DINO: Divergent Node Cloning for Sustained Redundancy in HPC

Arash Rezaei and Frank Mueller
North Carolina State University, Raleigh, NC.
mueller@csc.ncsu.edu

***Abstract*—Soft faults like silent data corruption and hard faults like hardware failures may cause a high-performance computing (HPC) job of thousands of processes to nearly cease to make progress due to recovery overheads. Redundant computing has been proposed as a solution at extreme scale by allocating two or more processes to perform the same task. However, current redundant computing approaches do not repair failed replicas. Thus, SDC-free execution is not guaranteed after a replica failure and the job may finish with incorrect results. Replicas are logically equivalent, yet may have divergent runtime states during job execution, which complicates on-the-fly repairs for forward recovery. In this work, we present a redundant execution environment that quickly repairs hard failures via Divergent Node cloning (DINO) at the MPI task level. DINO contributes a novel task cloning service integrated into the MPI runtime system that solves the problem of consolidating divergent states among replicas on-the-fly. Experimental results indicate that DINO can recover from failures nearly instantaneously, thus retaining the redundancy level throughout job execution. The cloning overhead, depending on the process image size and its transfer rate, ranges from 5.60 to 90.48 seconds. To the best of our knowledge, the design and implementation for repairing failed replicas in redundant MPI computing is unprecedented.**

I. Introduction

Reliability has been highlighted as a key challenge for next generation supercomputers [1], [3], [5]. Node failures are commonly due to hardware or software faults. Hardware faults may result from aging, loss of power, and operation beyond temperature thresholds. Software faults can be due to bugs (some of which may only materialize at scale), complex software component interactions and race conditions that surface only for rare parallel execution interleavings of tasks [4].

One resilience method is redundant computing [2], [8], [7], [9]. It aims at improving reliability and availability of systems by allocating two or more components to perform the same work. A recent study [13] showed that redundancy could be a very viable and even cost-effective approach for HPC on the cloud. Their approach to combine checkpointing and redundancy on Amazon EC2 using a variable-cost spot market provides up to 7 times cheaper execution compared to the on demand default market. Redundancy provides tolerance not only against hard faults but also soft faults, such as silent data corruptions (SDCs), which do not stop application execution as they are undetectable. SDCs may manifest at application completion by producing wrong results or, prior to that, wrong interim results. A study at CERN raised concerns over the significance of SDC in memory, disk and RAID [14]. Their results indicate that SDC rates are orders of magnitude larger than manufacture specifications. Schroeder et al.'s study [16] of the DRAM errors on a large scale over the course of 2.5 years concludes that more than 8% of DIMMs are affected by errors per year. SDCs can be detected by Dual Modular Redundancy (DMR) and can be corrected with voting under Triple Modular Redundancy (TMR) [9].

Current approaches for redundant computing do not provide a sustained redundancy level during job execution when processes are hit by failures. When a replica fails, either the application deadlocks (RedMPI [9]) or other replicas ensure that the application can progress in execution [2]. The latter requires at least one healthy replica, i.e., should all replicas of an MPI task fail, then the entire job fails. Note that after a replica failure, even if the job can continue its execution, the SDC detection module cannot guarantee application correctness (e.g., an undetected SDC might occur). Checkpoint/Restart (CR) is another popular method for tolerating hard errors but cannot handle soft errors. CR takes snapshots of all processes and saves them to storage. Should a hard error occur, all processes re-load the last snapshot into memory and the application continues. Elliott et al. [6] show that CR will eventually take longer than redundancy due to recomputation, restart and I/O cost. At scale, this makes capacity computing (maximizing the throughput of smaller jobs) more efficient than capability computing (using all nodes of an exascale machine). E.g., at 80,000 CPU sockets, dual redundancy will finish twice the number of jobs that can be handled without redundancy. This includes a redundancy overhead of 0-30% longer time (due to additional messages) with hashing protocols [9], which has no impact on bandwidth for Dragonfly networks since original and replica sphere exchange full messages independently. As hash messages are small, they add latency but do not impact bandwidth. Since twice the jobs finished under dual redundancy, this amounts to the *same* energy.

This work targets tightly-coupled parallel applications/jobs executing on HPC platforms using MPI-style message passing [10]. We use the term *rank* to refer to an MPI task/process. In MPI-style programming, each MPI process is associated with a unique integer value identifying the rank. Suppose there is a job with n ranks that requires t hours to complete without any failures. This is called plain execution time. We consider systems with r levels of redundancy (at the rank level). Our system then consists of $r \times n$ ranks, where n logical MPI tasks are seen by the user while redundant replicas remain transparent. There is no difference between replicas of the same task in terms of functionality as they perform the same operations. We also assume the availability of a small pool of spare nodes. Spare nodes are in a powered state but initially do not execute any jobs. We assume that a fault detector is provided by the system. We focus on the recovery phase and consider works in failure detection [18], [12] orthogonal to our work.

We introduce node cloning as a means to sustain a given redundancy level. (We use the terms node / MPI task cloning synonymously.) The core idea is to recover from hard errors with the assistance of a healthy replica. A healthy replica is cloned onto a spare node to take over the role of the failed process in "mid-flight". To address shortcomings in current redundant systems, we provide the following contributions:

• We devise a generic high performance node cloning service under divergent node execution (DINO) for recovery. DINO clones a process onto a spare node in a live fashion. We integrate DINO into the MPI runtime under redundancy as a reactive method that is triggered by the MPI runtime to forward recover from hard errors, e.g., node crash or hardware failure.

• We propose a novel Quiesce algorithm to overcome divergence in execution without excessive message logging. Execution of replicas is not in a lock-step fashion, i.e., can diverge. Our approach establishes consistency through a novel, multicast variant of the traditional bookmark protocol [15] and resolves inconsistencies through exploiting the symmetry property of redundant computing.

• We evaluate DINO's performance for MPI benchmarks. The time to regain dual redundancy after a hard error varies from 5.60 seconds to 90.48 seconds depending on process image size and cross-node transfer bandwidth, which is short enough to make our approach practical.

II. Design of DINO

DINO has a generic process cloning service at its core. Node cloning creates a copy of a given running process on a *spare* node. The cloning mechanism itself is MPI agnostic and is applied to processes encapsulating MPI tasks in this work. DINO considers the effect of cloning on the MPI runtime system, as detailed later. Fig. 1 shows how the system retains dual redundancy in case of a failure. A and A' are logically equivalent and both perform the same computation. They run on nodes 0 and 1, respectively. Ranks B and B' on nodes 2 and 3 are also replicas. If node 2 (B) fails, its replica (B') on node 3 (*source* node) is cloned onto node 4 (a *spare* node) on-the-fly. The newly created rank B'' takes over the role of failed rank B and the application recovers from the loss of redundancy. At the end of node cloning, B' and B'' are in the same state from the viewpoint of the application, but not necessarily from another rank's point of view due to stale B references. The quiesce algorithm resolves such inconsistencies.

The process B'' is created on node 4 as follows. While B' performs its normal execution, its memory is "live copied" page by page to B''. This happens in an iterative manner (see Section IV). When we reach a state where few changes in dirty pages remain to be sent, the communication channels are drained. This is necessary to keep the system of all communication processes in a consistent state. After this, rank B's execution is briefly paused so that the last dirty pages, linkage information, and credentials are sent to node 4. Rank B'' receives and restores this information and then is ready to take over the role of failed rank B. Then, communication channels are resumed and execution continues normally. Between

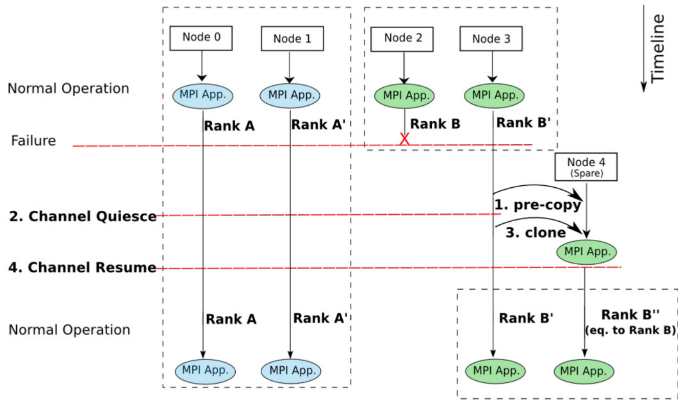


Fig. 1: Node cloning: Application w/ 2 ranks under dual redundancy

channel draining and channel resumption, no communication may proceed. This is also necessary for system consistency with respect to message passing.

The short time interval between error detection and the end of DINO recovery is a "vulnerability window" where undetected SDCs may occur. The vulnerability window depends on the process image size and is evaluated in Section V.

III. Quiesce Algorithm

The purpose of the Quiesce algorithm is to resolve the communication inconsistencies inside DINO at the library level and provide transparent and consistent recovery to the application layer. The inconsistencies are rooted in the state divergence of replicas. Blocking operations impose limited divergence. But non-blocking operations can easily create scenarios where the state of replicas differs largely as there is no enforced state synchronization among replicas.

Let us assume that rank B has failed and rank B' is cloned to create B'', which takes over the work of B. Ranks B' and B'' are in the same state, but any other ranks may still assume B'' to have the state of B.

First, the outgoing channels of B' and of any other ranks that have initiated a send to B' are cleared. We use a modified version of bookmark exchange protocol [15] to identify in-flight messages and drain them. After this step, we can be sure that any ongoing communication with rank B' is completed.

The next two steps (steps 2 and 3) are to identify the state of communication with regards to rank B'', and to guarantee the correctness of DINO recovery. We exploit the symmetry property of redundant computing stating that every rank receives the same number of messages from members of a replica set (e.g., B and B'). The same rule applies to the number of messages sent to a given replica set. This property is the basis for resolving the inconsistencies in stages 2 and 3. Every rank keeps a vector of the number of messages sent to other ranks ($Sent[]$) and received from other ranks ($Received[]$) along with the message signatures.

In step 2, every rank X (other than B') resolves its possible communication inconsistency due to sends to B. Three cases are distinguished (see Fig. 2, left side): (1) If bookmarks match ($Sent[B] == Sent[B']$), then B, B', and B'' are in the same state from the point of view of X, and no action is needed.

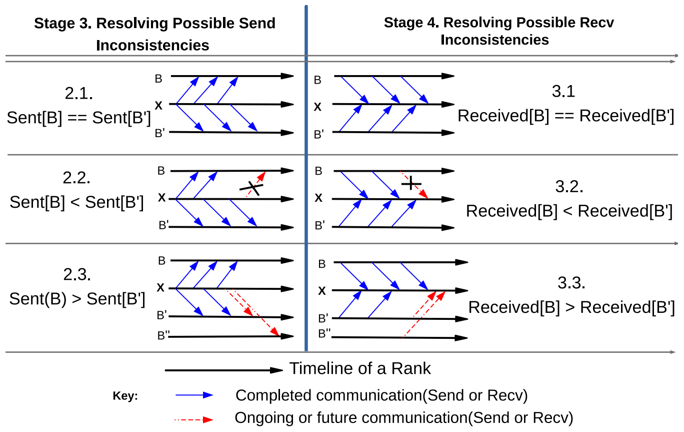

Stage 3. Resolving Possible Send Inconsistencies
Stage 4. Resolving Possible Recv Inconsistencies
2.1. Sent[B] == Sent[B']
3.1 Received[B] == Received[B']
2.2. Sent[B] < Sent[B']
3.2. Received[B] < Received[B']
2.3. Sent(B) > Sent[B']
3.3. Received[B] > Received[B']
B
X
B'
B''
Timeline of a Rank
Key:
Completed communication(Send or Recv)
Ongoing or future communication(Send or Recv)

Fig. 2: A view of Steps 2 and 3 of the Quiesce algorithm

(2) If B lagged behind B' ($Sent[B] < Sent[B']$), then sends from X to B are in transit/will be issued (Fig. 2 part 3.2). Since B has been removed and B'' is ahead (has already seen these messages), they are silently suppressed (skipped). (3) If B was ahead of B' ($Sent[B] > Sent[B']$), then there exist messages in transit/to be sent from X to B' (Fig. 2 part 3.3). Since B'' is in the same state as B', these messages need to be sent to B'' as well.

In step 3, the same procedure is performed on the receive counters (see Fig. 2, right side). The 3 cases are symmetric to the send cases: (1) if bookmarks match ($Received[B] == Received[B']$) then B, B', and B'' are in the same state from the point of view of X, and no further action is needed. (2) If B' was ahead of B ($Received[B] < Received[B']$) then X is expecting messages from B (Fig. 2 part 4.2). Since B does not exist anymore and B'' will not send them (as it is ahead), these receives silently complete (skipped). X will be provided with the corresponding messages from B' (without comparing them with the one from B'' that was never received). (3) If B' lagged behind B ($Received[B] > Received[B']$), then X is expecting messages from B' (Fig. 2 part 4.3). Since B'' and B' are in the same state, both will send those messages, even though X has already received a copy from B. Thus, messages from B'' are silently absorbed (up to the equalization point).

IV. Implementation

1. Pre-copy. This phase transfers a snapshot of the memory pages in the process address space communicated to the spare node while normal execution of the process continues on the source node (see Fig. 3). We use TCP sockets to create a communication channel between *source* and *spare* nodes. The pre-copy approach borrows concepts from [17] (under Linux 2.4), but adapted to Linux to 2.6. Vital meta data, including the number of threads, is transferred.[1] The spare node receives the memory map from the pre-copy thread. All non-zero pages are transferred and respective page dirty bits are cleared in the first iteration. In subsequent iterations, only dirty pages are transferred after consulting the dirty bit. We apply a patch to the Linux kernel to shadow the dirty bit inside page table entry (PTE) and keep track of the transferred memory pages. The pre-copy phase terminates when the number of transferred pages reaches a threshold (1MB in our current setting).

[1]We assume that applications maintain a constant sized thread pool after initialization, e.g., OpenMP implementations. Cloning applies to the execution phase after such thread pool creation.

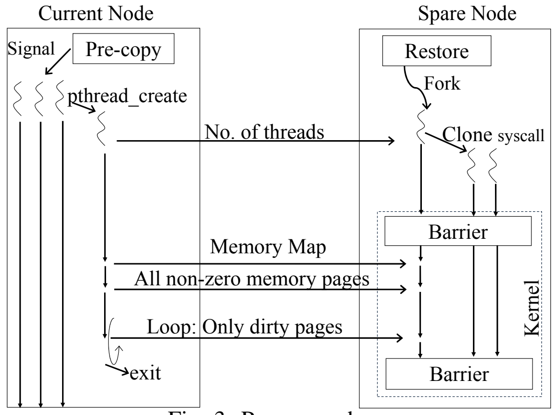


Fig. 3: Pre-copy phase

2. Channel Quiesce. The purpose of this phase is to create a settle point with the shadow process. This includes draining all in-flight MPI messages. The runtime system also needs to stop posting new send/receive requests. We build this phase on top of the functionality for message draining provided by the CR module of Open MPI [11]. The equalization stage described in Section III is implemented in this step.

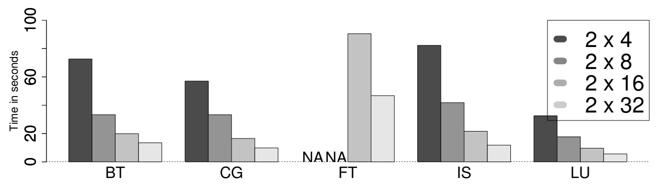
3. Clone. This phase stops the process for a short time to transfer a consistent image of its recent changes to the `restore` tool. The memory map and updated memory pages are transferred and stored at the corresponding location in the address space of B''. Then, credentials are transferred and permissions are set. Restoration of CPU-specific registers is performed in the next phase. The signal stack is sent next and the sets of blocked and pending signals are installed. Inside the kernel, we use a barrier at this point to ensure that all threads have received their register values before any file recovery commences. In short, different pieces of information are transferred to fully create the state of the process.

4. Channel Resume. In this phase, processes re-establish their communication channels with the recovered sphere. All processes receive updated job mapping information, reinitialize their Infiniband driver and publish their endpoint information.

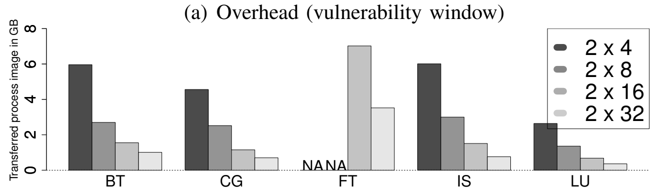
V. Experimental Results

The node cloning experiments require insertion of our kernel module into the Linux kernel. This permission is not granted on large-scale supercomputers maintained by NSF or DOE. Thus, we conducted the experiments on a 108-node cluster with QDR Infiniband. Each node is equipped with two AMD Opteron 6128 processors (16 cores total) and 32GB RAM running CentOS 5.5, Linux kernel 2.6.32 and Open MPI 1.6.1. The experiments are demonstrating failure recovery rather than exploring compute capability for extreme scale due to the limitations of our hardware platform. Hence, we exploit one process per node in all experiments. Experiments were repeated five times and average values of metrics are reported.

We consider 5 MPI benchmarks: (BT, CG, FT, IS, LU) from the NAS Parallel Benchmarks (NPB). We use input class D for NPB. We present results for 4, 8, 16 and 32 processes under dual redundancy (CG, IS, LU). We use 4, 9, 16, 25 processes for BT (square numbers are required by the benchmark). FT with 4 and 8 processes could not be executed due to memory limitations. Due to lack of support from the Infiniband driver to cancel outstanding requests



(a) Overhead (vulnerability window)



(b) Transferred process image

Fig. 4: DINO recovery from 1 fault injection for different MPI benchmarks (1 rank per physical compute node)

without invalidating the whole work queue and lack of safe re-initialization, current experiments are performed with marker messages. Every process receives a message indicating the fault injection and acts accordingly. One rank mimics the failure by performing a `SIGSTOP`. Then the Cloning APIs are used to start the clone procedure and the discussed steps in Section II are performed: Pre-copy, Quiesce, Clone, Resume.

Fig. 4(a) and 4(b) depict the overhead and transferred memory size, respectively. NPB are strong scaling applications and the problem size is constant in a given class. Therefore, the transferred memory size and consequently time decreases when the number of processes increases. Overhead is the time that takes to transfer the memory pages, create a fully working MPI rank on the spare node and resume the communication. This is also the vulnerability window where only one copy of the process exists (e.g., only B' in Fig. 1).

FT has the largest process image. The size of memory for FT with 16 processes is 7GB and takes 90.48 sec to transfer, while it takes 46.75 sec with 32 processes to recover from a failure when transferring 3.52GB of memory. LU has the smallest process image among NPB, its memory size ranges from 2.64GB to 0.36GB with transfer times of 32.51 sec to 5.60 sec for 4 to 32 processes, respectively.[2] The relative standard deviation in these experiments is less 7% in all cases.

VI. Conclusion

We introduced DINO, a quick forward recovery method from failures in redundant computing. DINO contributes a novel live node cloning service with multicast variant of the bookmark algorithm and a corresponding Quiesce algorithm for consistency among *diverging* communicating tasks. With its integration into the MPI runtime system, DINO allows a redundant job to retain its redundancy level via cloning throughout job execution. Experimental results with multiple MPI benchmarks indicate low overhead for failure recovery.

Acknowledgment

This work was supported in part by grants from Lawrence Berkeley National Laboratory and NSF grants 1058779 and 0958311. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Computer Science program under contract number DE-AC02-05CH11231. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

[2]In the current implementation, the memory is copied one page at a time. This lowers the performance of the cloning operation. A larger buffer size might increase the performance as the effective bandwidth could be increased.

References

[1] Keren Bergman et al. Exascale computing study: Technology challenges in achieving exascale systems, September 2008.

[2] Ron Brightwell, Kurt Kurt Ferreira, and Rolf Riesen. Transparent redundant computing with MPI. In *Euro-Par*, pages 208–218, 2010.

[3] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *Int. J. High Perform. Comput. Appl.*, 23(4):374–388, Nov 2009.

[4] Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. Software Aging Analysis of the Linux Operating System. In *ISSRE*, pages 71–80, 2010.

[5] Jack Dongarra et al. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011.

[6] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining Partial Redundancy and Checkpointing for HPC. In *International Conference on Distributed Computing Systems*, Macau, China, June 18-21 2012.

[7] Christian Engelmann and Swen Böhm. Redundant Execution of HPC Applications with MR-MPI. In *International Conference on Parallel and Distributed Computing and Networks*, pages 31–38, February 15-17, 2011.

[8] Kurt Ferreira, Jon Stearley, James H. Laros, III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G. Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Supercomputing*, pages 44:1–44:12, 2011.

[9] D. Fiala, F. Mueller, C. Engelmann, K. Ferreira, and R. Brightwell. Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing. In *Supercomputing*, 2012.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[11] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *IPDPS*, March 2007.

[12] A. Mahmood and E.J. McCluskey. Concurrent error detection using watchdog processors-a survey. *Computers, IEEE Transactions on*, 37(2):160–174, Feb 1988.

[13] Aniruddha Marathe et al. Exploiting Redundancy for Cost-effective, Time-constrained Execution of HPC Applications on Amazon EC2. In *HPDC*, pages 279–290, 2014.

[14] Bernd Panzer-Steindel. Data Integrity. Technical Report 1.3, CERN, 2007.

[15] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. The LAM/MPI Checkpoint/Restart framework: System-initiated checkpointing. In *LACSI Symposium, Sante Fe*, pages 479–493, 2003.

[16] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: A large-scale field study. *SIGMETRICS Perform. Eval. Rev.*, 37(1):193–204, June 2009.

[17] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in HPC environments. In *Supercomputing*, pages 1–12, 2008.

[18] Keun Soo Yim, Z. Kalbarczyk, and R.K. Iyer. Pluggable Watchdog: Transparent Failure Detection for MPI Programs. In *International Parallel and Distributed Processing Symposium*, pages 489–500, May 2013.