

Fault Tolerance & PetaScale Systems: Current Knowledge, Challenges and Opportunities

Franck Cappello

INRIA

fci@lri.fr

CCGSC Workshop, September 2008, Asheville, USA

40 ans
la révolution de l'information

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

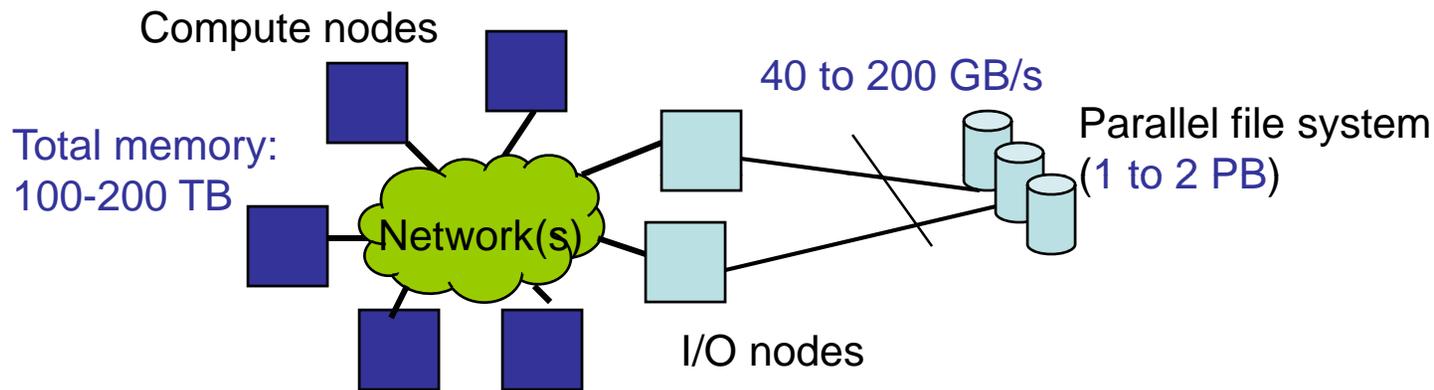
 INRIA

Agenda

- Why Fault Tolerance is Challenging?
- What are the main reasons behind failures?
- Rollback Recovery Protocols?
- Reducing Rollback Recovery Time?
- Rollback Recovery without stable storage?
- Alternatives to Rollback Recovery?
- Where are the opportunities?

Classic approach for FT: Checkpoint-Restart

Typical “Balanced Architecture” for PetaScale Computers



QuickTime™ and a TIFF (Uncompressed) decompressor are needed to see this picture.



TACC RoadRunner

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

Roac			Ckpt-rest on local disc does not help to reduce the ckpt time --> need to save (async.) checkpoint on stable storage.
LLNL			Restarting a failed node from local storage does help since it needs 10 to 100 hours (MTTR) in case of hardware failure.
XXX			
IDRIS BG/P	100 TF	<u>~30 min</u>	IDRIS



Failure rate and #sockets

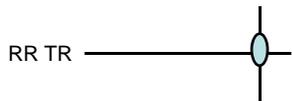
In Top500 machine performance X2 per year (See Jack slide on top500)

--> more than Moore's law and increase of #cores in CPUs

If we consider #core X 2, every 18, 24 and 30 months AND fixed Socket MTTI:

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

Figures from
Garth Gibson



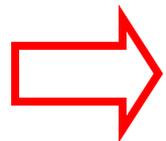
1h. wall



We may reach the 1h. wall as soon as in 2012-2013

Another projection from CHARNG-DA LU gives similar results

Projections may be correct or wrong! Can we take the risk of not considering them?



It's urgent to optimize Rollback-Recovery for PetaScale systems and to investigate alternatives.

Understanding Approach: Failure logs

- The computer failure data repository (CFDR)
 - <http://cfd.r.usenix.org/>
 - From 96 until now...
 - HPC systems+Google
- failure logs from LANL, NERSC, PNNL, ask.com, SNL, LLNL, etc.
 - ex: LANL released root cause logs for: 23000 events causing apps. Stop on 22 Clusters (5000 nodes), over 9 years

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

What are the main reasons of failures?

- 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): 

Hardware

50%

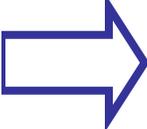
QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

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

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

Conclusion1: Both Hardware and Software failures have to be considered

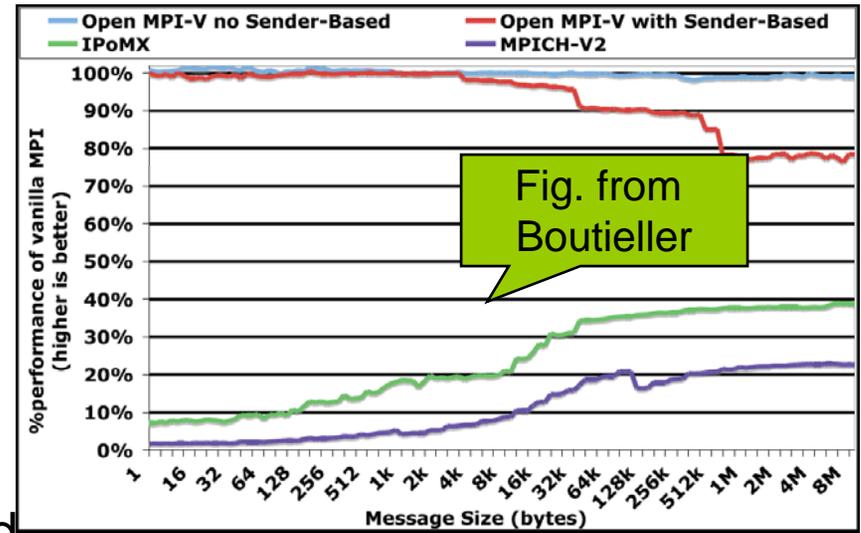
Conclusion2: Oliner: logging tools fail too, some key info is missing, better filtering (correlation) is needed

 **FT system should cover all causes of failures**
(Rollback Recovery is consistent with this requirement*)

Roll-Back Recovery Protocols?

- Classic approach (**MPICH-V**) implements **Message Logging at the device level**: all messages are copied
- High speed MPI implementations use **Zero Copy** and decompose Recv in:
a) Matching, b) Delivery
- **OpenMPI-V** implements **Mes. Log. within MPI**: different event types are managed differently, distinction between determ. and non determ. events, optimized mem. copy

Bandwidth of OpenMPI-V compared to others



OpenMPI-V Overhead on NAS (Myri 10g)



⇒ Coordinated and message logging protocols have been improved --> improvements are probably still possible but very difficult to obtain!

dt.c.b4 cg.c.b4 lu.c.b4 mg.c.b4 sp.c.b4

NAS Kernel

Reducing Checkpoint size 1/2

- **Incremental Checkpointing:**

A runtime monitor detects memory regions that have not been modified between two adjacent CKPT. and omit them from the subsequent CKPT. OS Incremental Checkpointing uses the memory management subsystem to decide which data change between consecutive checkpoints

Fraction of Memory Footprint Overwritten during Main Iteration

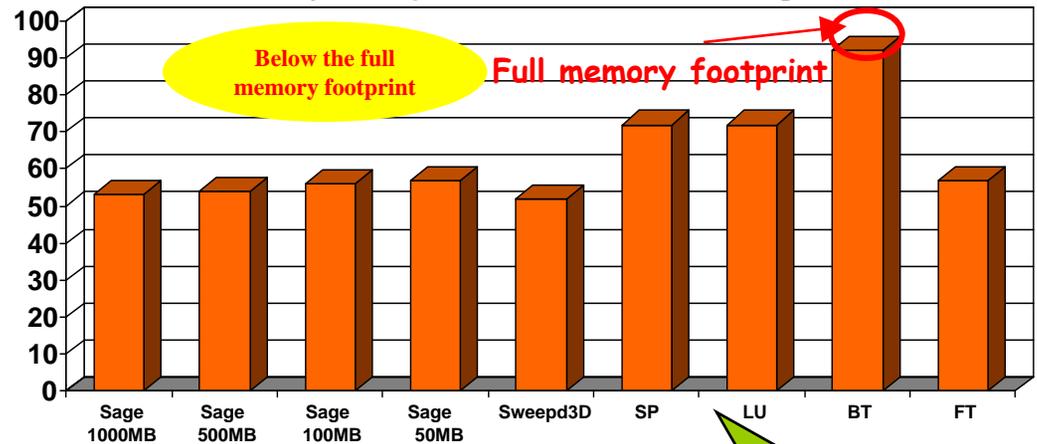


Fig. from J.-C. Sancho

- **Application Level Checkpointing**

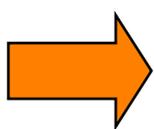
“Programmers know what data to save and when to save the state of the execution”. Programmer adds dedicated code in the application to save the state of the execution.

Few results available:

Bronevetsky 2008: MDCASK code of the ASCI Blue Purple Benchmark

→ Hand written Checkpointer eliminates 77% of the application state

Limitation: impossible to optimize checkpoint interval (interval should be well chosen to avoid large increase of the exec time --> cooperative checkpointing)



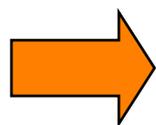
Challenge (not scientific): establish a base of codes with Application Level Checkpointing

Reducing Checkpoint size 2/2

Compiler assisted application level checkpoint

- From Plank (compiler assisted memory exclusion)
- User annotate codes for checkpoint
- The compiler detects dead data (not modified between 2 CKPT) and omit them from the second checkpoint. →
- Latest result (Static Analysis 1D arrays) excludes live arrays with dead data:
--> 45% reduction in CKPT size for mdcask, one of the ASCI Purple benchmarks

- Inspector Executor (trace based) checkpoint (INRIA study)
Ex: DGETRF (max gain 20% over IC)
Need more evaluation



Challenge: Reducing checkpoint size (probably one of the most difficult problems).

100%

Fig. from G. Bronevetsky

QuickTime™ and a TIFF (Uncompressed) decompressor are needed to see this picture.

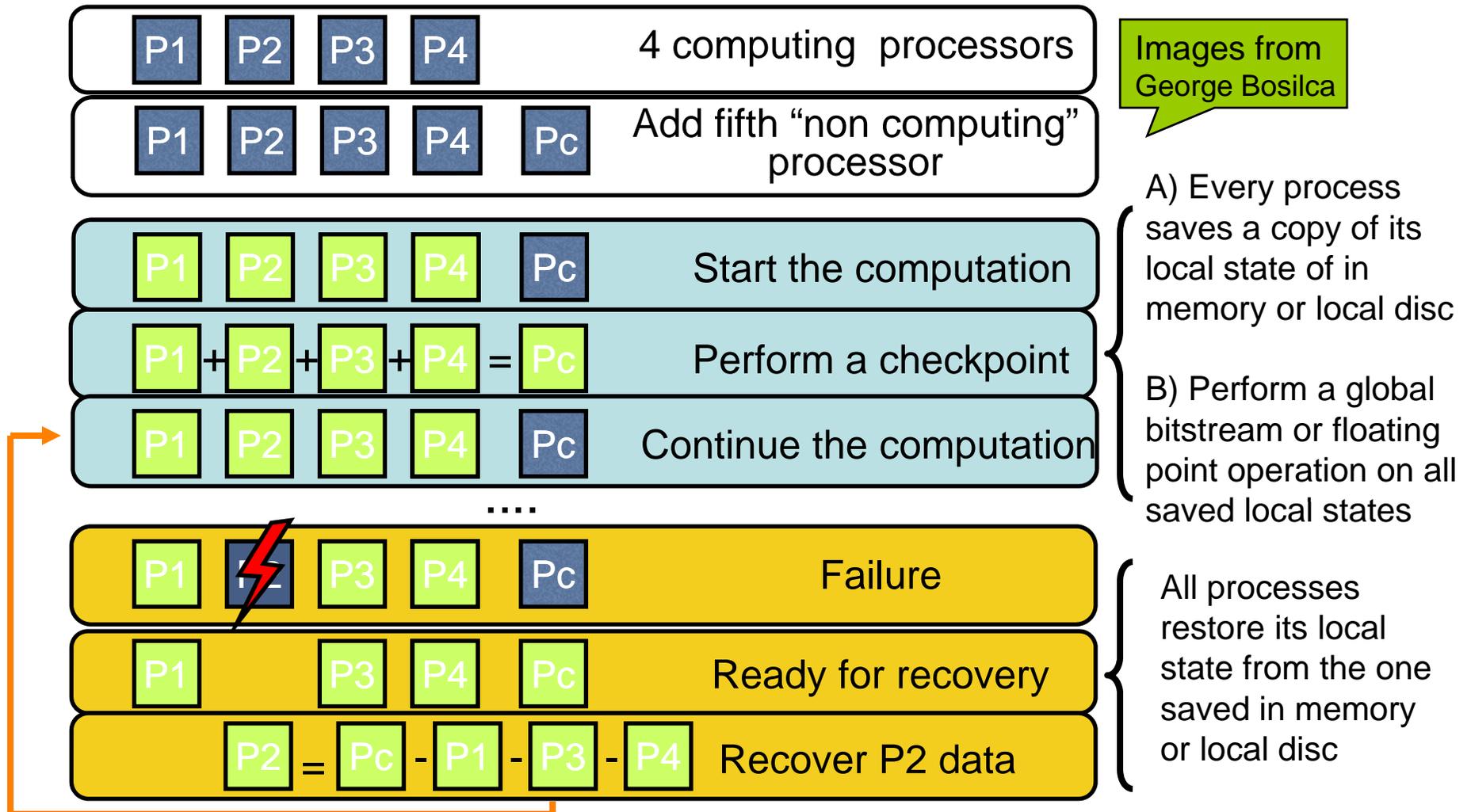
22%



Diskless Checkpointing 1/2

Principle: Compute a checksum of the processes' memory and store it on spare processors

Advantage: does not require ckpt on stable storage.

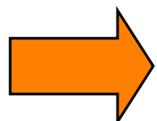


Diskless Checkpointing 2/2

Images from
CHARNG-DA LU

- Could be done at application and system levels
- Process data could be considered (and encoded) either as bit-streams or as floating point numbers. Computing the checksum from bit-streams uses operations such as parity. Computing checksum from floating point numbers uses operations such as addition
- Can survive multiple failures of arbitrary patterns
Reed Solomon for bit-streams and weighted checksum for floating point numbers (sensitive to round-off errors).
- Work with with incremental ckpt.
- Need spare nodes and double the memory occupation (to survive failures during ckpt.) --> increases the overall cost and #failures
- **Need coordinated checkpointing or message logging protocol**
- Need very fast encoding & reduction operations
- **Need automatic Ckpt protocol or program modifications**

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.



Challenge: experiment more Diskless CKPT and in very large machines (current result are for ~1000 CPUs)

Proactive Operations

- Principle: predict failures and trigger preventive actions when a node is suspected
- Many researches on proactive operations assume failures could predicted.

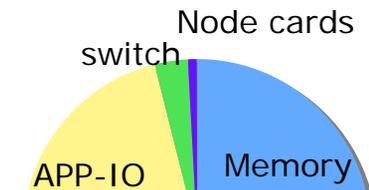
Only few papers are based on actual data.

- Most of researches refer 2 papers published in 2003 and 2005 on a 350

CP Traces from either a rather small system (350 CPUs) or the first 100 days of a large system not yet stabilized

BG/L prototype

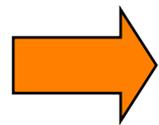
Graphs from
R. Sahoo



A lot of fatal failures
(up to >35 a day!)

Everywhere in
the system

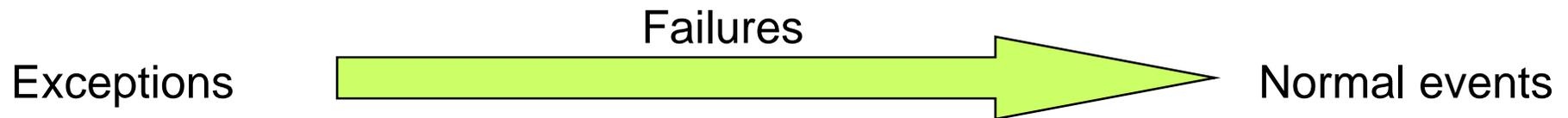
From many
sources



Challenge: Analyze more traces, Identify more correlations, Improve predictive algorithms

Opportunities

May come from a strong modification of the problem statement:



From system side: “Alternatives FT Paradigms”:

- Replication (mask the effect of failure),
- Self-Stabilization (forward recovery: push the system towards a legitimate state),
- Speculative Execution (commit only correct speculative state modifications),

From applications&algorithms side: “Failures Aware Design”:

- Application level fault management (FT-MPI: reorganize computation)
- Fault Tolerance Friendly Parallel Patterns (Confine failure effects),
- Algorithmic Based Fault tolerance (Compute with redundant data),
- Naturally Fault Tolerant Algorithms (Algorithms resilient to failures).

Since these opportunities have received only little attention (recently), they need further explorations in the context of PetaScale systems.

Does Replication make sense?

Slide from
Garth Gibson

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

- Need investigation on the processes slowdown with high speed networks
- Currently too expensive (double the Hardware & power consumption)
- Design new parallel architectures with very cheap and low power nodes
- Replicate only nodes that are likely to fail --> failure prediction

“Algorithmic Based Fault Tolerance”

In 1984, Huang and Abraham, proposed the ABFT to detect and correct errors in some matrix operations on systolic arrays.

ABFT encodes data & redesign algo. to operate on encoded data. Failure are detected and corrected off-line (after execution).

From G. Bosilca

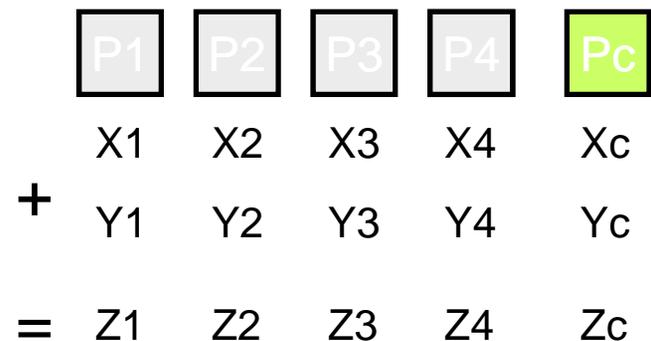
ABFT variation for on-line recovery (runtime detects failures + robust to failures):

- Similar to Diskless ckpt., an extra processor is added, P_{i+1} , store the checksum of data: (vector X and Y in this case)

$$X_c = X_1 + \dots + X_p, Y_c = Y_1 + \dots + Y_p.$$

$$X_f = [X_1, \dots, X_p, X_c], Y_f = [Y_1, \dots, Y_p, Y_c],$$

- Operations are performed on X_f and Y_f instead of X and Y : $Z_f = Y_f + Z_f$



- Compared to diskless checkpointing, the memory AND CPU of P_c take part of the computation):

- No global operation for Checksum!
- No local checkpoint!

Works for many Linear Algebra operations:

Matrix Multiplication: $A * B = C \rightarrow A_c * B_r = C_f$

LU Decomposition: $C = L * U \rightarrow C_f = L_c * U_r$

Addition: $A + B = C \rightarrow A_f + B_f = C_f$

Scalar Multiplication: $c * A_f = (c * A)_f$

Transpose: $A_f^T = (A^T)_f$

Cholesky factorization & QR factorization

“Naturally fault tolerant algorithm”

Natural fault tolerance is the ability to tolerate failures through the mathematical properties of the algorithm itself, without requiring notification or recovery.

The algorithm includes natural compensation for the lost information.

Figure from
A. Geist

For example, an iterative algorithm may require more iterations to converge, but it still converges despite lost information

Assumes that a maximum of 0.1% of tasks may fail

QuickTime™ et un
décompresseur TIFF (LZW)
sont requis pour visionner cette image.

Ex1 : Meshless iterative methods+chaotic relaxation
(asynchronous iterative methods)

Meshless formulation of 2-D
finite difference application

Ex2: Global MAX (used in iterative methods to determine convergence)

QuickTime™ et un
décompresseur TIFF (LZW)
sont requis pour visionner cette image.

This algorithm share some features
with SelfStabilization algorithms:
detection of termination is very hard!
→it provides the max « eventually »...
BUT, it does not tolerate Byzantine
faults (SelfStabilization does for
transient failures + acyclic topology)

Wrapping-up

Fault tolerance is becoming a major issue for users of large scale parallel systems.

Many Challenges:

- Reduce the cost of Checkpointing (checkpoint size & time)
- Design better logging and analyzing tools
- Design less expensive replication approaches
- Integrate Flash mem. tech. while keeping cost low and MTTI high
- Investigate scalability of Diskless Checkpointing
- Collect more traces, Identify correl., new predictive algo.

Opportunities may come from Failure Aware application Design and the investigation of Alternatives FT Paradigms, in the context of HPC applications.

Questions?

- Novel methods and algorithms (inherently tolerant to failures)
- FT Aware Paradigm & Design
 - Replication, SelfStab., Speculative?
Specific hardware?
 - Rewrite Apps. in MW or D&C
- Optimizations:
 - Checkpoint Storage & transfers (Additional Local disk, Flash mem.)
 - Reduce ckeckpoint size
 - Diskless checkpointing
 - Proactive actions (Prediction)

