CS 3580 - Advanced Topics in Parallel Computing

Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing

Mohammad Hasanzadeh-Mofrad

University of Pittsburgh

October 03, 2017

Article information

Conference: **SC 12** Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis

Title: Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing

Authors: **David Fiala**, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell

Affiliation: North Carolina State University

Citation (as of October 2017): 178

https://scholar.google.com/citations?user=nkN-PNYAAAAJ&hl=en&oi=sra

Gist of the Paper

- Motivation: When dealing Silent Data Corruptions (SDCs) if Checkpoint/Restart (C/C) overheads exceed 50% of a job, **redundancy** is actually cheaper than traditional C/R.
- Contributions:
  - Design of new **SDC detection and correction protocol** at the communication layer.
  - A study about challenges and cost of SDC protection using redundancy.
    - Detection
    - Correction
  - Limit the overhead of SDC detection by reducing the relevant footprint
  - Assessing fault injection scenarios through hardware and software fault injection.
  - A live SDC tracking framework using MPI communication
- Assumptions:
  - Most critical data is communicated during/after computation which is the scope of importnant calculations.

High End Computing (HEC) Challenges

- Nowadays, faults have became a norm in massive clusters of cores with numerous reasons such as:
  - Hardware: I/O, memory, processor, power supply, switch failure
  - Software: Operating system, runtime, unscheduled maintenance
  - A failure in each category is enough to interrupt the application.
- And as frequent as:
  - Servers tend to crash twice a year (2 – 4%)
  - 1 – 5% of disk drives die every year
  - DRAM errors occurs in 2% of all DIMM[1] per year
- And the solutions are:
  - Checkpoint/Restart (C/R) methods for long-running applications

David Fiala et al. “Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing.“ SC 12

1 Dual In-line Memory Module (DIMM)

Checkpoint/Restart (C/R) Efficiency

- There is a rapid decay for the amount of useful work when
  - **The cluster is larger**

168 hours job with 5 year MTBF[1]

#Nodes	Work	Checkpt	Recomp.	Restart
100	96%	1%	3%	0%
1,000	92%	7%	15	0%
10,000	75%	15%	6%	4%
100,000	35%	20%	10%	35%

  - **The job is longer**
    - A 168 hours job
    - With a node MTBF of 5 years
    - On a 100K nodes cluster
    - 35% useful work
    - The reminder is spent on C/R

100K nodes job with varied MTBF

Job work	MTBF	Work	Checkpt	Recomp.	Restart
168 hours	5 years	35%	20%	10%	35%
700 hours	5 years	38%	18%	9%	43%
5,000 hours	1 year	5%	5%	5%	85%

David Fiala et al. “Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing.“ SC 12

1 Mean Time Between Failures (MTBF)

So, what's the Outcome of previous tables?

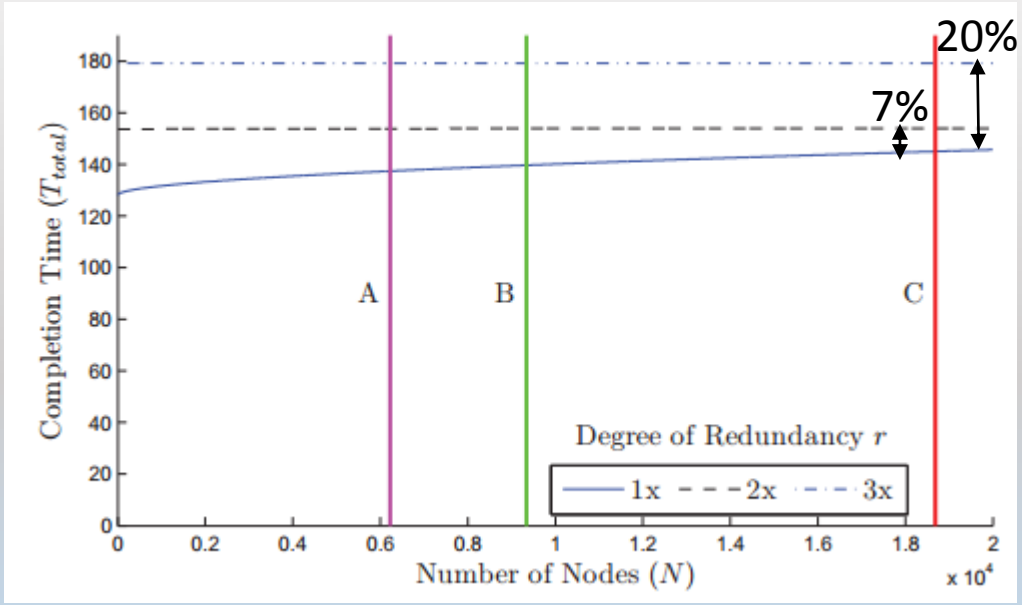
- **Redundancy in computing can significantly revert the useful work decay.** By doubling up the compute nodes so that every node N has a replica N'.
- **Redundancy scale**: As more nodes are added to the system, the probability for simultaneous failure of a primary N and its replica rapidly decreases.
- Thus, when restart and rework overheads exceed 50% of a job, redundancy is actually cheaper than traditional C/R.

Failure taxonomy

- Fail-stop faults: Faults detectable by monitoring of hardware or software.
- Silent Data Corruption (SDC): When data corruption goes undetected, it becomes a **Silent Data Corruption (SDC)** and poses a high risk to the application since it is not recoverable and is corrupted the correct state of data. Data corruption is an unintentional change in a bit (bit flips) or an unrecoverable read error.
- SDC examples
  - **Single bit flip** which can be detected by ECC in caches but no in register files or ALU
  - **Double bit flips** which ECC cannot correct them and force an instant reboot after them
- Good news: With SDCs occurring at significant rates, not every bit flips result in faults e.g. flips in stale data or cod

Modeling Redundancy

- Jaguar supercomputer[1]
  - Processor: 224,256 x86-based AMD Opteron
  - Operating system: Cray Linux Environment
  - MTBF: 52 hours == 50 years per node
  - Active: Since 2005
  - TOP500: 2009, and 2010
- 128 hours job
  - 1x, 2x, and 3x redundancy
  - Different number of cores



- **Conclusions**:
  - Dual or triple redundancy is more efficient compare to running a job two or three times
  - Dual redundancy would have the lowest cost, yet there is an additional cost of SDC correction at the triple redundancy.
  - Double redundancy enables SDC detection
  - Triple redundancy recovering from SDC detection

1. https://en.wikipedia.org/wiki/Jaguar_(supercomputer)
2. David Fiala et al. “Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing.“ SC 12

RedMPI design

- The authors proposed RedMPI, an MPI library capable detecting and correcting SDC faults. RedMPI creates replicas of **primary task** and **performs comparison of messages to detect corrupt messages**.
- **Double Redundant RedMPI** can detect divergent messages between replicas. The corrupted results will be invalidated and replaced with the good results of the correct replica.
- **Triple Redundant RedMPI** can correct the corrupted messages and presents an additional level of redundancy when having two consecutive faults. It has a voting mechanism to identify corrupted messages.
- RedMPI does not monitor memory spaces of replicas because it is much more costly compare to comparing MPI messages.
- RedMPI limited to the critical messages communicated during/after computation

RedMPI Design

A. Point-to-Point Message Verification
B. Assumptions
C. Implementation Notes

Point-to-Point Message Verification

- For any given rank, a P2P message sent by MPI_Isend should be the same for primary MPI process and its replicas.
- How to verify correctness?
  - Byte by byte message comparison or comparing hashes of messages (to reduce network bandwidth) between primary process and replicas.
- Where to verify correctness?
  - **Sender-side:** The sender process will verify the correctness of data. All replicas send messages to verify their content, then the verified data will be sent to the receiving replica. This incurs overhead especially in the case of matching messages.
  - **Receiver-side:** The receiver process will verify the correctness of data. This way is faster because it reduces message latency prior to sending the message.

Assumptions

- Support for Collectives/wildecards/non-deterministic MPI
- MPI-1 support
- RedMPI does not support MPI I/O functionality

Implementation Notes

- A transparent profiling layer is implemented which wraps MPI calls and add additional logic to support replications for MPI calls i.e. Comm_rank, Comm_size, Send, Recv,and etc.
- When launching a job with RedMPI, a *multiple* of the original number of desired processes will be launched.
  - E.g. An MPI job with 128 processes will require 256 or 384 processes for dual and triple redundancy.

Implementation Notes – All-to-all Protocol

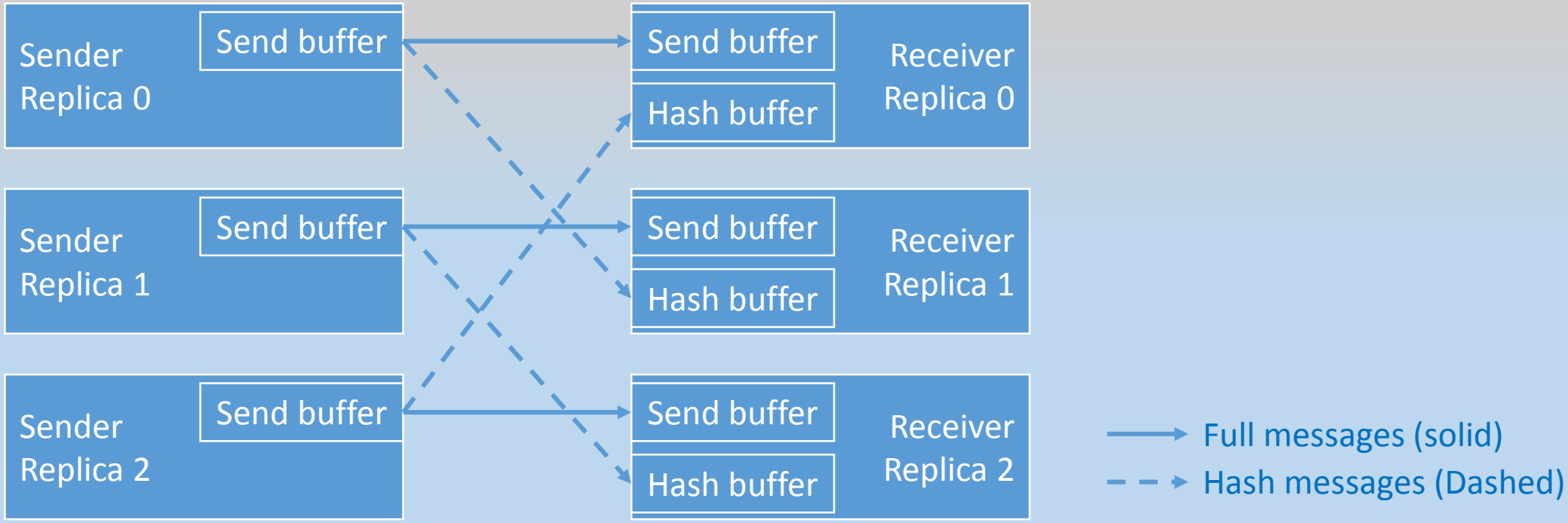
- The RedMPI receiver side protocol (All-to-all)
  - All sender replicas send message to all receiver replicas. Thus, each sender sends three messages, on for each replica of receiver.
  - Originally, in MPI, MPI_Test or MPI_wait completes a message transmission. But, RedMPI uses **MPI_Request** to verify correct message reception of MPI_Irecv
- **In RedMPI with dual redundancy,** messages received from the replica should be matched. If not, application will be terminated.
- **In RedMPI with triple redundancy or better,** voting is used to determine the replica sent the corrupted data and copy of the correct message will be sent to that replica.

Implementation Notes – MsgPlusHash

- The redMPI primary mode of operation with message verification at receiver side (MsgPlusHash)
  - A copy of the message will be sent to the receiver replica and other replicas will receive the hash of message.
  - The message will be transmitted only once while the additional hash is later use to verify each receiver's message.
  - **MsgPlusHash is more efficient**, because for double redundancy it is not necessary to send two full messages just for verifying the data correctness. Once a message is received, its hash is computed and checked against the received hash to assure correctness.
  - The messages are sent to the receiver with same replica rank as the sender.
  - The hashes are sent to the receiver with rank equal to sender's replica rank plus one

Implementation Notes: All-to-all versus MsgPlusHash

- **All-to-all** needs to send *n*^*r* message where *n* is the number of messages and *r* is the number of replicas.
- **MsgPlusHash** needs to send *n***r* + (*n*) messages where *n* is the number of hashes which is very small compare to the original message.



Fault Injection Design

- A fault injector is implemented within RedMPI that is able to inject SDCs that ECC is unable to detect or correct them.
- It can inject fault with the probability of 1/x by picking a message and corrupting it by flipping one of its bits. It actually modifies both memory allocated for a message and the message itself.

Experimental Framework

- RedMPI is deployed on a medium-sized cluster with
  - 96 machines
  - AMD Opteron dual socket
  - 8 cores per docket (16 cores per machine)
  - 32GB RAM per machine
  - 40 Gb/s InfiniBand fat tree interconnect MPI transport
  - **Replicas are not in the same machine**
  - Processor counts and communication patterns are studied for calculating the time
  - RedMPI is assessed using a suite of strong and weak scaling applications:
    - Weak scaling benchmarks: LAMMPS, ASCI Sweep3D, and HPCCG
    - Strong scaling benchmarks: NAS Parallel benchmarks (NPB)

Results (Reduced version of Table III - XI)

- Runtime of weak and strong scaling suites
- C, D, E are NBP input class where E is the largest
- Size is the number of processes
- 1x is the baseline
- 2x is double redundancy
- 3x is triple redundancy
- 2x OV is the overhead of 2x/1x
- 3x OV is the overhead of 3x/1x

TABLE V
SWEEP3D

Size	1x [s]	2x [s]	3x [s]	2x OV	3x OV
128	390.30	389.49	393.05	-0.2%	0.7%
256	428.17	427.53	431.20	-0.1%	0.7%
512	488.08	488.93	494.09	0.2%	1.2%

TABLE VIII
NPB EP

Size	1x [s]	2x [s]	3x [s]	2x OV	3x OV
128-D	72.31	72.63	72.74	0.4%	0.6%
256-E	579.94	581.02	581.27	0.2%	0.2%
512-E	289.80	290.83	291.30	0.4%	0.5%

TABLE XI
NPB MG

Size	1x [s]	2x [s]	3x [s]	2x OV	3x OV
128-E	339.17	340.41	429.67	0.4%	26.7%
256-E	168.56	170.68	171.48	1.3%	1.7%
512-E	66.97	68.35	69.29	2.1%	3.5%

Results – SDC Fault Propagation (Fig. 4 – 10)

- Two sets of RedMPI with dual redundancy
  - A control set which does not receive any faults
  - A test set which receive faults
- RedMPI detect faults in victim set but allows application to continue.
- X-axis is the application progress
- Left Y-axis is the number of corrupted messages
- Right Y-axis is the number of corrupted nodes
- They conclude that protecting applications at the MPI message level is an appropriate method to detect, isolate, and prevent further corruption.

Results – SDC Fault Propagation (Fig. 4 – 10)

Fig. 5. NPB BT Overview of Corrupt Nodes and Messages

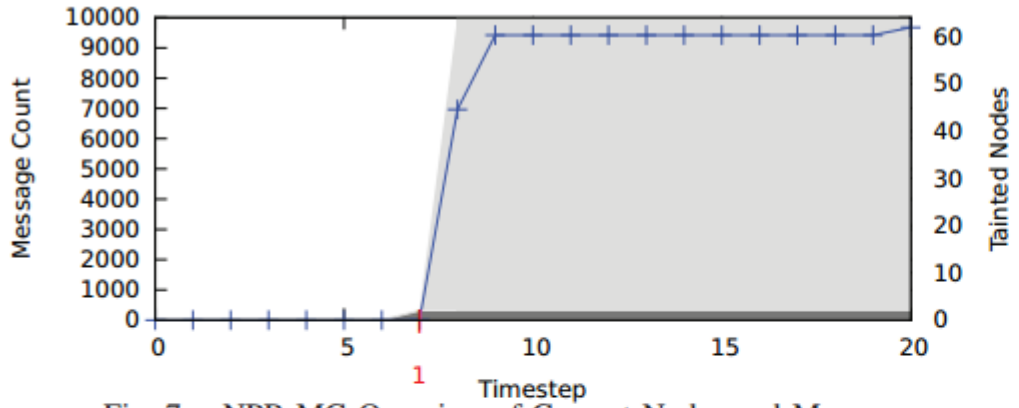
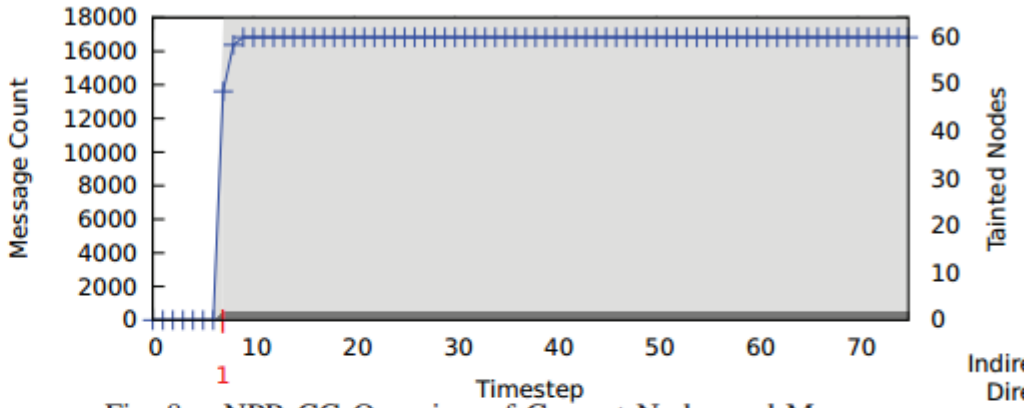


Fig. 6. NPB SP Overview of Corrupt Nodes and Messages



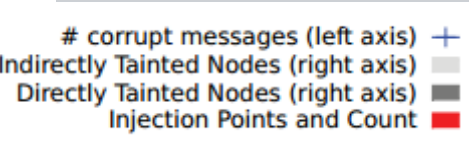


Fig. 7. NPB MG Overview of Corrupt Nodes and Messages

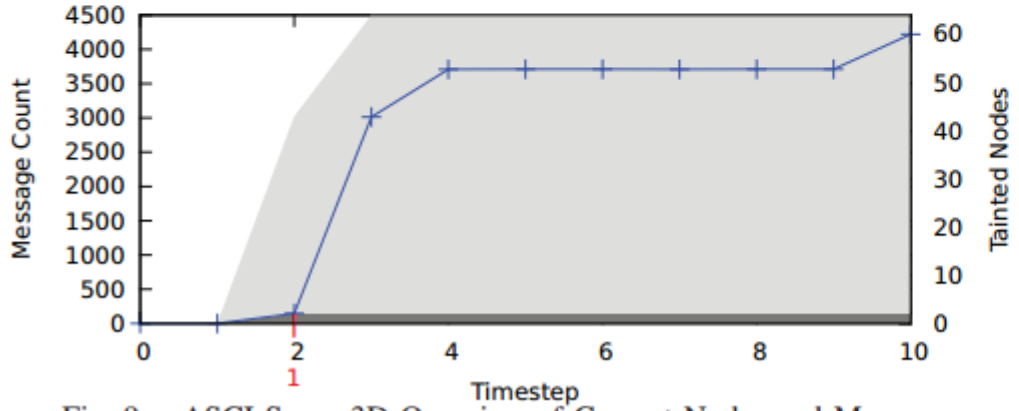


Fig. 8. NPB CG Overview of Corrupt Nodes and Messages

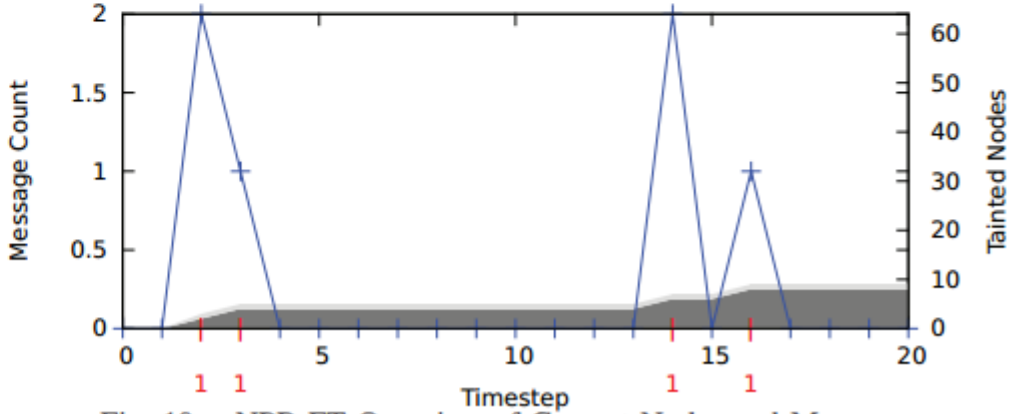


Fig. 9. ASCI Sweep3D Overview of Corrupt Nodes and Messages

Fig. 10. NPB FT Overview of Corrupt Nodes and Messages

Results – SDC Fault Protection

- Run CG benchmark
- 64 processes and a replication degree of 3 (192 physical processors)
- Generating faults with the frequency of 1/5,000,000
- Except for three simultaneous injections, RedMPI is able to detect 100,000 corrupted messages
- Within 20 iterations of CG, it has been observed that the runtime for MsgPlusHash is 0.31 seconds less than the original runtime without RedMPI.
- The actual of overhead of RedMPI occurs when it corrects a SDC. So, the overhead is a function of number of detected invalid messages.