An overview of fault-tolerant techniques for HPC

Thomas Hérault[1] & Yves Robert[1,2]

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

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

http://graal.ens-lyon.fr/~yrobert/sc12tutorial.pdf

SC'2012 Tutorial

Thanks

INRIA & ENS Lyon

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

UT Knoxville

- George Bosilca
- Aurélien Bouteiller
- Jack Dongarra

Others

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

Outline

Outline

Outline

Exascale platforms (courtesy Jack Dongarra)

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

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

Exascale platforms

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

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

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

Exascale platforms

- Hierarchical
  - 10^5 or 10^6 nodes
  - Each node equipped with 10^4 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 = ... (between failures)

Exascale
$\neq$ Petascale $\times 1000$

Scenario for 2015

- Phase-Change memory
  - read bandwidth 100GB/sec
  - write bandwidth 10GB/sec
- Checkpoint size 128GB
- C: checkpoint save time: $C = 12sec$
- R: checkpoint recovery time: $R = 1.2sec$
- D: down/reboot time: $D = 15sec$
- p: total number of (multicore) nodes: $p = 2^8$ to $p = 2^{20}$
- MTBF $\mu = 1$ week, 1 month, 1|10|100|1000 years (per node)

Distribution of parallel jobs

Number of processors required by typical jobs: *two-stage log-uniform distribution biased to powers of two* (says Dr. Feitelson)

- Let $p = 2^Z$ for simplicity
- Probability that a job is sequential: $\alpha_0 = p_1 \approx 0.25$
- Otherwise, the job is parallel, and uses 2^j processors with identical probability
- **Steady-state** utilization of whole platform:
  - all processors always active
  - constant proportion of jobs using any number of processors

Platform throughput with optimal checkpointing period

μ =1 week	p	Throughput
	2^8	91.56%
	2^{11}	73.75%
	2^{14}	20.07%
	2^{17}	2.51%
	2^{20}	0.31%

μ =1 month	p	Throughput
	2^8	96.04%
	2^{11}	88.23%
	2^{14}	62.28%
	2^{17}	10.66%
	2^{20}	1.33%

μ =1 year	p	Throughput
	2^8	98.89%
	2^{11}	96.80%
	2^{14}	90.59%
	2^{17}	70.46%
	2^{20}	15.96%

μ =10 years	p	Throughput
	2^8	99.65%
	2^{11}	99.00%
	2^{14}	97.15%
	2^{17}	91.63%
	2^{20}	74.01%

μ =100 years	p	Throughput
	2^8	99.89%
	2^{11}	99.69%
	2^{14}	99.11%
	2^{17}	97.45%
	2^{20}	92.56%

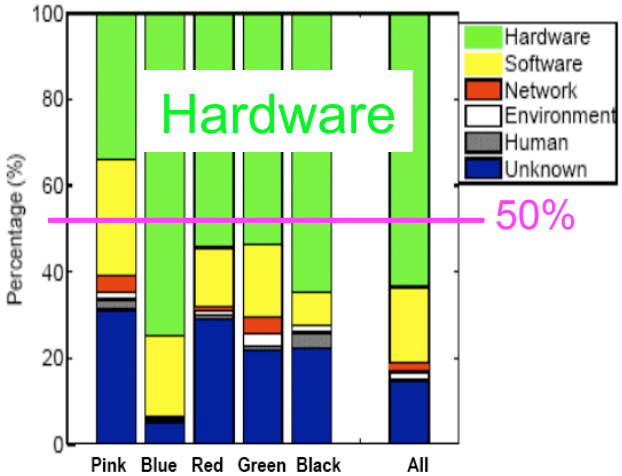
μ =1000 years	p	Throughput
	2^8	99.97%
	2^{11}	99.90%
	2^{14}	99.72%
	2^{17}	99.20%
	2^{20}	97.73%

Outline

Error sources (courtesy Franck Cappello)

Sources of failures

- Analysis of error and failure logs
- In 2005 (Ph. D. of CHARNG-DA LU) : "Software halts account for the most number of outages (59-84 percent), and take the shortest time to repair (0.6-1.5 hours). Hardware problems, albeit rarer, need 6.3-100.7 hours to solve."
- In 2007 (Garth Gibson, ICPP Keynote):



Relative frequency of root cause by system type.

- In 2008 (Oliner and J. Stearley, DSN Conf.):

Type	Raw		Filtered	
	Count	%	Count	%
Hardware	174,586,516	98.04	1,999	18.78
Software	144,899	0.08	6,814	64.01
Indeterminate	3,350,044	1.88	1,832	17.21

Software errors: Applications, OS bug (kernel panic), communication libs, File system error and other.

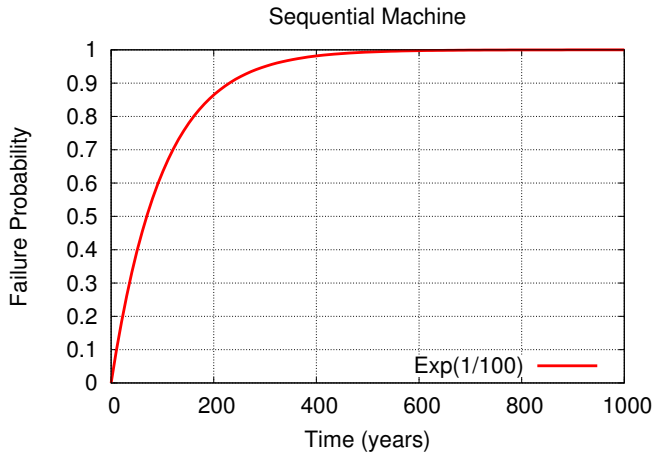
Hardware errors, Disks, processors, memory, network

Conclusion: Both Hardware and Software failures have to be considered

A few definitions

- Many types of faults: software error, hardware malfunction, memory corruption
- Many possible behaviors: silent, transient, unrecoverable
- Restrict to faults that lead to application failures
- This includes all hardware faults, and some software ones
- Will use terms *fault* and *failure* interchangeably

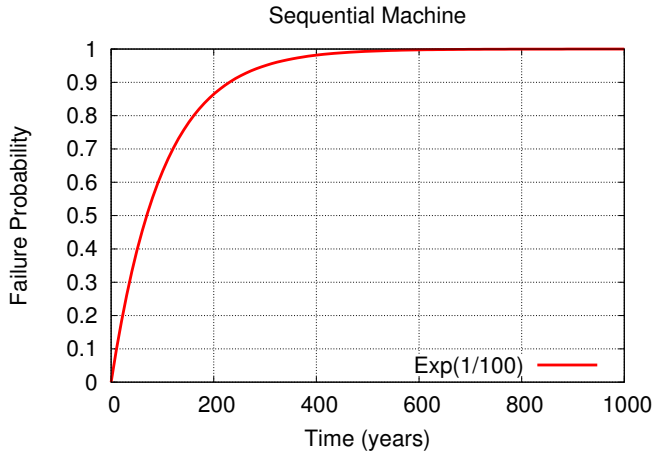
Failure distributions: (1) Exponential



$Exp(\lambda)$: 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 $= \frac{1}{\lambda}$

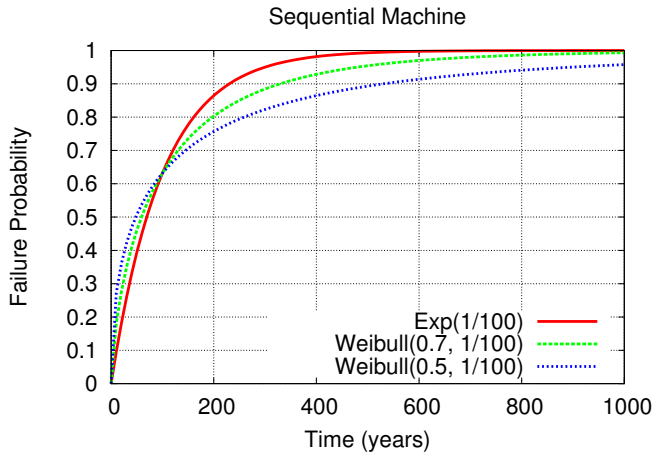
Failure distributions: (1) Exponential



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) $\mu = \mathbb{E}(X) = \frac{1}{\lambda}$

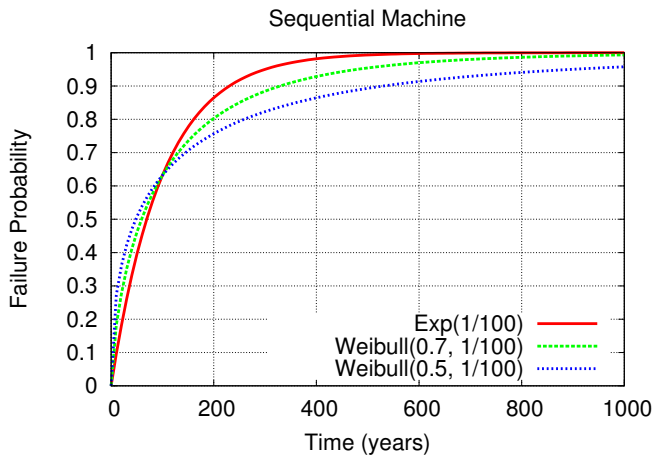
Failure distributions: (2) Weibull



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?

- Well, it depends ☹

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?

- Well, it depends ☹

With rejuvenation

- Rebooting all p processors after a failure
- Platform failure distribution
  $\Rightarrow$ minimum of p IID processor distributions
- With p distributions $Exp(\lambda)$:

$$\min\left(Exp(\lambda_1), Exp(\lambda_2)\right) = Exp(\lambda_1 + \lambda_2)$$

$$\mu = \frac{1}{\lambda} \Rightarrow \mu_p = \frac{\mu}{p}$$

- With p distributions $Weibull(k, \lambda)$:

$$\min_{1..p}\left(Weibull(k, \lambda)\right) = Weibull(k, p^{1/k}\lambda)$$

$$\mu = \frac{1}{\lambda}\Gamma(1 + \frac{1}{k}) \Rightarrow \mu_p = \frac{\mu}{p^{1/k}}$$

Without rejuvenation

- Rebooting only faulty processor
- Platform failure distribution
  $\Rightarrow$ superposition of p IID processor distributions
- Simple formula for arbitrary distributions:

$$\mu_p = \frac{\mu}{p}$$

  with p processors of MTBF μ
- Rejuvenation does not matter for Exponential
- Rejuvenation harmful for Weibull with $k < 1$

MTBF with p processors (1/2)

Theorem: $\mu_p = \frac{\mu}{p}$ for arbitrary distributions

With one processor:

- $n(F)$ = number of failures until time F is exceeded
- X_i iid random variables for inter-arrival times, with $\mathbb{E}(X_i) = \mu$
- $\sum_{i=1}^{n(F)-1} X_i \leq F \leq \sum_{i=1}^{n(F)} X_i$
- Wald's equation: $(\mathbb{E}(n(F)) - 1)\mu \leq F \leq \mathbb{E}(n(F))\,\mu$
- $\lim_{F \to +\infty} \frac{\mathbb{E}(n(F))}{F} = \frac{1}{\mu}$

MTBF with p processors (2/2)

Theorem: $\mu_p = \frac{\mu}{p}$ for arbitrary distributions

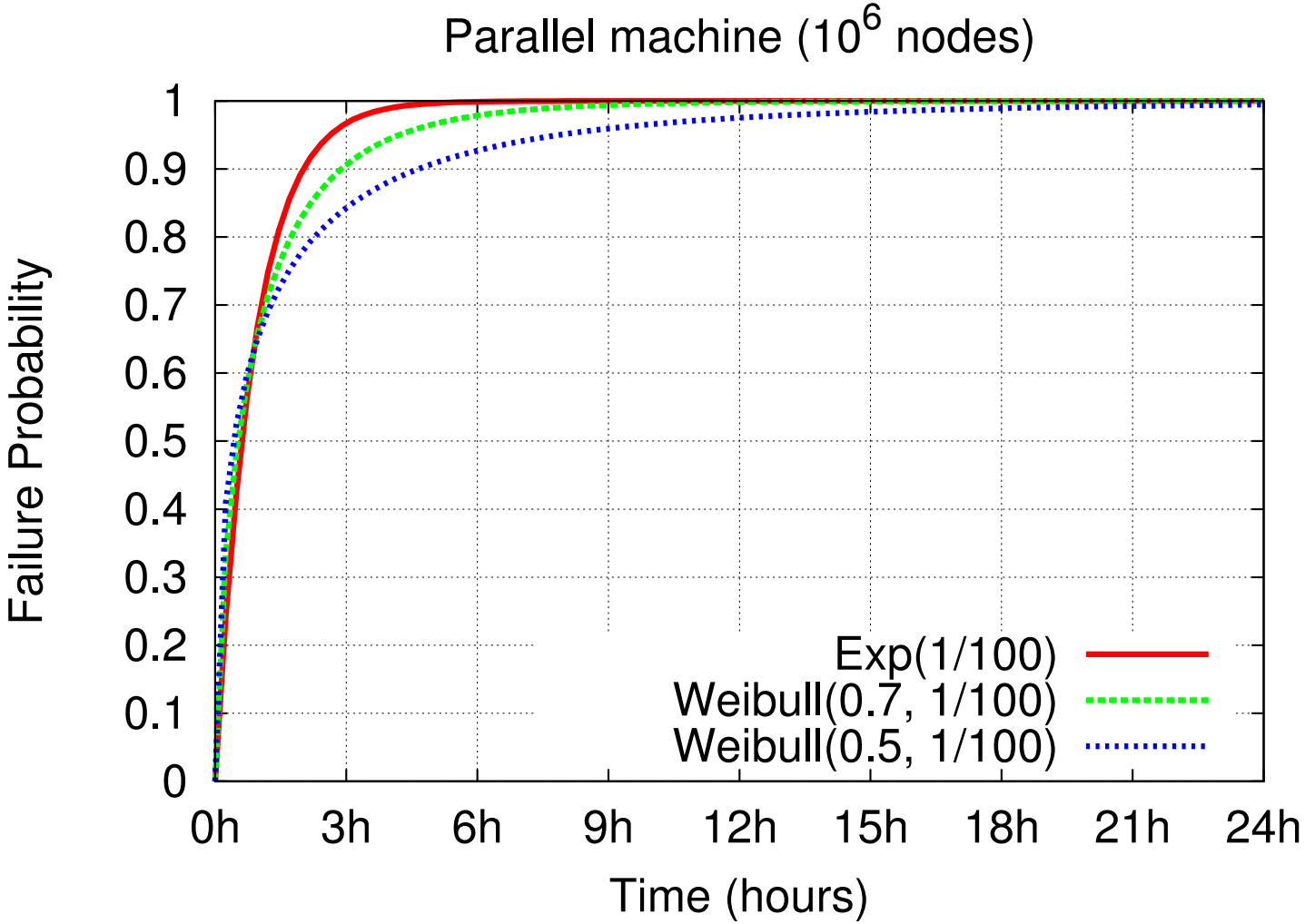
With p processors:

- $n(F)$ = number of platform failures until time F is exceeded
- $n_q(F)$ = number of those failures that strike processor q
- $n_q(F) + 1$ = number of failures on processor q until time F is exceeded (except for processor with last-failure)
- Y_i iid random variables for platform inter-arrival times, with $\mathbb{E}(Y_i) = \mu_p$
- $\lim_{F \to +\infty} \frac{n(F)}{F} = \frac{1}{\mu_p}$ as above
- $\lim_{F \to +\infty} \frac{n(F)}{F} = \frac{p}{\mu}$ because $n(F) = \sum_{q=1}^{p} n_q(F)$
- Hence $\mu_p = \frac{\mu}{p}$

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?



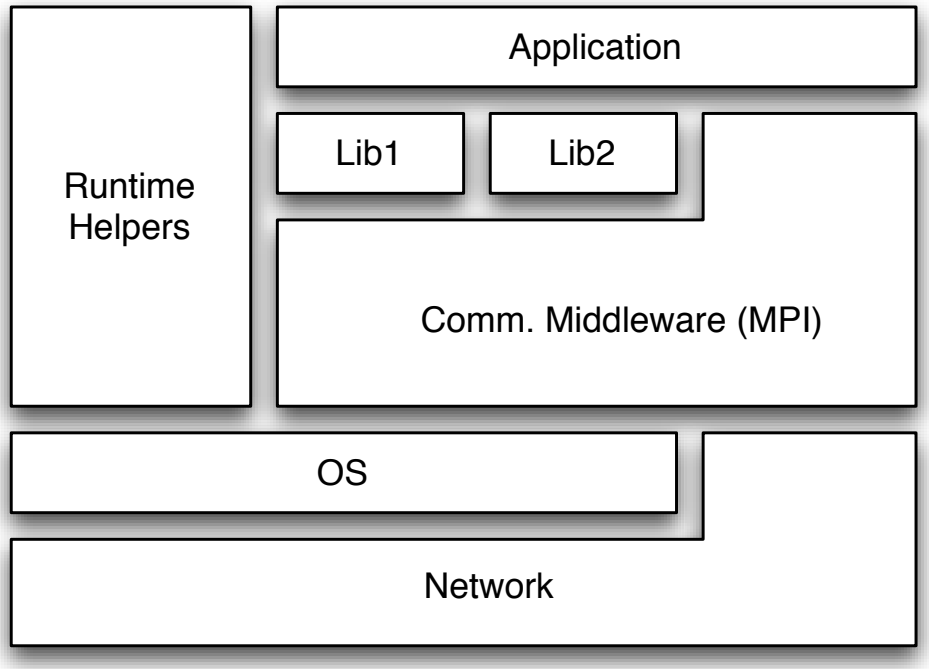
Outline

Motivation

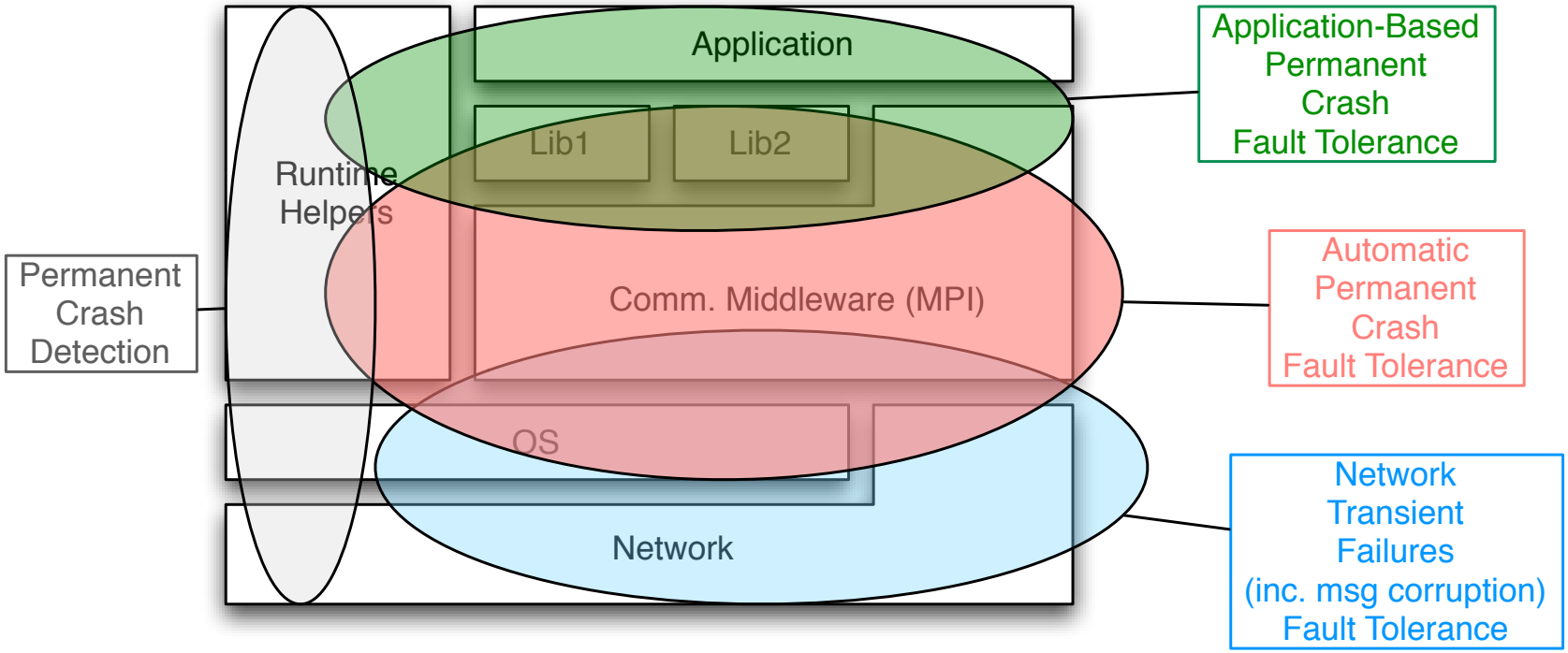
Motivation

- The more general, the less efficient. Hence the less general, the more efficient
- Naturally Fault Tolerant Applications

Fault Tolerance Software Stack



Fault Tolerance Software Stack



Outline

HPC – MPI

HPC

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

> *[...] it is the job of the implementor of the MPI subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, such failures will be reflected as errors in the relevant communication call. Similarly, MPI itself provides no mechanisms for handling processor failures.*
>
> – MPI Standard 2.2, p. 22

HPC – MPI

HPC

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

> *This document does not specify the state of a computation after an erroneous MPI call has occurred.*
>
> – MPI Standard 2.2, p. 23

HPC – MPI

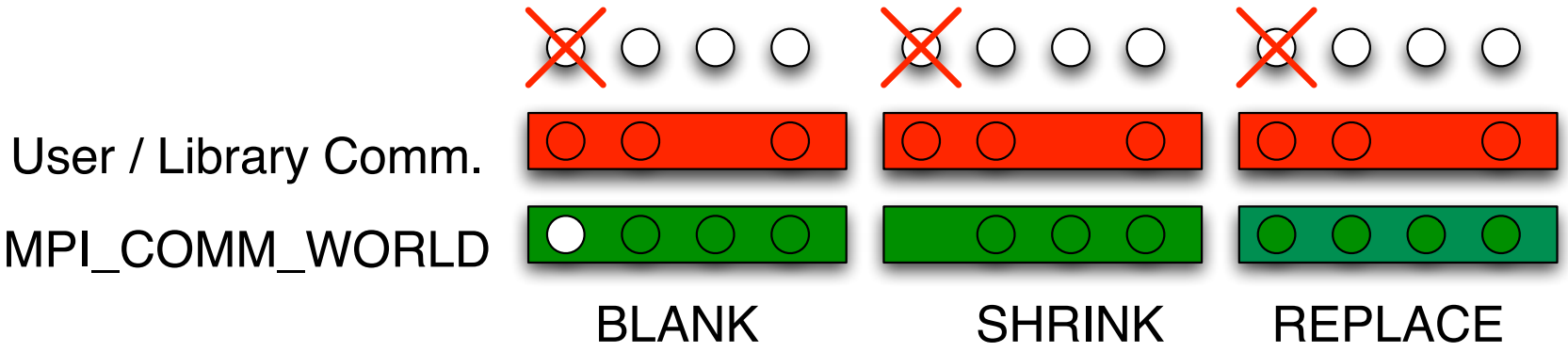
MPI Implementations

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

FT Middleware in HPC

- Not MPI. Sockets, PVM... CCI? `http://www.olcf.ornl.gov/center-projects/common-communication-interface/`
- FT-MPI: `http://icl.cs.utk.edu/harness/`, 2003
- MPI-3.x-FT proposal (Open MPI, MPICH)
- Checkpoint on Failures: the rejuvenation in HPC

FT-MPI



Models

- FT MPI models: BLANK, REPLACE, SHRINK
- Destroys all communicators, re-create COMM_WORLD
- Collective semantics:
  - Weak synchronization
  - Strong synchronization

MPI-3.x-FT proposal

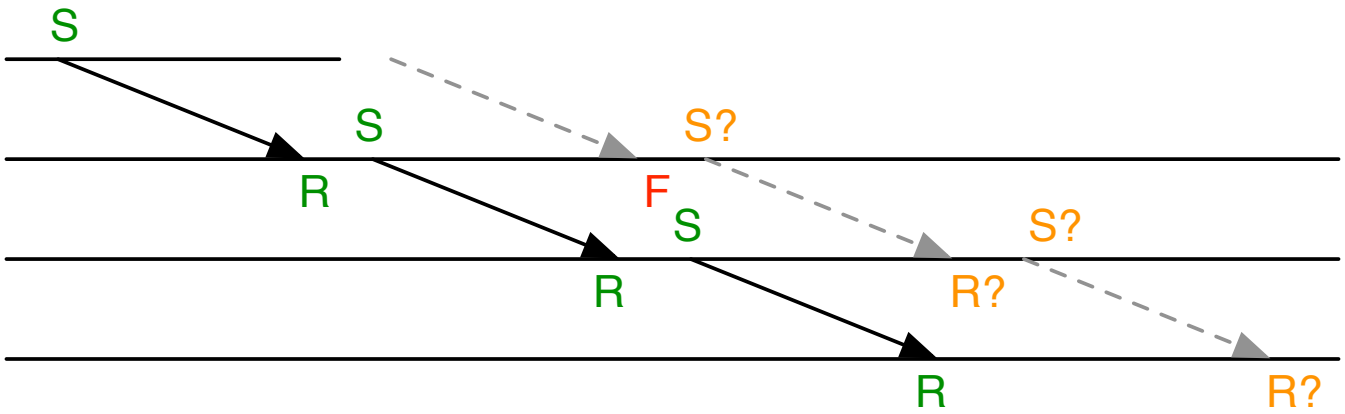
Point to Point

- Failure is reported if peer is dead
- `ANY_SOURCE`: report failure
- `MPI_COMM_FAILURE_ACK(comm)` stop reporting failures that are known at the moment of the call
- `MPI_COMM_FAILURE_GET_ACKED(comm, &group)` know the list of failures acknowledged until now

Collective

- Collective do not block because of failure.
- If operation cannot complete locally, and error must be reported

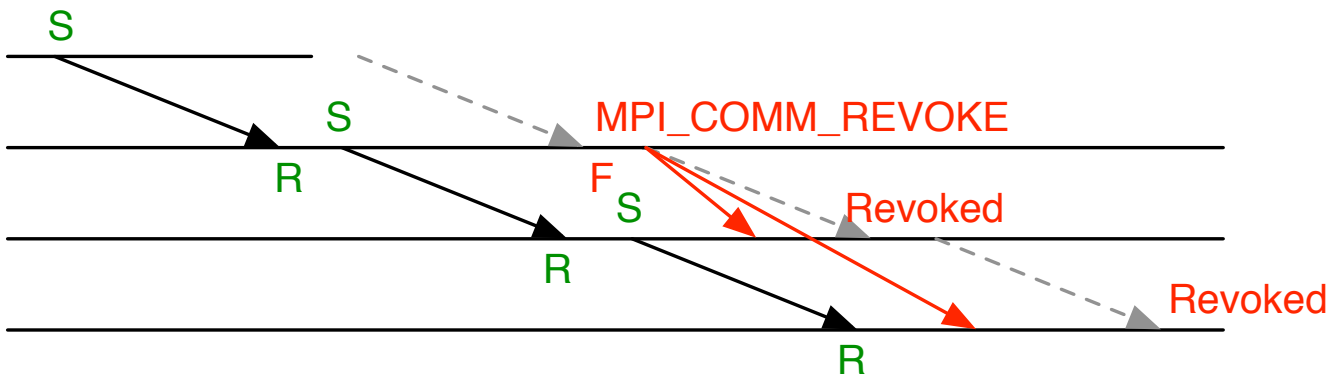
MPI-3.x-FT proposal



Additional MPI Calls – Revoke

- Transitive Closure of communications → possible deadlocks
- `MPI_COMM_REVOKE(comm)`: (globally) makes a communicator un-usable for communication (ever)
  - Similar to `MPI_ABORT`, but does not abort the process: aborts the communicator
  - *eventually* abort

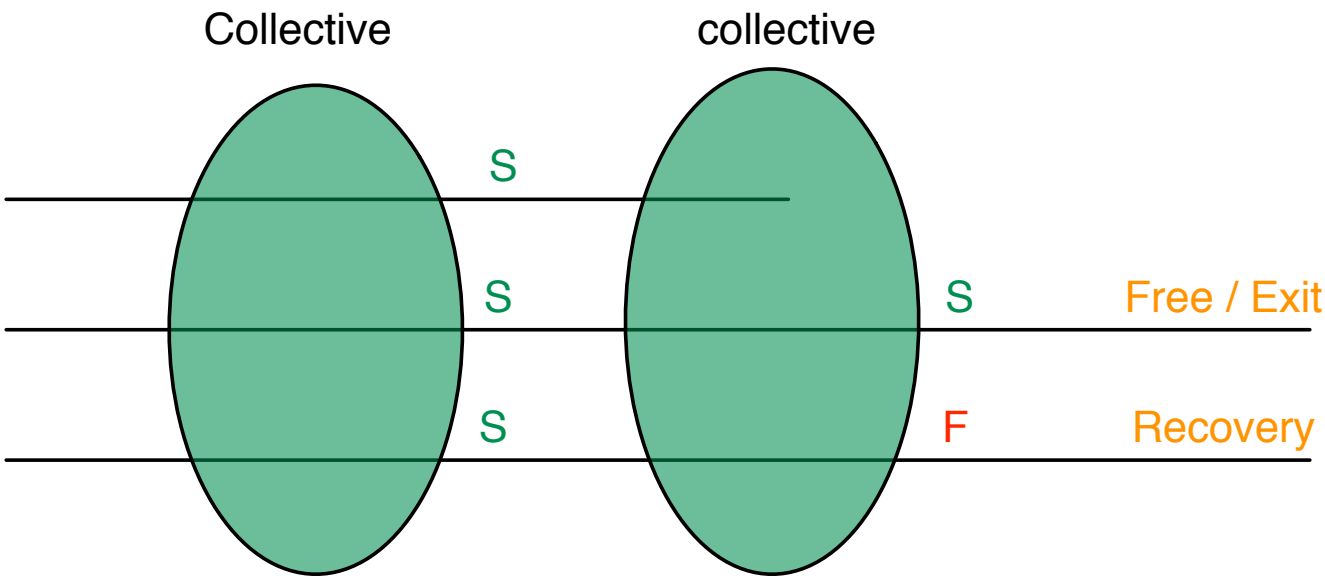
MPI-3.x-FT proposal



Additional MPI Calls – Revoke

- Transitive Closure of communications → possible deadlocks
- `MPI_COMM_REVOKE(comm)`: (globally) makes a communicator un-usable for communication (ever)
  - Similar to `MPI_ABORT`, but does not abort the process: aborts the communicator
  - *eventually* abort

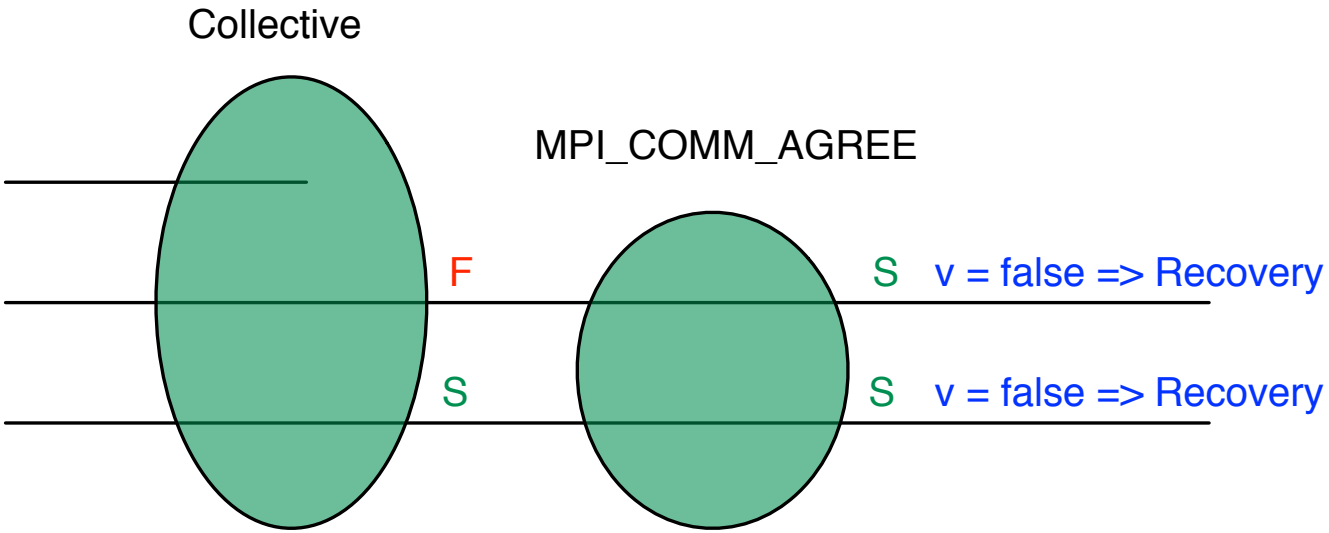
MPI-3.x-FT proposal



Additional MPI Calls – Agree

- Collectives do not need to report errors consistently
- Completion of operations: when to free a communicator?
- `MPI_COMM_AGREE(comm, &v)`: consistent value

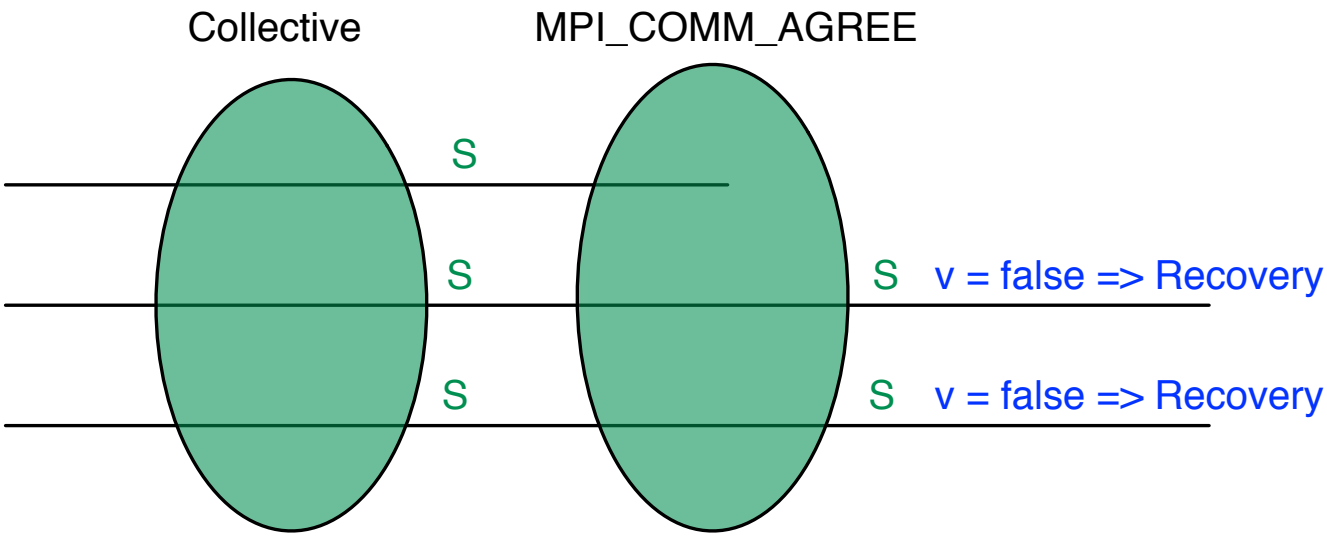
MPI-3.x-FT proposal



Additional MPI Calls – Agree

- `MPI_COMM_AGREE(comm, &v)`: consistent value
  - When returning success, $\forall p, v_p$ after the call $= \wedge_{p \in comm} v_p$
  - Processes dead before entering the call are considered to participate with value `false`

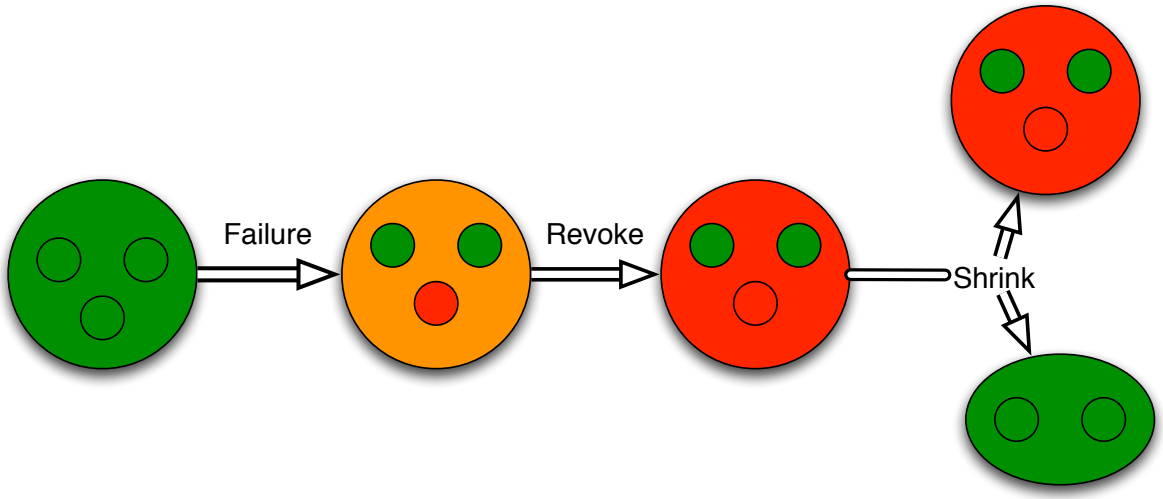
MPI-3.x-FT proposal



Additional MPI Calls – Agree

- `MPI_COMM_AGREE(comm, &v)`: consistent value
  - Processes dead inside the call participate either with their value or `false`
  - This call cannot return an error because of a process failure: must succeed despite process failures

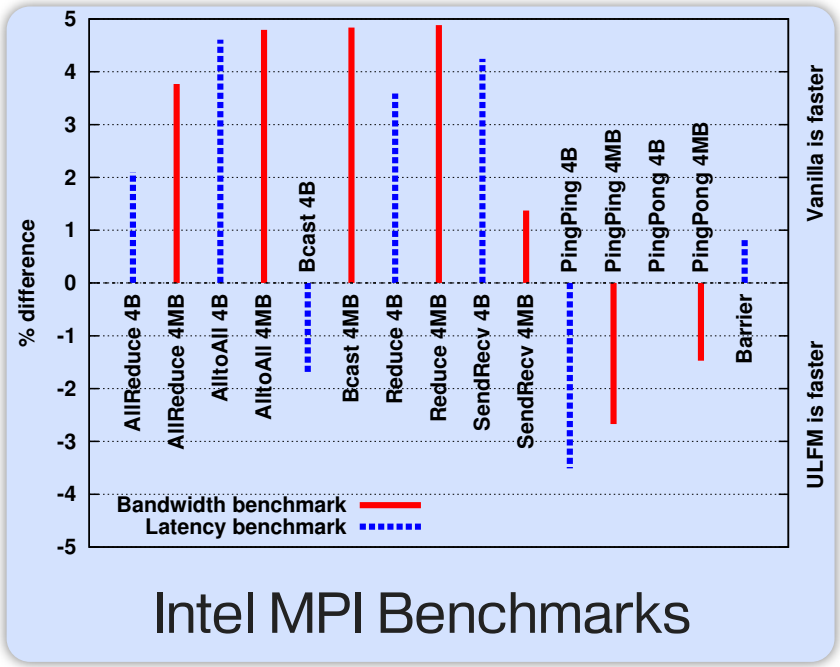
MPI-3.x-FT proposal



Additional MPI Calls – Repair

- What has been revoked, remains revoked
- Exhaustion of communicators → ways to create new communicators
- `MPI_COMM_SHRINK(&revoked_comm, &new_comm)`: creates a new communicator without dead processes
- Succeeds despite failures

MPI-3.x-FT proposal

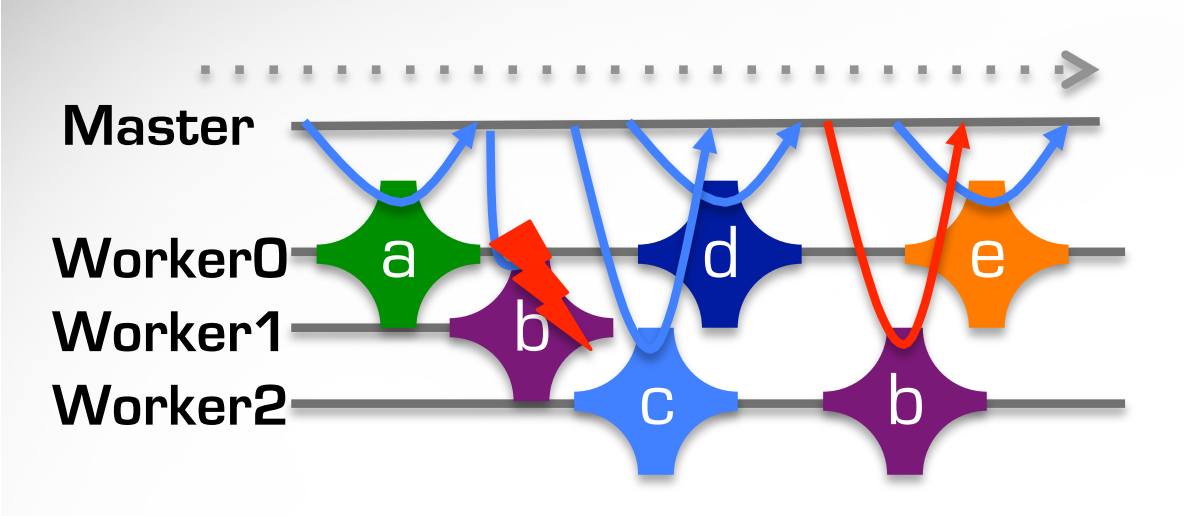


Open MPI - ULFM support

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

Outline

Master/Worker



Worker

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

Master/Worker

Master

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

FT Master

Fault Tolerant Master

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

FT Master

Fault Tolerant Master

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

FT Master

Fault Tolerant Master

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

FT Master

Fault Tolerant Master

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

FT Master

Fault Tolerant Master

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

Outline

Iterative Algorithm

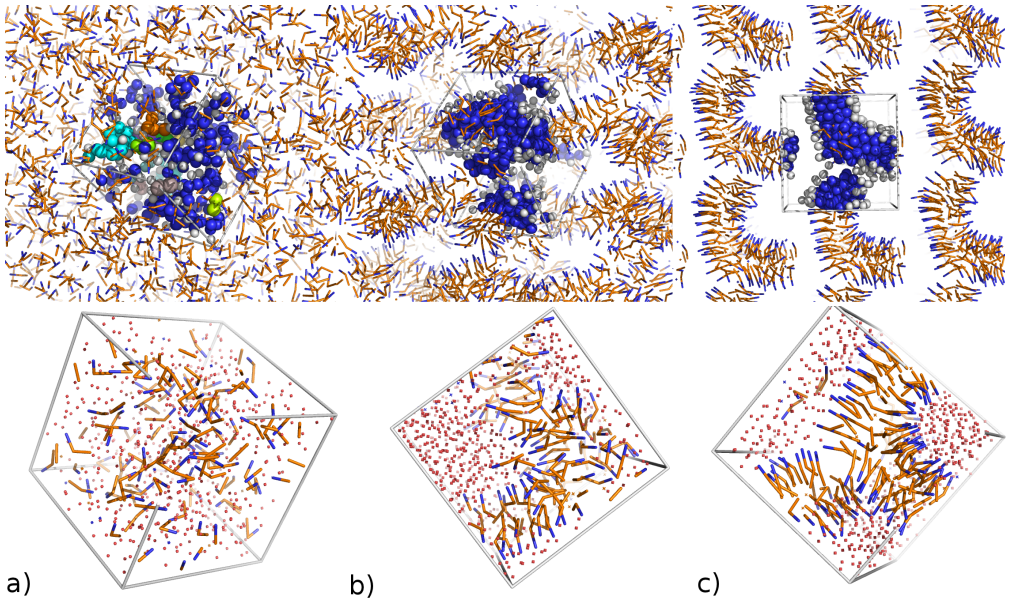
```
while( gnorm > epsilon ) {
     iterate();
     compute_norm(&lnorm);
     rc = MPI_Allreduce( &lnorm, &gnorm, 1,
                         MPI_DOUBLE, MPI_MAX, comm);
     if( (MPI_ERR_PROC_FAILED == rc) ||
         (MPI_ERR_COMM_REVOKED == rc) ||
         (gnorm <= epsilon) ) {
        allsuceeded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

Iterative Algorithm

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

Outline

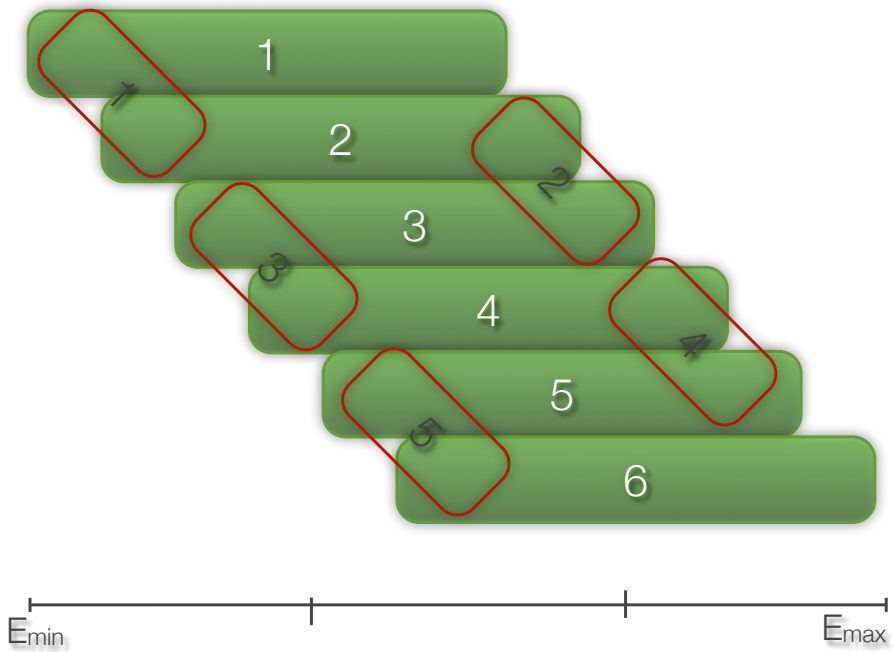
Domain Decomposition



Computational Chemistry Application

- Wang Landau Polymer Freezing and Collapse
- Developed at the UGA and UTK [T. Vogel, W. Bland]
- Simulating molecular interaction in skin cells
- Uses two types of communicators: within energy window, and between energy windows.

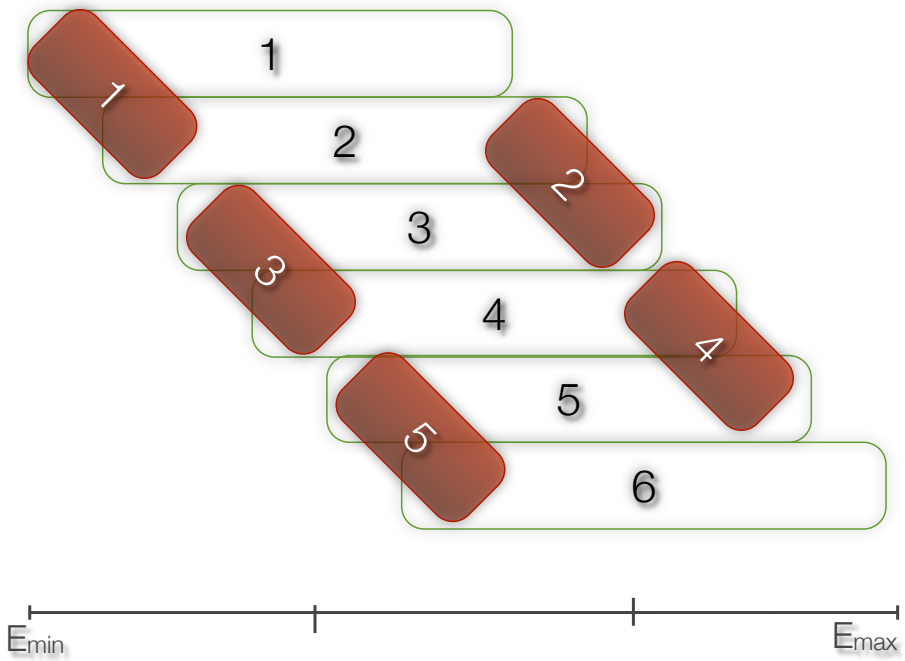
Domain Decomposition



Application Model

- Green Communicator:
  - Includes all processes in own energy window
  - 2x `MPI_Allreduce` per iteration

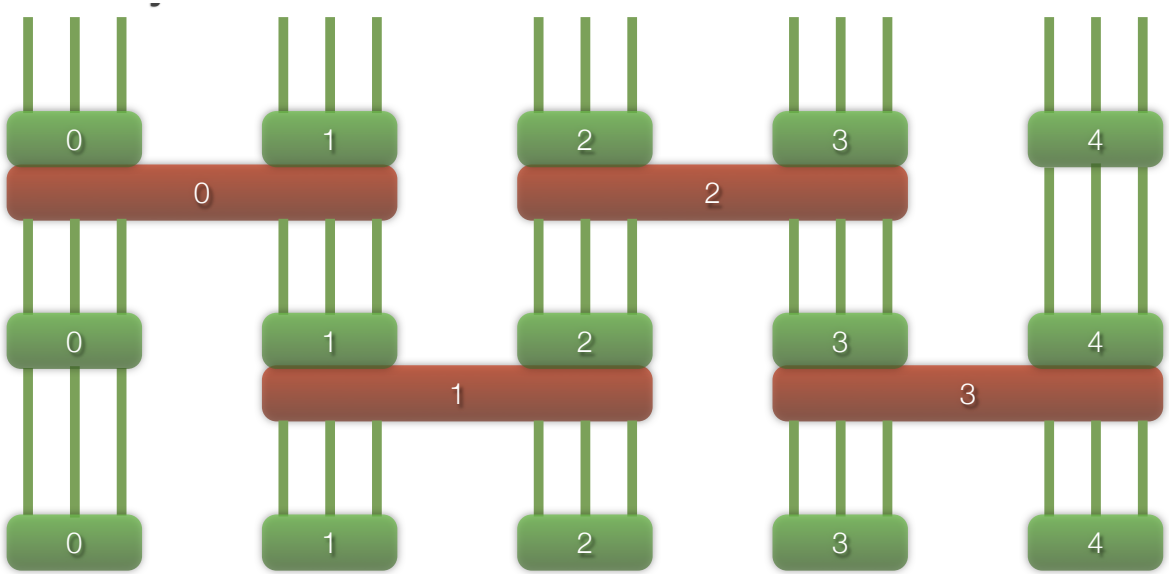
Domain Decomposition



Application Model

- Red Communicator:
  - Includes all processes in own and neighboring energy window
  - 1x `MPI_Scatter` per iteration
  - 2x `MPI_Send_recv_replace` per iteration
  - Many `MPI_Send` and `MPI_Recv` per iteration

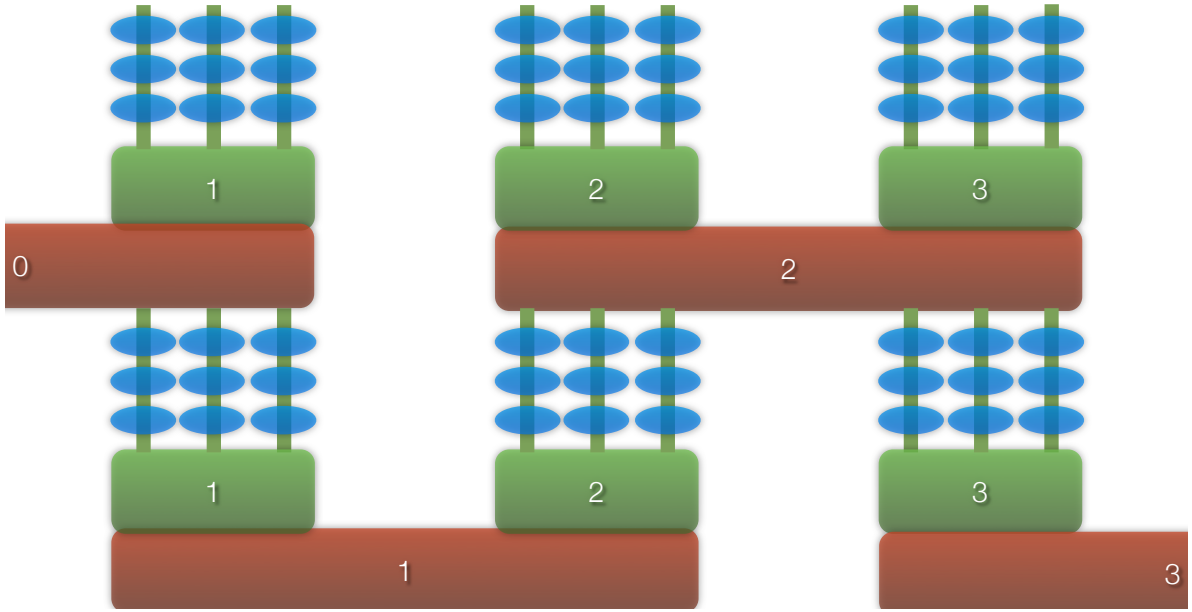
Domain Decomposition



Fault Tolerance in Wang Landau

Long Independent Computation on each Processor
Periodic AllReduce on the communicator of the energy window
Immediately after, Scatter, and many p2p on neighboring energy windows

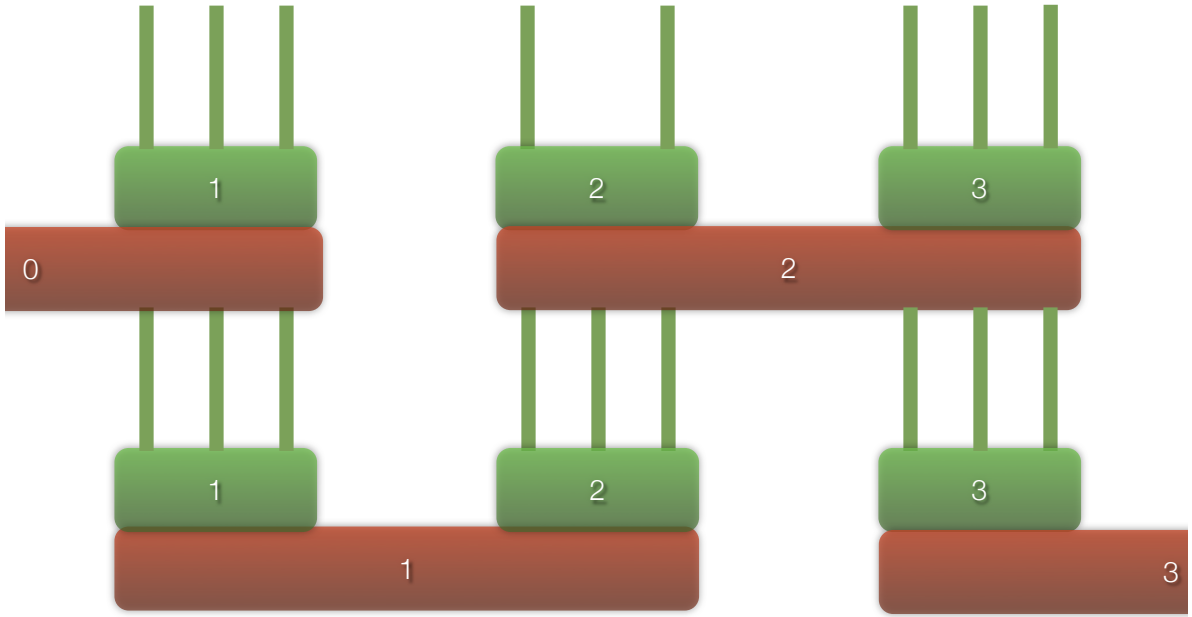
Domain Decomposition



Fault Tolerance in Wang Landau

Dataset protected by small cheap application-specific checkpoints (stored on neighbors)

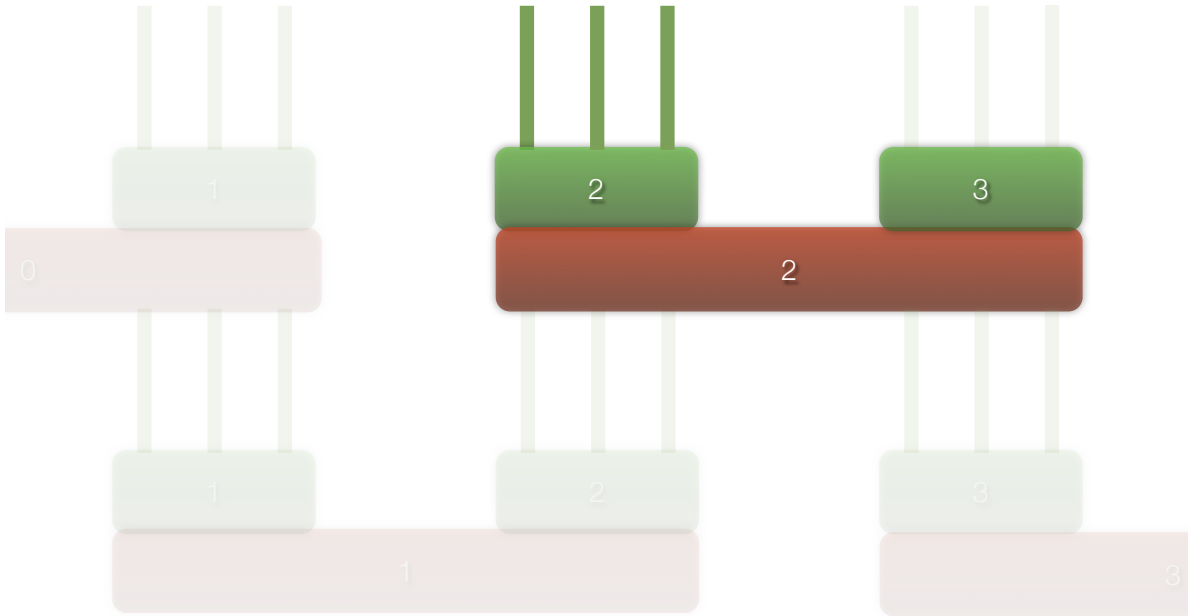
Domain Decomposition



Fault Tolerance in Wang Landau

Revoke; Shrink; Spawn; Merge; Reorder;
load checkpoint

Domain Decomposition

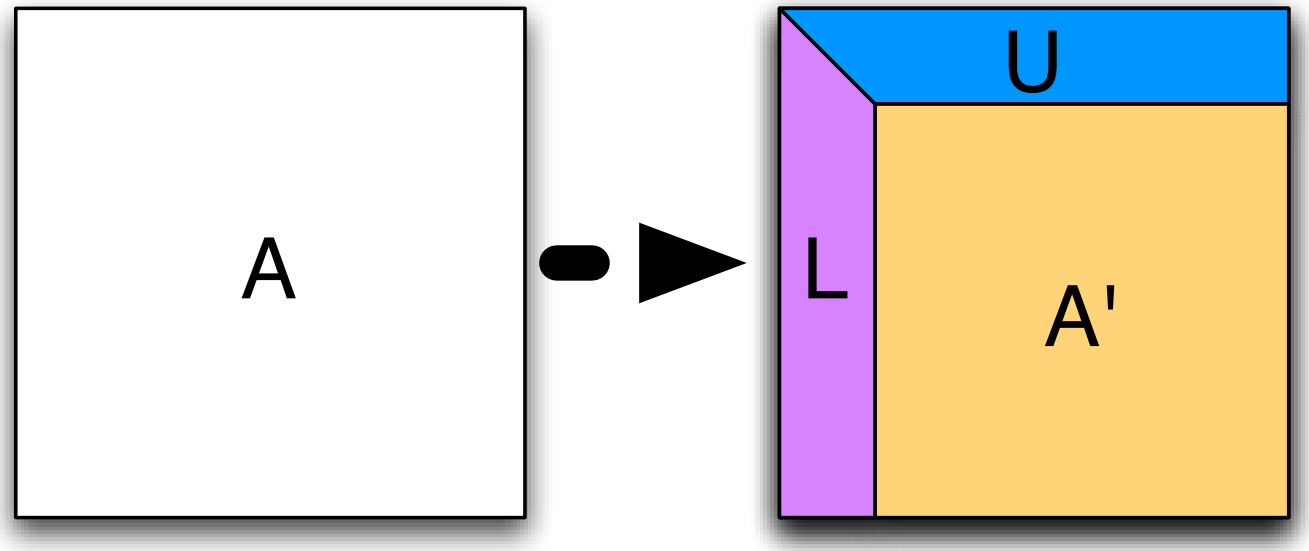


Fault Tolerance in Wang Landau

Revoke; Free; Connect; Accept; Merge; Reorder

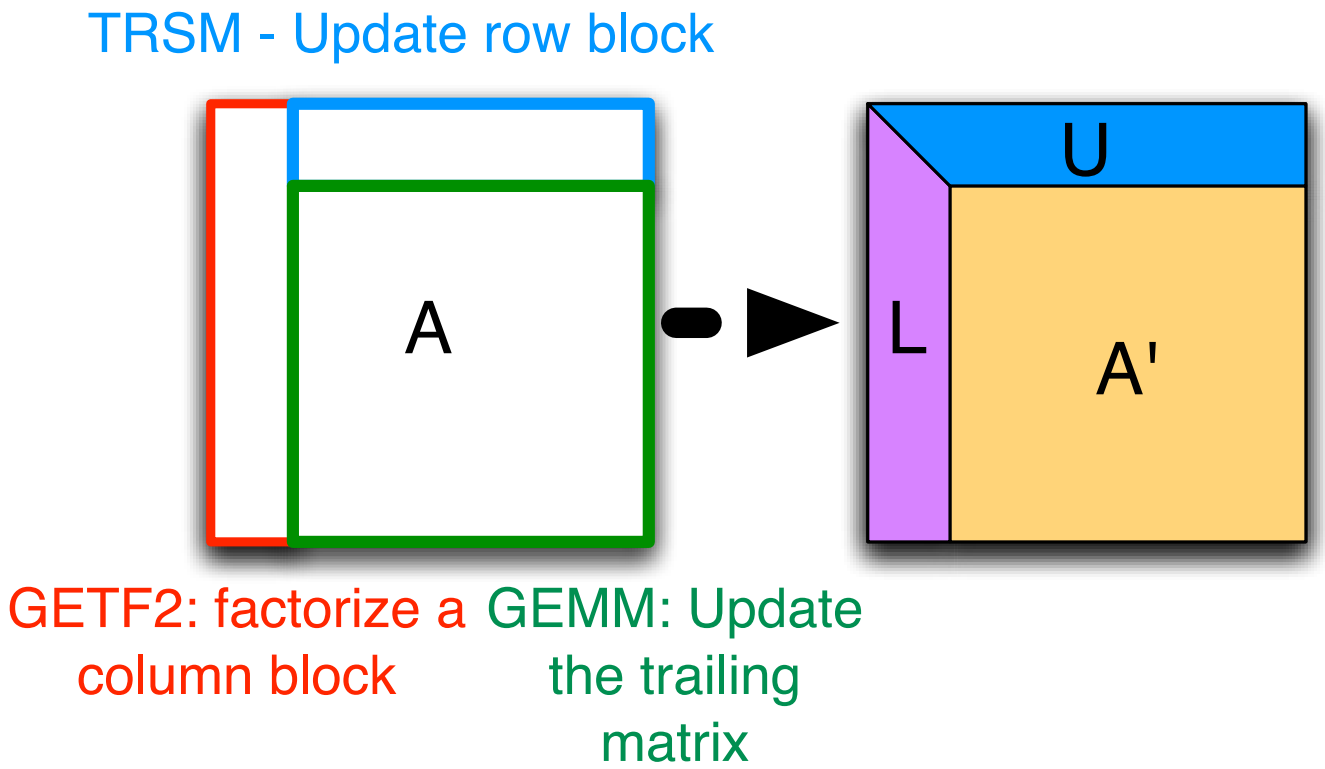
Outline

Example: recursive block LU/QR factorization



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

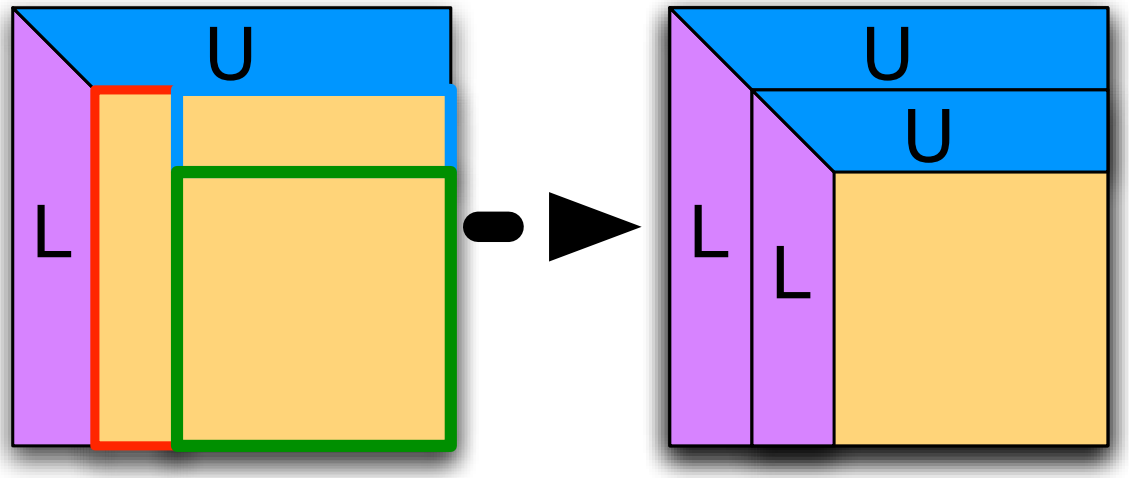
Example: recursive block LU/QR factorization



- 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: *recursive* block LU/QR factorization

TRSM - Update row block

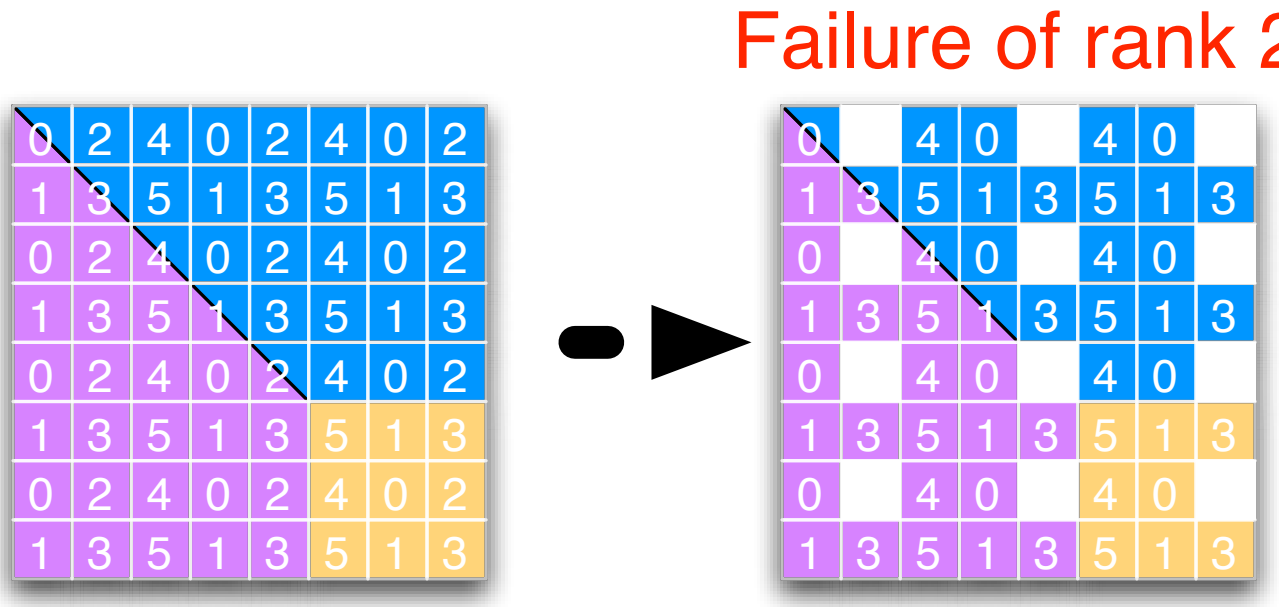


GETF2: factorize a column block

GEMM: Update the trailing matrix

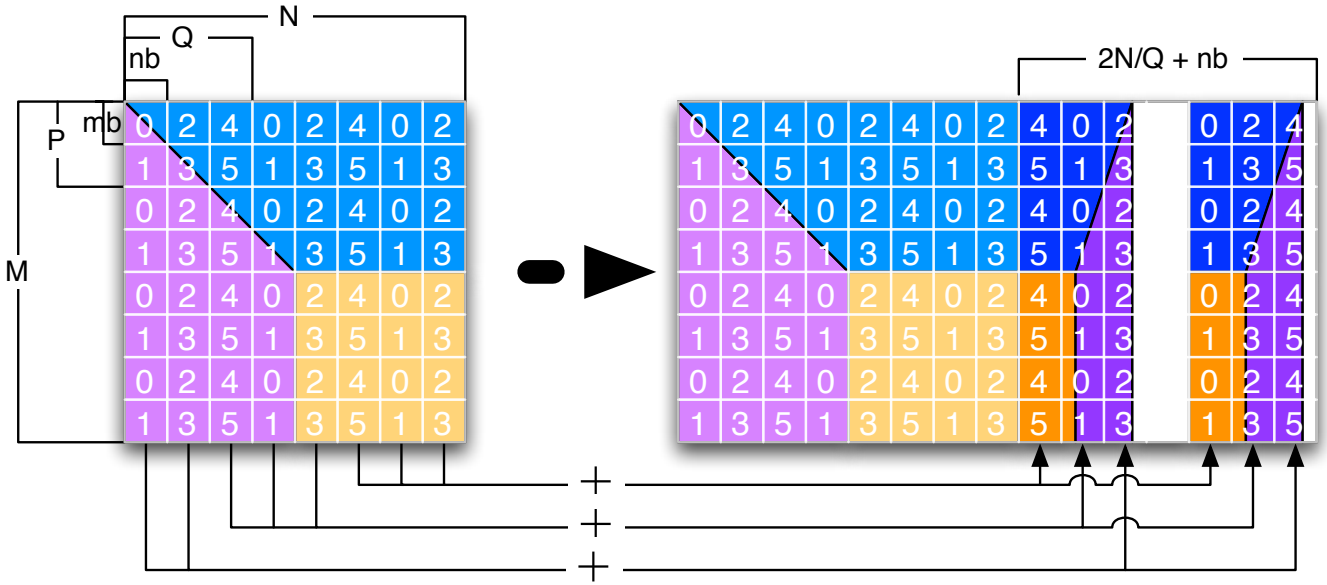
- Solve $A \cdot x = b$ (hard)
- Transform A into a LU factorization
- Solve $L \cdot y = B \cdot b$, then $U \cdot x = y$

Example: *recursive* block LU/QR factorization



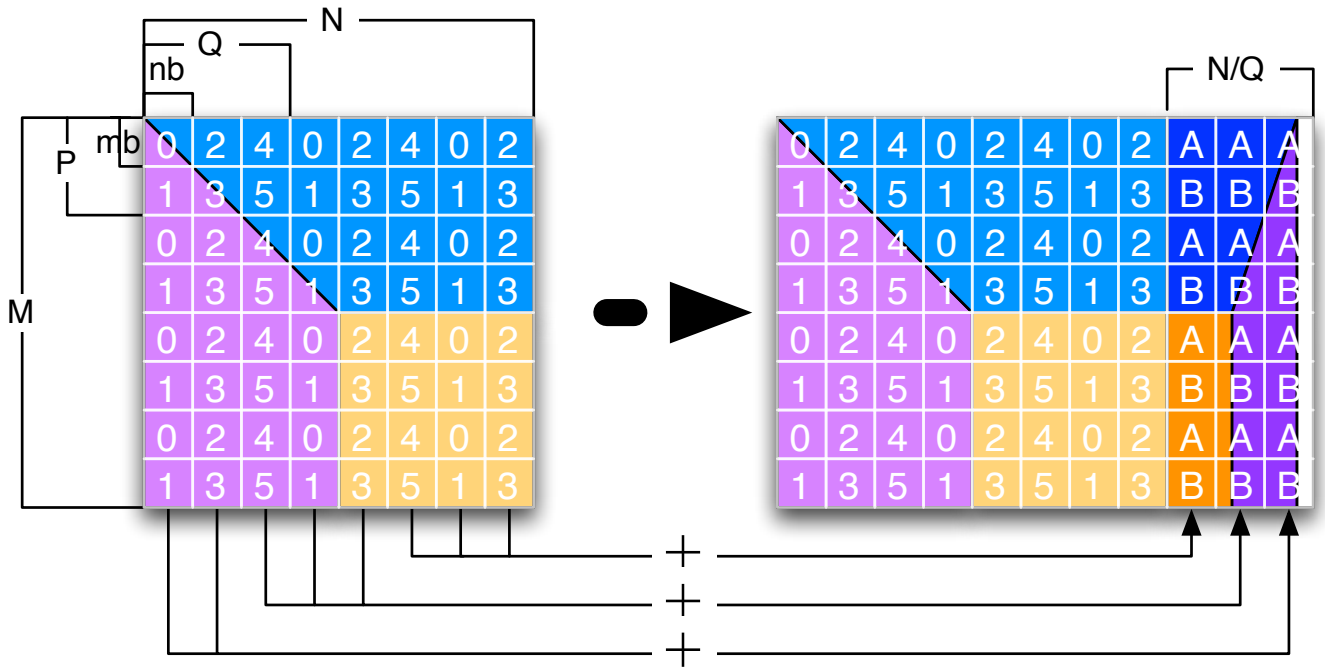
- 2D Block Cyclic Distribution (here 2×3)
- A single failure $\Rightarrow$ many data lost

Algorithm Based Fault Tolerant LU decomposition



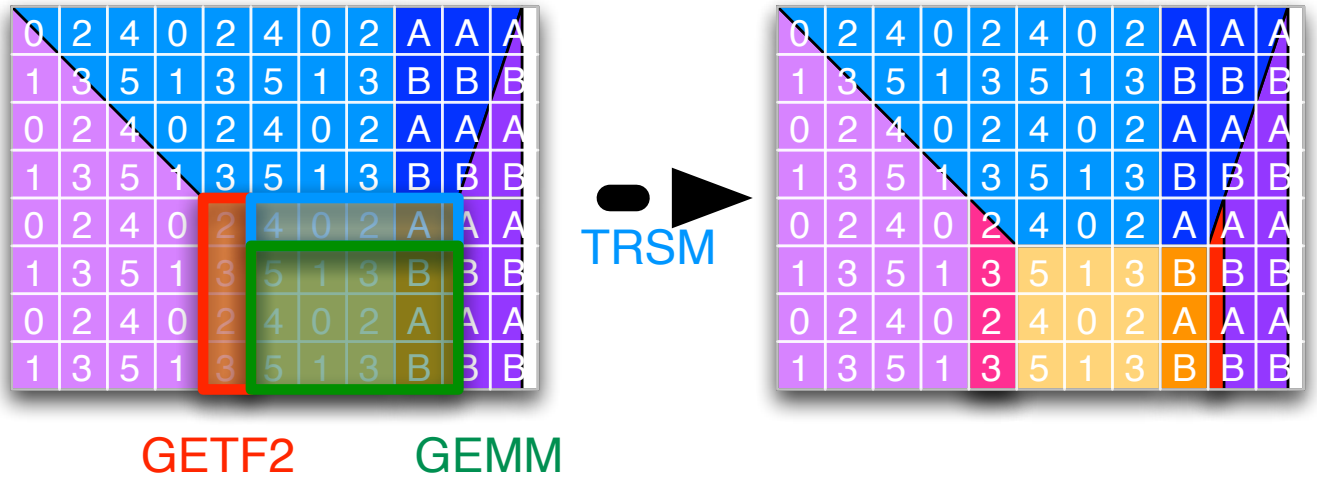
- Checksum: invertible operation on the data of the row / column
  - Checksum blocks are doubled, to allow recovery when data and checksum are lost together

Algorithm Based Fault Tolerant LU decomposition



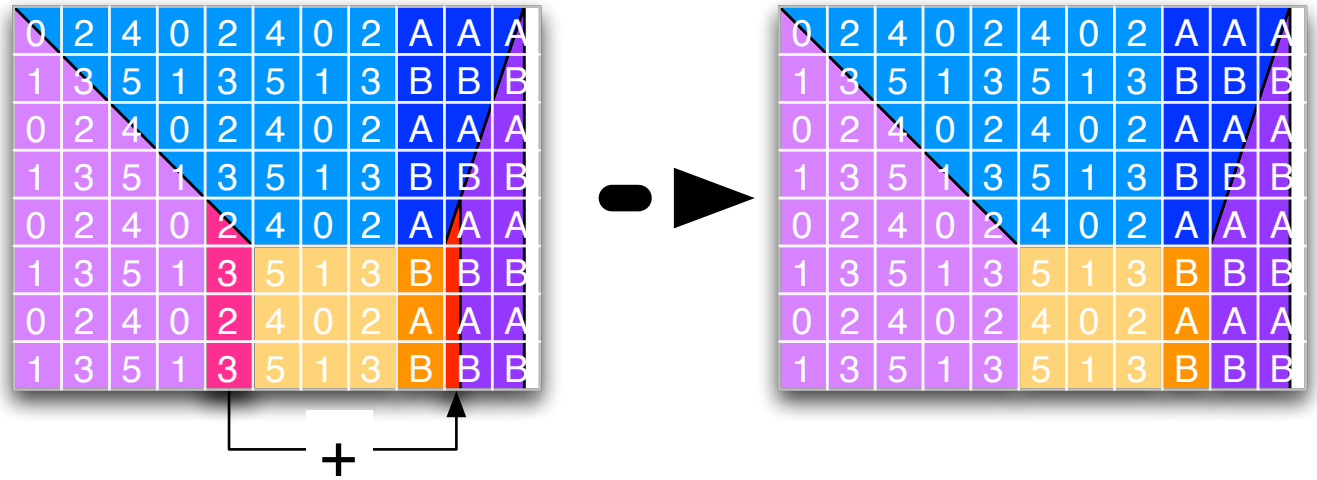
- Checksum: invertible operation on the data of the row / column
  - Checksum replication can be avoided by dedicating computing resources to checksum storage

Algorithm Based Fault Tolerant LU decomposition



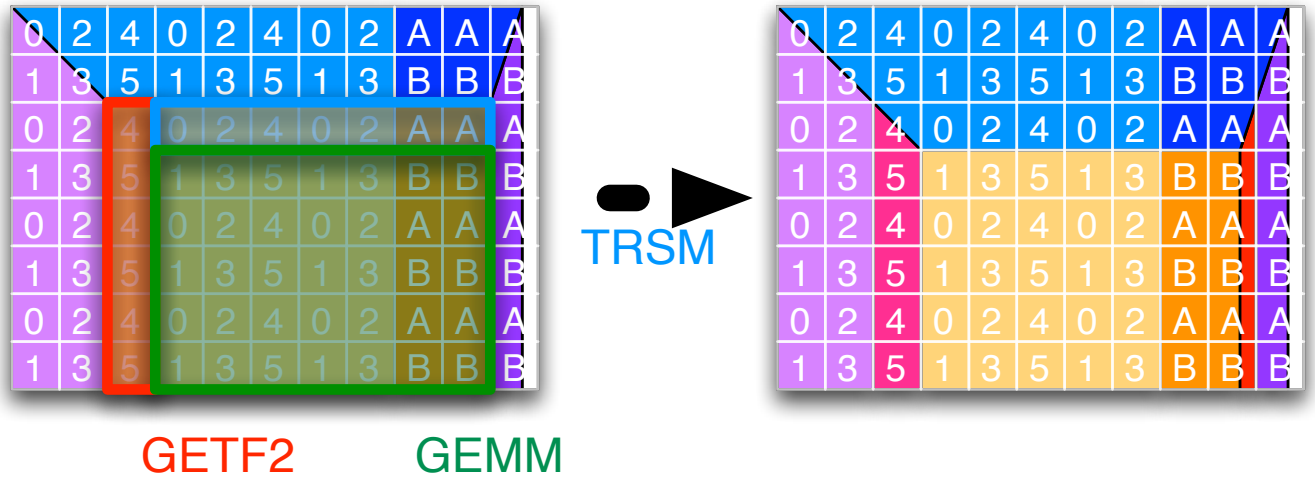
- Checksum: invertible operation on the data of the row / column
  - Idea of ABFT: applying the operation on data and checksum preserves the checksum properties

Algorithm Based Fault Tolerant LU decomposition



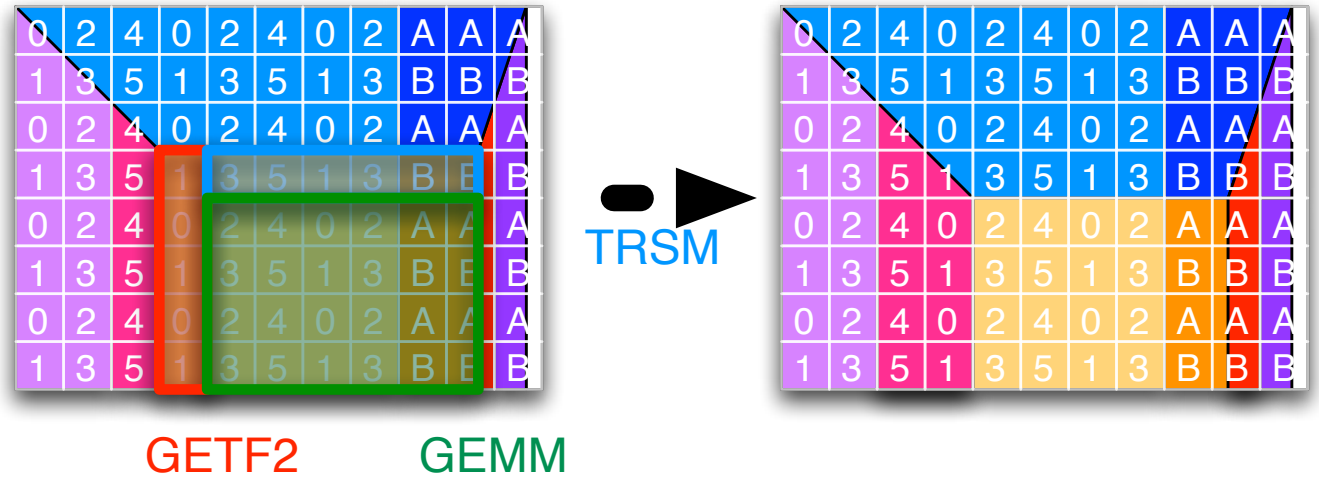
- Checksum: invertible operation on the data of the row / column
  - For the part of the data that is not updated this way, the checksum must be re-calculated

Algorithm Based Fault Tolerant LU decomposition



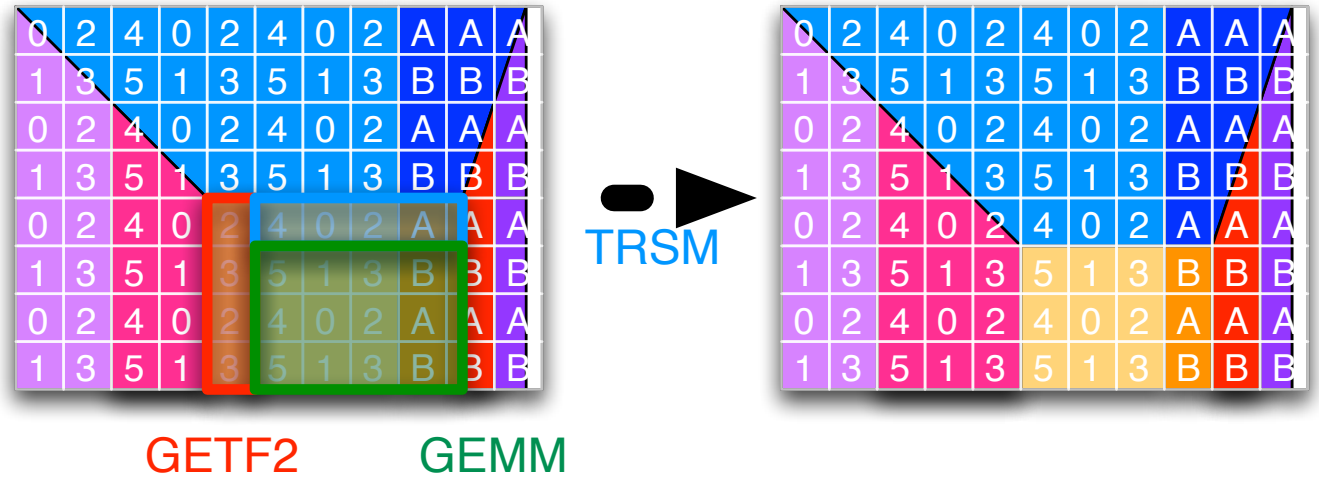
- Checksum: invertible operation on the data of the row / column
  - To avoid slowing down all processors and panel operation, group checksum updates every q block columns

Algorithm Based Fault Tolerant LU decomposition



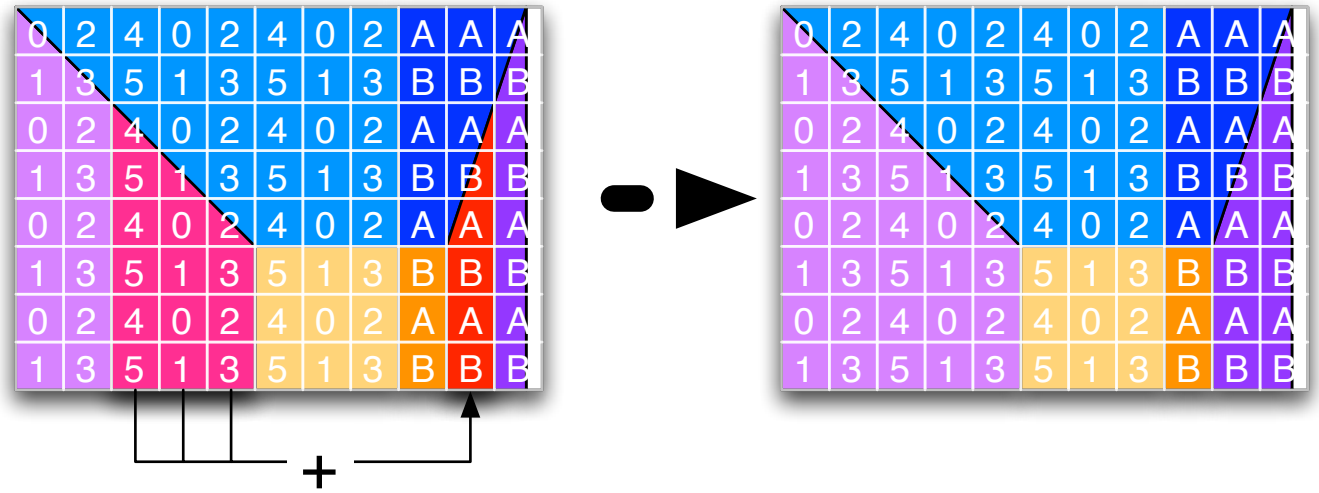
- Checksum: invertible operation on the data of the row / column
  - To avoid slowing down all processors and panel operation, group checksum updates every q block columns

Algorithm Based Fault Tolerant LU decomposition



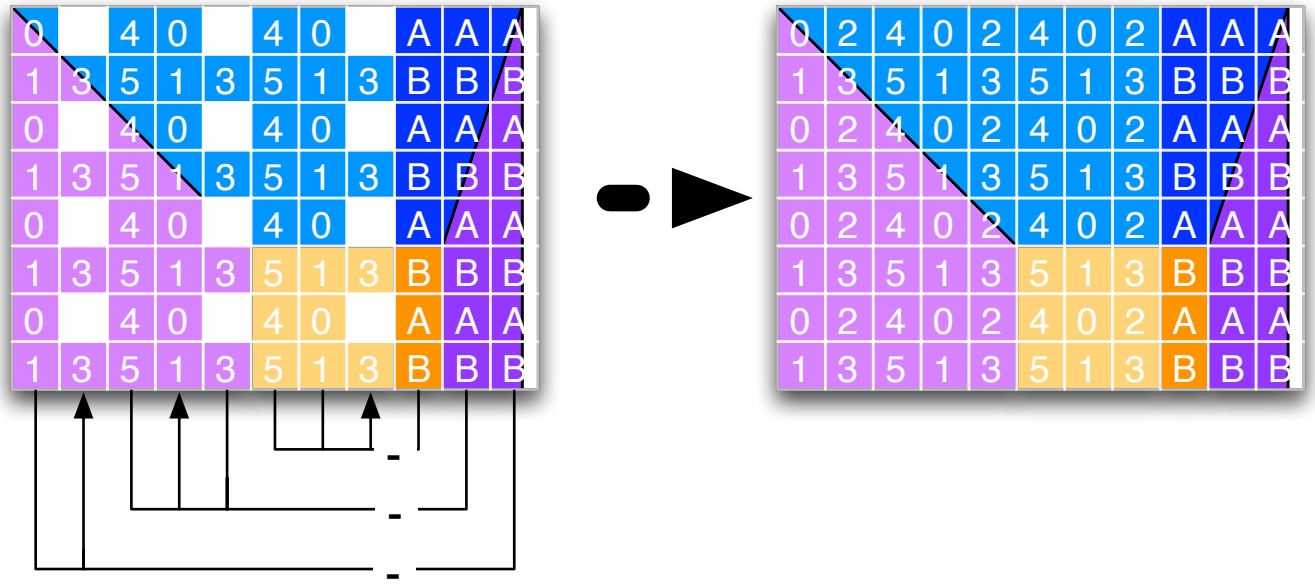
- Checksum: invertible operation on the data of the row / column
  - To avoid slowing down all processors and panel operation, group checksum updates every q block columns

Algorithm Based Fault Tolerant LU decomposition



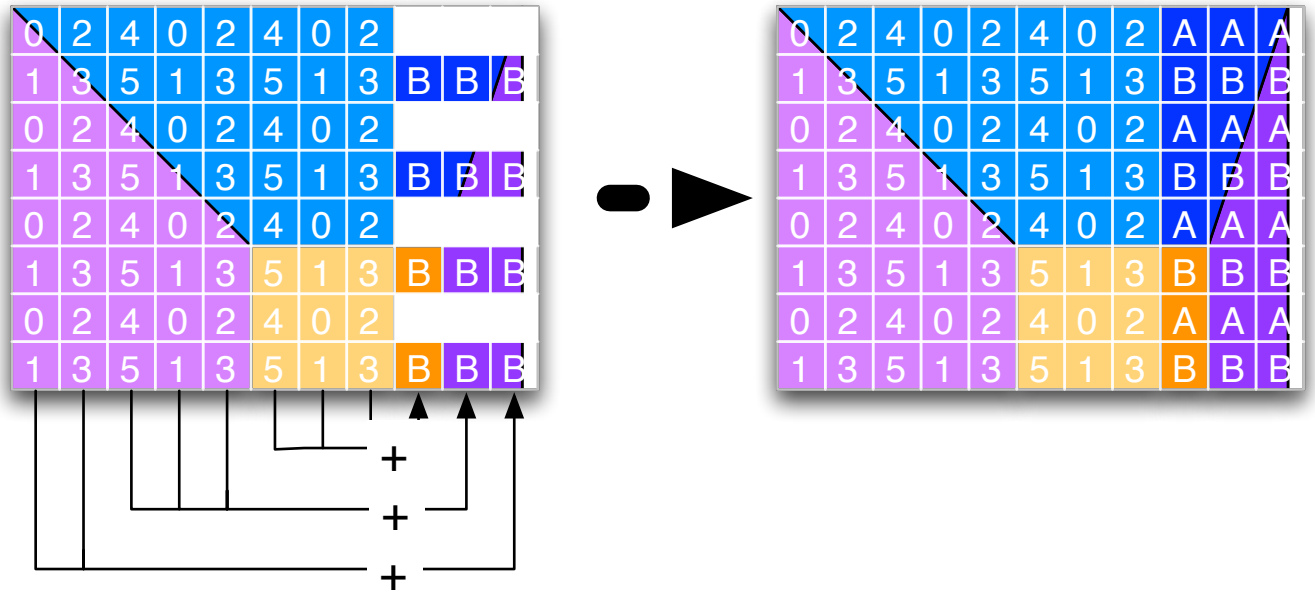
- Checksum: invertible operation on the data of the row / column
  - Then, update the missing coverage. Keep checkpoint block column to cover failures during that time

Algorithm Based Fault Tolerant LU decomposition



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

Algorithm Based Fault Tolerant LU decomposition



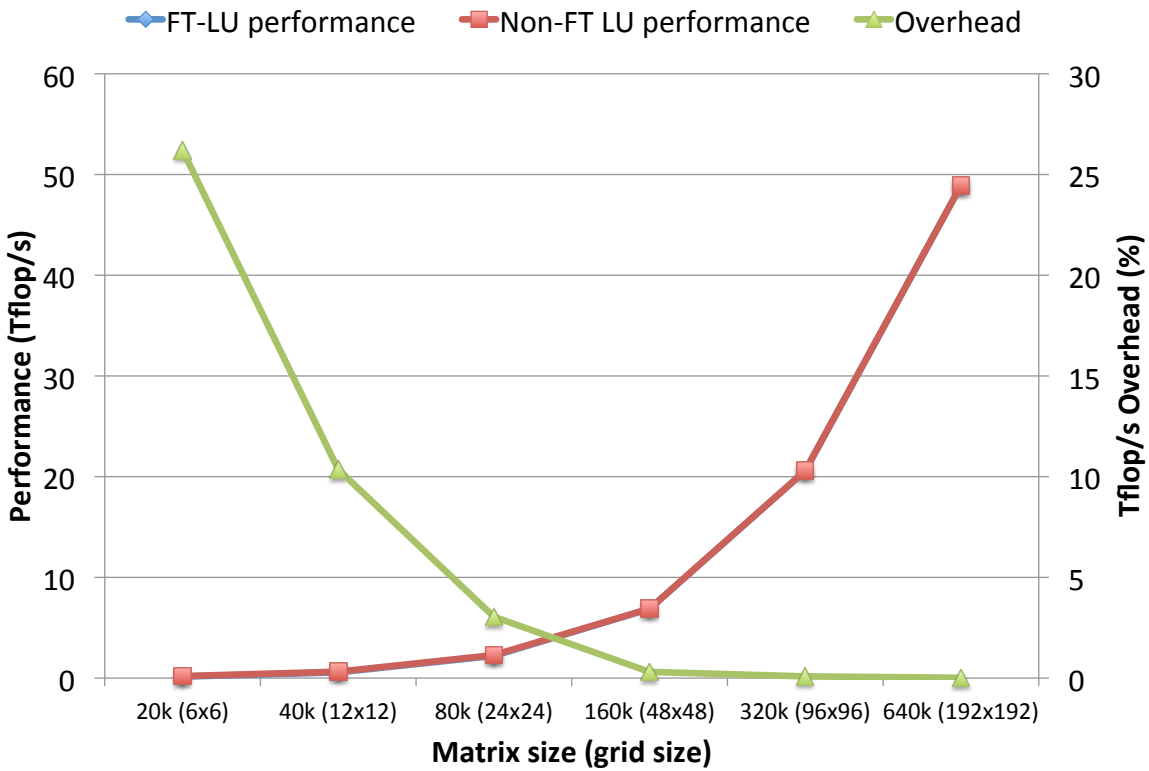
- In case of failure, conclude the operation, then
  - Missing Checksum = Sum(Existing Data)s

ABFT LU decomposition: implementation

MPI Implementation

- PBLAS-based: need to provide "Fault-Aware" version of the library
- Cannot enter recovery state at any point in time: need to complete ongoing operations despite failures
- Need to test the return code of each and every MPI-related call: code explosion

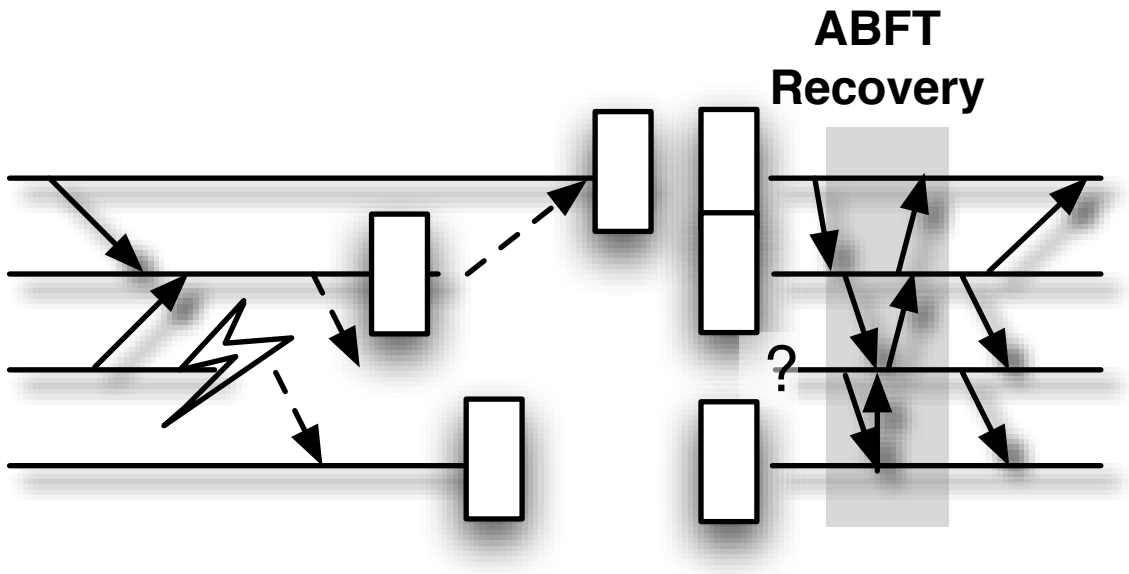
ABFT LU decomposition: performance



MPI-3.x ULFM Performance

- Open MPI with ULFM; Kraken supercomputer;

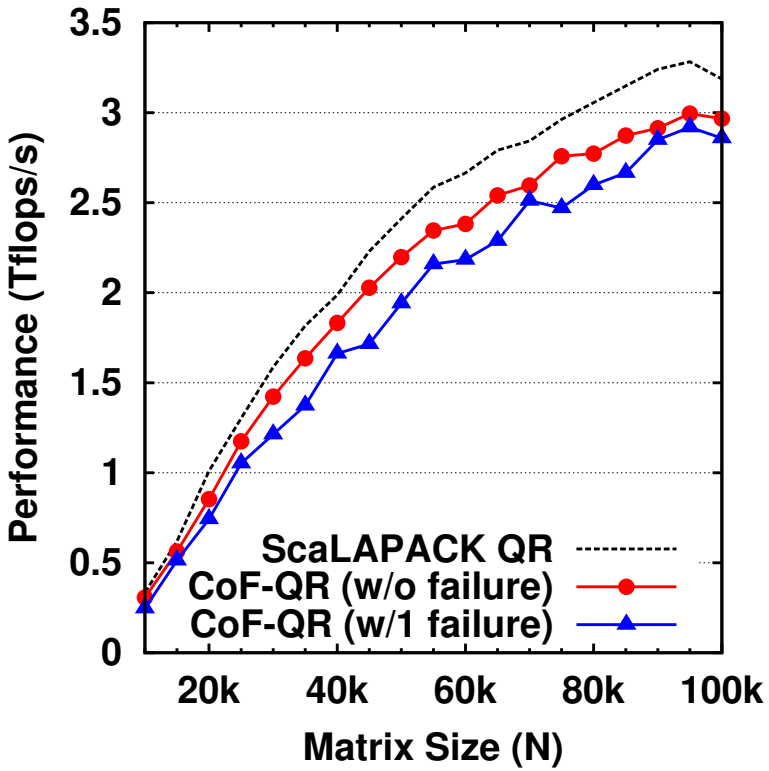
ABFT LU decomposition: implementation



Checkpoint on Failure - MPI Implementation

- FT-MPI / MPI-3.x FT: not easily available on large machines
- Checkpoint on Failure = workaround

ABFT QR decomposition: performance



Checkpoint on Failure - MPI Performance

- Open MPI; Kraken supercomputer;

Outline

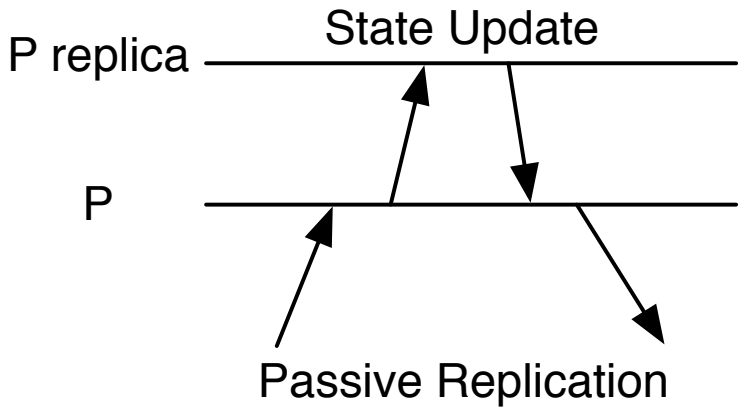
Maintaining Redundant Information

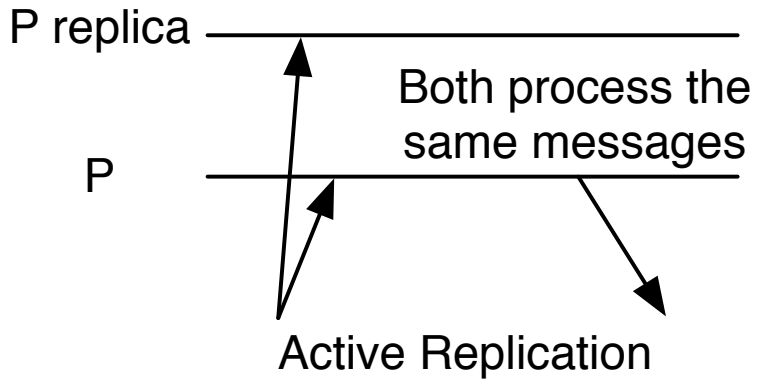
Goal

- General Purpose Fault Tolerance Techniques: work despite the application behavior
- Two adversaries: Failures & Application
- Use automatically computed redundant information
  - At given instants: checkpoints
  - At any instant: replication
  - Or anything in between: checkpoint + message logging

Outline

Replication

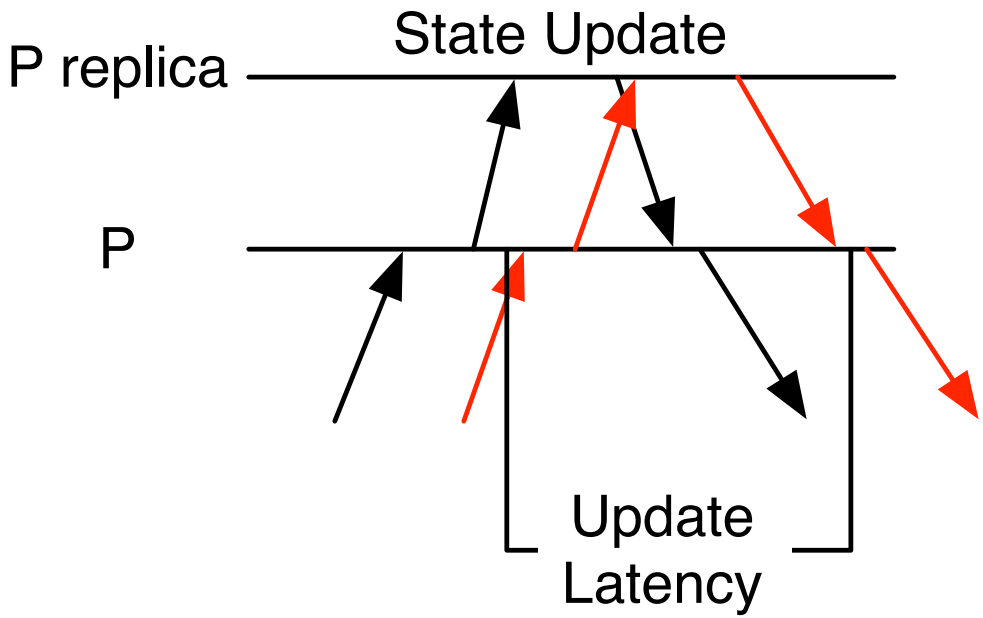




Idea

- Each process is replicated on a resource that has small chance to be hit by the same failure as its replica
- In case of failure, one of the replicas will continue working, while the other recovers
- Passive Replication / Active Replication

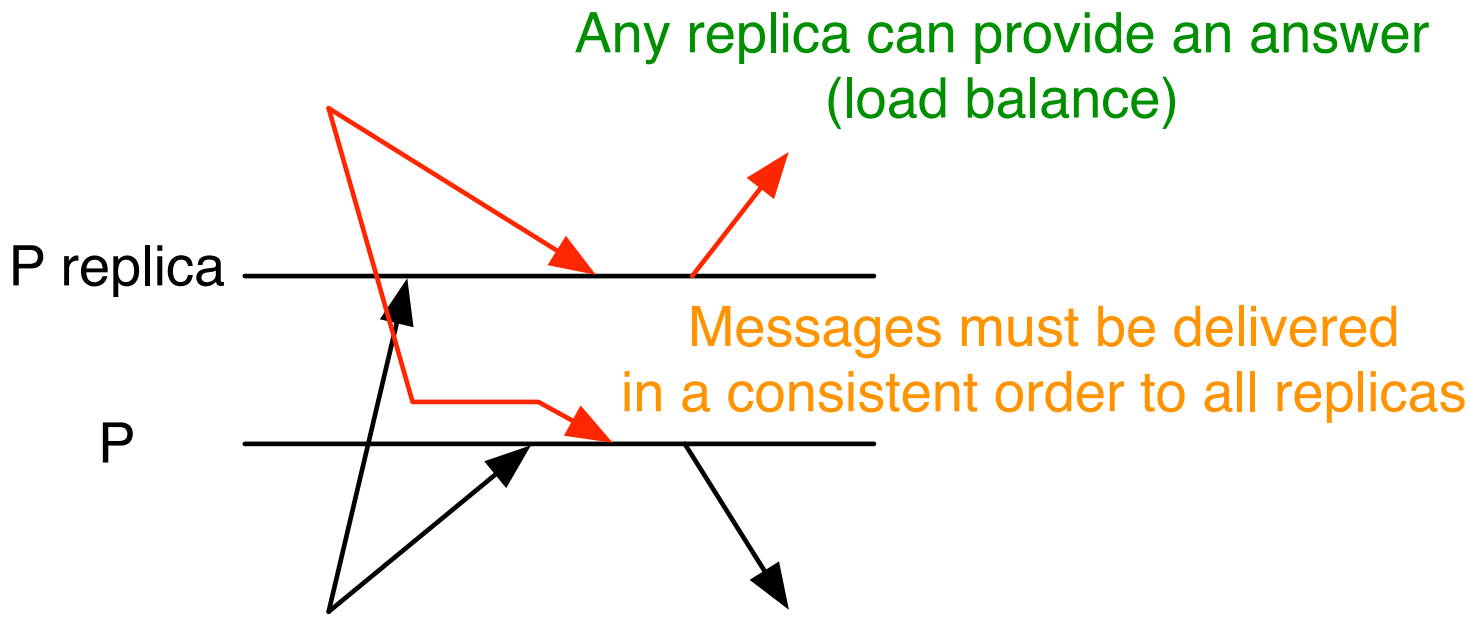
Replication



Challenges

- Passive replication: latency of state update
- Active replication: ordering of decision → internal additional communications
- By nature: replication → at most 50% machine efficiency

Replication



Challenges

- Passive replication: latency of state update
- Active replication: ordering of decision → internal additional communications
- By nature: replication → at most 50% machine efficiency

Outline

Process Checkpointing

Goal

- Save the current state of the *process*
  - Protocols save the current state of the *application*

Techniques

- User-level checkpointing
- System-level checkpointing
- Blocking call
- Asynchronous call

User-level checkpointing

User code serializes the state of the application in a file.

- 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 the size of memory and in the size of modified files

Asynchronous Checkpointing

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

User-level approach: critical sections, when needed

Storage

Remote Reliable Storage

Intuitive. I/O intensive. Disk usage.

Memory Hierarchy

- local memory
- local disk (SSD, HDD)
- remote disk

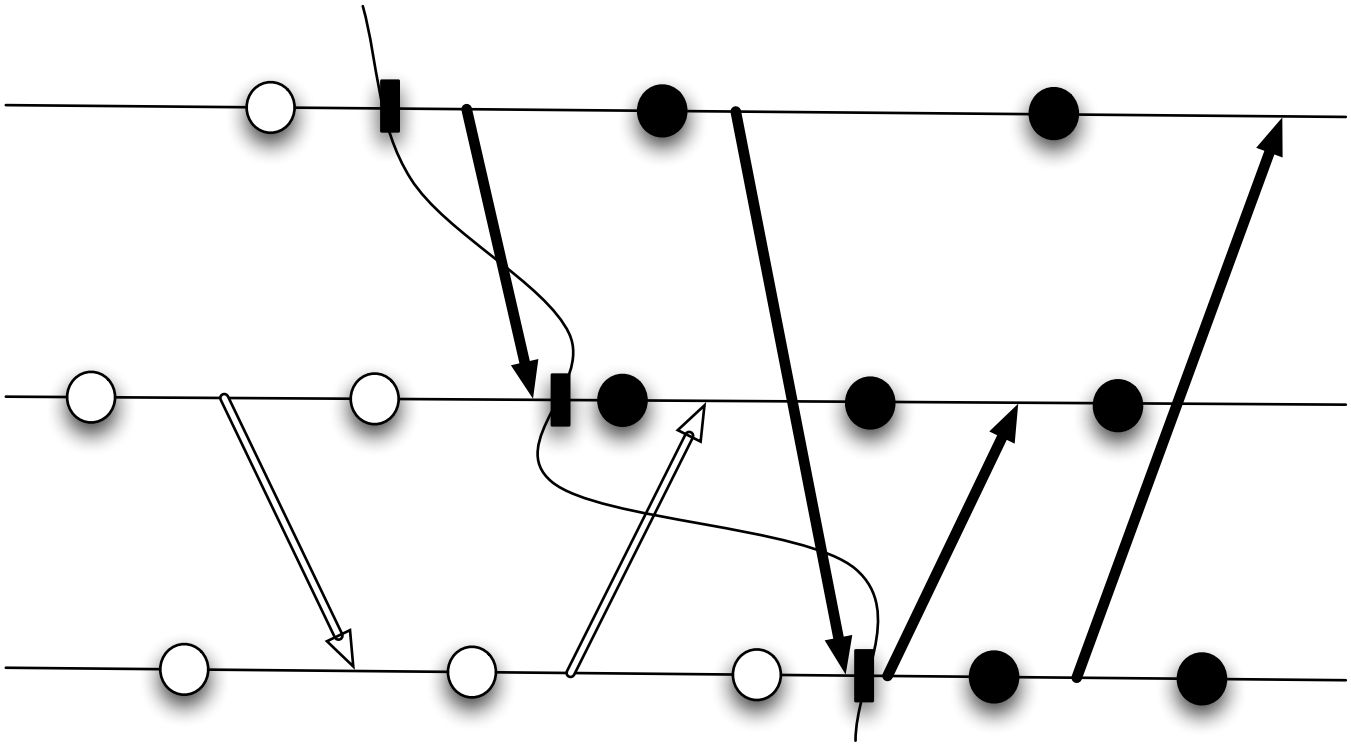
Checkpoint is valid when finished on reliable storage

Distributed Memory Storage

- In-memory checkpointing
- Disk-less checkpointing

Outline

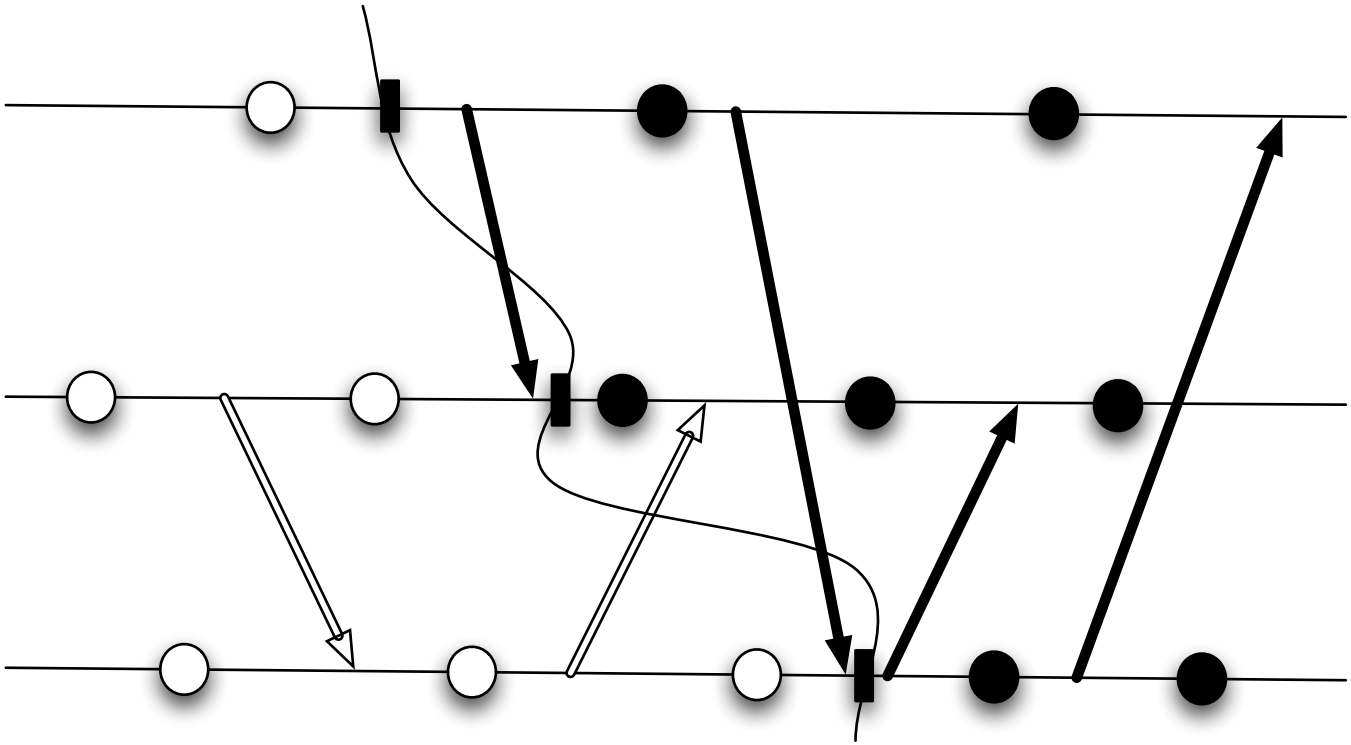
Coordinated Checkpointing Idea



Definition (Orphan Message)

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

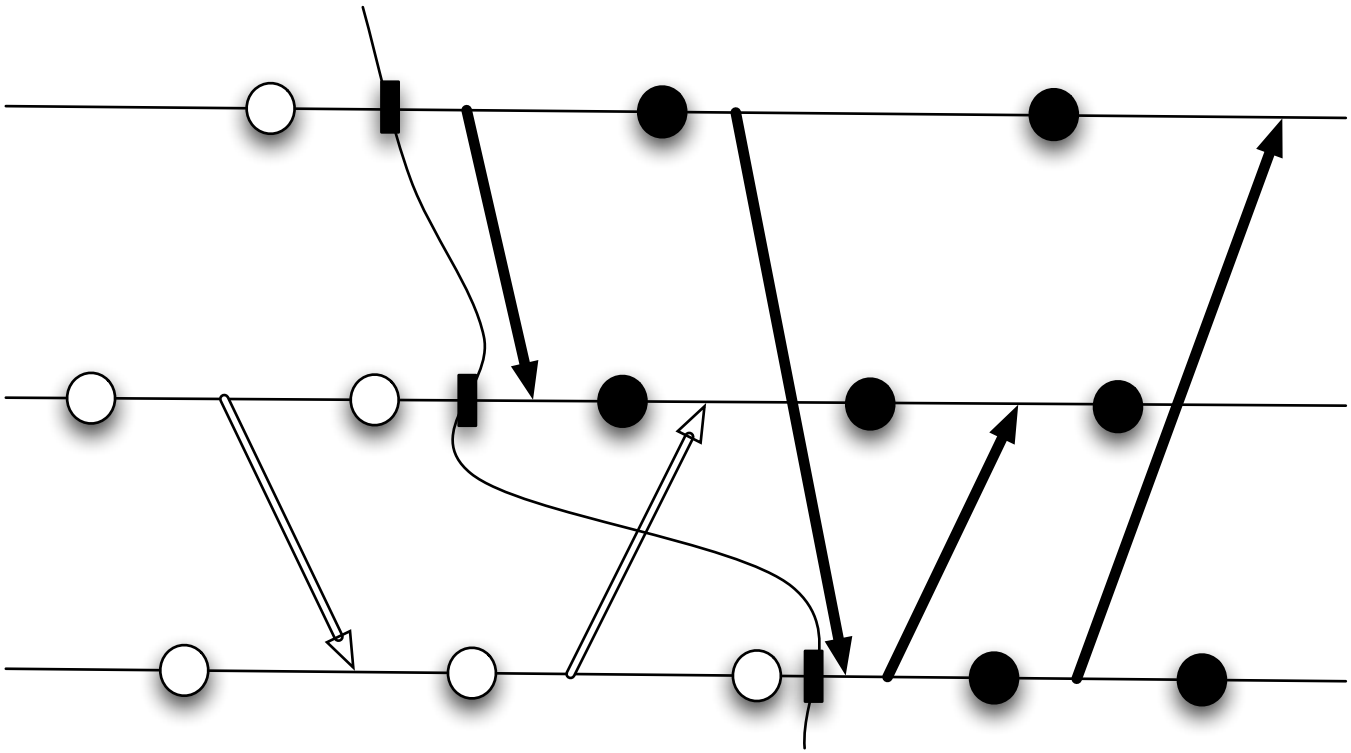
Coordinated Checkpointing Idea



Definition (Missing Message)

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

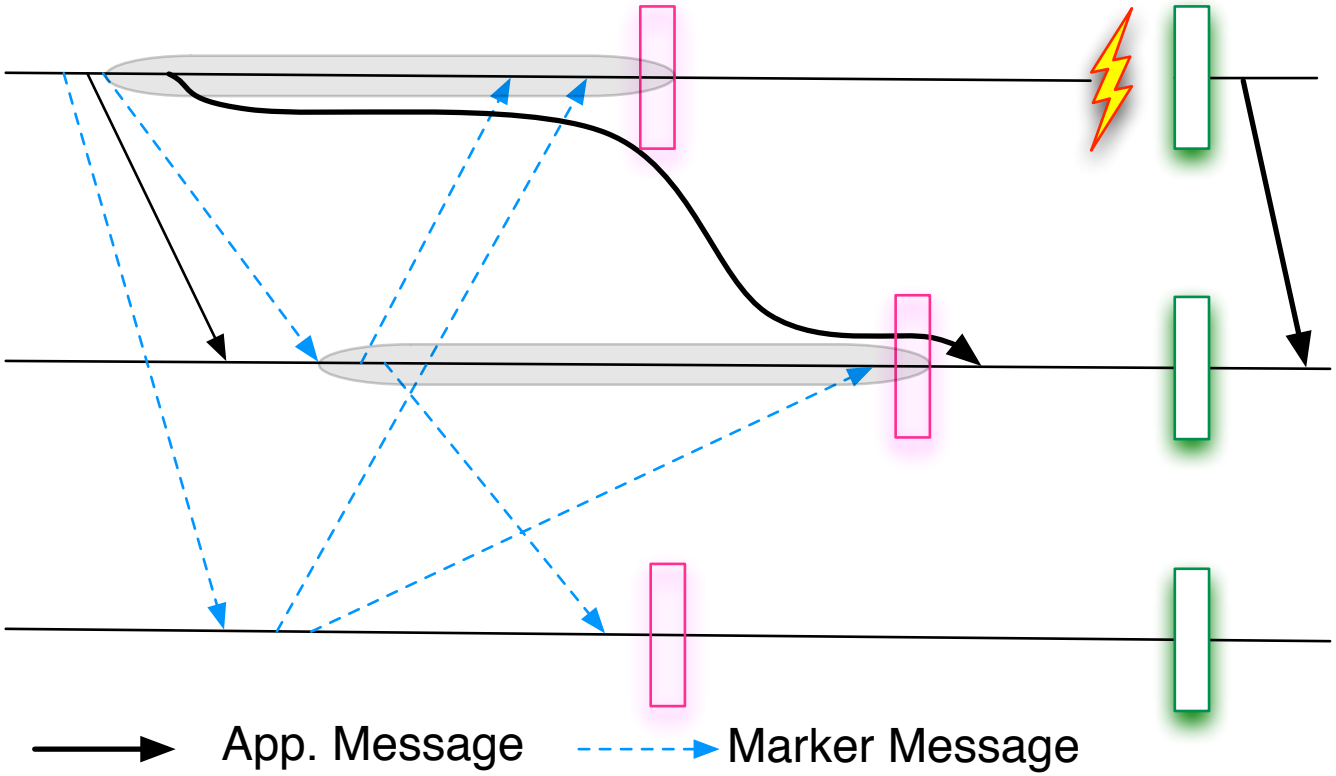
Coordinated Checkpointing Idea



Create a consistent view of the application

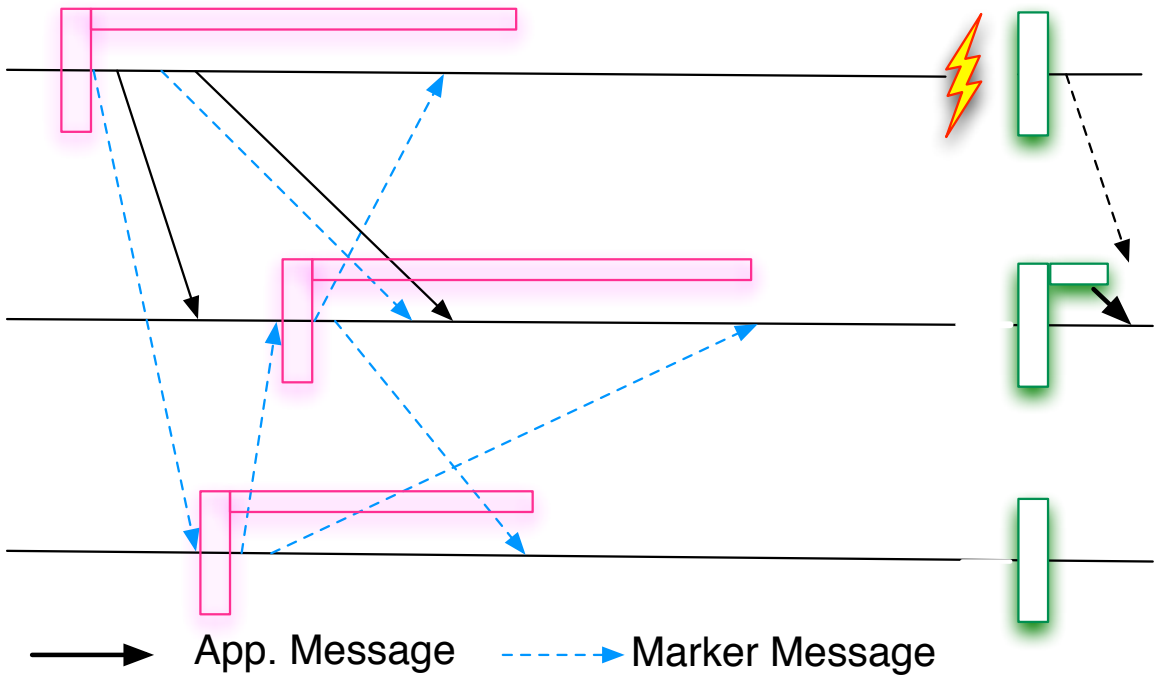
- Messages belong to a checkpoint wave or another
- All communication channels must be flushed (all2all)

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

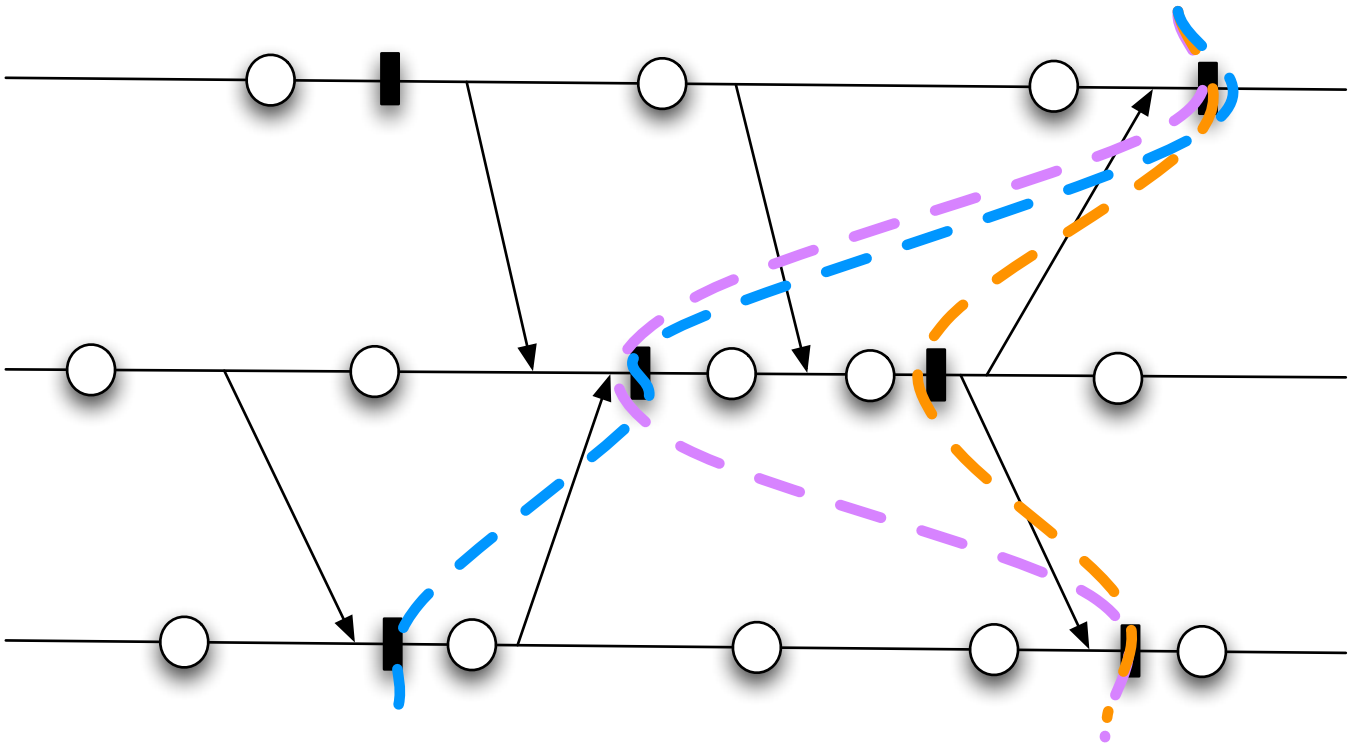
- Flush of communication channels
  - conservative approach. One Message per open channel / One message per channel
- Preemptive checkpointing usually required
  - Can have a user-level checkpointing, but requires one that be called any time

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

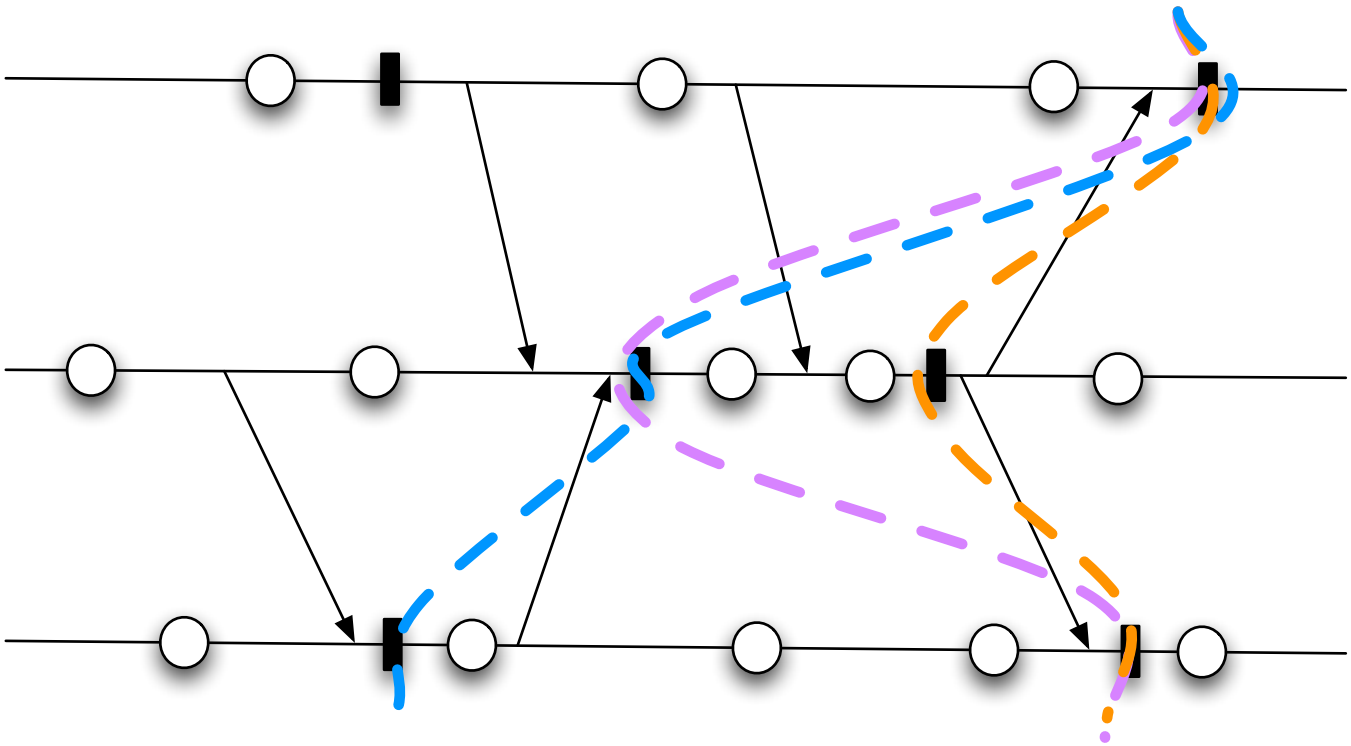
Outline

Uncoordinated Checkpointing Idea



Processes checkpoint independently

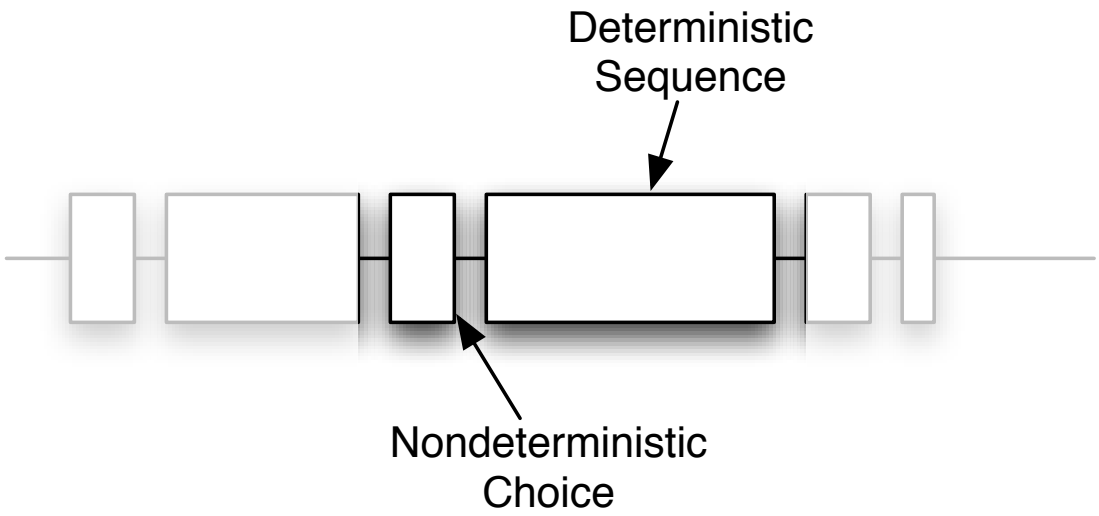
Uncoordinated Checkpointing Idea



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

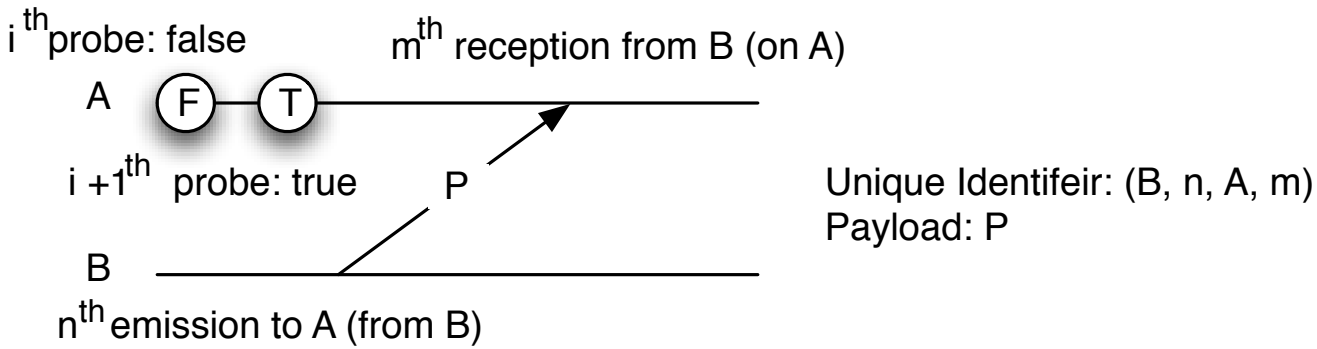
Piece-wise Deterministic Assumption



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

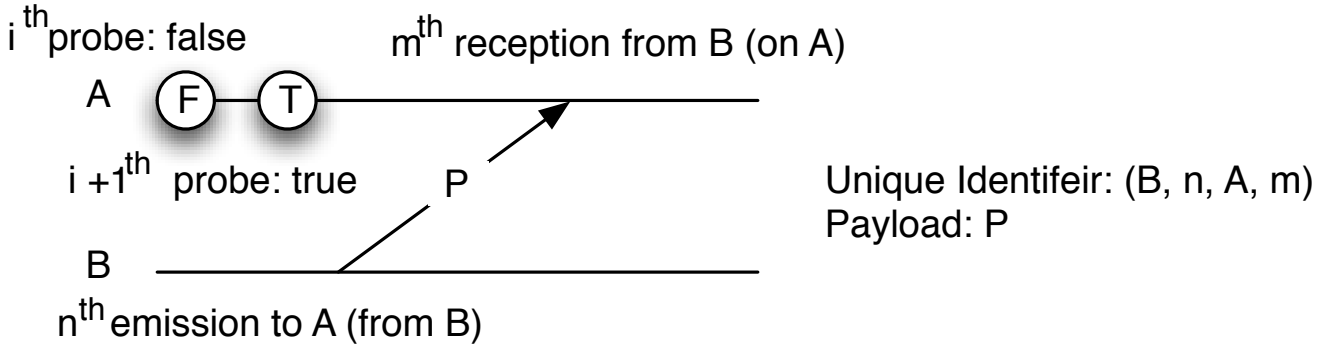
Message Logging



Message Logging

By replaying the sequence of messages and test/probe with the same result that it obtained in the initial execution (from the last checkpoint), one can guide the execution of a process to its exact state just before the failure

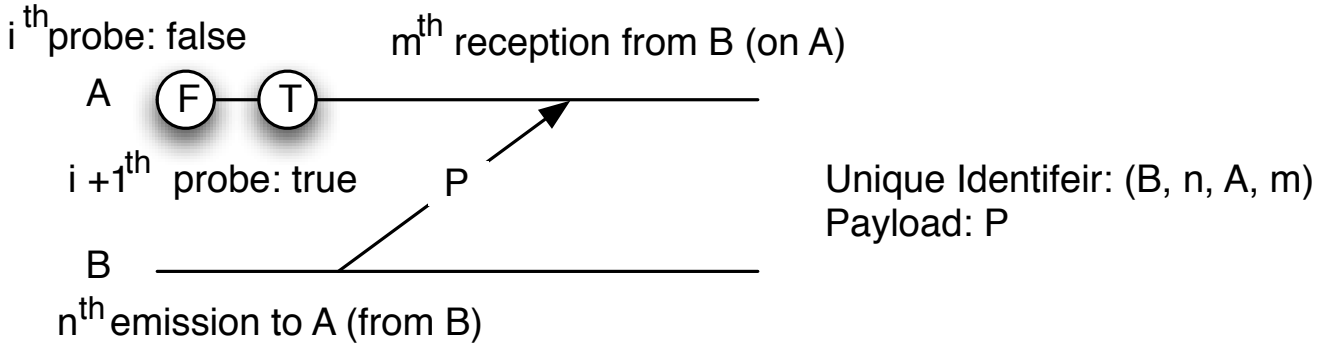
Message Logging



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

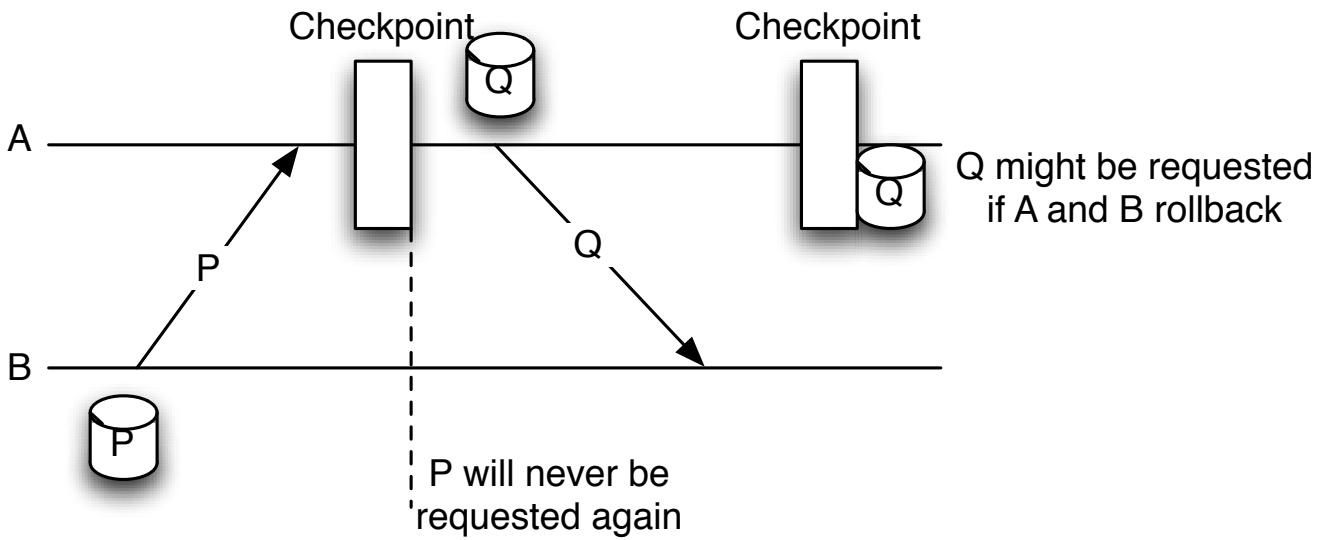
Message Logging



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

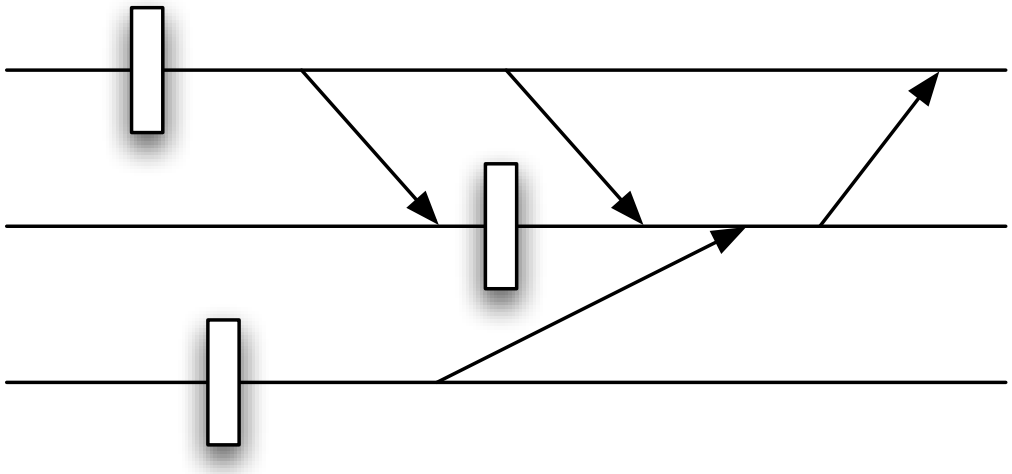
Message Logging



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

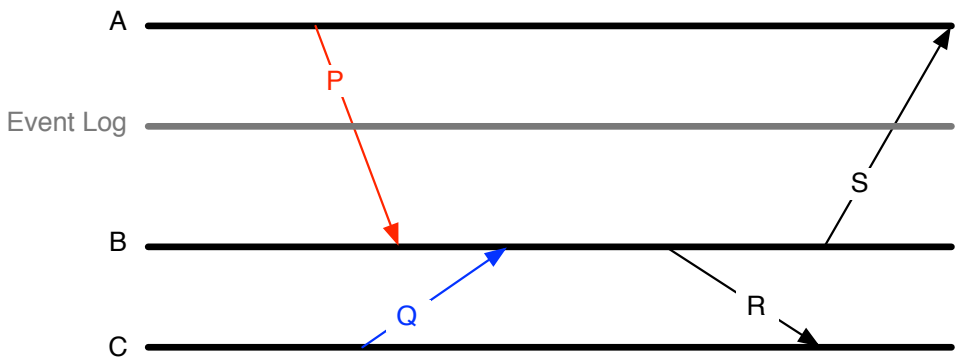
Message Logging



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)

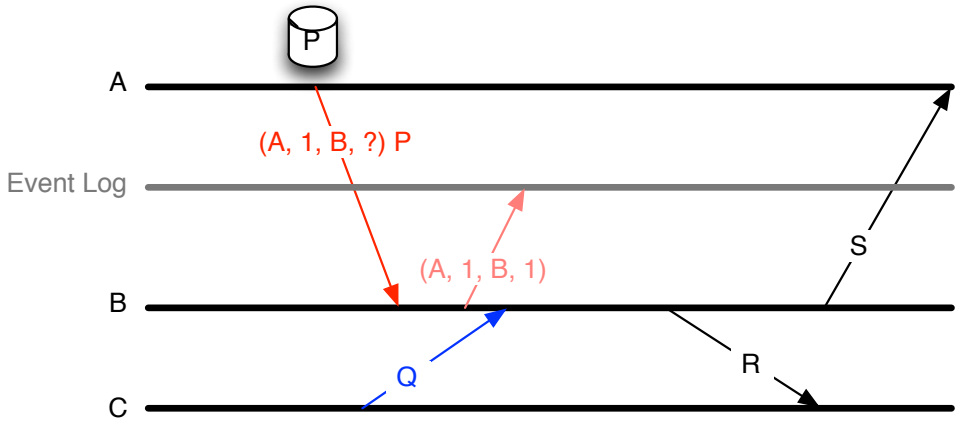
Optimistic Message Logging



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)

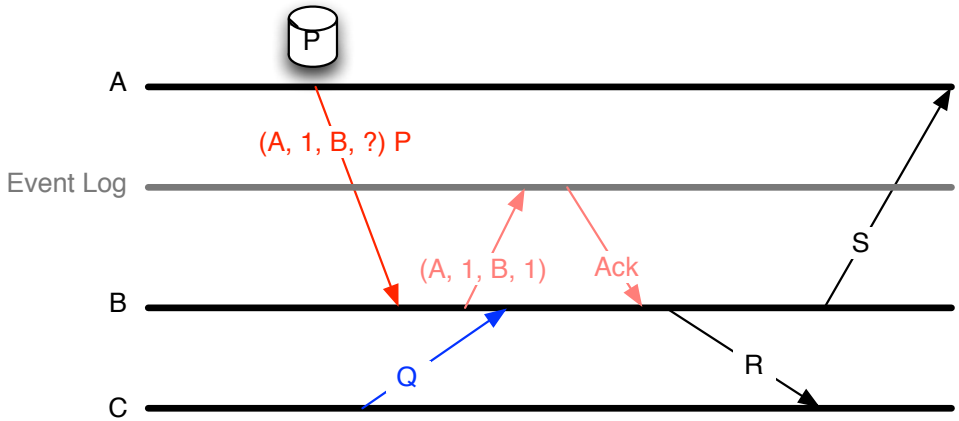
Optimistic Message Logging



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)

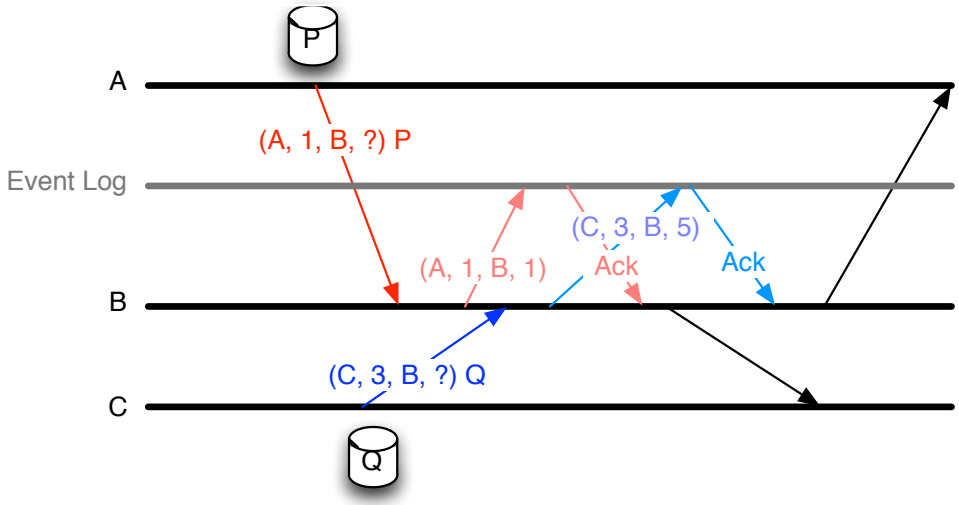
Optimistic Message Logging



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)

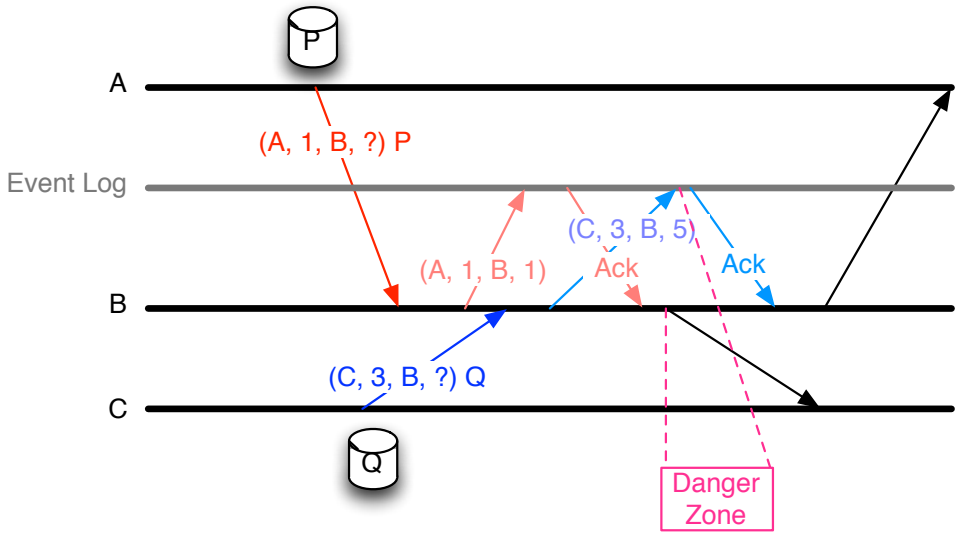
Optimistic Message Logging



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)

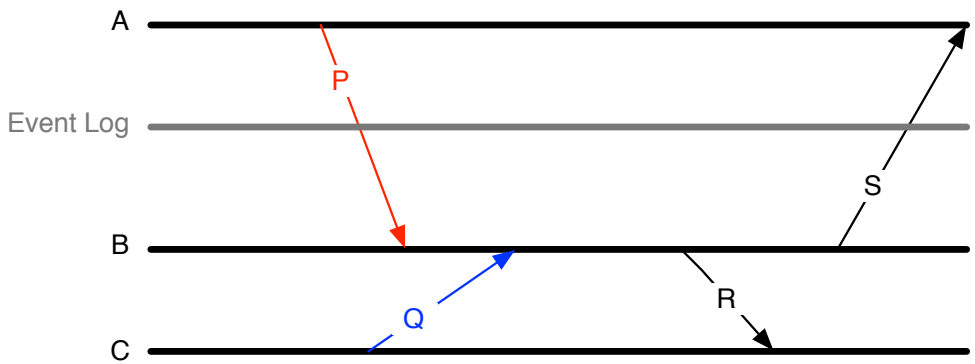
Optimistic Message Logging



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)

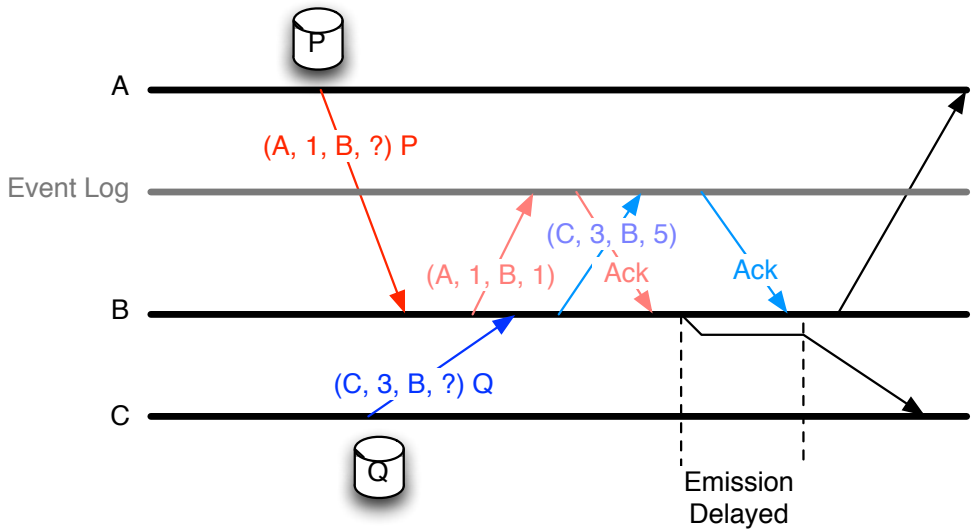
Pessimistic Message Logging



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

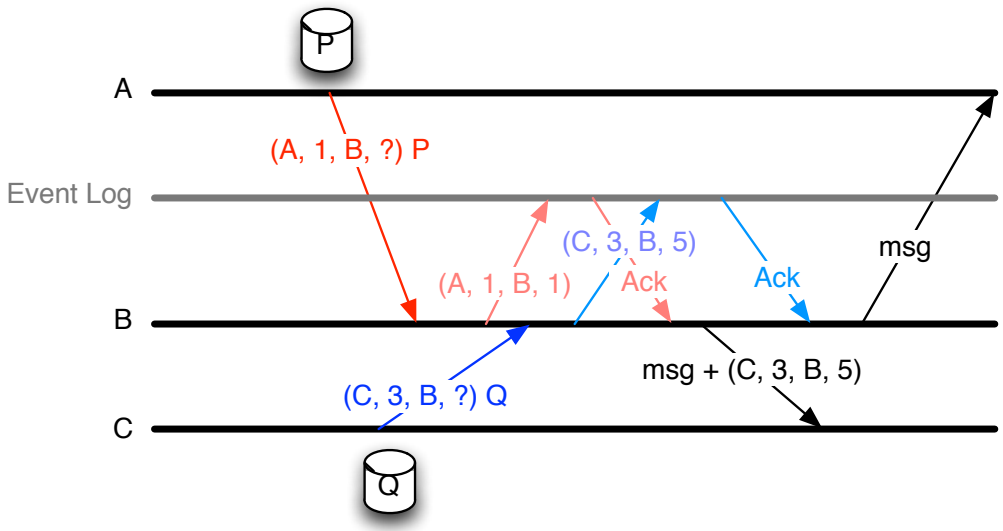
Pessimistic Message Logging



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

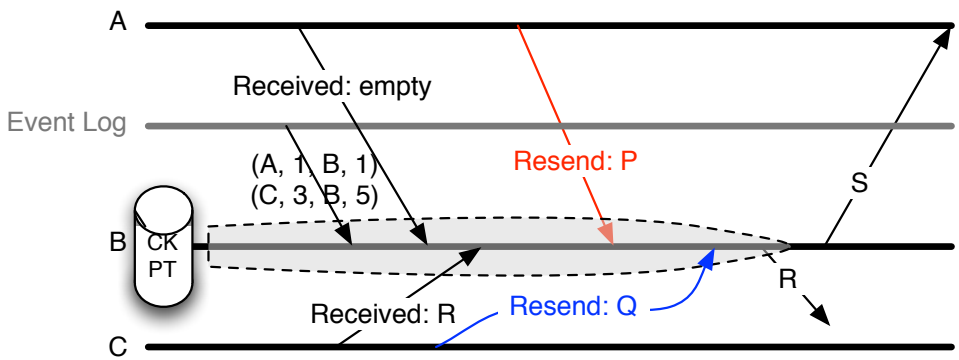
Causal Message Logging



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: global communication + potential storage system

Recover in Message Logging



Recovery

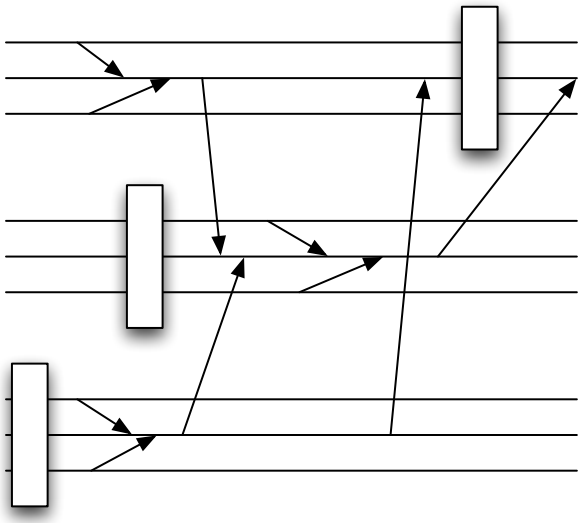
- Collect the history (from event log / event log + peers for Causal)
- Collect Id of last message sent
- Emitters resend, deliver in history order
- Fake emission of sent messages

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

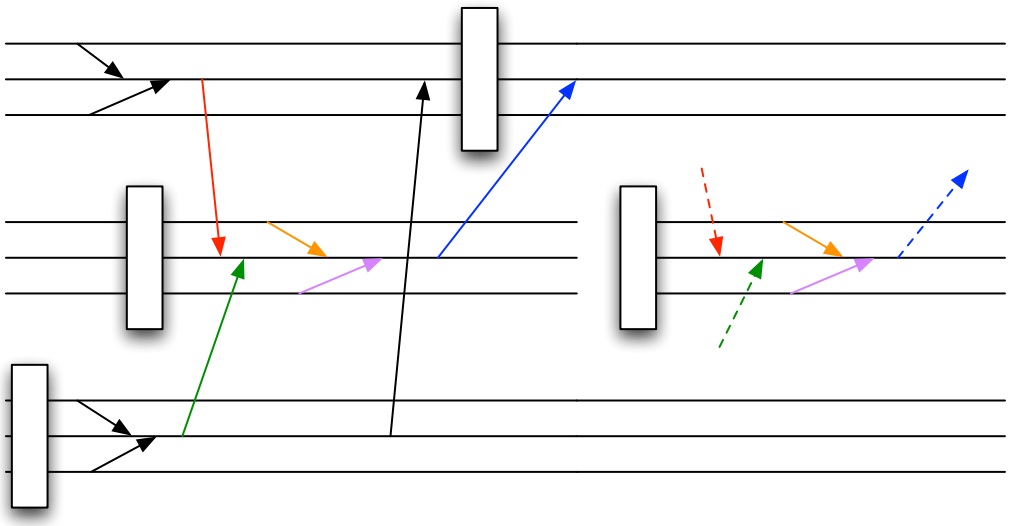
Hierarchical Protocols



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

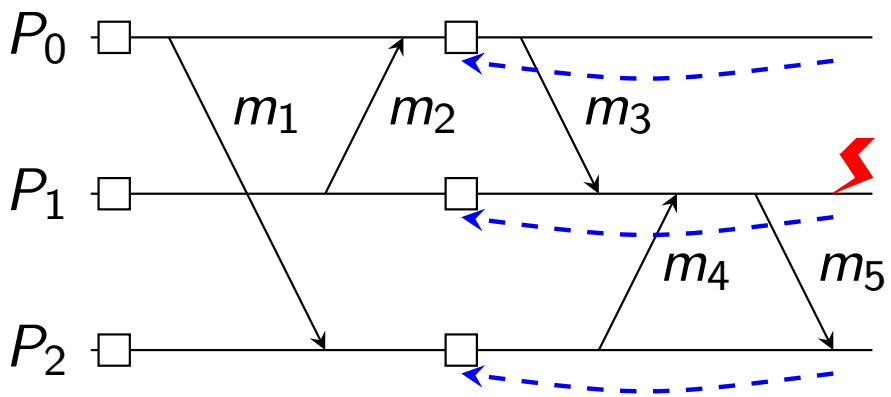
Strategies to reduce the amount of event log

- 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

Outline

Recap: coordinated checkpointing protocols

- Coordinated checkpoints over all processes
- Global restart after a failure



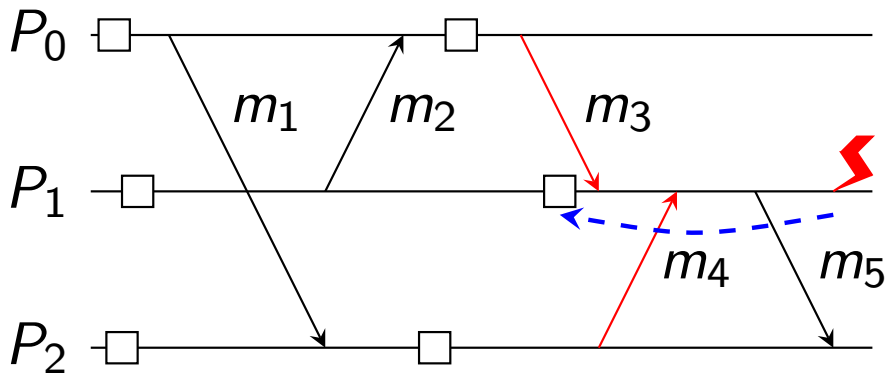
☺ No risk of cascading rollbacks

☺ No need to log messages

☹ All processors need to roll back

Recap: message logging protocols

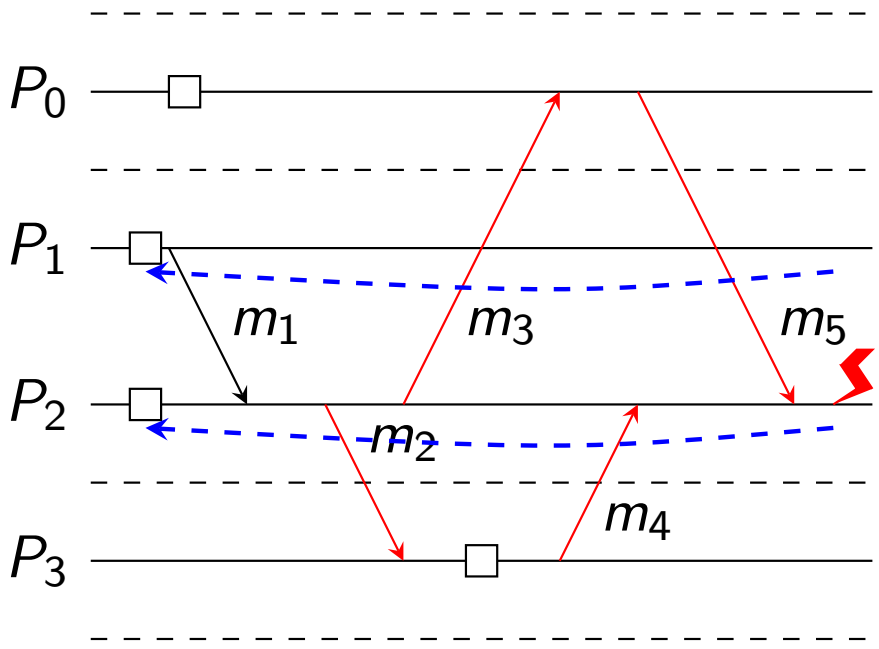
- Message content logging (sender memory)
- Restart of the failed process



- ☺ No cascading rollbacks
- ☺ Number of processes to roll back
- ☹ Memory occupation
- ☹ Overhead

Recap: hierarchical protocols

- Clusters of processes
- Coordinated checkpointing protocol within clusters
- Message logging protocols between clusters
- Only processors from failed group need to roll back



☹ Need to log inter-groups messages
- Slowdowns failure-free execution
- Increases checkpoint size/time

☺ Faster re-execution with logged messages

Which checkpointing protocol to use?

Coordinated checkpointing

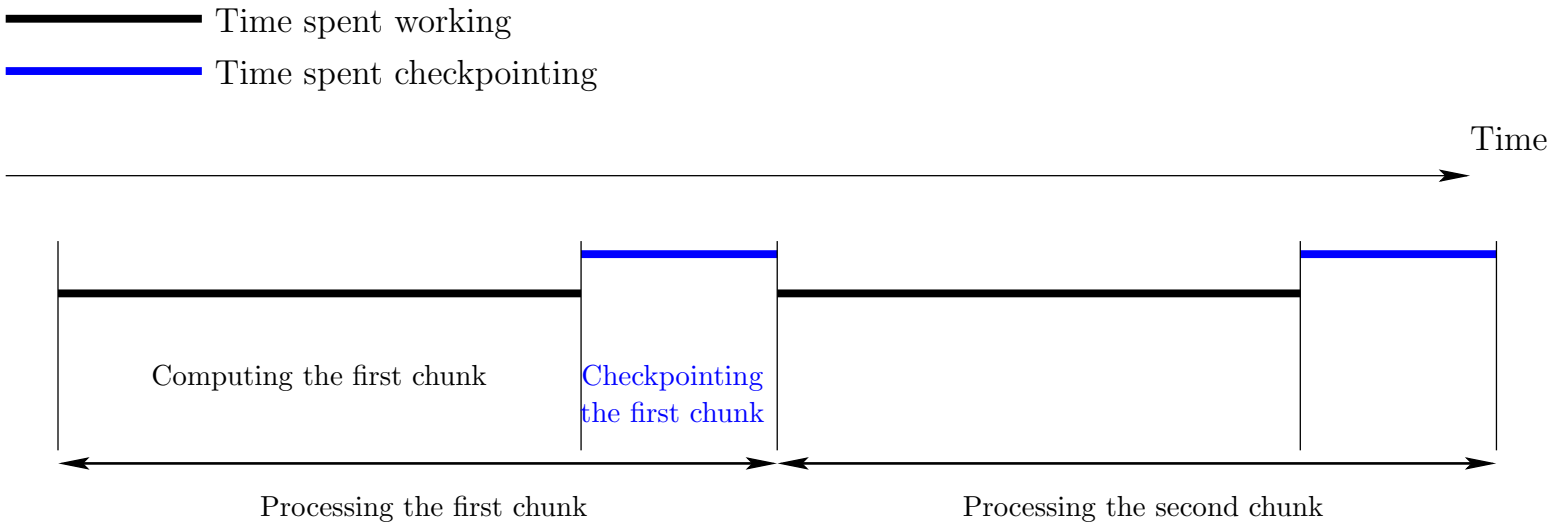
- ☺ No risk of cascading rollbacks
- ☺ No need to log messages
- ☹ All processors need to roll back
- ☹ Rumor: May not scale to very large platforms

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

Outline

Checkpointing cost



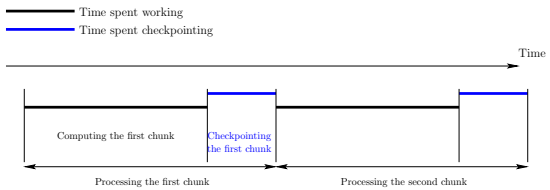
Blocking model: while a checkpoint is taken, no computation can be performed

Framework

- Periodic checkpointing policy of period T
- Independent and identically distributed failures
- Applies to a single processor with MTBF $\mu = \mu_{ind}$
- Applies to a platform with p processors with MTBF $\mu = \frac{\mu_{ind}}{p}$
  - coordinated checkpointing
  - tightly-coupled application
  - progress $\Leftrightarrow$ all processors available

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

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

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

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

$$\text{WASTE}[FF] = \frac{C}{T}$$

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 par failures

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

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

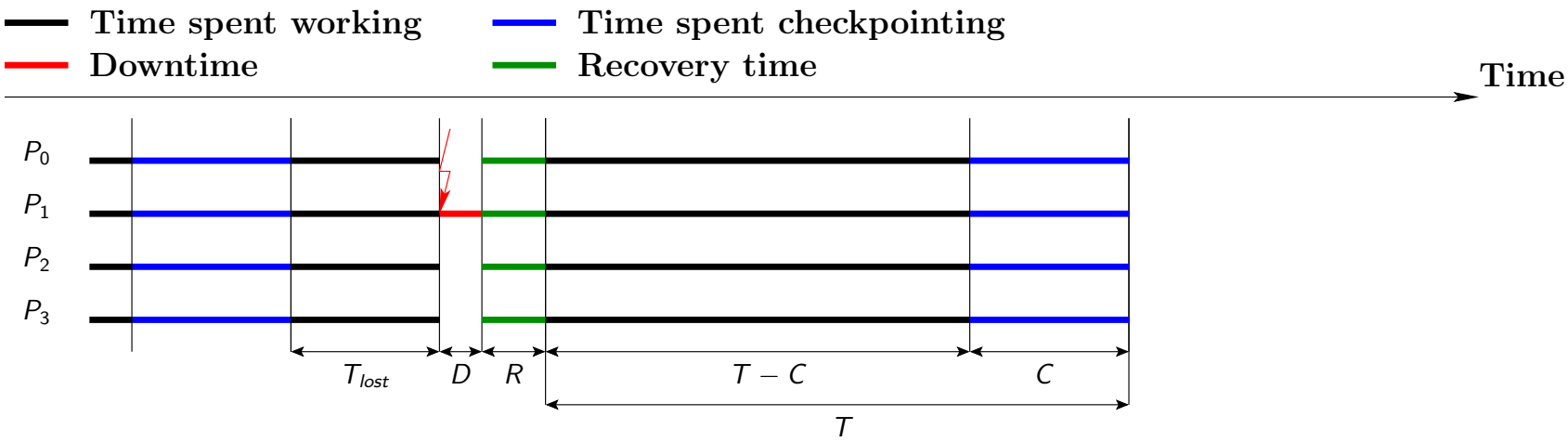
$$\text{TIME}_{\text{final}} = \text{TIME}_{\text{FF}} + N_{faults} \times T_{\text{lost}}$$

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

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

T_{lost}?

Computing T_{lost}



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

$\Rightarrow$ Instants when periods begin and failures strike are independent

$\Rightarrow$ Valid for all distribution laws, regardless of their particular shape

Waste due to failures

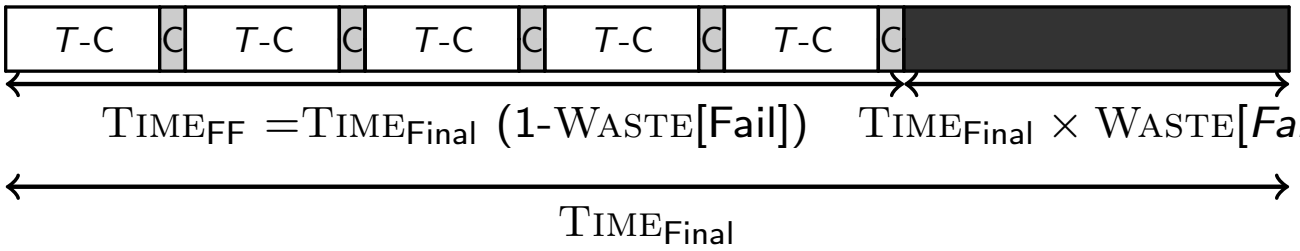
$$\text{TIME}_{\text{final}} = \text{TIME}_{\text{FF}} + N_{faults} \times T_{\text{lost}}$$

$$\text{TIME}_{\text{final}} = \text{TIME}_{\text{FF}} + \frac{\text{TIME}_{\text{final}}}{\mu} \times \left(D + R + \frac{T}{2} \right)$$

$$\text{WASTE}[fail] = \frac{\text{TIME}_{\text{final}} - \text{TIME}_{\text{FF}}}{\text{TIME}_{\text{final}}}$$

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

Total waste



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

$$(1 - \text{WASTE}[\mathit{fail}])(1 - \text{WASTE}[\mathit{FF}])\text{TIME}_{\text{final}} = \mathit{Time}[\mathit{base}]$$

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

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

Waste minimization

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

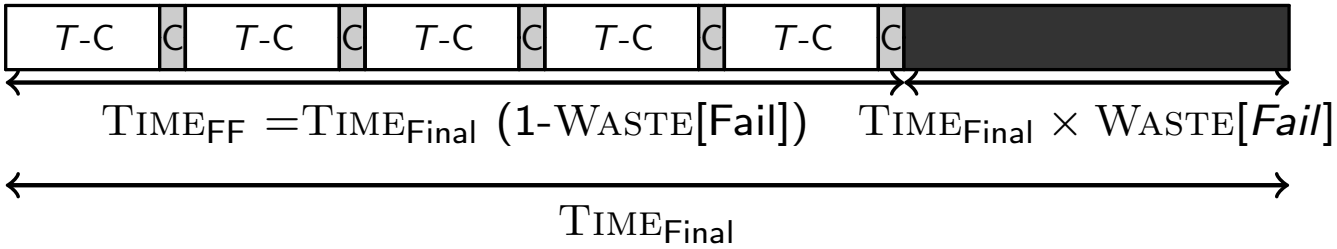
$$\text{WASTE} = \frac{u}{T} + v + wT$$

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

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

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

Comparison with Young/Daly



$$\big(1 - \textsc{Waste}[\mathit{fail}]\big)\,\textsc{Time}_{\mathsf{final}} = \textsc{Time}_{\mathsf{FF}}$$
$$\Rightarrow T = \sqrt{2(\mu - (D+R))C}$$

Daly: $\textsc{Time}_{\mathsf{final}} = \big(1 + \textsc{Waste}[\mathit{fail}]\big)\,\textsc{Time}_{\mathsf{FF}}$
$$\Rightarrow T = \sqrt{2(\mu + (D+R))C} + C$$

Young: $\textsc{Time}_{\mathsf{final}} = \big(1 + \textsc{Waste}[\mathit{fail}]\big)\,\textsc{Time}_{\mathsf{FF}}$ and $D = R = 0$
$$\Rightarrow T = \sqrt{2\mu C} + C$$

Comparison with Young/Daly

p	μ	Young		Daly		Previous formula		Optimal
2^{10}	3849609	68567	(1.4 %)	68573	(1.4 %)	67961	(0.5 %)	67640
2^{11}	1924805	48660	(2.0 %)	48668	(2.0 %)	48052	(0.7 %)	47720
2^{12}	962402	34584	(3.0 %)	34595	(3.0 %)	33972	(1.1 %)	33589
2^{13}	481201	24630	(4.2 %)	24646	(4.3 %)	24014	(1.6 %)	23631
2^{14}	240601	17592	(6.0 %)	17615	(6.2 %)	16968	(2.3 %)	16594
2^{15}	120300	12615	(8.6 %)	12648	(8.9 %)	11982	(3.1 %)	11618
2^{16}	60150	9096	(12.3 %)	9142	(12.9 %)	8449	(4.3 %)	8101
2^{17}	30075	6608	(17.7 %)	6673	(18.9 %)	5941	(5.8 %)	5614
2^{18}	15038	4848	(25.7 %)	4940	(28.1 %)	4154	(7.7 %)	3858
2^{19}	7519	3604	(37.7 %)	3733	(42.6 %)	2869	(9.6 %)	2618
2^{20}	3759	2724	(56.2 %)	2903	(66.4 %)	1929	(10.6 %)	1744

Approximated vs. optimal period for Exponential distributions

Validity of the approach (1/3)

Technicalities

- $\mathbb{E}(N_{faults}) = \frac{\text{TIME}_{\text{final}}}{\mu}$ and $\mathbb{E}(T_{\text{lost}}) = \frac{T}{2}$
  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 ☺

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*: non-predicted faults

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

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 all three event types)

$$\frac{(1-r)}{\mu} = \frac{1}{\mu_{NP}}$$

$$\frac{r}{\mu} = \frac{p}{\mu_P}$$

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

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:** $\frac{1}{\mu_{NP}} \left[D + R + \frac{T}{2} \right]$
3. **Predictions:** $\frac{1}{\mu_P} \left[p(C + D + R) + (1 - p)C \right]$

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

$$T_{opt} \approx \sqrt{\frac{2\mu C}{1 - r}}$$

Outline

Exponential failure distribution

1. Expected execution time for a single chunk
2. Expected execution time for a sequential job
3. Expected execution time for a parallel job

Expected execution time for a single chunk

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

Recursive Approach

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

Expected execution time for a single chunk

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

Recursive Approach

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

Expected execution time for a single chunk

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

Recursive Approach

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

Expected execution time for a single chunk

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

Recursive Approach

$$\mathbb{E}(T(W)) = \mathcal{P}_{\text{succ}}(W + C)\,(W + C)$$

Expected execution time for a single chunk

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

Recursive Approach

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

Expected execution time for a single chunk

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

Recursive Approach

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

Expected execution time for a single chunk

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

Recursive Approach

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

Expected execution time for a single chunk

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

Recursive Approach

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

Expected execution time for a single chunk

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

Recursive Approach

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

Computation of $\mathbb{E}(T(W, C, D, R, \lambda))$

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

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

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

Checkpointing a sequential job

- $\mathbb{E}(T(W)) = e^{\lambda R}\left(\frac{1}{\lambda} + D\right)\left(\sum_{i=1}^{K} e^{\lambda(W_i+C)} - 1\right)$
- Optimal strategy uses same-size chunks (convexity)
- $K_0 = \frac{\lambda W}{1+\mathbb{L}(-e^{-\lambda C-1})}$ where $\mathbb{L}(z)e^{\mathbb{L}(z)} = z$ (Lambert function)
- Optimal number of chunks K^* is $\max(1, \lfloor K_0 \rfloor)$ or $\lceil K_0 \rceil$

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

- Can also use Daly's second-order approximation

Checkpointing a parallel job

- p processors $\Rightarrow$ distribution $Exp(\lambda_p)$, where $\lambda_p = p\lambda$
- Use $W(p)$, $C(p)$, $R(p)$ in $\mathbb{E}_{opt}(T(W))$ for a distribution $Exp(\lambda_p = p\lambda)$
- Job types
  - Perfectly parallel jobs: $W(p) = W/p$.
  - Generic parallel jobs: $W(p) = W/p + \delta W$
  - Numerical kernels: $W(p) = W/p + \delta W^{2/3}/\sqrt{p}$
- Checkpoint overhead
  - Proportional overhead: $C(p) = R(p) = \delta V/p = C/p$ (bandwidth of processor network card/link is I/O bottleneck)
  - Constant overhead: $C(p) = R(p) = \delta V = C$ (bandwidth to/from resilient storage system is I/O bottleneck)

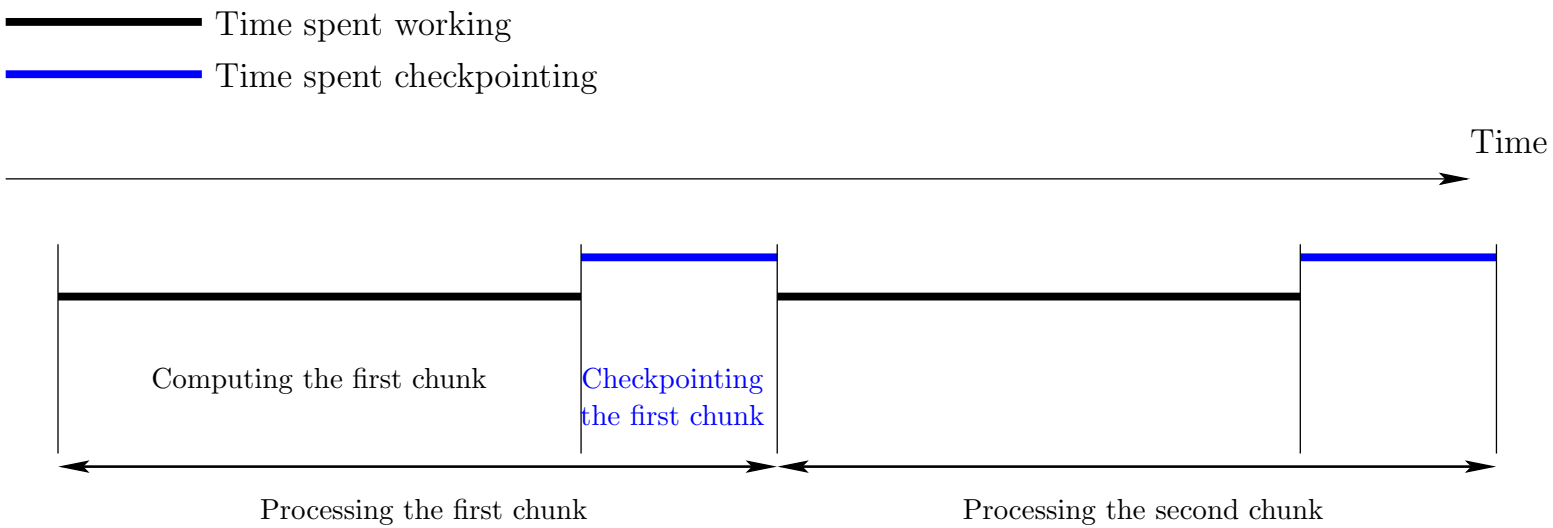
Weibull failure distribution

- No optimality result known
- Heuristic: maximize expected work before next failure
- Dynamic programming algorithms
  - Use a time quantum
  - Trim history of previous failures

Outline

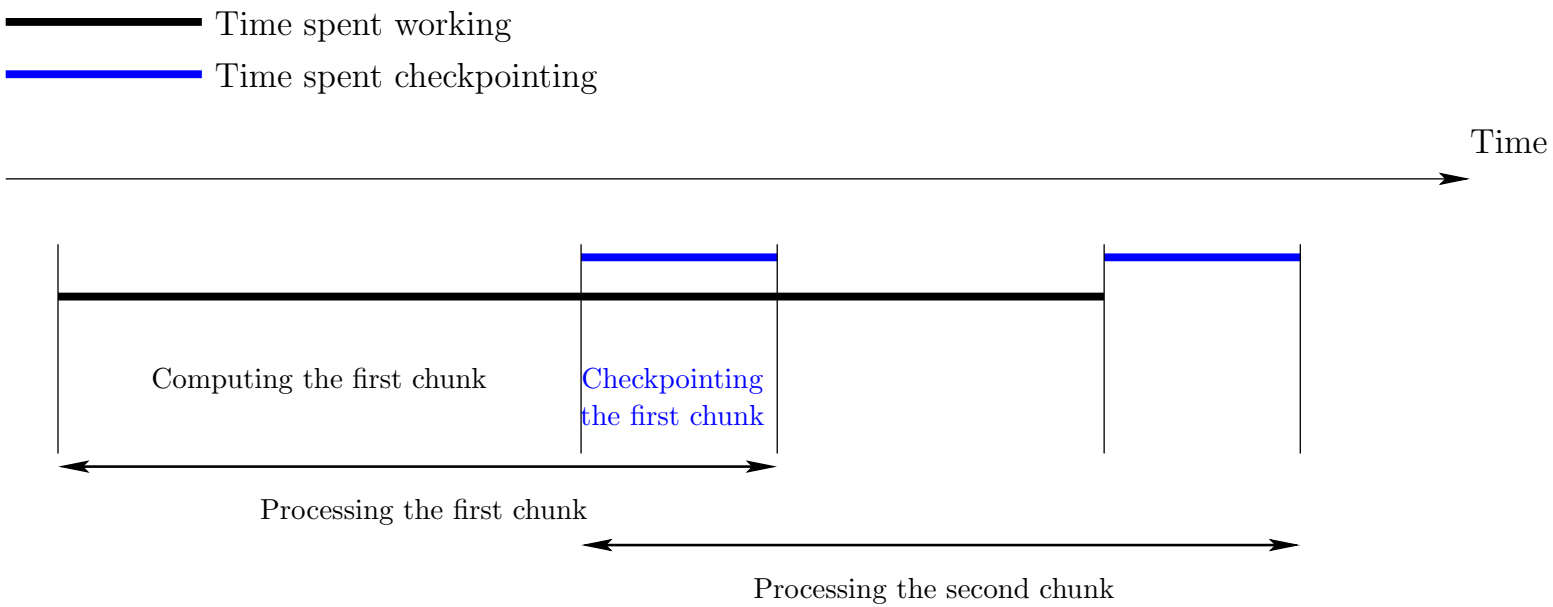
Outline

Checkpointing cost



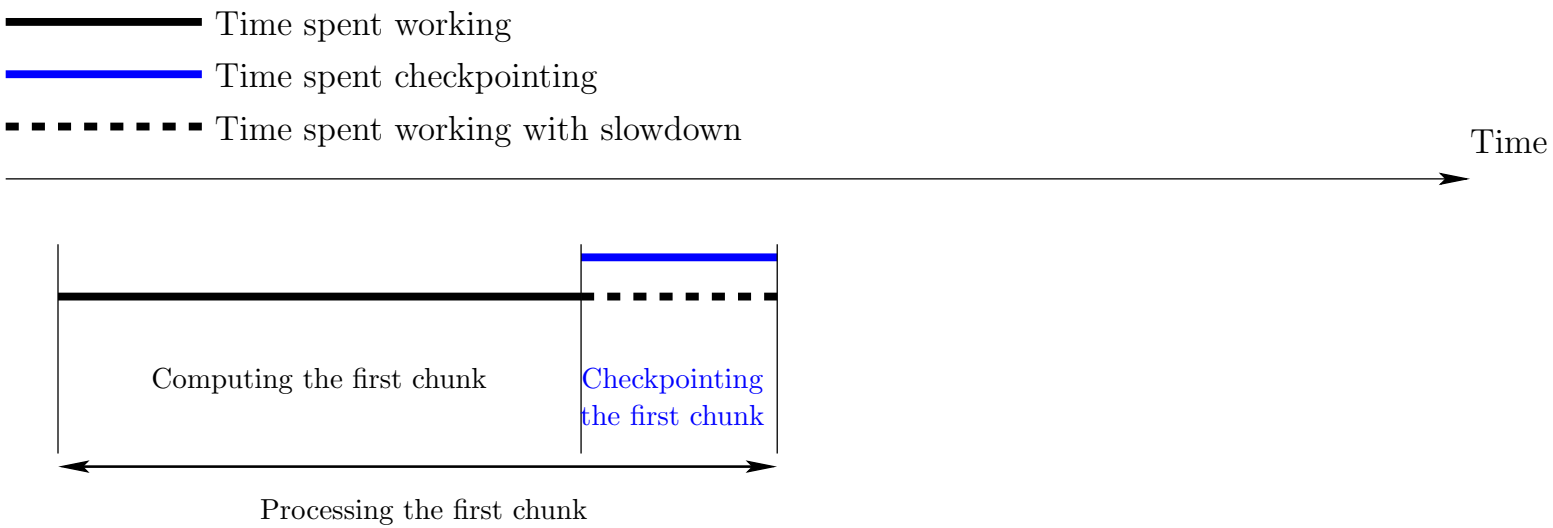
Blocking model: checkpointing blocks all computations

Checkpointing cost



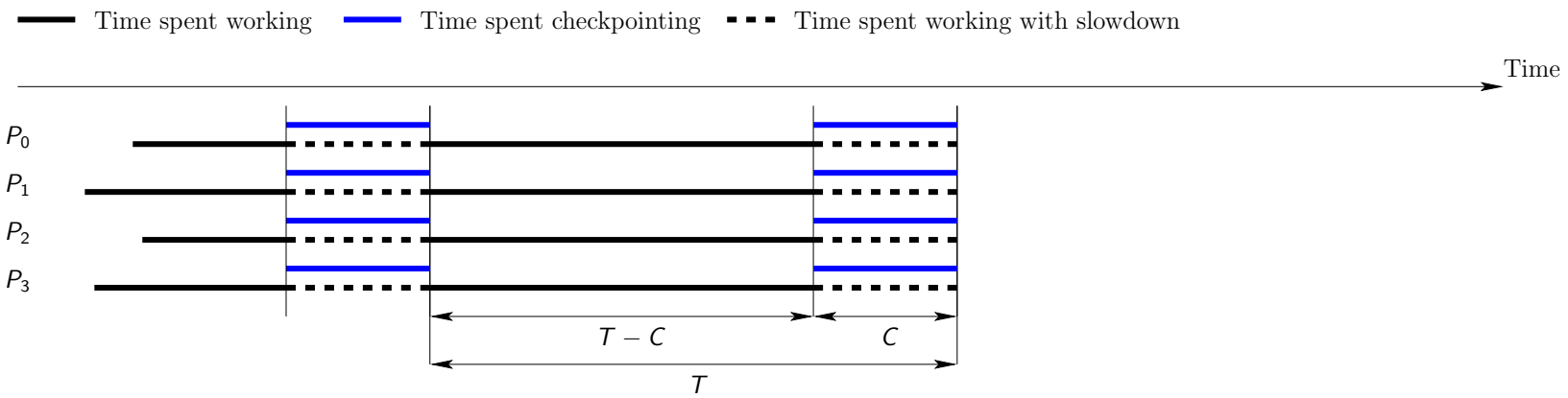
Non-blocking model: checkpointing has no impact on computations (e.g., first copy state to RAM, then copy RAM to disk)

Checkpointing cost



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 ($0 \leq \alpha \leq 1$)

Waste in fault-free execution

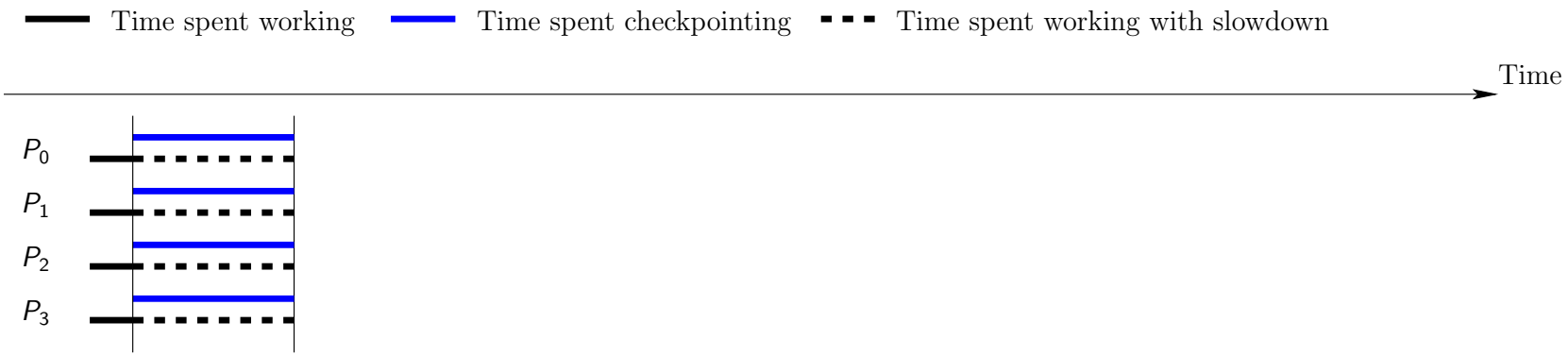


Time elapsed since last checkpoint: T

Amount of computations executed: $(T - C) + \alpha C$

$$\text{WASTE}[FF] = \frac{T-((T-C)+\alpha C)}{T} = \frac{(1-\alpha)C}{T}$$

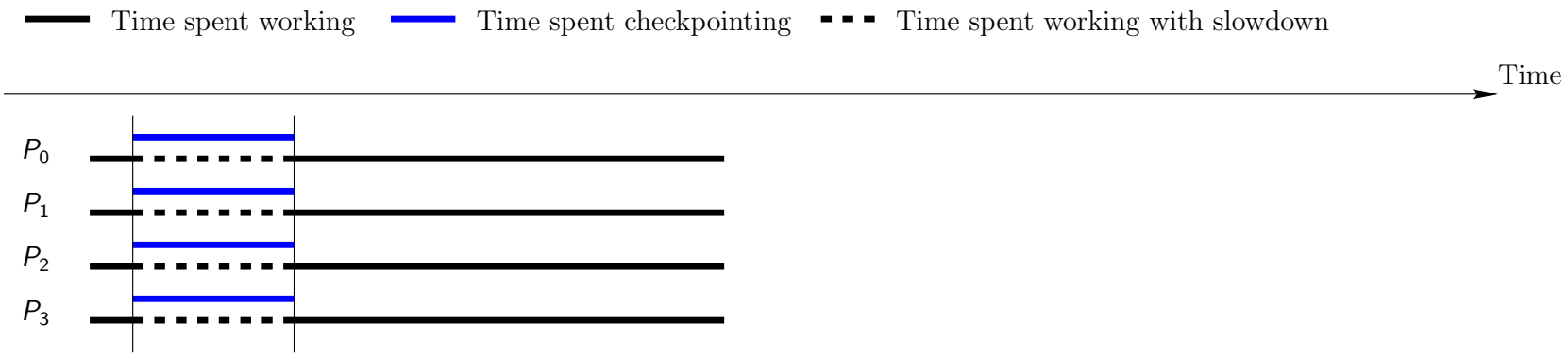
Waste due to failures



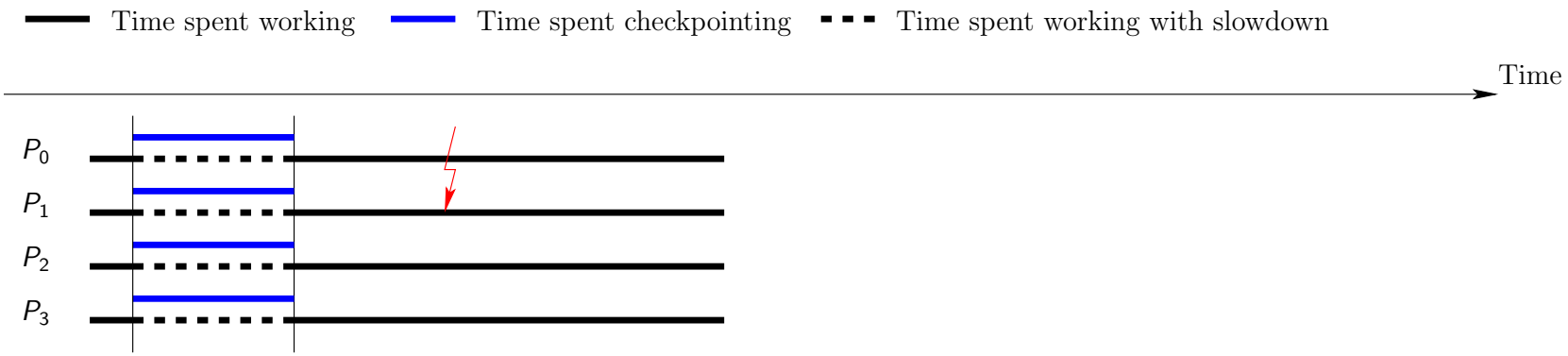
Failure can happen

1. During computation phase
2. During checkpointing phase

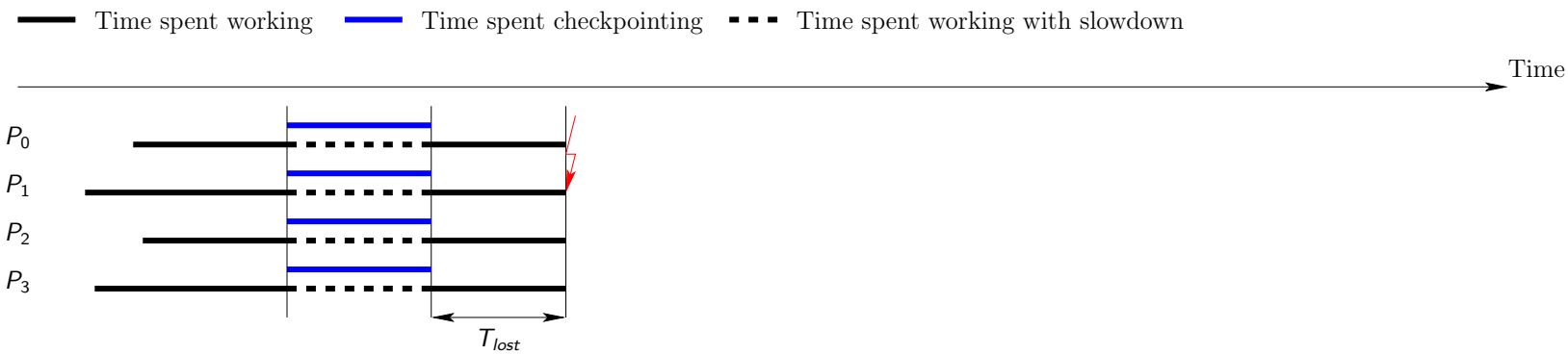
Waste due to failures



Waste due to failures

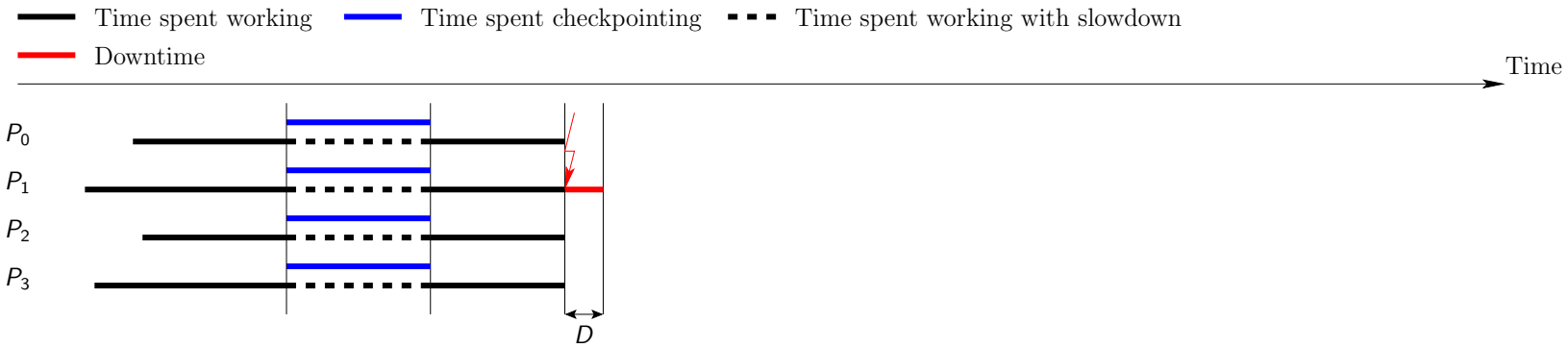


Waste due to failures

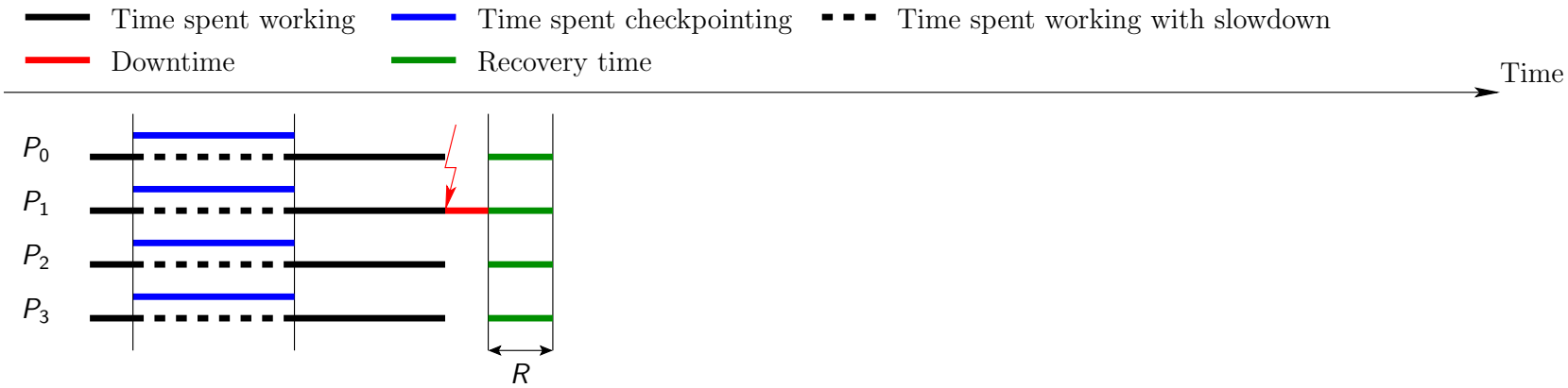


Coordinated checkpointing protocol: when one processor is victim of a failure, all processors lose their work and must roll back to last checkpoint

Waste due to failures in computation phase

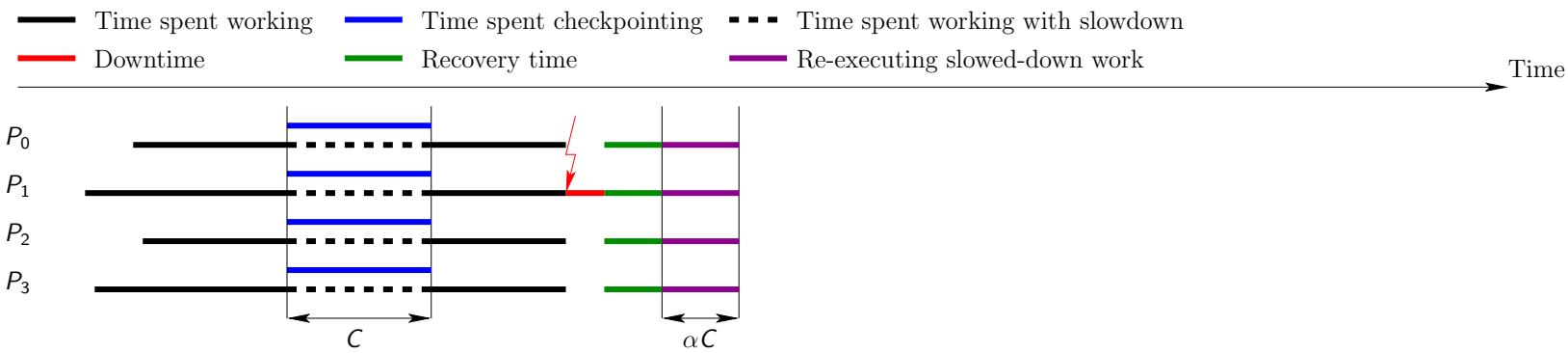


Waste due to failures in computation phase



Coordinated checkpointing protocol: all processors must recover from last checkpoint

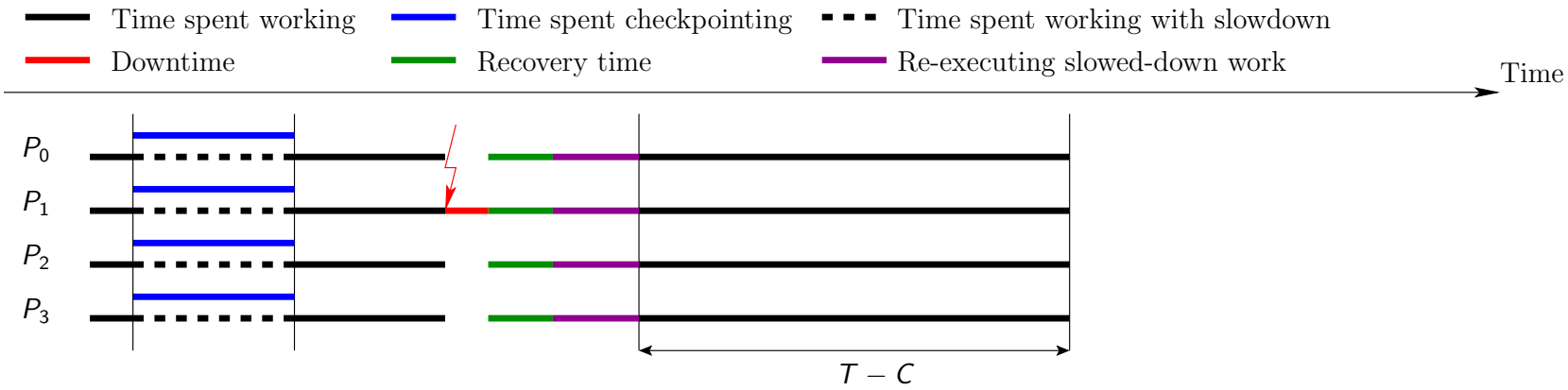
Waste due to failures in computation phase



Redo the work destroyed by the failure, that was done in the checkpointing phase before the computation phase

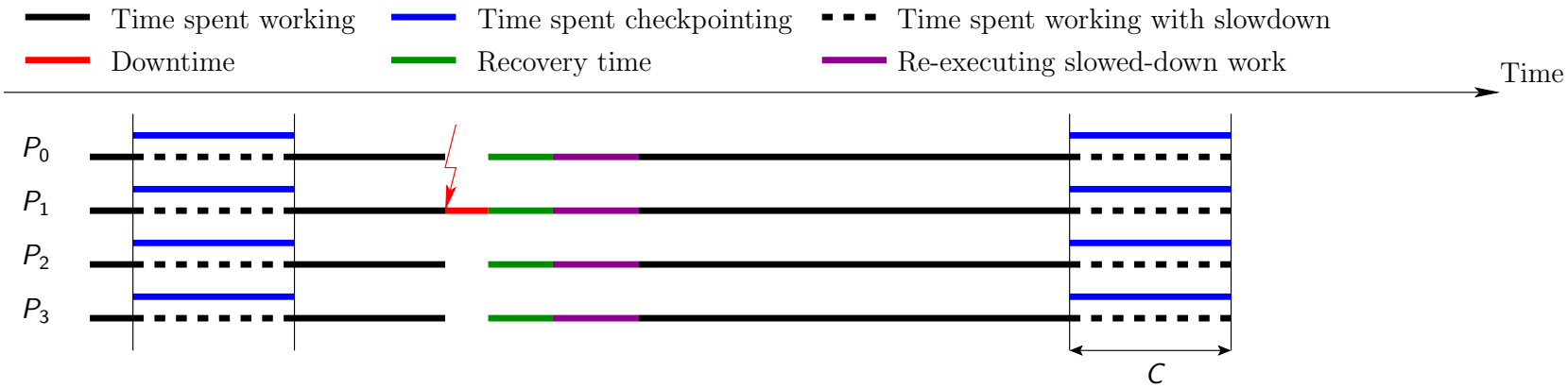
But no checkpoint is taken in parallel, hence this re-execution is faster than the original computation

Waste due to failures in computation phase



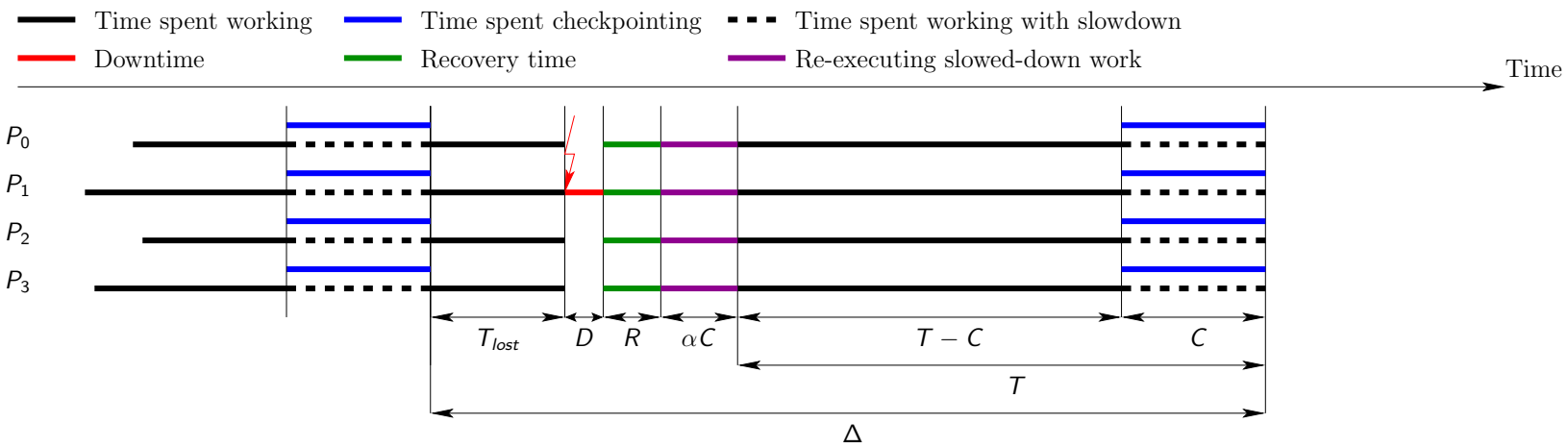
Re-execute the computation phase

Waste due to failures in computation phase



Finally, the checkpointing phase is executed

Waste due to failures in computation phase



RE-EXEC: $\Delta - T = T_{lost} + \alpha C$

Expectation: $T_{lost} = \frac{1}{2}(T - C)$

$$\text{RE-EXEC}_{coord-fail-in-work} = \frac{T - C}{2} + \alpha C$$

Waste due to failures

- Failure in the computation phase (probability: $\frac{T-C}{T}$)

$$\text{RE-EXEC}_{coord-fail-in-work} = \alpha C + \frac{T-C}{2}$$

- Failure in the checkpointing phase (probability: $\frac{C}{T}$)

$$\text{RE-EXEC}_{coord-fail-in-checkpoint} = \alpha C + T - C + \frac{C}{2}$$

$$\text{RE-EXEC}_{coord} = \frac{T-C}{T}\left(\frac{T-C}{2} + \alpha C\right) + \frac{C}{T}\left(T - \frac{C}{2} + \alpha C\right)$$

$$= \alpha C + \frac{T}{2}$$

Total waste

$$\text{WASTE}[FF] = \frac{(1-\alpha)C}{T}$$

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

$$\text{WASTE} = \text{WASTE}[FF] + \text{WASTE}[fail] - \text{WASTE}[FF]\text{WASTE}[fail]$$

Optimal period

$$T_{\text{opt}} = \sqrt{2(1-\alpha)(\mu - (D+R))C}$$

Sanity check: previous result if $\alpha = 0$ (blocking checkpoint)

Total waste

$$\text{WASTE}[FF] = \frac{(1-\alpha)C}{T}$$

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

$$\text{WASTE} = \text{WASTE}[FF] + \text{WASTE}[fail] - \text{WASTE}[FF]\text{WASTE}[fail]$$

Optimal period

$$T_{\text{opt}} = \sqrt{2(1-\alpha)(\mu - (D+R))C}$$

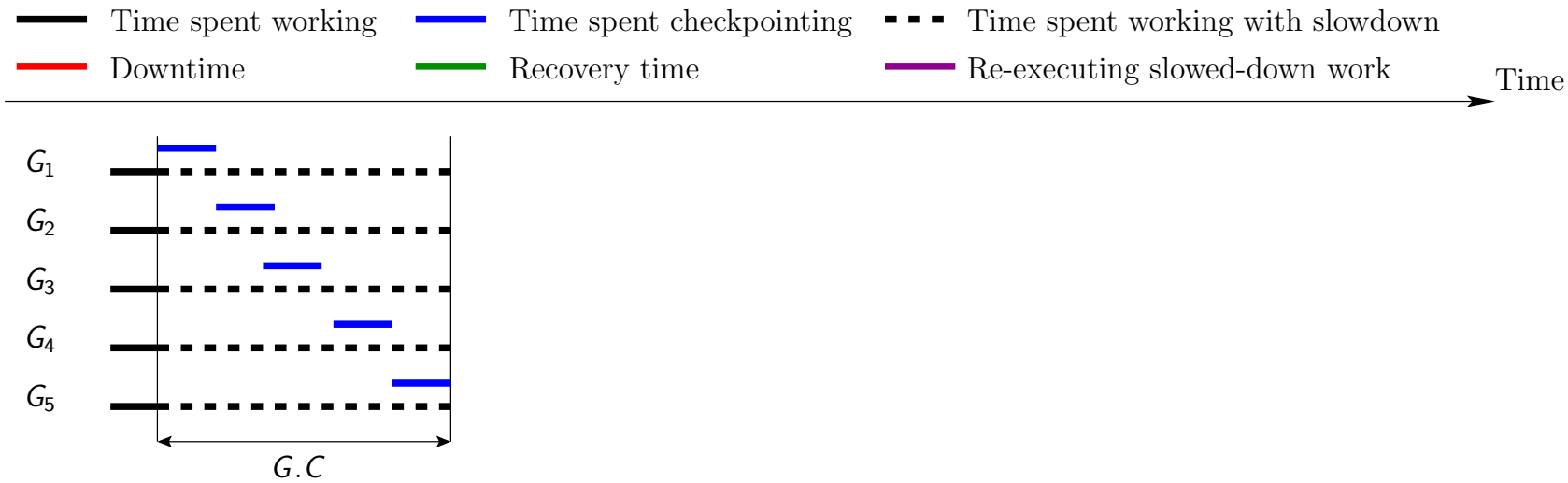
Sanity check: previous result if $\alpha = 0$ (blocking checkpoint)

Outline

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

Waste in fault-free execution

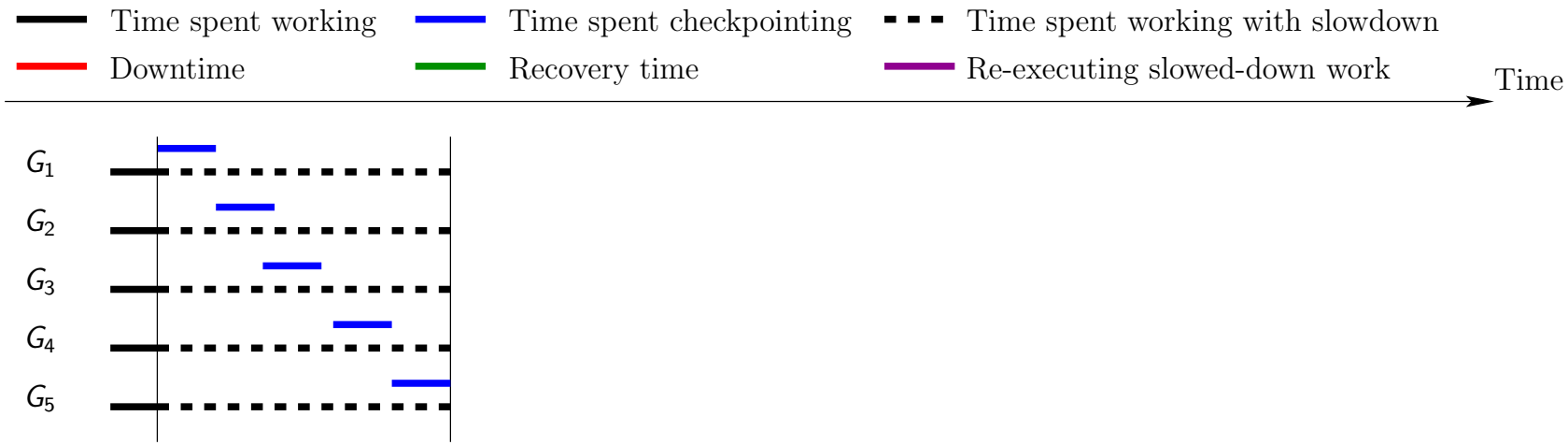


When a group checkpoints, its own computation speed is slowed-down

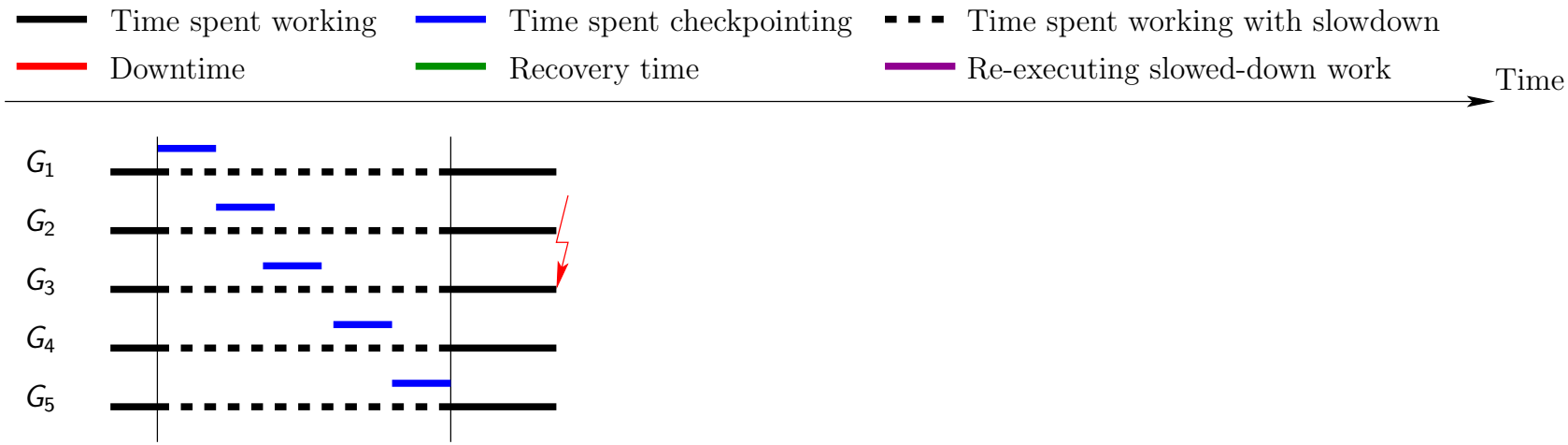
This holds for all groups because of the tightly-coupled assumption

$\text{WASTE}[FF] = \frac{T - \text{WORK}}{T}$ where $\text{WORK} = T - (1 - \alpha)GC(q)$

Waste due to failures

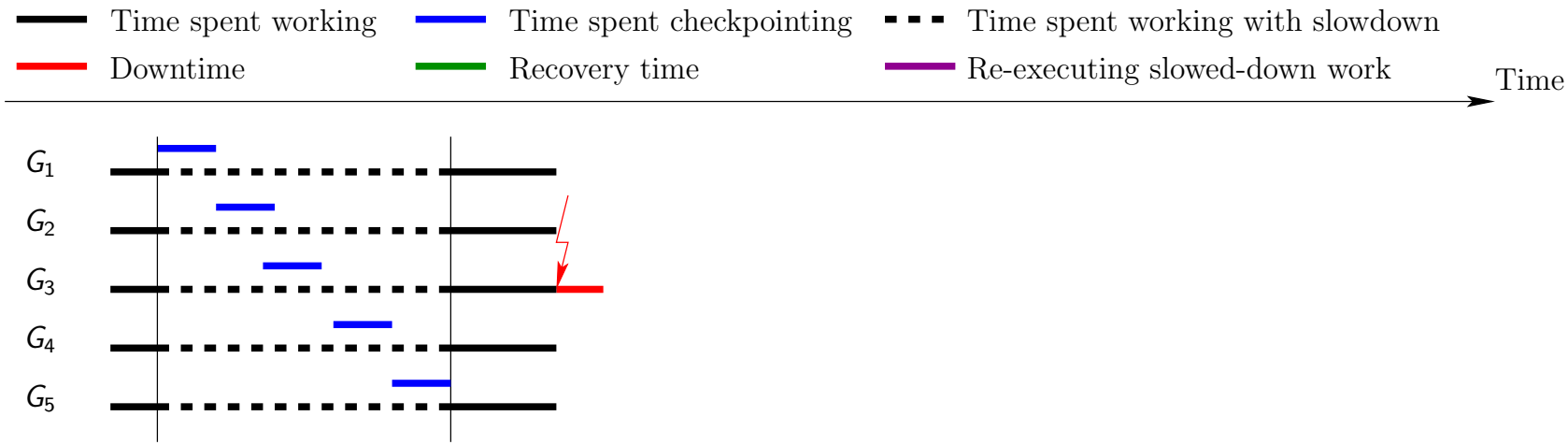


Waste due to failures



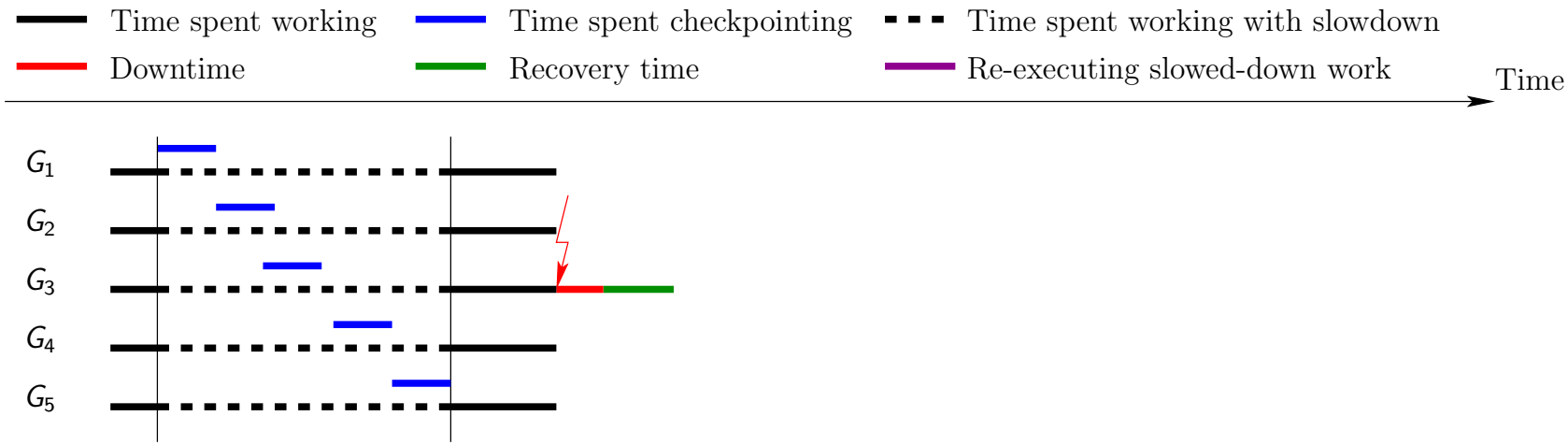
Group g is struck by a failure

Waste due to failures



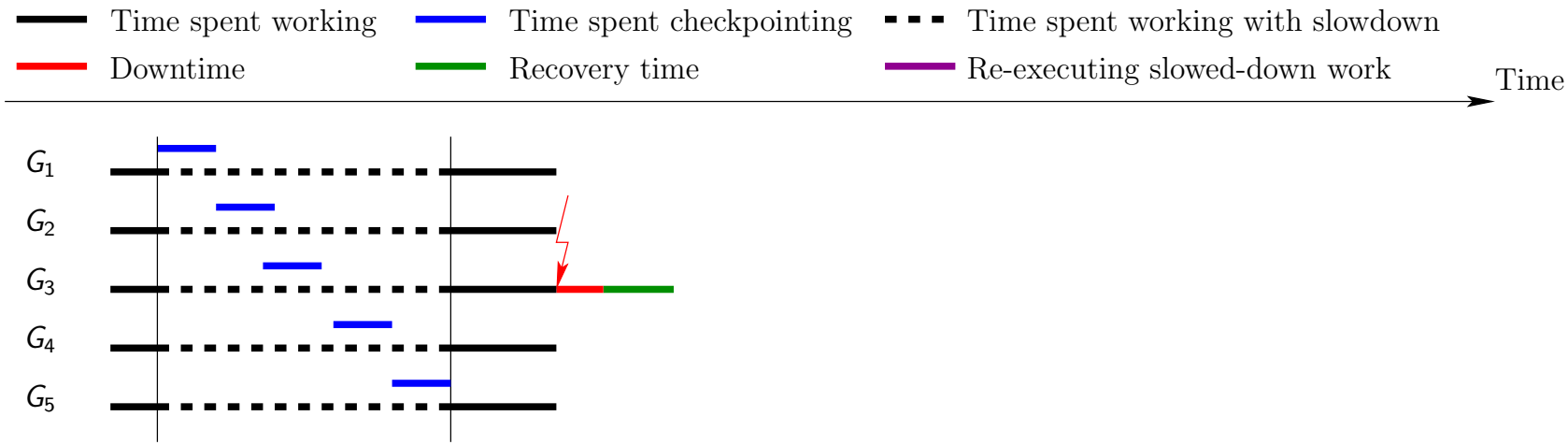
Tightly-coupled model: while one group is in downtime, none can work

Failure during computation phase



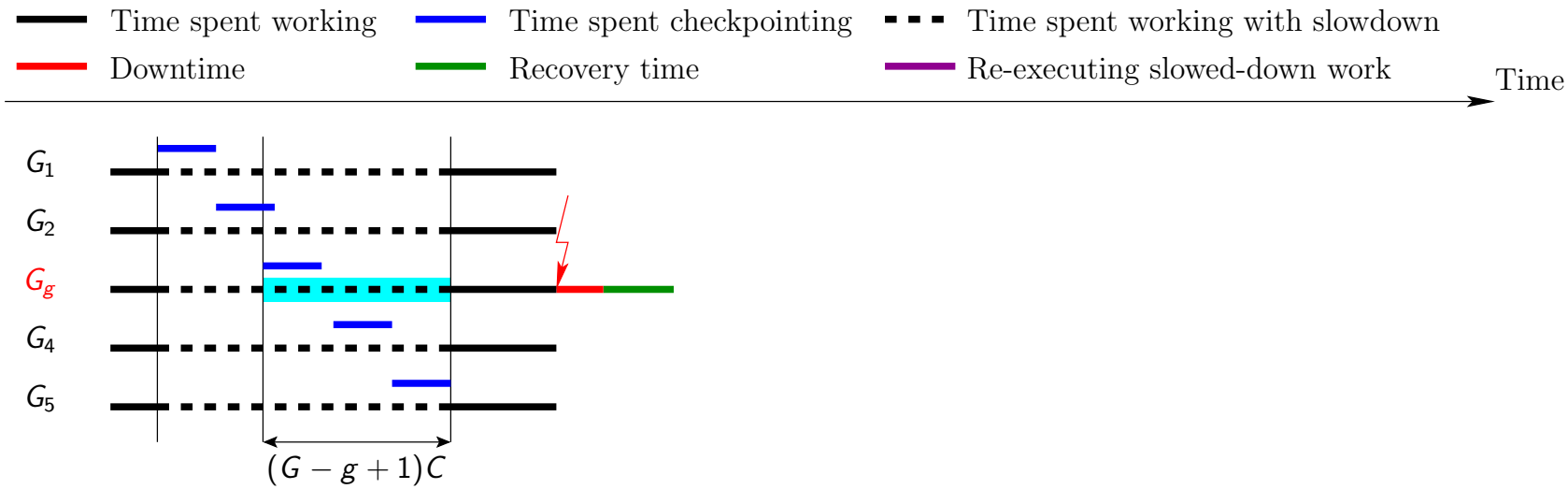
Tightly-coupled model: while one group is in recovery, none can work

Failure during computation phase



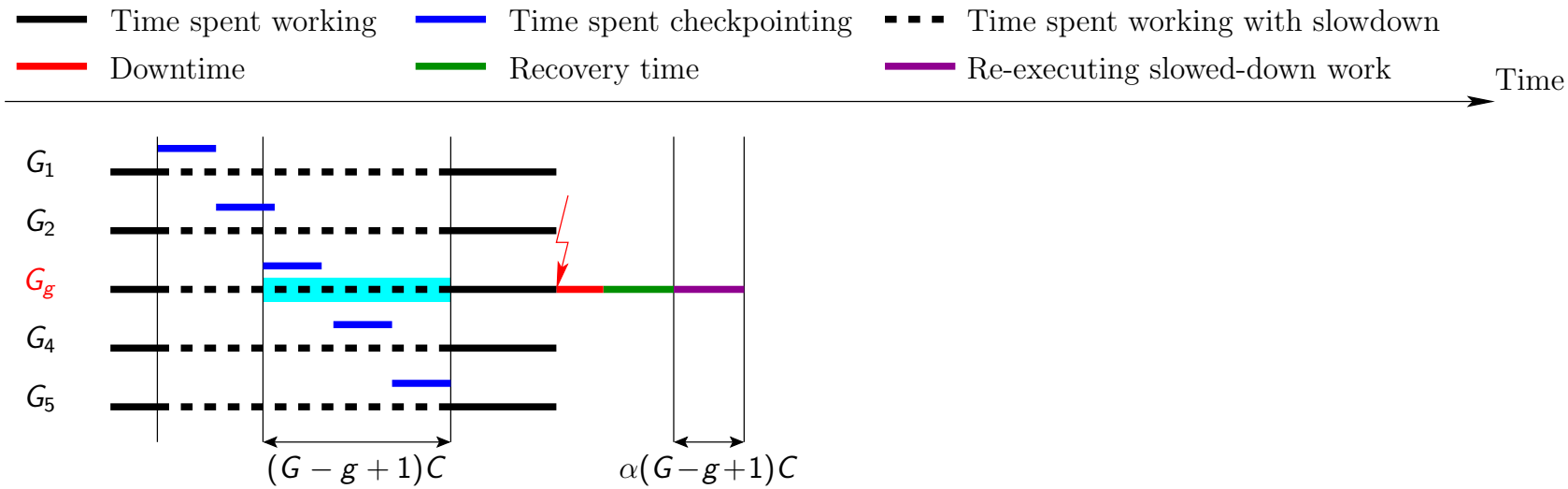
Groups must have completed the same amount of work in between two consecutive checkpoints, independently of the fact that a failure may or may not have struck the platform in between these checkpoints. Hence, no checkpointing is possible during the rollback.

Failure during computation phase



Redo work done during previous checkpointing phase and that was destroyed by the failure

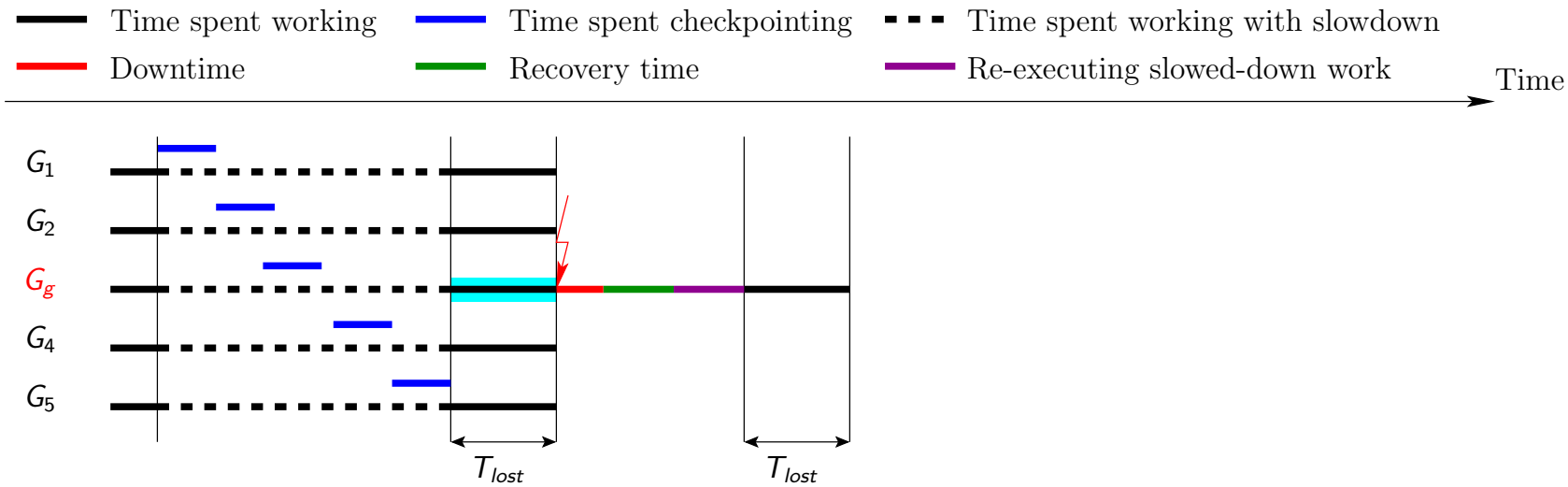
Failure during computation phase



Redo work done during previous checkpointing phase and that was destroyed by the failure

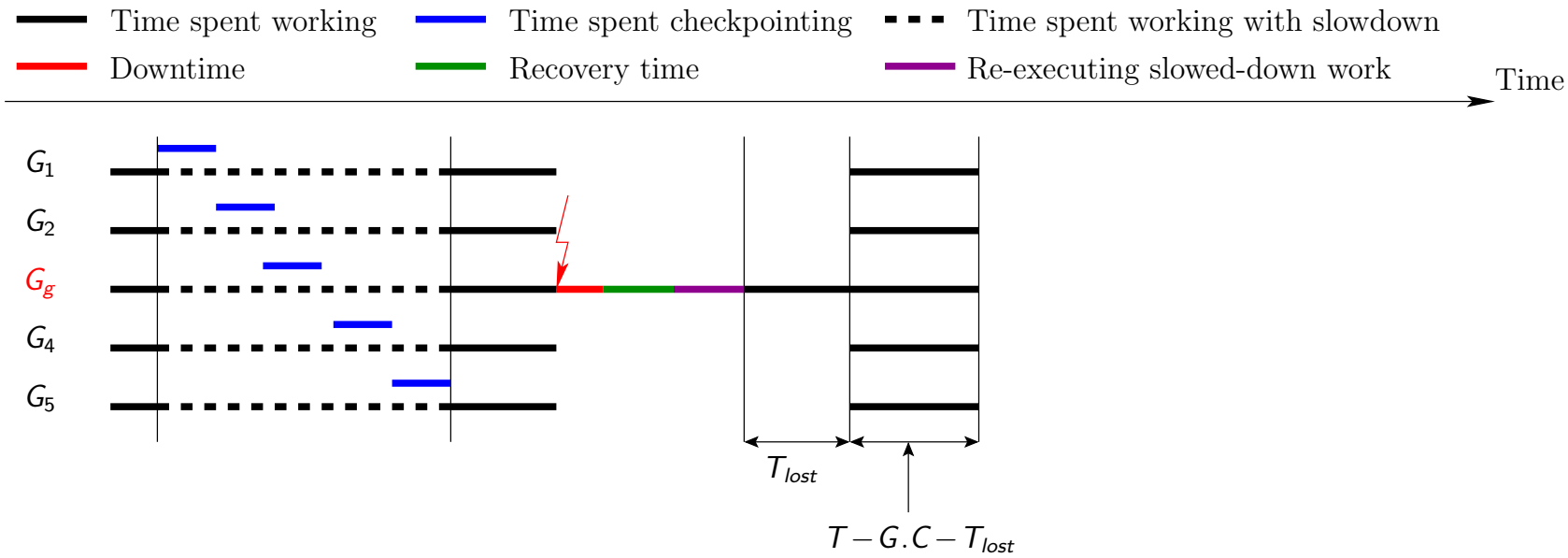
But no checkpoint is taken in parallel, hence this re-execution is faster than the original computation

Failure during computation phase



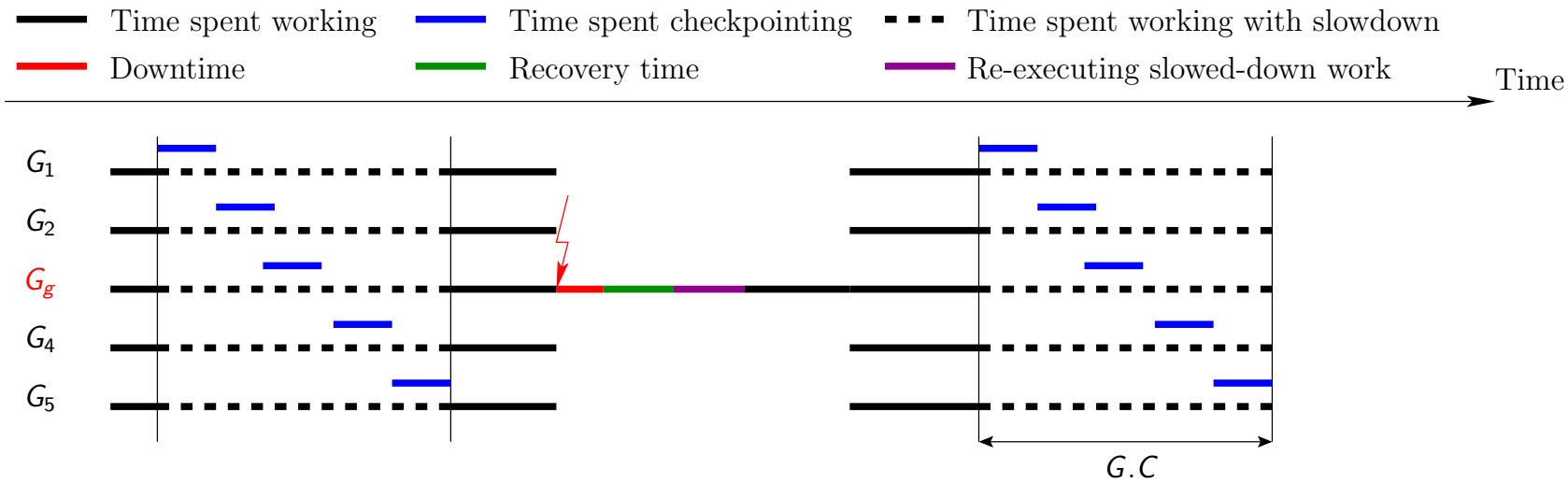
Redo work done in computation phase and that was destroyed by the failure

Failure during computation phase



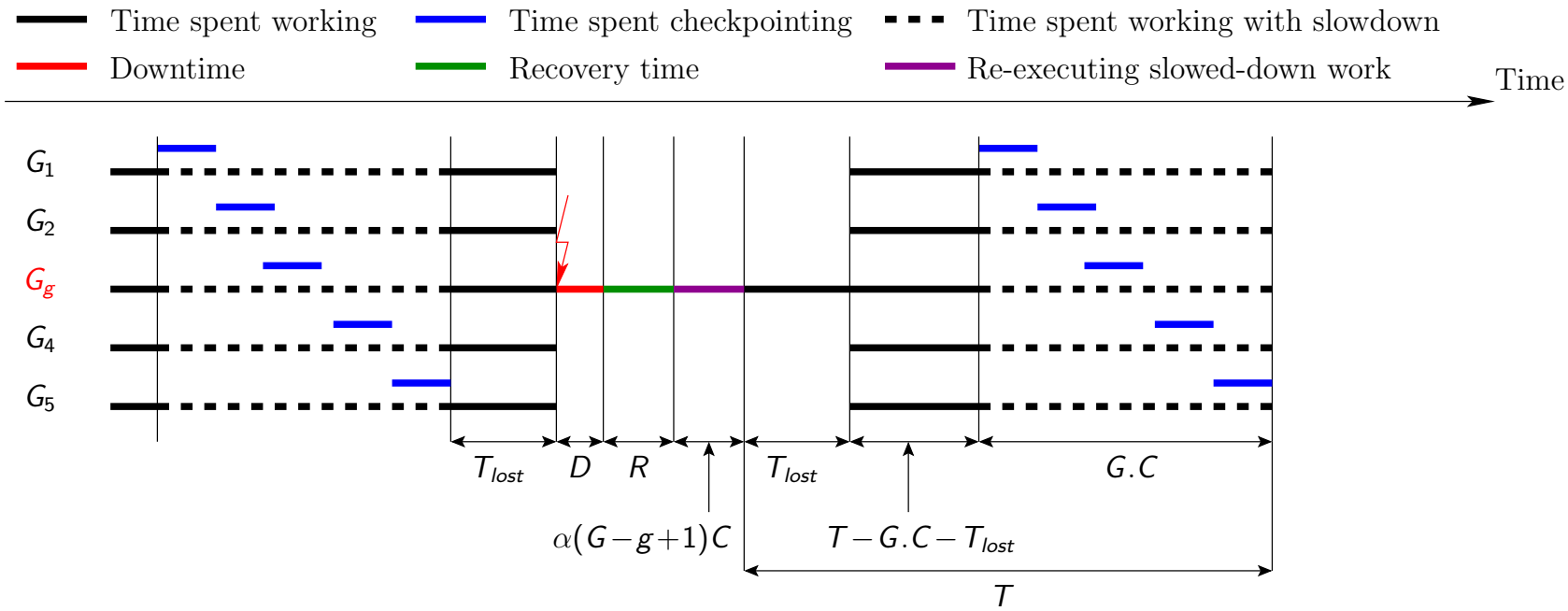
Failing group g has reached the point where it was struck.
All groups now resume execution in parallel and complete the computation phase

Failure during computation phase



Finally, perform checkpointing phase

Failure during computation phase

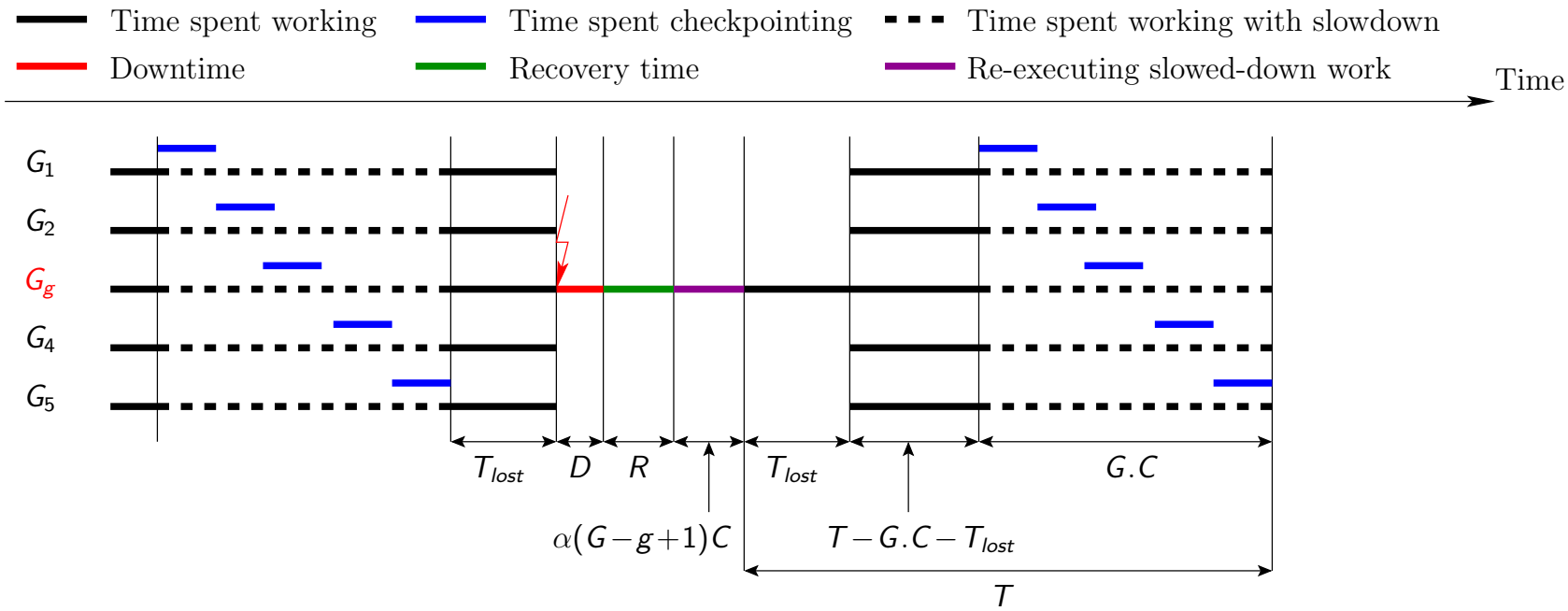


RE-EXEC: $T_{lost} + \alpha(G - g + 1)C$

Expectation: $T_{lost} = \frac{1}{2}(T - G.C)$

Expected RE-EXEC: $\frac{T-G.C}{2} + \alpha(G - g + 1)C$

Failure during computation phase



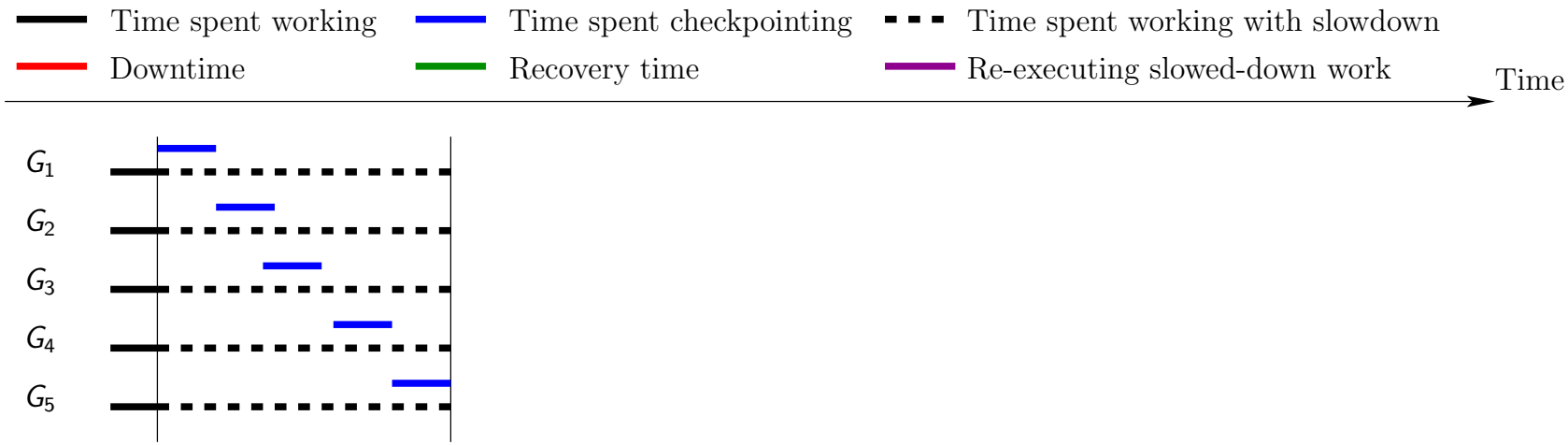
Expected RE-EXEC: $\frac{T-G.C}{2} + \alpha(G-g+1)C$

Averaging over groups: $\text{RE-EXEC}_{comp} =$

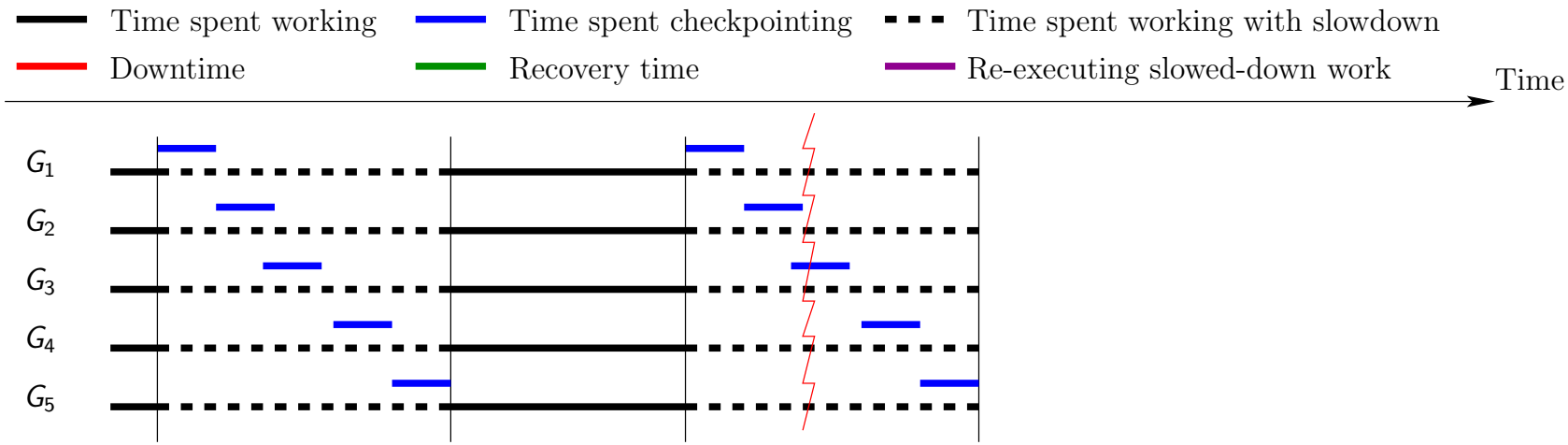
$$\frac{1}{G}\sum_{g=1}^{G}\left[\frac{T-G.C(q)}{2} + \alpha(G-g+1)C(q)\right]$$

$$= \frac{T-G.C(q)}{2} + \alpha\frac{G+1}{2}C$$

Failure during checkpointing phase



Failure during checkpointing phase



When does the failing group fail?

1. Before starting its own checkpoint
2. While taking its own checkpoint
3. After completing its own checkpoint

Average waste for failures during checkpointing phase

- Compute expected RE-EXEC when group g fails
- Average over groups:

$$\frac{1}{G}((g-1).\text{RE-EXEC}_{before\_ckpt} + 1.\text{RE-EXEC}_{during\_ckpt} + (G-g).\text{RE-EXEC}_{after\_ckpt})$$

Final result (skipping details):

$$\text{RE-EXEC}_{ckpt} = \frac{G+1}{2G}T + \frac{\alpha C(q)(G+3)}{2} + \frac{C(q)(1-2\alpha)}{2G} - \frac{C(q)(G+1)}{2}$$

Total waste

$$\textsc{Waste}[FF] = \frac{T - \textsc{Work}}{T} \text{ with } \textsc{Work} = T - (1-\alpha)GC(q)$$

$$\textsc{Waste}[fail] = \frac{1}{\mu}\left(D(q) + R(q) + \textsc{Re-Exec}\right) \text{ with}$$

$$\textsc{Re-Exec} = \frac{T - GC(q)}{T}\textsc{Re-Exec}_{comp} + \frac{GC(q)}{T}\textsc{Re-Exec}_{ckpt}$$

$$\textsc{Waste} = \textsc{Waste}[FF] + \textsc{Waste}[fail] - \textsc{Waste}[FF]\textsc{Waste}[fail]$$

Minimize $\textsc{Waste}$ subject to:

- $GC(q) \leq T$ (by construction)
- $T \leq \gamma\mu$ (capping period as before)
- Gets complicated! Use computer algebra software ☹

Accounting for message logging: Impact on work

- ☹ Logging messages slows down execution:
  $\Rightarrow$ WORK becomes λWORK, where $0 < \lambda < 1$
  Typical value: $\lambda \approx 0.98$
- ☺ Re-execution after a failure is faster:
  $\Rightarrow$ RE-EXEC becomes $\frac{\text{RE-EXEC}}{\rho}$, where $\rho \in [1..2]$
  Typical value: $\rho \approx 1.5$

$$\text{WASTE}[FF] = \frac{T - \lambda\text{WORK}}{T}$$

$$\text{WASTE}[fail] = \frac{1}{\mu}\left(D(q) + R(q) + \frac{\text{RE-EXEC}}{\rho}\right)$$

Accounting for message logging: Impact on checkpoint size

- Inter-groups messages logged continuously
- Checkpoint size increases with amount of work executed before a checkpoint
- $C_0(q)$: Checkpoint size of a group without message logging

$$C(q) = C_0(q)(1 + \beta \textsc{Work}) \Leftrightarrow \beta = \frac{C(q) - C_0(q)}{C_0(q)\textsc{Work}}$$

$$\textsc{Work} = \lambda(T - (1 - \alpha)GC(q))$$

$$C(q) = \frac{C_0(q)(1 + \beta\lambda T)}{1 + GC_0(q)\beta\lambda(1 - \alpha)}$$

- Constraint $GC(q) \leq T$ translates into

$$GC_0(q)\beta\lambda\alpha \leq 1 \text{ and } T \geq \frac{GC_0(q)}{1 - GC_0(q)\beta\lambda\alpha}$$

Three case studies

Coord-IO

Coordinated approach: $C = C_{\text{Mem}} = \frac{\text{Mem}}{\text{b}_{io}}$
where Mem is the memory footprint of the application

Hierarch-IO

Several (large) groups, *I/O-saturated*
$\Rightarrow$ groups checkpoint sequentially

$$C_0(q) = \frac{C_{\text{Mem}}}{G} = \frac{\text{Mem}}{G\text{b}_{io}}$$

Hierarch-Port

Very large number of smaller groups, *port-saturated*
$\Rightarrow$ some groups checkpoint in parallel
Groups of q_{min} processors, where $\text{q}_{\text{min}}\text{b}_{port} \geq \text{b}_{io}$

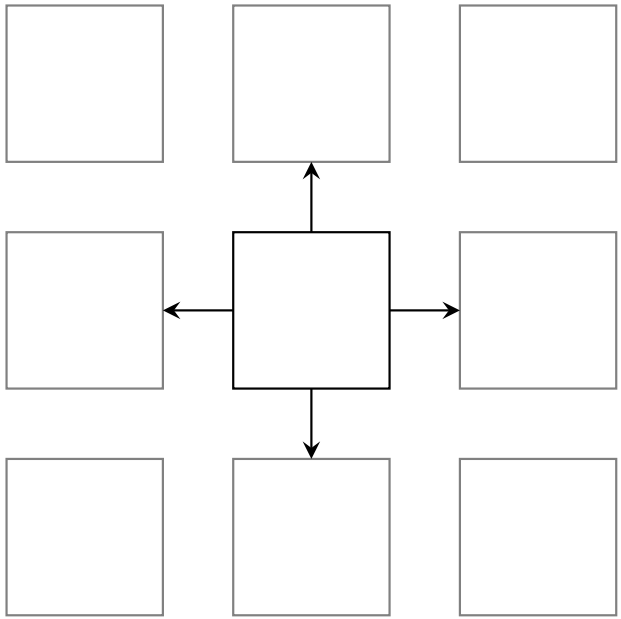
Three applications

1. 2D-stencil
2. Matrix product
3. 3D-Stencil
   - Plane
   - Line

Computing β for 2D-Stencil

$$C(q) = C_0(q) + Logged\_Msg = C_0(q)(1 + \beta \text{WORK})$$

Real $n \times n$ matrix and $p \times p$ grid
$Work = \frac{9b^2}{s_p}$, $b = n/p$
Each process sends a block to its 4 neighbors



HIERARCH-IO:

- 1 group = 1 grid row
- 2 out of the 4 messages are logged
- $\beta = \frac{2s_p}{9b^3}$

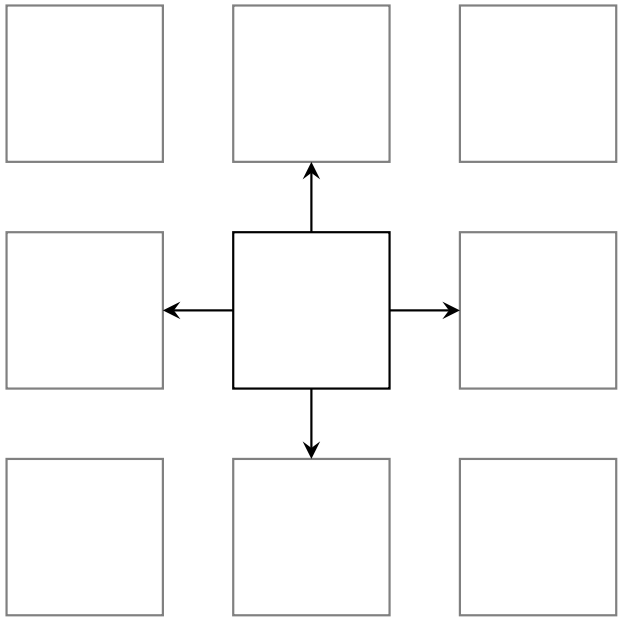
HIERARCH-PORT:

- β doubles

Computing β for 2D-Stencil

$$C(q) = C_0(q) + Logged\_Msg = C_0(q)(1 + \beta \text{WORK})$$

Real $n \times n$ matrix and $p \times p$ grid
$Work = \frac{9b^2}{s_p}$, $b = n/p$
Each process sends a block to its 4 neighbors



HIERARCH-IO:

- 1 group = 1 grid row
- 2 out of the 4 messages are logged
- $\beta = \frac{2s_p}{9b^3}$

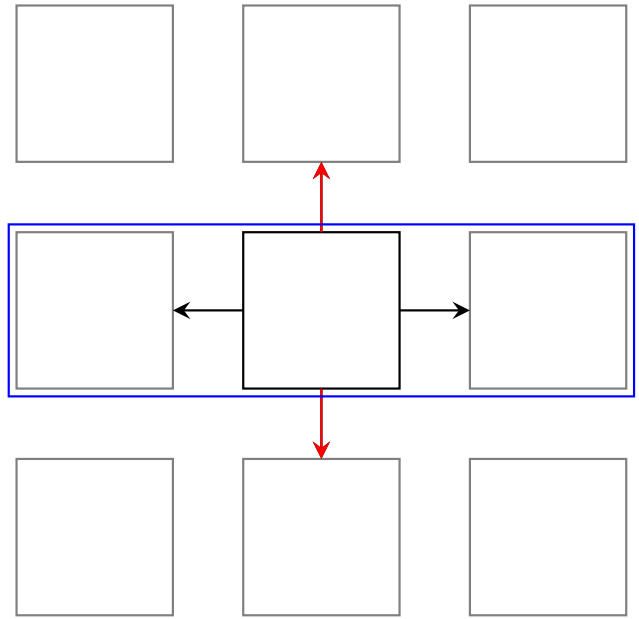
HIERARCH-PORT:

- β doubles

Computing β for 2D-Stencil

$$C(q) = C_0(q) + Logged\_Msg = C_0(q)(1 + \beta \text{WORK})$$

Real $n \times n$ matrix and $p \times p$ grid
$Work = \frac{9b^2}{s_p}$, $b = n/p$
Each process sends a block to its 4 neighbors



HIERARCH-IO:

- 1 group = 1 grid row
- 2 out of the 4 messages are logged
- $\beta = \frac{2s_p}{9b^3}$

HIERARCH-PORT:

- β doubles

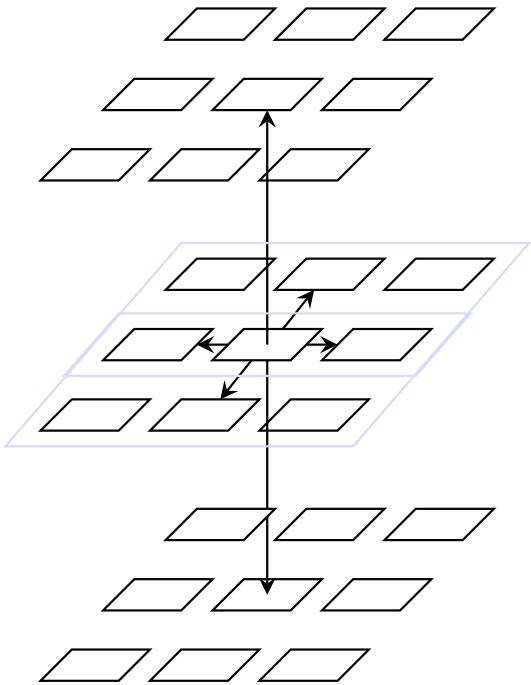
Computing β for Matrix Product

- 3 matrices of size $n \times n$ partitioned across a $p \times p$ processor grid
- Mem $= 24n^2$ (in bytes)
- Each processor holds three matrix blocks of size $b = n/p$
- At each iteration (Cannon's algorithm):
  - shift one block vertically and one horizontally
  - perform a matrix product
- (Parallel) work for one iteration is $\text{WORK} = \frac{2b^3}{s_p}$

1. HIERARCH-IO: one group per grid row: $\beta = \frac{s_p}{6b^3}$
2. HIERARCH-PORT: groups of size q_{min}: $\beta = \frac{s_p}{3b^3}$

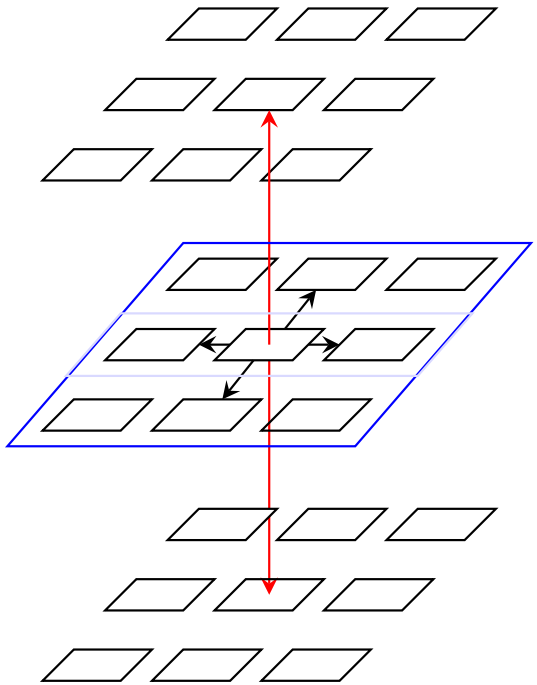
Computing β for 3D-Stencil

- Each processor has a cube, $b = n/p$
  Parallel work: $\text{WORK} = \frac{27b^3}{s_p}$
- HIERARCH-IO-PLANE:
  group = horizontal plane of size p^2
  $\beta = \frac{2s_p}{27b^3}$
- HIERARCH-IO-LINE:
  group = horizontal line of size p
  $\beta = \frac{4s_p}{27b^3}$
- HIERARCH-PORT: groups of size q_{min}:
  $\beta = \frac{6s_p}{27b^3}$



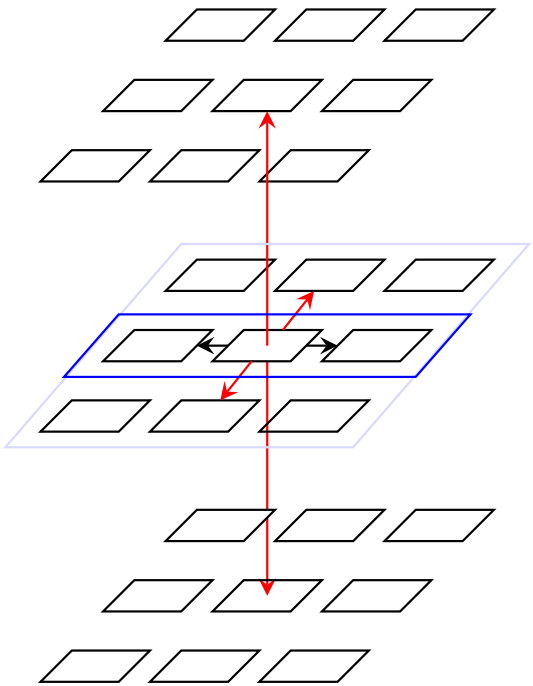
Computing β for 3D-Stencil

- Each processor has a cube, $b = n/p$
  Parallel work: $\text{WORK} = \frac{27b^3}{s_p}$
- HIERARCH-IO-PLANE:
  group = horizontal plane of size p^2
  $\beta = \frac{2s_p}{27b^3}$
- HIERARCH-IO-LINE:
  group = horizontal line of size p
  $\beta = \frac{4s_p}{27b^3}$
- HIERARCH-PORT: groups of size q_{min}:
  $\beta = \frac{6s_p}{27b^3}$



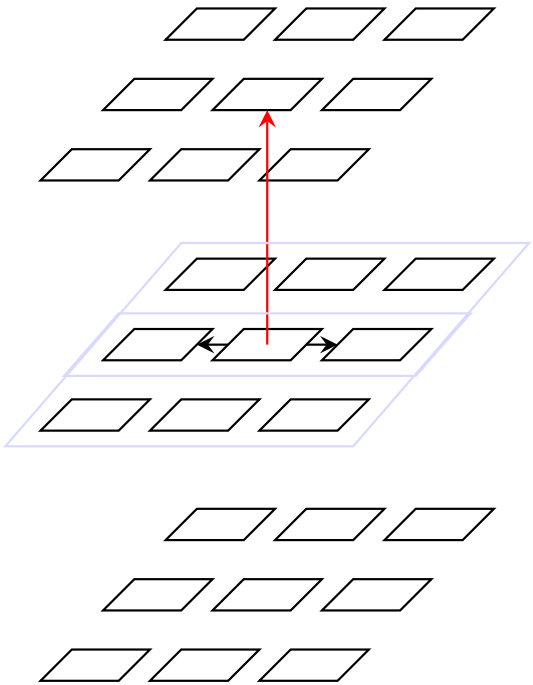
Computing β for 3D-Stencil

- Each processor has a cube, $b = n/p$
  Parallel work: $\text{WORK} = \frac{27b^3}{s_p}$
- HIERARCH-IO-PLANE:
  group = horizontal plane of size p^2
  $\beta = \frac{2s_p}{27b^3}$
- HIERARCH-IO-LINE:
  group = horizontal line of size p
  $\beta = \frac{4s_p}{27b^3}$
- HIERARCH-PORT: groups of size q_{min}:
  $\beta = \frac{6s_p}{27b^3}$



Computing β for 3D-Stencil

- Each processor has a cube, $b = n/p$
  Parallel work: $\text{WORK} = \frac{27b^3}{s_p}$
- HIERARCH-IO-PLANE:
  group = horizontal plane of size p^2
  $\beta = \frac{2s_p}{27b^3}$
- HIERARCH-IO-LINE:
  group = horizontal line of size p
  $\beta = \frac{4s_p}{27b^3}$
- HIERARCH-PORT: groups of size q_{min}:
  $\beta = \frac{6s_p}{27b^3}$



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

Four platforms: 2D-Stencil and Matrix-Product

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

Four platforms: 2D-Stencil and Matrix-Product

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

Four platforms: 3D-Stencil

Name	Scenario	G	β for 3D-Stencil
Titan	Coord-IO	1	/
	Hierarch-IO-Plane	26	0.001476
	Hierarch-IO-Line	675	0.002952
	Hierarch-Port	1,246	0.004428
K-Computer	Coord-IO	1	/
	Hierarch-IO-Plane	44	0.003422
	Hierarch-IO-Line	1,936	0.006844
	Hierarch-Port	17,626	0.010266
Exascale-Slim	Coord-IO	1	/
	Hierarch-IO-Plane	100	0.003952
	Hierarch-IO-Line	10,000	0.007904
	Hierarch-Port	200,000	0.011856
Exascale-Fat	Coord-IO	1	/
	Hierarch-IO-Plane	46	0.001834
	Hierarch-IO-Line	2,116	0.003668
	Hierarch-Port	33,333	0.005502

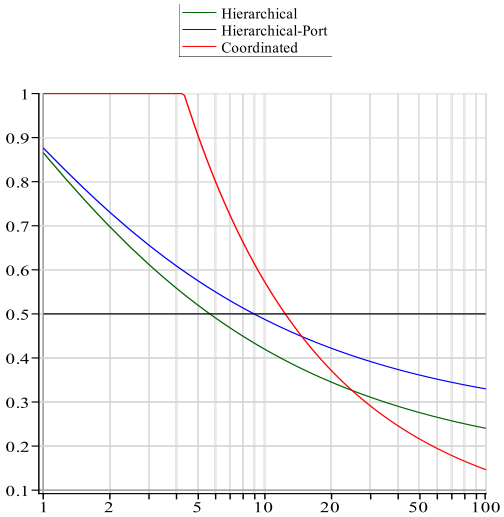
Four platforms: 3D-Stencil

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

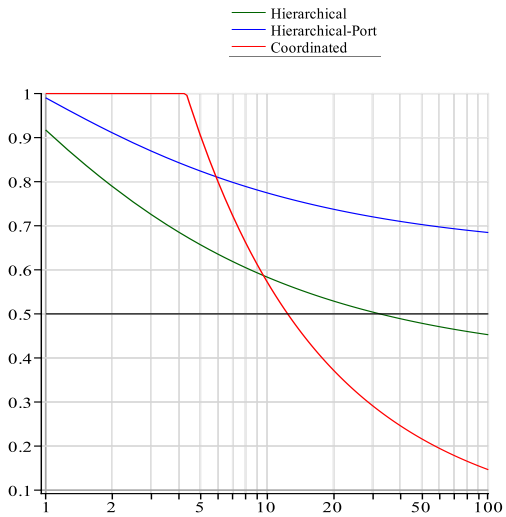
- Large time to dump the memory
- Using $1\%C$
- Comparing with $0.1\%C$ for exascale platforms
- $\alpha = 0.3$, $\lambda = 0.98$ and $\rho = 1.5$

Plotting formulas – Platform: Titan

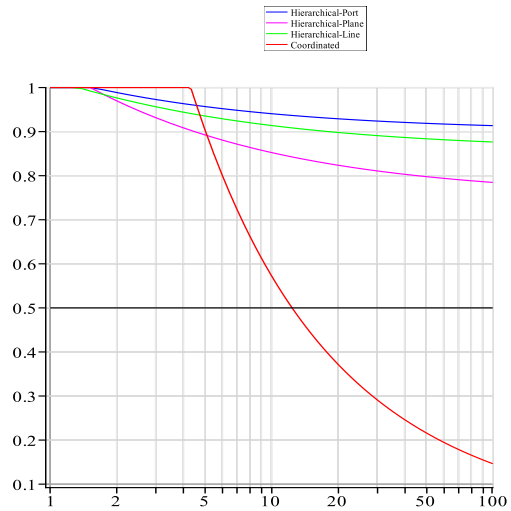
Stencil 2D



Matrix product



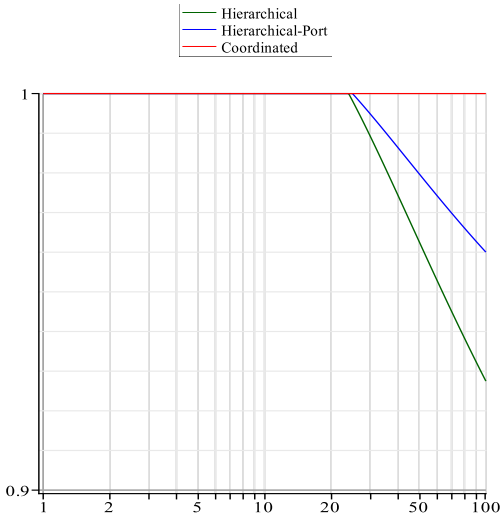
Stencil 3D



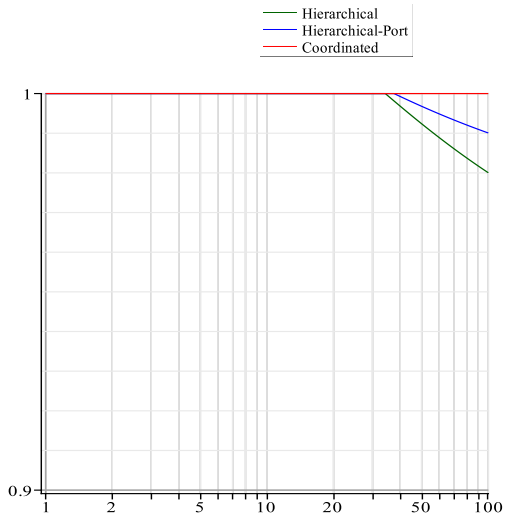
Waste as a function of processor MTBF μ

Platform: K-Computer

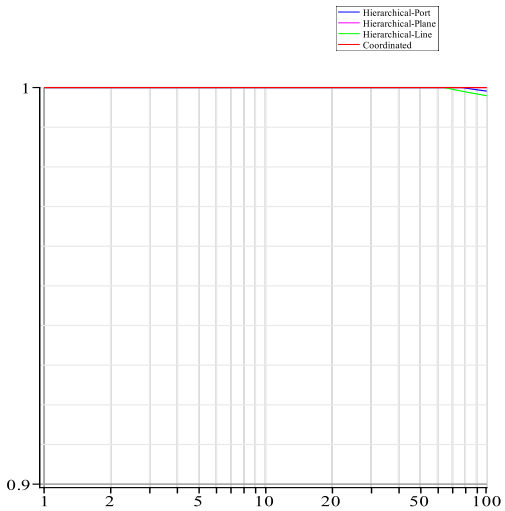
Stencil 2D



Matrix product



Stencil 3D



Waste as a function of processor MTBF μ

Plotting formulas – Platform: Exascale

$\textsc{Waste} = 1$ for all scenarios!!!

Plotting formulas – Platform: Exascale

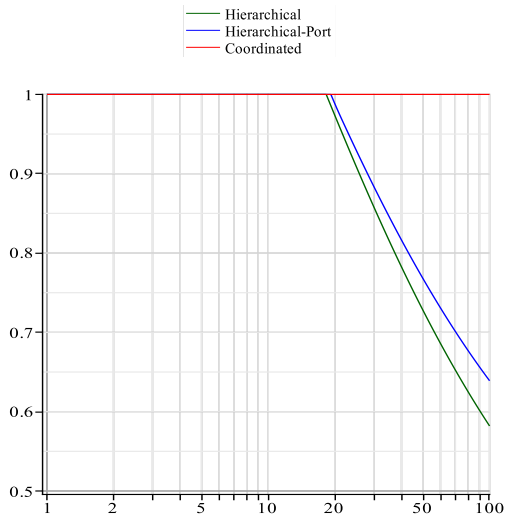
WASTE = ... for all s... narios!!!

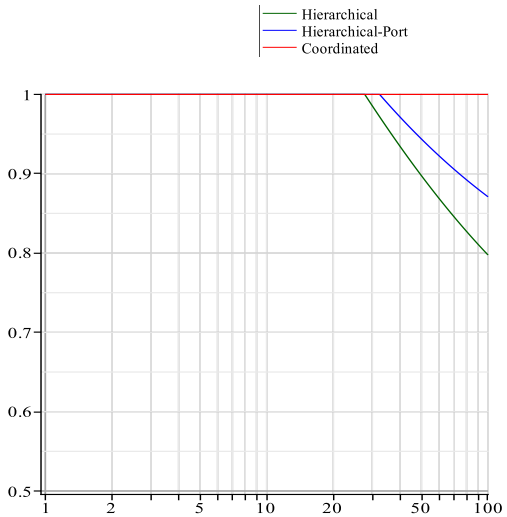
Goodbye Exascale?!

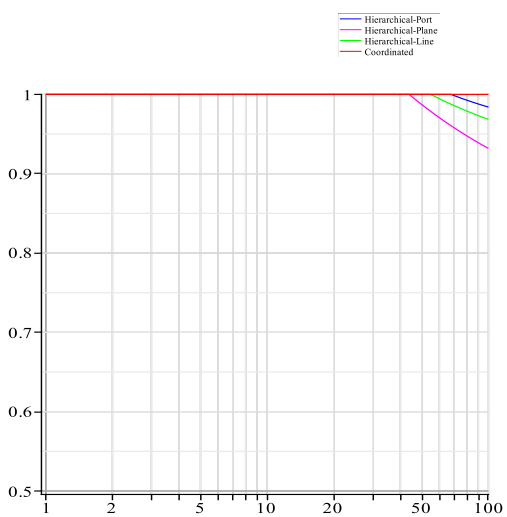
Plotting formulas – Platform: Exascale with $C = 1,000$

Stencil 2D | Matrix product | Stencil 3D

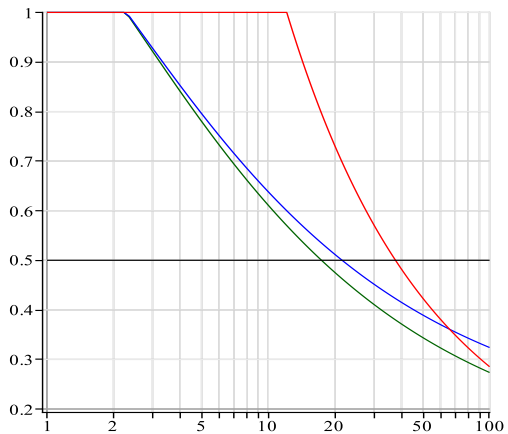
Exascale-Slim

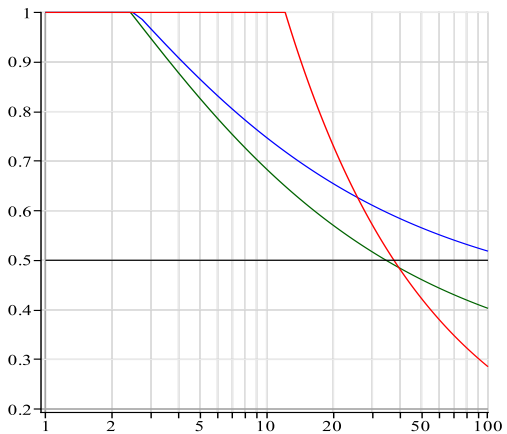


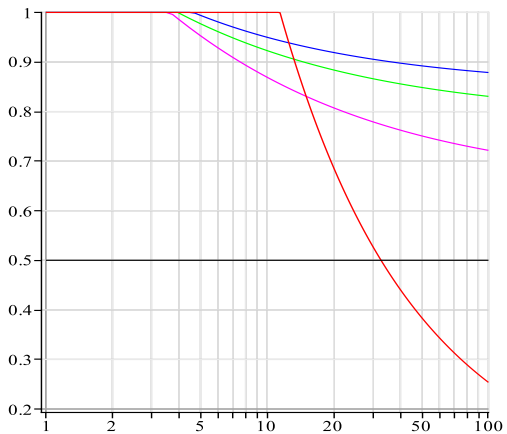




Exascale-Fat





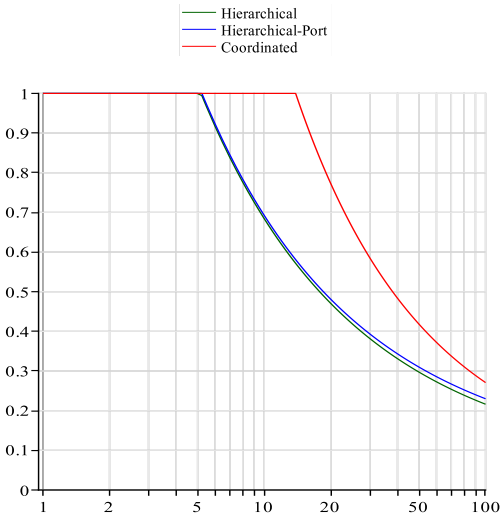


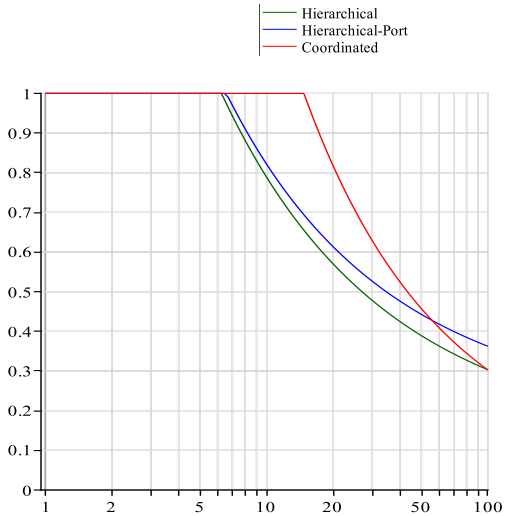
Waste as a function of processor MTBF μ, $C = 1,000$

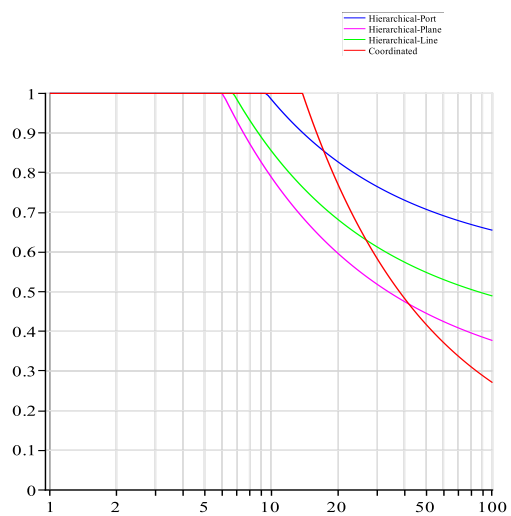
Plotting formulas – Platform: Exascale with $C = 100$

Stencil 2D | Matrix product | Stencil 3D

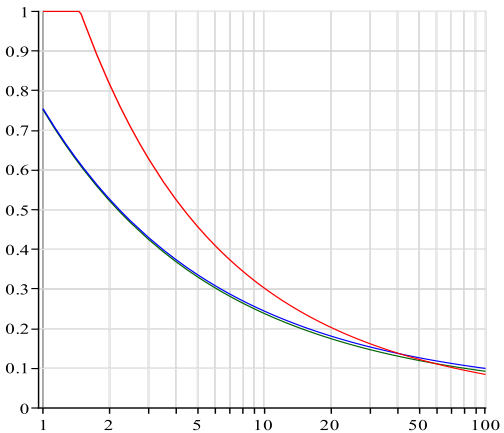
Exascale-Slim

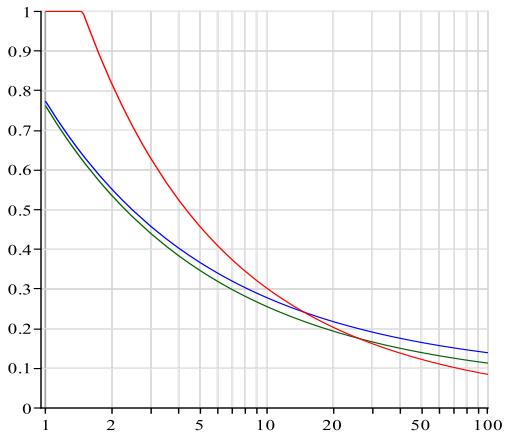


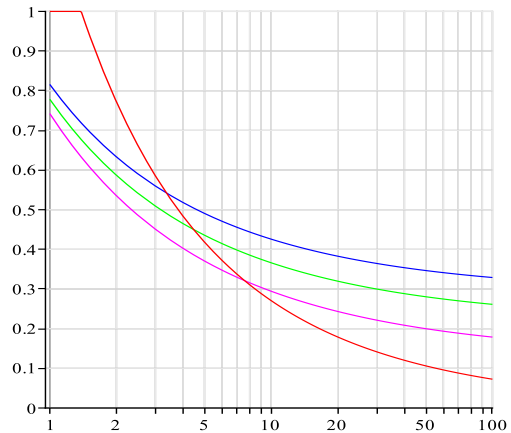




Exascale-Fat



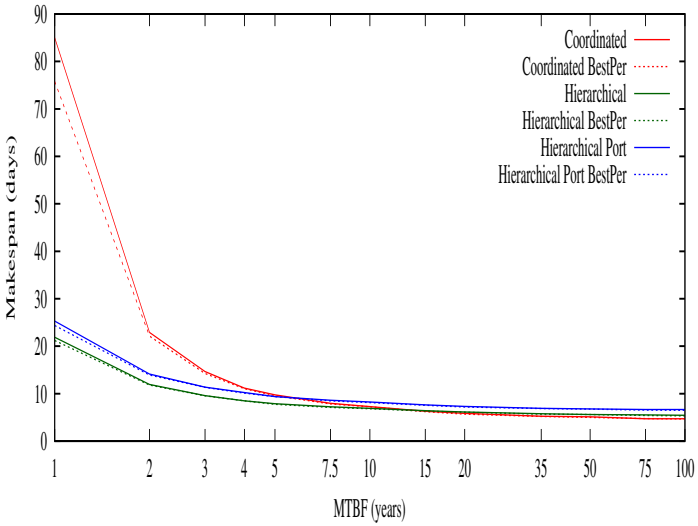




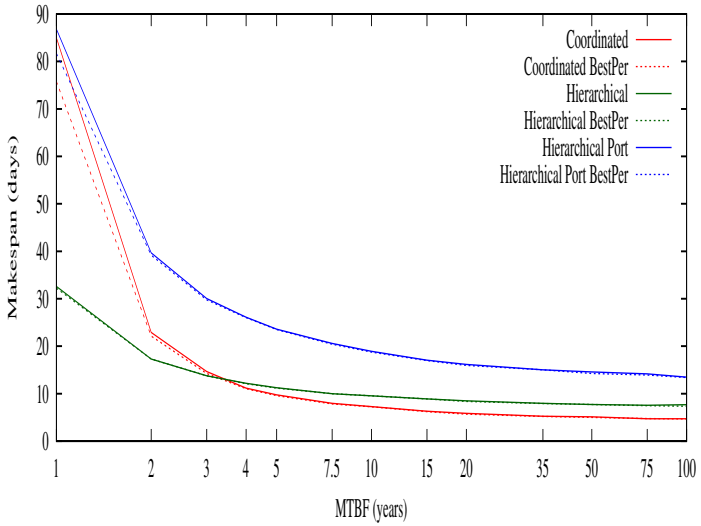
Waste as a function of processor MTBF μ, $C = 100$

Simulations – Platform: Titan

Stencil 2D



Matrix product



Makespan (in days) as a function of processor MTBF μ

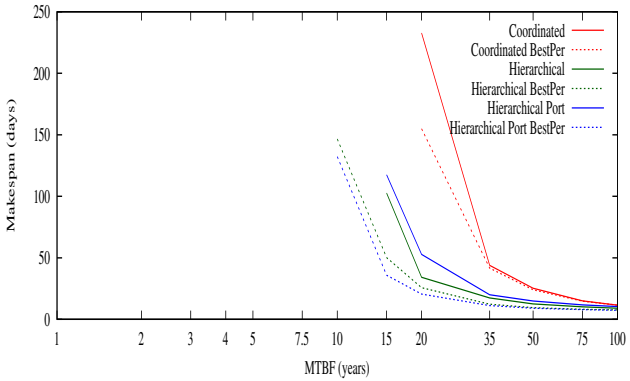
Simulations – Platform: Exascale with $C = 1,000$

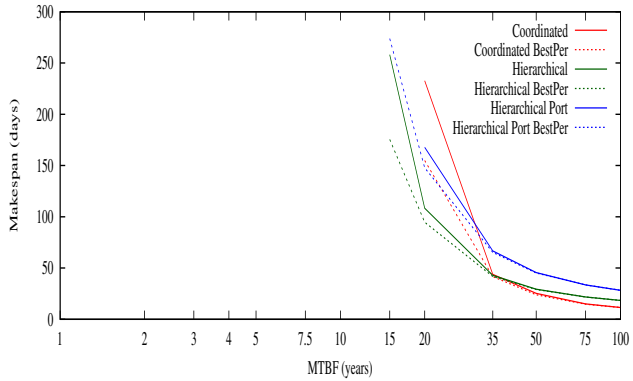
Stencil 2D

Matrix product

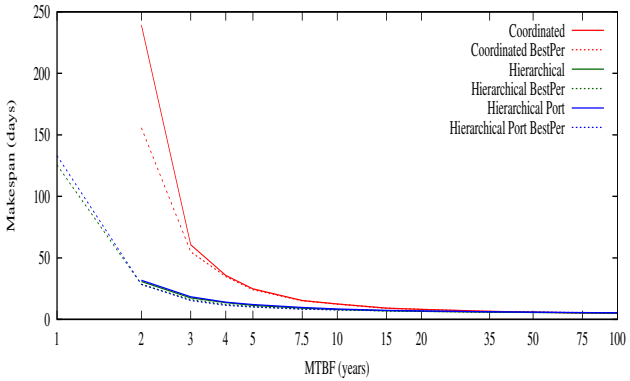
Coordinated
Coordinated BestPer
Hierarchical
Hierarchical BestPer
Hierarchical Port
Hierarchical Port BestPer

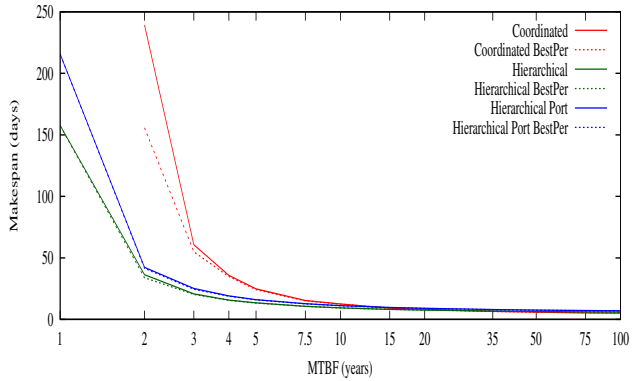
Exascale-Slim





Exascale-Fat





Makespan (in days) as a function of processor MTBF μ, $C = 1,000$

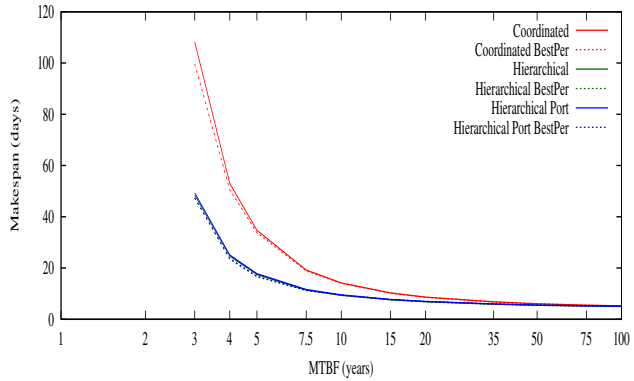
Simulations – Platform: Exascale with $C = 100$

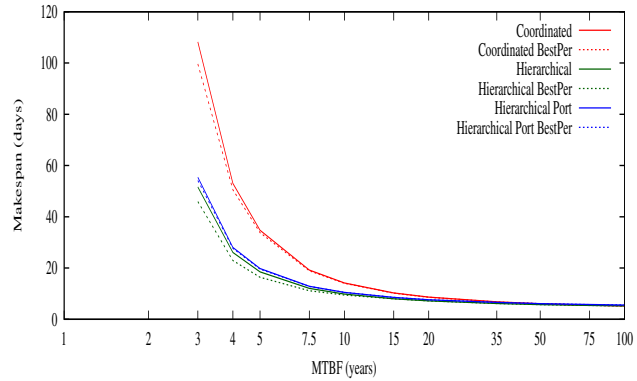
Stencil 2D

Matrix product

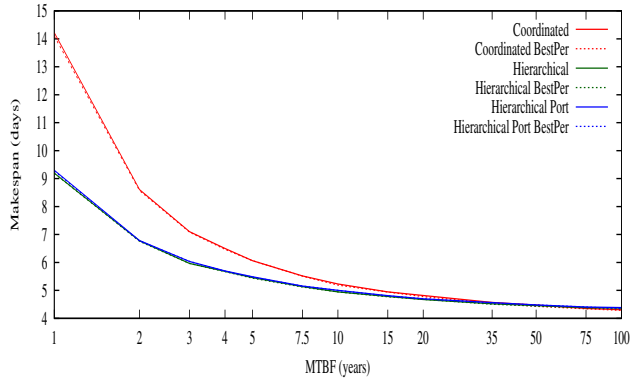
Coordinated
Coordinated BestPer
Hierarchical
Hierarchical BestPer
Hierarchical Port
Hierarchical Port BestPer

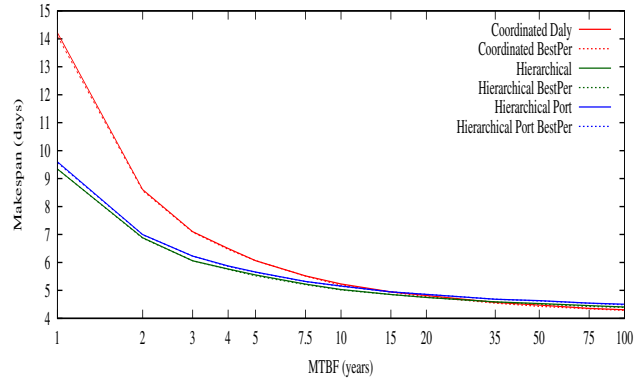
Exascale-Slim





Exascale-Fat

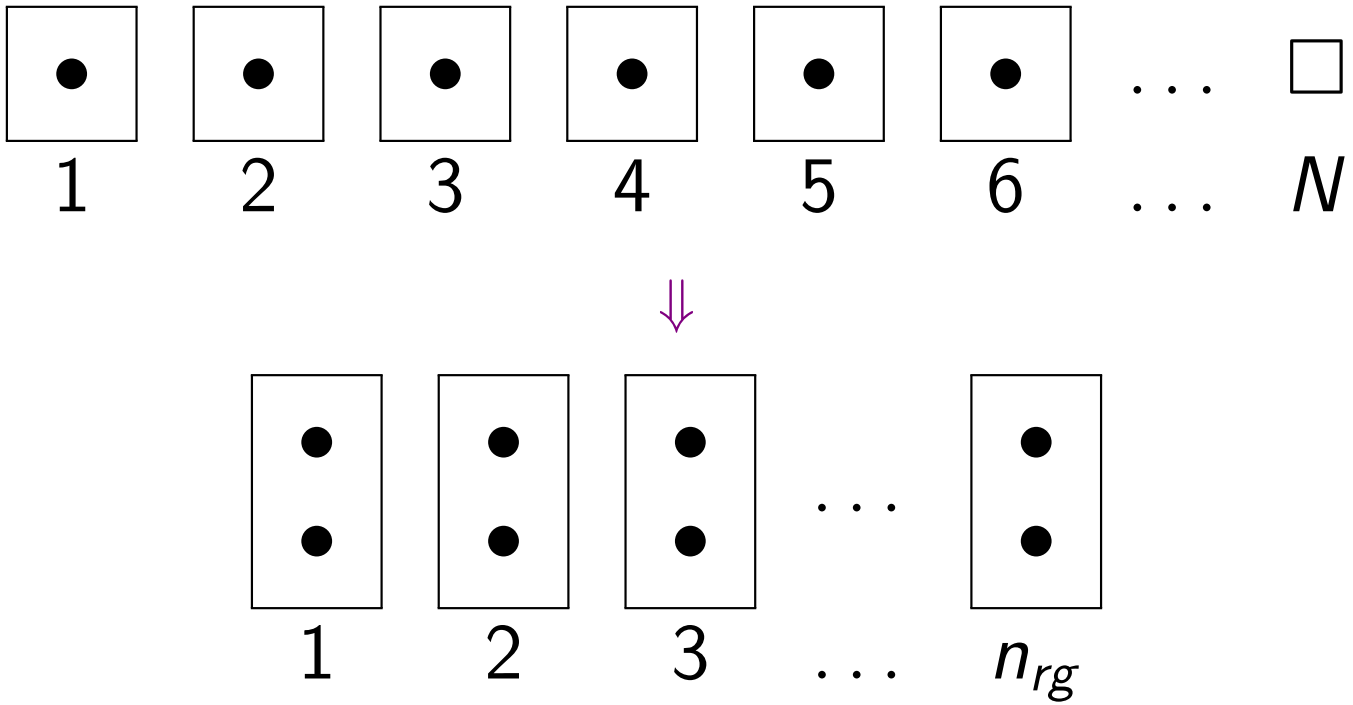




Makespan (in days) as a function of processor MTBF μ, $C = 100$

Outline

Process replication



- Each process replicated $g \geq 2$ times $\rightarrow$ *replica-group*
- n_{rg} = number of replica-groups ($g \times n_{rg} = N$)
- Study for $g = 2$ by Ferreira et al., SC'2011

Number of failures to bring down application

- $MNFTI^{\mathrm{ah}}$ Count each failure hitting any of the $g \cdot n_{rg}$ initial processors, including those *already hit* by a failure
- $MNFTI^{\mathrm{rp}}$ Count failures that hit *running processors*, and thus effectively kill replicas.

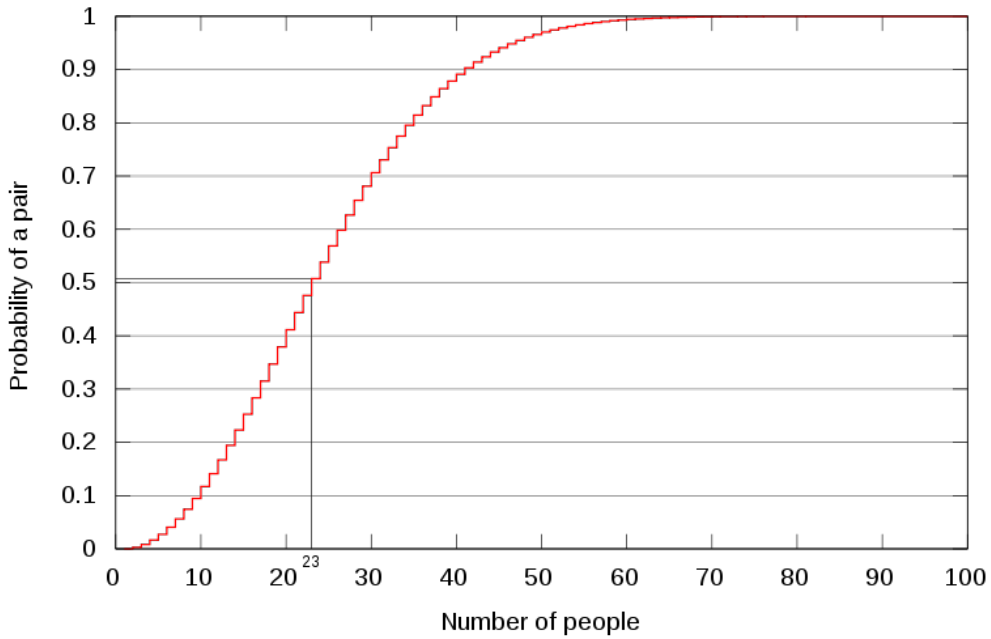
$$MNFTI^{\mathrm{ah}} = 1 + MNFTI^{\mathrm{rp}}$$

Number of failures to bring down application

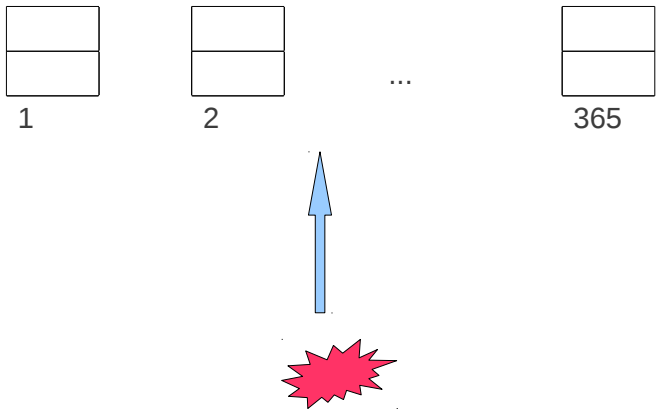
- $MNFTI^{\text{ah}}$ Count each failure hitting any of the $g \cdot n_{rg}$ initial processors, including those *already hit* by a failure
- $MNFTI^{\text{rp}}$ Count failures that hit *running processors*, and thus effectively kill replicas.

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

Analogy with birthday problem

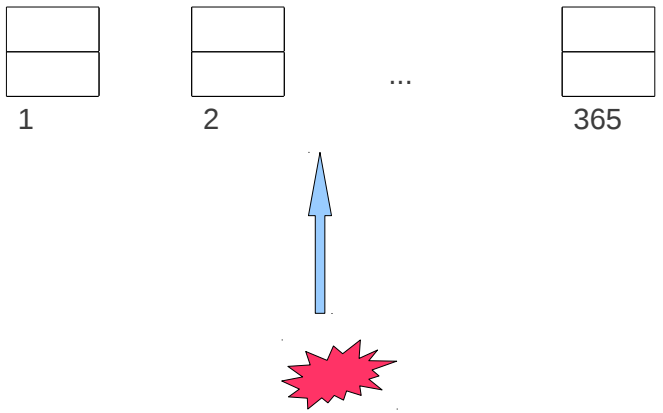


Analogy with birthday problem



$n = n_{rg}$ bins, throw balls until one bin gets two balls

Analogy with birthday problem

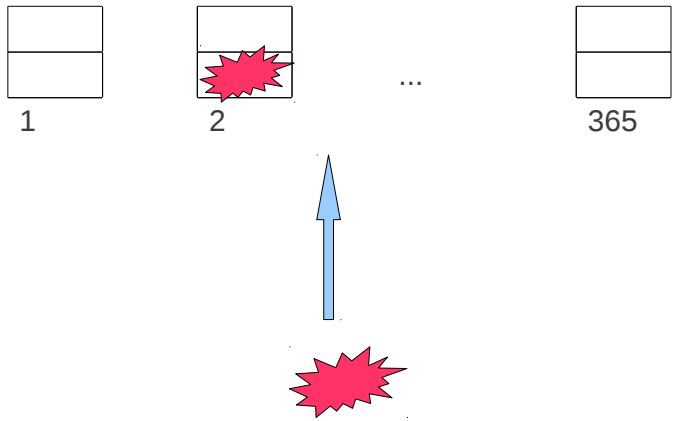


$n = n_{rg}$ bins, throw balls until one bin gets two balls

Expected number of balls to throw:

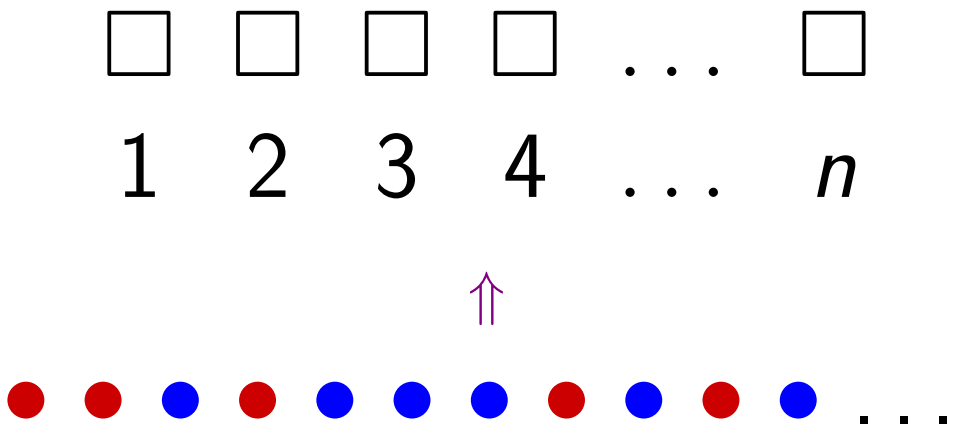
$$Birthday(n) = 1 + \int_0^{+\infty} e^{-x}(1 + x/n)^{n-1} dx$$

Analogy with birthday problem



But second failure may hit already struck replica ☹

Analogy with birthday problem



$n = n_{rg}$ bins, red and blue balls

$MNFTI^{\text{ah}}$ = expected number of balls to throw until one bin gets one ball of each color

Exponential failures, $g = 2$

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

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

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

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

Exponential failures, $g = 2$ (cont'd)

Proof

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

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

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

Exponential failures, $g = 2$ (cont'd)

Proof

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

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

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

Exponential failures, $g = 2$ (cont'd)

Proof

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

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

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

Failure distribution

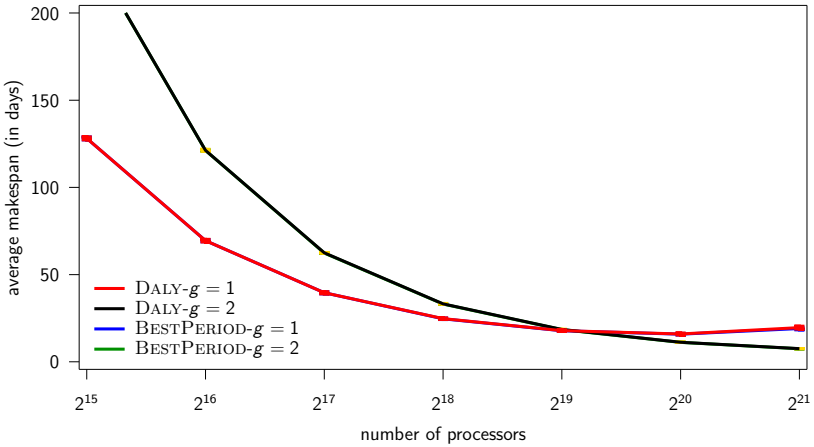
$R(t)$ probability that application still running at time t

- All replica-groups have at least one replica running
- Exponential: $R(t) = \left(1 - \left(1 - e^{-\lambda t}\right)^g\right)^{n_{rg}}$
- Weibull: $R(t) = \left(1 - \left(1 - e^{-\left(\frac{t}{\lambda}\right)^k}\right)^g\right)^{n_{rg}}$
- Can use dynamic programming algorithms for sequential/parallel jobs

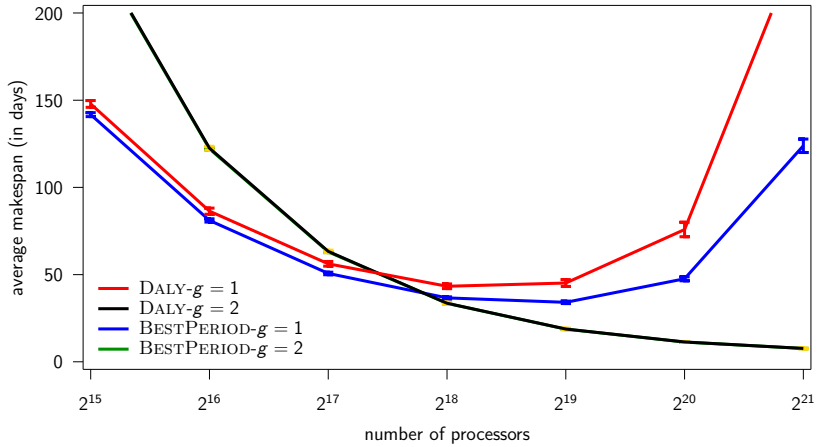
MTTI

- $MTTI = \int_0^{+\infty} R(t)\mathrm{d}t \rightarrow$ closed-form formulas
- Assess the impact of replication for various scenarios ☺

Failure distribution



(a) Exponential



(b) Weibull, $k = 0.7$

Crossover point for replication when $\mu = 125$ years

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-3.x 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 ☺

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 ☺

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

Exascale
• Toward Exascale Resilience, Cappello F. et al., IJHPCA 23, 4 (2009)
• The International Exascale Software Roadmap, Dongarra, J., Beckman, P. et al., IJHPCA 25, 1 (2011)

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

Coordinated Checkpointing Distributed snapshots: determining global states of distributed systems, Chandy K.M., Lamport L., ACM Trans. Comput. Syst. 3, 1 (1985)

Message Logging A survey of rollback-recovery protocols in message-passing systems, Elnozahy E.N. et al., ACM Comput. Surveys 34, 3 (2002)

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

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