$\mathbb{H}_{opt} = \sqrt{2C\lambda_N} + o(\lambda^{\frac{1}{2}}) = \Theta(\lambda^{\frac{1}{2}}) \quad (2)$$

Recall that $\lambda_N = N\lambda = \frac{1}{\mu_N} = \frac{N}{\mu}$

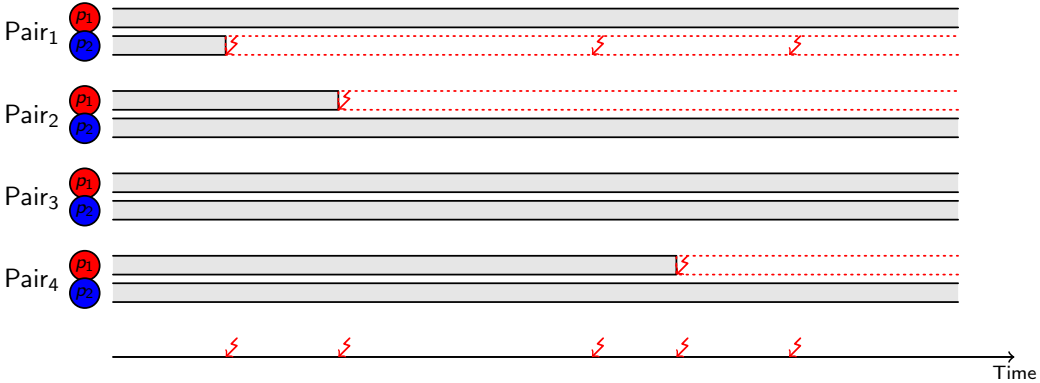
Replication

- Full replication: efficiency $< 50\%$
- Can replication+checkpointing be more efficient than checkpointing alone?
- Study by Ferreira et al. [SC'2011]: yes
- Revisited by Hussain, Znati and Melhem [SC'2018]: yes

Model by Ferreira et al. [SC' 2011]

- Platform with $N = 2b$ processors arranged into b pairs
- Parallel application with b processes, each replicated
- When a replica is hit by a failure, it is not restarted
- Application fails when both replicas in one pair have been hit

Example



Mean Time To Interruption

- $n_{\text{fail}}(2b)$ expected number of failures to interrupt the applications
- MTTI M_{2b} = Mean Time to Interruption
  $\Rightarrow$ replaces MTBF from the application perspective

$$M_{2b} = n_{\text{fail}}(2b) \times \mu_{2b} = n_{\text{fail}}(2b) \times \frac{\mu}{2b} = \frac{n_{\text{fail}}(2b)}{2\lambda b} \tag{3}$$

Mean Time To Interruption

- $n_{\text{fail}}(2b)$ expected number of failures to interrupt the applications
- MTTI M_{2b} = Mean Time to Interruption
  $\Rightarrow$ replaces MTBF from the application perspective

$$M_{2b} = n_{\text{fail}}(2b) \times \mu_{2b} = n_{\text{fail}}(2b) \times \frac{\mu}{2b} = \frac{n_{\text{fail}}(2b)}{2\lambda b} \quad (3)$$

Proposition

$$n_{fail}(2b) = 1 + 4^b / \binom{2b}{b} \approx \sqrt{\pi b}$$

Checkpointing

$$\text{No Replication} \quad T_{\text{opt}} = \sqrt{2\mu_N C} \tag{4}$$

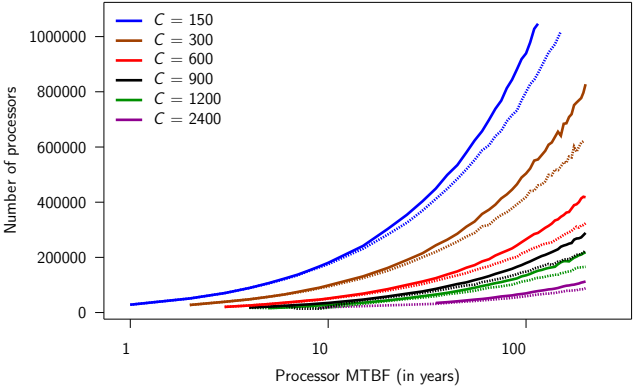
$$\text{Full Replication} \quad T_{\text{opt}} = \sqrt{2M_N C} \tag{5}$$

Comparison

- N processors, no replication
  $\text{THROUGHPUT}_{\text{Std}} = N(1 - \text{WASTE}) = N\left(1 - \sqrt{\frac{2C}{\mu_N}}\right)$
- $N = 2b$, b replica-pairs
  $\text{THROUGHPUT}_{\text{Rep}} = \frac{N}{2}\left(1 - \sqrt{\frac{2C}{M_N}}\right)$
- Platform with $N = 2^{20}$ processors $\Rightarrow M_N = 1284.4$
  $\mu = 10$ years $\Rightarrow$ better if C shorter than 6 minutes

Weibull distribution with $k = 0.7$

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



- Study by Ferrreira et al. favors replication
- Replication beneficial if small μ + large C + big N

Outline

Definitions

- Instantaneous error detection ⇒ fail-stop failures, e.g. resource crash
- Silent errors (data corruption) ⇒ detection latency

Silent error detected only when corrupt data is activated and modifies application behavior

- Includes some software faults, some hardware errors (soft errors in L1 cache, ALU), double bit flip
- Cannot always be corrected by ECC memory

Probability distributions for silent errors

?

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

(a.k.a, scale is the enemy)

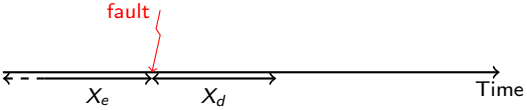
Probability distributions for silent errors

?

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

(a.k.a, scale is the enemy)

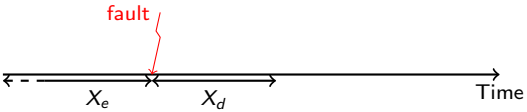
Coupling checkpointing with detectors



Error and detection latency

- 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?

Coupling checkpointing with detectors



Error and detection latency

- 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!

Detectors

- 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)

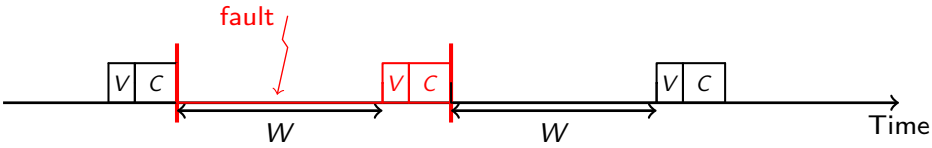
On-line ABFT scheme for PCG

```
Compute r^(0) = b − Ax^(0), z^(0) = M^−1 r^(0), p^(0) = z^(0),
    and ρ_0 = r^(0)^T z^(0) for some initial guess x^(0)
checkpoint: A, M, and b
for i = 0, 1, ...
    if ( (i>0) and (i%d = 0) )
        if ( p^(i+1)^T q^(i) / (||p^(i+1)||.||q^(i)||) > 10^−10
          or ||r^(i+1) + Ax^(i+1) − b|| / (||b||.||A||) > 10^−10 )
            recover: A, M, b, i, ρ_i,
                     p^(i), x^(i), and r^(i).
        else if ( i%(cd) = 0 )
            checkpoint: i, ρ_i, p^(i), and x^(i)
        endif
    endif
    q^(i) = Ap^(i)
    α_i = ρ_i / p^(i)^T q^(i)
    x^(i+1) = x^(i) + α_i p^(i)
    r^(i+1) = r^(i) − α_i q^(i)
    solve Mz^(i+1) = r^(i+1), where M = M^T
    ρ_{i+1} = r^(i+1)^T z^(i+1)
    β_i = ρ_{i+1} / ρ_i
    p^(i+1) = z^(i+1) + β_i p^(i)
    check convergence; continue if necessary
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	Silent errors
Pattern	$T = W + C$	$T = W + V + C$
$\text{WASTE}[FF]$	$\frac{C}{T}$	?
$\text{WASTE}[fail]$	$\frac{1}{\mu}(D + R + \frac{T}{2})$	?
Optimal	$T_{\text{opt}} = \sqrt{2C\mu}$	?
$\text{WASTE}[opt]$	$W_{\text{opt}} = \sqrt{\frac{2C}{\mu}}$	?

Solution

Fail-stop error

- $W_{FF} = \frac{C}{T}$
- $W_{fail} = \frac{1}{\mu}(D + R + \frac{T}{2}) \approx \frac{T}{2\mu}$
- $W_{tot} \approx W_{FF} + W_{fail} = \frac{C}{T} + \frac{T}{2\mu}$
- $T_{\text{opt}} = \sqrt{2C\mu}$ and $W_{\text{opt}} = \sqrt{\frac{2C}{\mu}}$

Silent error ???

Solution

Fail-stop error

- $W_{FF} = \frac{C}{T}$
- $W_{fail} = \frac{1}{\mu}(D + R + \frac{T}{2}) \approx \frac{T}{2\mu}$
- $W_{tot} \approx W_{FF} + W_{fail} = \frac{C}{T} + \frac{T}{2\mu}$
- $T_{\text{opt}} = \sqrt{2C\mu}$ and $W_{\text{opt}} = \sqrt{\frac{2C}{\mu}}$

Silent error ???

- $W_{FF} = \frac{V+C}{T}$
- $W_{fail} = \frac{1}{\mu}(D + R + T) \approx \frac{T}{\mu}$
- $W_{tot} \approx W_{FF} + W_{fail} = \frac{V+C}{T} + \frac{T}{\mu}$
- $T_{\text{opt}} = \sqrt{(V + C)\mu}$ and $W_{\text{opt}} = 2\sqrt{\frac{V+C}{\mu}}$

Now with both fail-stop and silent errors?

- μ_f MTBF of fail-stop errors
- μ_s MTBF of silent errors

Solution ???

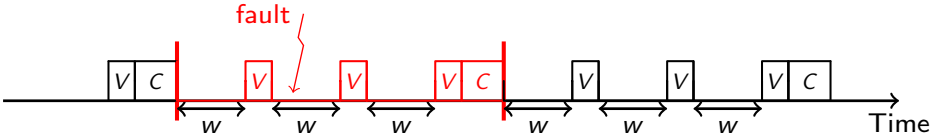
Now with both fail-stop and silent errors?

- μ_f MTBF of fail-stop errors
- μ_s MTBF of silent errors

Solution ???

- $W_{FF} = \frac{V+C}{T}$
- $W_{fail} \approx \frac{T}{\mu_f} + \frac{T}{2\mu_s}$
- $W_{tot} \approx \frac{V+C}{T} + T(\frac{1}{\mu_f} + \frac{1}{2\mu_s})$
- $T_{\text{opt}} = \sqrt{\frac{V+C}{\frac{1}{\mu_f}+\frac{1}{2\mu_s}}}$

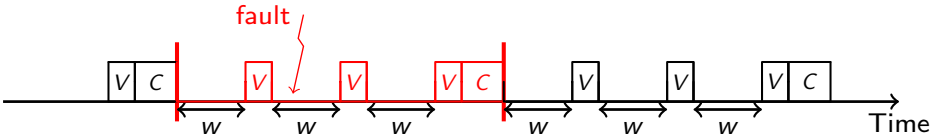
Now with 1 checkpoint and 3 verifications



Only silent errors

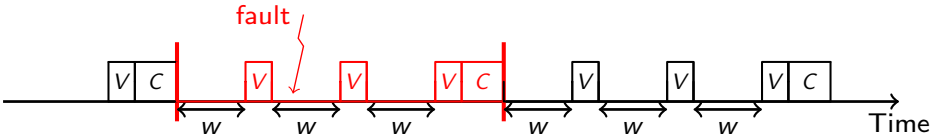
Base Pattern	$\textsc{Waste}[opt] = 2\sqrt{\frac{C+V}{\mu}}$
New Pattern	?

Now with 1 checkpoint and 3 verifications (solution)



- $W_{FF} = \frac{3V+C}{T}$
- $W_{fail} \approx \frac{1}{\mu}(\frac{1}{3} \times \frac{T}{3} + \frac{1}{3} \times \frac{2T}{3} + \frac{1}{3} \times \frac{3T}{3}) = \frac{2T}{3\mu}$
- $T_{\text{opt}} = \sqrt{\frac{3}{2}(3V + C)\mu}$
- $W_{\text{opt}} = 2\sqrt{\frac{2(3V+C)}{3\mu}}$

Now with 1 checkpoint and 3 verifications



Only silent errors

Base Pattern	$\text{WASTE}[opt] = 2\sqrt{\frac{C+V}{\mu}}$
New Pattern	$\text{WASTE}[opt] = 2\sqrt{\frac{2(3V+C)}{3\mu}}$

New pattern better for $V \leq \frac{C}{3}$

Outline

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 ata-analytics/machine learning approaches

Application-specific detectors

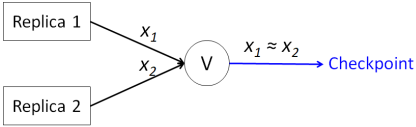
Do you believe it?

- 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?

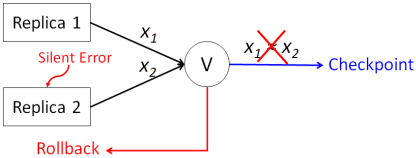
- **Error detection (duplication)**:



- **Error correction (triplication)**:

Why Is Replication Useful?

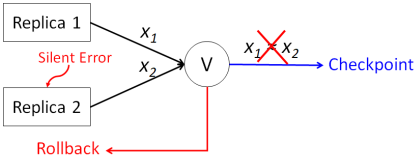
- **Error detection (duplication)**:



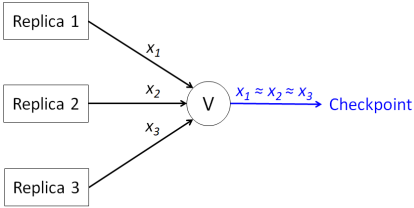
- **Error correction (triplication)**:

Why Is Replication Useful?

- **Error detection (duplication)**:

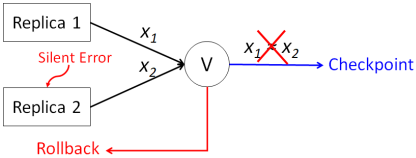


- **Error correction (triplication)**:

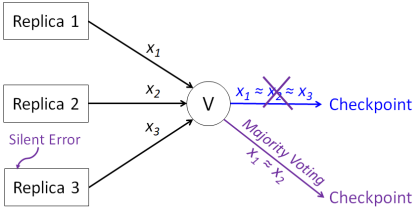


Why Is Replication Useful?

- **Error detection (duplication)**:

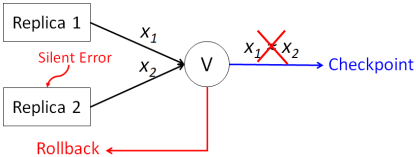


- **Error correction (triplication)**:

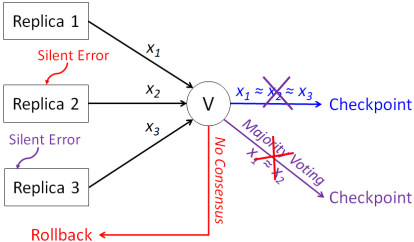


Why Is Replication Useful?

- **Error detection (duplication)**:

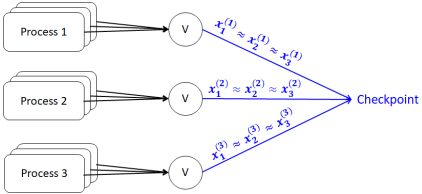


- **Error correction (triplication)**:

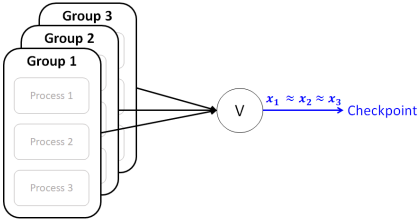


Two Replication Modes

- **Process Replication**:

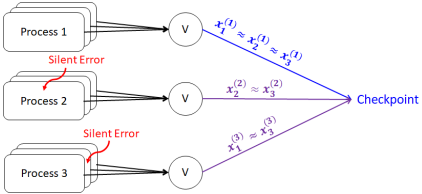


- **Group Replication**:

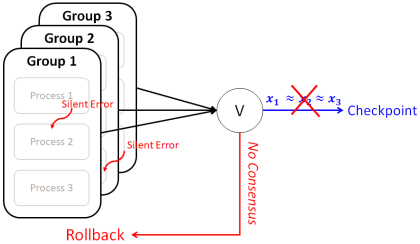


Two Replication Modes

- **Process Replication**:



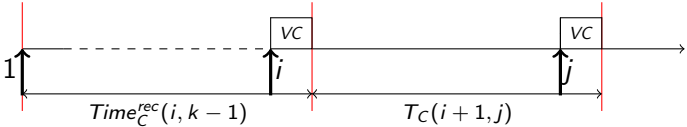
- **Group Replication**:



Dynamic programming for linear chains of tasks

- $\{T_1, T_2, \ldots, T_n\}$: 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



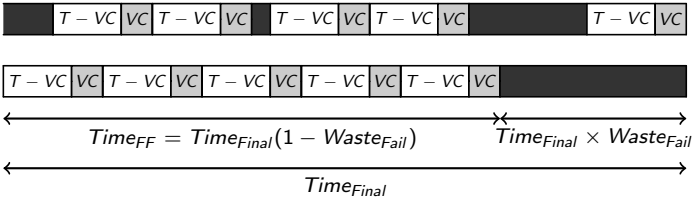
From T_1 up to T_n and checkpoint: $Time_C^{rec}(n)$

From T_1 up to T_j and checkpoint: $Time_C^{rec}(j)$

$$Time_C^{rec}(j) = \min_{0 \le i < j} \{ Time_C^{rec}(i) + T_C^{SF}(i+1, j) \}$$

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

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

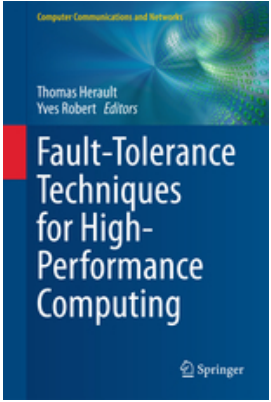
Silent errors

- 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 = comprehensive survey, freely available as LAWN 289 (LApack Working Note)

Outline

Hands-on

Hands-On

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

It is available online:

`http://fault-tolerance.org/sc19`

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

Outline

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?

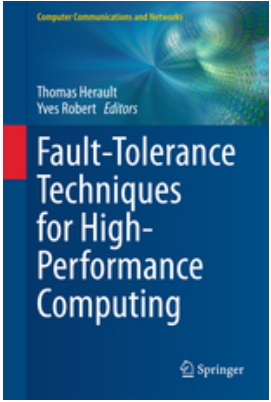
Conclusion

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 SC19 tutorial evaluation form

`http://bit.ly/sc19-eval`