Rejuvenating Shadows: Fault Tolerance with Forward Progress

Xiaolong Cui, Taieb Znati, Rami Melhem
Computer Science Department
University of Pittsburgh
Pittsburgh, USA
Email: {mclarencui, znati, melhem}@cs.pitt.edu

***Abstract*—**

***Keywords*-Rejuvenation; Leaping; Extreme-scale computing; Forward progress; Reliability;**

I. Introduction

The exponential increase of computation power, enabled by exascale computing infrastructure, is vital to accelerate discoveries across the spectrum of scientific fields and successfully address challenges in a broad spectrum of practical and increasingly complex applications that have profound impacts on society. To achieve the level of fidelity these applications require, the exascale computing infrastructure is expected to deliver 50 times the performance of todays Petaflop HPC systems. Efforts are underway to meet this daunting challenge in the next decade. The path to exascale computing, however, involves several major road blocks and numerous challenges inherent to the complexity and scale of these systems. A key challenge for exascale computing systems stems from the stringent requirement, set by the US Department of Energy, to operate in a power envelope of 20 Megawatts. It is worth noting that simply scaling current HPC technology to achieve exascale performance will increase power consumption to 100 MW, resulting in unsustainable energy costs. Innovative design approaches, such as software and hardware co-design, need to be explored to integrate all components of an exascale infrastructure and investigate trade-offs to increase computational power by a factor of 10, while limiting the increase of power consumption to only a factor of 1.

The trend toward shrinking transistor geometries, coupled with the variability in chip manufacturing, which is highly likely to decrease significantly the reliability of computing and communication components, presents yet another key challenge on the path toward exascale computing. The sheer number of components in future exascale computing, order of magnitudes higher than in existing HPC systems, will lead to frequent system failures, significantly limiting computational progress in large scale applications [1], [2]. It is projected that the Mean Time Between Failures (MTBF) of future extreme-scale systems will be at the order of hours or even minutes, resulting in the occurrence of several failures on a daily basis [3]. New paradigms must be developed to cope with frequent faults and ensure that applications run to completion. The need to reduce time-to-solution for a wide range of compute- and data-intensive applications, while adhering to stringent power constraints and high-levels of resilience, further compound the exascale computing challenge.

The current approach to resilience relies on checkpointing and rollback recovery, whereby the execution state of an application is periodically saved to a stable storage, so that execution can be restarted from a recent checkpoint after a failure occurs [4], [5], [6]. As the rate of failure increases, however, the time required to periodically checkpoint and restart an application approaches the systems MTBF, which leads to a significant drop in the efficiency of checkpointing and rollback recovery [7]. A number of approaches have been proposed to address this shortcoming, focusing on hybrid in-memory and multilevel checkpointing methods, which rely both on coordinated and uncoordinated checkpointing and the use of NVRAM and DRAM for storage, to reduce overhead and improve execution time [8]. Although the additional power consumption remains low, the cost of additional memory may be prohibitive in exascale environments. Furthermore, the rollback to a consistent global state still requires that each node performs snapshots at different time and maintains a log of all incoming messages.

A second approach to achieve resilience in exascale, referred to as state machine replication, exploits hardware redundancy to overcome failures by executing simultaneously multiple instances of the same task on separate processors [9], [10], [11]. The physical isolation of processors ensures that faults occur independently, thereby enhancing tolerance to failure. The approach, however, suffers low efficiency, as it dedicates 50% of the computing infrastructure to the execution of replicas. Furthermore, achieving exascale performance, while operating within the 20 MW power cap, becomes challenging and may lead to high energy costs. To address these shortcomings, the *Shadow Replication* computational model, which explores radically different fault tolerance methodologies, has been proposed to address resilience in extreme-scale, failure-prone computing environments [12]. The basic tenet of Shadow Replication is to associate with each main process a suite of coordinated shadow processes, physically isolated from their associated main processes, to hide failures, ensure system level consis-

tency and meet the performance requirements of the underlying application. The size of the suite and computational attributes of the shadows depend on the elasticity of the domain application with respect to computational fidelity, fault-tolerance and bounds on time-to-solution. In failure-prone exascale environments, full replication, whereby multiple shadows run in parallel as exact replicas of their associated main process may be necessary in order to comply with stringent requirements of the application. For elastic applications whose performance requirements include multiple attributes, it may be desirable to trade response time to reduce energy consumption. In this case, a single shadow that runs as a replica of its associated main, but at a lower execution rate, may be sufficient to achieve acceptable response time. Similarly, trading fidelity to meet stringent response time requirements, in failure prone environment, can be achieved by running in parallel a single shadow, not as a full, but *differential* replica of its associated main. The shadow executes at the same speed as the main process, but at a lower computational resolution.

Lazy Shadowing, described in [13], is an adaptive power-aware instance of Shadowing Replication, which strikes a balance between energy consumption and time-to-completion in error-prone exascale computing infrastructure. Experimental results demonstrate its ability to achieve higher performance and significant energy savings in comparison to existing approaches in most cases. Despite its viability to tolerate failure in a wide range of exascale systems, Lazy Shadowing assumes that either the main or the shadow fails, but not both. Consequently, the resiliency of the system decreases as failure increases. Furthermore, when failure occurs, shadows are designed to substitute for their associated main. The tight coupling and ensuing fate sharing between a main and its shadow increases the implementation complexity of Lazy Shadowing. Finally, crash failures significantly reduce the efficiency of the system to deal with failures. In this paper, we introduce a new fault-tolerant computational model, referred to as *Rejuvenating Shadows*, to reduce Lazy Shadowing's vulnerability and enhance its performance. In this new model, shadows are no longer replicas, which are promoted to substitutes for their associated main processes, upon the occurrence of a failure. Instead, a shadow is a *rescuer* whose role is to restore their associated main to its exact state before failure. There are several benefits that can be obtained by using Rejuvenation as a building block of the Rejuvenating Shadow computational model to enhance system tolerance to failure. It ensures that the vulnerability of the system to failure does not increase, after a failure occurs. Furthermore, it can be used to handle different types of failure, including transient and crash failures. The main contributions of this paper are as follows:

- Proposal of Rejuvenating Shadows as an enhanced scheme of Lazy Shadowing that incorporates rejuvenation techniques for consistent reliability and improved performance.
- An full-feature implementation of Rejuvenating Shadows for Message Passing Interface.
- An implementation of application-level in-memory Checkpointing/Restart for comparison.
- A thorough evaluation of the overhead and performance of the implementation with various benchmark applications.

The rest of the paper is organized as follows. We begin with a survey on related work in Section II. Section III introduces system design and fault model, followed by discussion on implementation details in Section IV. Section V presents empirical evaluation results. Section VI concludes this work and points out future directions.

II. RELATED WORK

The study of fault tolerance has been fruitful for large-scale High Performance Computing [14]. Checkpointing/restart periodically saves the execution state to stable storage, with the anticipation that computation can be restarted from a saved checkpoint in the case of a failure. Assuming that all non-deterministic events can be identified, message logging protocols allow a system to recover beyond the most recent consistent checkpoint by combining checkpointing with logging of non-deterministic events [4]. Another way to cope with faults is proactive fault tolerance, which relies on a prediction model to forecast fault ahead of time and take preventive measures, such as task migration or checkpointing the application [15], [16]. Algorithm-based fault tolerance (ABFT) uses redundant information inherent of its coding of the problem to achieve resilience. Although the efficiency of ABFT can be orders of magnitude better than that of general techniques, it lacks generality [14].

For the past 30 years, disk-based coordinated checkpointing has been the primary fault tolerance mechanism in production HPC systems [11]. To achieve a globally consistent state, all processes coordinate with one another to produce individual states that satisfy the "happens before" relationship [17]. Coordinated Checkpointing gains its popularity for its simplicity and ease of implementation. Its major drawback, however, is the lack of scalability [18]. Uncoordinated checkpointing allows processes to record their states independently, thereby reducing the overhead during fault free operation [19], [20]. However, the scheme requires that each process maintains multiple checkpoints, necessary to construct a consistent state during recovery, and complicates the garbage collection scheme. It can also suffer the well-known domino effect [21].

One of the largest overheads in any checkpointing technique is the time to save checkpoint to stable storage. To mitigate this issue, multiple optimization techniques have been proposed, including incremental checkpointing, semi-blocking checkpointing, in-memory checkpointing, and

multi-level checkpointing [22], [23], [24], [25], [26], [27], [28]. Although well-explored, these techniques have not been widely adopted in HPC environments due to implementation complexity.

Recently, process replication has been proposed as a viable alternative to checkpointing in HPC, as replication can significantly increase system availability and achieve higher efficiency than checkpointing in failure-prone systems [29], [30]. In addition, full and partial replication have also been used to augment existing checkpointing techniques, and to guard against silent data corruption [31], [32], [33], [34].

Many efforts aim at providing fault tolerance to MPI, which is the de facto programming paradigm for HPC. Checkpointing and message logging are supported by either building them from scratch or integrating with existing solutions [35], [36], [37], [38], [27]. Several replication schemes are implemented in MPI, with the runtime overhead being negligible or up to 70% depending upon the application communication patterns [39], [11], [34]. ULFM provides a set of MPI extensions that enable the deployment of application specific fault tolerance strategies [40].

III. REJUVENATING SHADOWS MODEL

We adopt the fail-stop fault model, which defines that a process halts in response to a failure and its internal state and memory contents are irretrievably lost [41]. Furthermore, we assume that the machine on which failure occurred can be rebooted (or equally, a spare machine is available) for starting a new process after the failure. Note that this is the same assumption for checkpointing/restart. Below we discuss the components of the Rejuvenating Shadows model and its design trade-offs.

A. Shadowing

Shadowing is the essential concept that associates each original process (referred to as main) with a replica process (referred to as shadow), which potentially lower its execution rate to save power. Then fault tolerance comes from the property that if one process fails, its associated process can continue to complete the task. For example, if a main fails, its shadow will be promoted to a new main and continue to carry out the task. In this work, however, we deviate from the original concept of shadow as a replica [13], and use a shadow as a "rescuer" to a main in the presence of failures. Specifically, if a main fails, a new main process will be rejuvenated from its associated shadow with our leaping technique, while the shadow remains as a shadow.

State consistency between mains and shadows is required both during normal execution and following a failure. We designed a protocol, as shown in Figure 1, to enforce sequential consistency, i.e., each shadow sees the same message order and operation results as its main. For each message, the main of the sender sends a copy of the message to each of the main and shadow of the receiver, and the shadow of the sender is

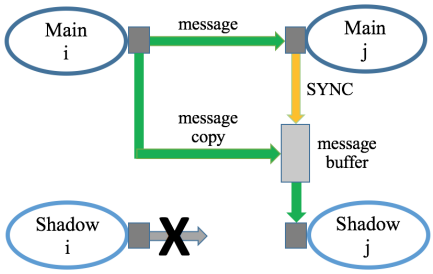


Figure 1: Consistency protocol for Rejuvenating Shadows.

suppressed from sending out messages. We assume that two copies of the same message are sent in an atomic manner, as this multicast functionality can be implemented within NIC. The SYNC message in Figure 1 is only used when there is potential non-determinism. This will be discussed in details in the next Section.

B. Leaping

Leaping is initially proposed as a technique to boost performance in [13]. As the shadows execute slower than mains, failure recovery introduces delay to the execution. Leaping opportunistically takes advantage of the recovery time and transfers state from each living main to its associated shadow. As a result, shadows achieve forward progress with minimal overhead, and recovery time for future failures, if any, is minimized.

Recently, we realized that leaping can also be used for rejuvenation (discussed below) to transfer state from a living shadow to a newly started main. Note that leaping always happens between a pair of main and shadow. To avoid ambiguity, the process which provides state in a leaping is referred to as target process, and the other process which receives and updates state is referred to as lagging process.

C. Rejuvenation

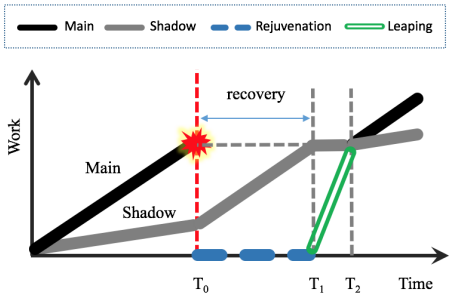
It has been discussed in [13] that each shadowed set can only tolerate one failure. After the first failure, all main processes in the shadowed set would lose their shadows and become vulnerable. Although quantitative study show that a second failure in a shadowed set is unlikely, in practice the system will become more and more vulnerable as failures occur. In many cases, it is too costly to take such risk, especially for long-running, large-scale, and mission-critical applications. Therefore, it is preferable to maintain the same level of resilience at all times.

Not surprisingly, vulnerability could be avoided by rejuvenation, in which we restart a new process for every failed process (either main or shadow). In this way, every main is always guaranteed to have an associated shadow, and shadow never needs to substitute a main. The problem, however, is that the newly launched process will start from the beginning and may lag far behind the other processes.

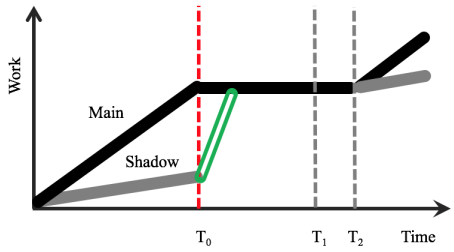
Later on when the new process needs to participate in a synchronization point, or when it needs to carry out a failure recovery, significant delay will incur as a result of its lag.

Fortunately, leaping is a natural solution. After a new process is started, we can use leaping to synchronize the new process' state with its associated living process. Hence, lag of the new process is resolved at the minimal cost of a leaping. Suppose a main M_i fails at T_0, Figure 2 illustrates the failure recovery process with rejuvenation. In order for its shadow S_i to speed up and finish the recovery as soon as possible, we will temporarily suspend the execution of the other shadows that collocate with S_i. Meanwhile, the failed machine is rebooted and then used to launch a new process for M_i. When S_i catches up with the state of the previous M_i before the failure at T_1, leaping can be used to advance the new M_i to the current state of S_i. The leaping is initiated after S_i catches up so as to avoid the need of message logging for restarting M_i. Because of the failure of M_i, the other mains will be blocked when they arrive at the next synchronization point, which is also assumed at T_0. During the idle time, we can opportunistically perform a leaping and transfer state from each living main to its associated shadow. Therefore, this leaping has minimal overhead as it overlaps with the recovery, as shown in Figure 2(b). Leaping for the shadows collocated with S_i are delayed until the recovery completes at T_1, since these shadows are suspended during the recovery. After the leaping finishes at T_2, all mains and shadow can resume normal execution with the same level of resilience as before the failure.

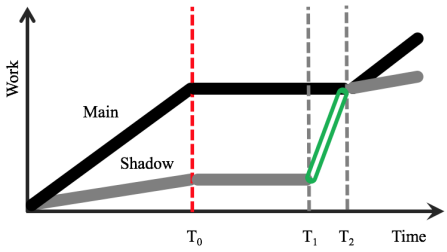
Figure 2 and the above analysis assume that the time for rebooting is no longer than the recovery time. If the new M_i is not yet ready when S_i catches up at T_1, however, we have two design choices: 1) S_i can continue execution and take the role of a main; or 2) S_i can wait for the rebooting to finish and the new M_i to launch, and then perform the leaping. The first option requires a shadow to starting sending out messages, as well as all the other processes to update their internal process mapping in order to correctly receive messages from this shadow (see Figure 1). This not only complicates the implementation, but also requires expensive global coordination that is detrimental to scalability. We therefore chose the second design.



(a) Faulty task



(b) Non-faulty tasks in different shadowed sets



(c) Non-faulty tasks in the same shadowed set

Figure 2: Recovery and rejuvenation after a main process fails.

IV. IMPLEMENTATION

This section presents the details of our full-feature implementation of rsMPI, which is an MPI library for Rejuvenating Shadows. Similar to rMPI and RedMPI [11], [34], rsMPI is implemented as a separate layer between MPI and user application. It uses the standard MPI profiling interface to intercept every MPI call and enforces Rejuvenating Shadows logic. In this way, we not only can take advantage of existing MPI performance optimization that numerous researches have spent years on, but also achieve portability across all MPI implementations that conform to the MPI specification. When used, rsMPI transparently spawns the shadow processes during the initialization phase, manages the coordination between main and shadow processes, and guarantees order and consistency for messages and non-deterministic MPI events.

A. MPI rank

A rsMPI world has 3 types of identities: main process, shadow process, and coordinator process that coordinates between main and shadow. A static mapping between srMPI rank and application-visible MPI rank is maintained so that each process can retrieve its identity. For example, if the user specifies N processes to run, rsMPI will translate it into $2N + K$ processes, with the first N ranks being the mains, the next N ranks being the shadows, and the last K ranks being the coordinators. We also statically group the processes into shadowed sets according to a user configuration file. Figure 3 shows an example rsMPI world with 16 application-visible processes grouped into 4 shadowed sets. Using the MPI profiling interface, we added wrapper

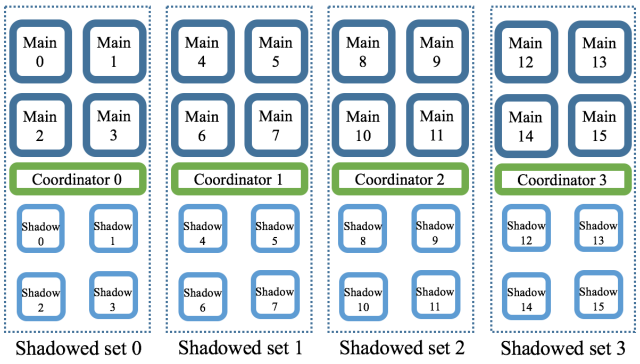


Figure 3: Logical organization of a MPI world for Rejuvenating Shadows. $N = 16$, $K = 4$.

for MPI_Comm_rank() and MPI_Comm_size(), so that each process (main or shadow) gets its correct execution path.

B. Execution rate control

While the mains always execute at maximum rate for HPC's throughput consideration, the shadows can be configured to execute slower by collocation [13]. When using rsMPI, the user has the flexibility to specify the desired collocation ratio (the number of shadows to collocate on each processing unit) through a configuration file that we provide. Accordingly, rsMPI will generate an MPI rankfile and provide it to the MPI runtime to control the process mapping. Note that rsMPI always maps the main and shadow processes of the same task onto different nodes. This is preferred as a fault on a node will not affect both of them. To minimize resource usage, each coordinator is collocated with all the shadows in the shadowed set. Since each coordinator simply waits for incoming control messages (discussed below) and does minimal work, it has negligible impact on the execution rate of the collocated shadows.

C. Coordination between mains and shadows

Each shadowed set has a coordinator process dedicated to coordination between the main and shadow processes in the shadowed set. Coordinators do not execute any application code, but just wait for srMPI defined control messages, and then carry out some coordination work accordingly. There are three types of control messages, i.e., termination, failure, and leaping. Correspondingly, each coordinator is responsible for three types of work:

- when a main finishes, it notifies its coordinator, which then forces the associated shadow to terminate.
- when a main fails, its associated shadow needs to catch up, so the coordinator will temporarily suspend the other collocated shadows, and resume their execution once the recovery is done.
- when a main or shadow initiates a leaping, either because of failure or buffer overflow, the coordinator triggers leaping at the associated shadow or main.

To separate control messages from data messages, rsMPI uses a dedicated MPI communicator, created during MPI initialization, for the control messages. In addition, to ensure fast response and minimize the number of messages, coordinators also use OS signals to communicate with their collocated shadows.

D. Message passing and consistency

We wrapped around every MPI communication function and implemented the consistency protocol described in Section III-A. For sending functions, such as MPI_Send() and MPI_Isend(), rsMPI requires the main to duplicate the sending while the shadow does no work. For receiving functions, such as MPI_Recv() and MPI_Irecv(), both the main and the shadow does one receiving from the main process at the sending side. Internally, collective communication in rsMPI uses point-to-point communication in a binomial tree topology, which demonstrates excellent scalability.

We assume that only MPI operations can introduce nondeterminism, and the SYNC message shown in Figure 1 is introduced to enforce determinism. MPI_ANY_SOURCE may result in different message receiving orders between a main and its shadow. To deal with this, we serialize MPI_ANY_SOURCE message receiving by having the main first do the receiving and then use a SYNC message to forward the message source information to its shadow, which then issues a receiving with the specific source. Other operations, such as MPI_Wtime() and MPI_Probe(), can be dealt with in a similar manner by forwarding the result from a main to its shadow.

E. Leaping

Checkpointing/restart requires each process to save its execution state, which can be used later to retrieve the computation. Leaping is similar to Checkpointing/restart in saving a process' state, but the state is always transferred between a pair of main and shadow. To reduce the size of data involved in saving a process' state, we choose to implement leaping in the same way as application-level checkpointing. rsMPI provides a routine for users to register any data as process state. Application developer could use domain knowledge to identify only necessary state data, or use compiler techniques to automate this [42].

rsMPI provides the following API for process state registration:

void leap_register_state(void *addr, int count, MPI_Datatype dt);

For each piece of data to be registered, three parameters are needed: a pointer to the address of the data, the number of data items, and the datatype. Internally, rsMPI uses a linked list to keep track of all registered data. During leaping,

the linked list is traversed to retrieve all registered data as the process state.

Coordination of leaping is easier than coordination of checkpointing, since leaping is always between a pair of main and shadow. To synchronize the leaping between a main and a shadow, the coordinator in the corresponding shadowed set is involved. For example, when a main detects failure of another main and initiates a leaping, it will send a control message to its coordinator, which then uses a signal to notify the associated shadow to participate in the leaping.

Different from Checkpointing where the process state is saved to storage, leaping directly transfers process state between a main and its shadow. Since MPI provides natural support for message passing between processes, rsMPI uses MPI messages to transfer process state. Although multiple pieces of data can be registered as a process' state, only a single message is needed to transfer the process state, as MPI supports derived datatypes. To prevent the messages carrying process state from mixing with application messages, rsMPI uses a separate communicator for transferring process state. With the synchronization of leaping by coordinator and the fast transfer of process state via MPI messages, the overhead of leaping is minimized.

A challenge in leaping lies in the need for maintaining state consistency across leaping. To make sure a pair of main and shadow stay consistent after a leaping, not only user-defined states should be transferred correctly, but also lower level states, such as program counter and message buffer, need to be updated correspondingly. Specifically, the lagging process needs to satisfy two requirements. Firstly, after leaping the lagging process should discard all obsolete messages before resuming normal execution. Secondly, the lagging process should resume execution at the same point as the target process. We discuss our solutions below, under the assumption that the application's main body consists of a loop, which is true in most cases.

To satisfy the second requirement, we restrict leaping to always occur at certain possible points, and uses internal counter to make sure that both lagging and target processes start leaping from the same point. For example, when a main process triggers a leaping and asks coordinator to notify its associated shadow, the coordinator will trigger the shadow's specific signal handler. The signal handler does not carry out leaping, but sets a flag for leaping and receives a counter value that indicates the leaping point from its main process. Then, the shadow will check the flag and compare the counter value at every possible leaping point. Only when both the flag is set and counter value matches will the shadow start leaping. In this way, it is guaranteed that after leaping the main and shadow will resume execution from the same point. To balance the trade-off between the implementation overhead and the flexibility of checking for when to perform leaping, we choose MPI receive operations as the possible leaping points.

There is no straightforward solution for the first problem, as the message buffer is maintained by MPI runtime and not visible to rsMPI. Alternatively, rsMPI borrows the idea of message logging to correctly discard all obsolete messages. During normal execution, both the main and shadow record the meta data (i.e., MPI source, tag, and communicator) for all received messages in the receiving order. During leaping, the meta data at the main is transferred to the shadow, so that the shadow knows about the messages that have been received by its main but not by itself. Then the shadow combines MPI probe and MPI receive operations to remove the messages from MPI runtime buffer in correct order.

V. EVALUATION

We deployed rsMPI on a medium sized cluster and utilized up to 21 nodes for testing and benchmarking. Each node consists of a 2-way SMPs with Intel Haswell E5-2660 v3 processors of 10 cores per socket (20 cores per node), and is configured with 128 GB RAM. Nodes are connected via 56 GB/s FDR InfiniBand. To maximize the compute capacity, we used up to 20 cores per node.

We used benchmark applications from both the Sandia National Lab Mantevo Project and NAS Parallel Benchmarks (NPB), and evaluated rsMPI with various problem sizes and number of processes. CoMD is a proxy for the computations in a typical molecular dynamics application. MiniAero is an explicit unstructured finite volume code that solves the compressible Navier-Stokes equations. Both MiniFE and HPCCG are proxy applications for unstructured implicit finite element codes, but HPCCG uses MPI_ANY_SOURCE receive operations and can be used to demonstrate rsMPI's capability of handling MPI nondeterministic events. IS, EP, and CG from NPB represent integer sort, embarrassingly parallel, and conjugate gradient applications, respectively. These applications cover key simulation workloads for US DOE, and represent both different communication patterns and computation-to-communication ratios.

A. Measurement of runtime overhead

While the hardware overhead for rsMPI is straightforward (e.g., collocation ratio of 4 results in the need for 25% more hardware cost), the runtime overhead of the enforced consistency protocol depend on applications. To measure this overhead we ran each benchmark application linked to srMPI multiple times and compared the average execution time with the baseline, where each application runs with original OpenMPI.

Figure 4 shows the comparison of the execution time between baseline and srMPI for the 7 applications. All the experiments are conducted with 256 application-visible processes. That is, the baseline always uses 256 MPI ranks compiled with the unmodified OpenMPI library, while rsMPI uses 256 mains together with 256 shadows which are

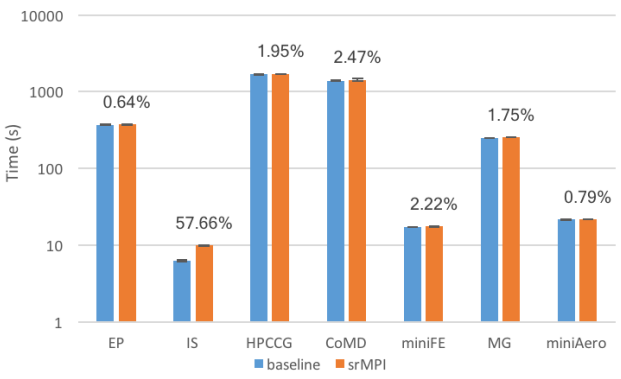


Figure 4: Comparison of execution time between baseline and rsMPI. 256 application-visible processes, and collocation ratio of 4 for srMPI.

invisible to the application. Each result shows the average execution time of 5 runs, the standard deviation, and srMPI's runtime overhead. The baseline execution time varies from seconds to half an hour, so we plotted the time in log-scale.

From the figure we can see that srMPI has comparable execution time to the baseline for all applications except IS. The reason for the large overhead of IS is that IS uses all-to-all communication and is largely communication-intensive. This is verified by adding fake computation to the application and we can see an immediate drop of the overhead to negligible level. We argue that communication-intensive applications like IS are not scalable, and as a result, they are not suitable for large-scale HPC. For all other applications, the overhead varies from 0.64% (EP) to 2.47% (CoMD). Even for HPCCG, which uses MPI_ANY_SOURCE and adds extra work to our consistency protocol, the overhead is only 1.95%, thanks to the non-blocking semantics of MPI_Send. Therefore, we conclude that srMPI's runtime overheads are modest for scalable HPC applications that exhibit a fair communication-to-computation ratio.

B. Scalability

In addition to measuring the runtime overhead at fixed application-visible process count, we also assessed both strong and weak scalability by varying the number of processes for the applications. Strong scaling is defined as how the execution time varies with the number of processes for a fixed total problem size. In contrast, weak scaling is defined as how the execution time varies with the number of processes for a fixed problem size per process.

Among the seven applications, HPCCG, CoMD, and miniAero allow us to vary the input so that we can perform both strong scaling and weak scaling test. The results for miniAero are similar to those of CoMD, so we only show the results for HPCCG and CoMD here. Figure 5 reveals that both HPCCG and CoMD have good strong scalability. By increasing the number of processes, we can always reduce the execution time for a fixed problem size. At the same time, srMPI's runtime overhead increases with the number of processes during the strong scalability test. At 256 processes, the overhead reaches 13.2% for CoMD, and 29.1% for HPCCG. This may seem to contradict with the results in Section V-A. It is expected, however, since increasing the number of processes while keeping a constant problem size increases the communication-to-computation ratio of the application. Hence, to keep rsMPI overheads reasonable, it is important to choose input sizes such that the ratio of communication-to-computation is balanced.

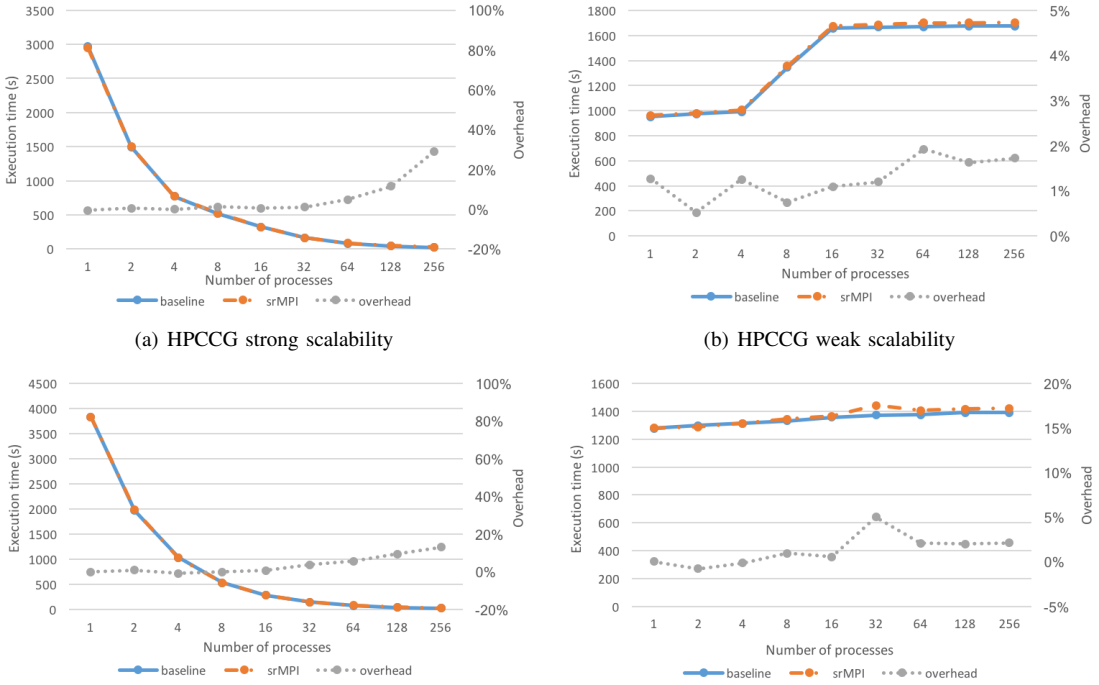
Comparing the baseline execution time between Figure 5(b) and Figure 5(d), it is obvious that HPCCG and CoMD have different weak scaling characteristics. Keeping the same problem size per process, the execution time for CoMD increases by 8.9% from 1 process to 256 processes, while the execution time is almost doubled for HPCCG. However, further analysis show that from 16 to 256 processes, the execution time increases by only 2.5% for CoMD, and 1.0% for HPCCG. We suspect that the results are not only determined by the scalability of the application, but also impacted by other factors, such as cache and memory contention on the same node, and network interference from other jobs running on the cluster. Remember that each node in the cluster has 20 cores and we always use all the cores of a node before adding another node. Therefore, it is very likely that the node level contention leads to the substantial increase in execution time for HPCCG. By analyzing the results from 16 to 256 processes, we believe both of HPCCG and CoMD are weak scaling applications.

Different from strong scalability test, there is no correlation between srMPI's runtime overhead and the number of processes during the weak scalability test. The overhead is always below 2.1%, except for the case of 32 processes for CoMD where the overhead is 5.0%.

C. Performance under failures

As one main goal of this work is to achieve fault tolerance, an integrated fault injector is required to evaluate the effectiveness and efficiency of rsMPI to tolerate failures during execution. To produce failures in a manner similar to naturally occurring process failures, our failure injector is designed to be distributed and co-exist with all rsMPI processes. Failure is injected by sending a specific signal to the target process.

Failure detection is beyond the scope of srMPI, and we assume the underlying hardware platform has a RAS system that provides this functionality. In our test system, we emulate a RAS system with a signal handler installed at every main and shadow. The signal handler catches failure signal sent from the failure injector, and uses a rsMPI defined failure message via a dedicated communicator to notify all other processes of the failure. Similar to ULFM, process in srMPI can only detect failure when it does an MPI receive



(a) HPCCG strong scalability

(b) HPCCG weak scalability

(c) CoMD strong scalability

(d) CoMD weak scalability

Figure 5: Scalability test for number of processes from 1 to 256. Collocation ratio is 4 for srMPI.

operation. In a srMPI receive, a process checks for failure messages before it does the actual MPI receive operation.

We also implemented checkpointing to compare with srMPI in the presence of failures. To be optimistic, we chose double in-memory checkpointing that is much more scalable then disk-based checkpointing [25]. Same as leaping in srMPI, our implementation provides an API for process state registration. This API requires the same parameters as leap_register_state(void *addr, int count, MPI_Datatype dt), but internally, it allocates extra memory in order to store the state of a "buddy" process. Another provided API is checkpoint(), which can be used to insert a checkpoint in the application code. For fairness, our implementation also uses MPI messages to transfer state. For both srMPI and checkpointing/restart, we assume a 60 seconds rebooting time after a failure. All experiments run with 256 application-visible processes, and the results are average of 5 runs.

Firstly, we tested the effectiveness of leaping. For each application, we identified the process state and register them with rsMPI. Figure 6 shows the execution time of HPCCG with a single failure injected at various locations. The blue solid line represents srMPI without any forced leaping, and the red dash line represents srMPI with periodic forced leaping. Note that the execution time is reduced compared to previous results because we reduced the number of iterations for the application main loop from 5000 to 150, so that there is no need for any forced leaping by buffer overflow.

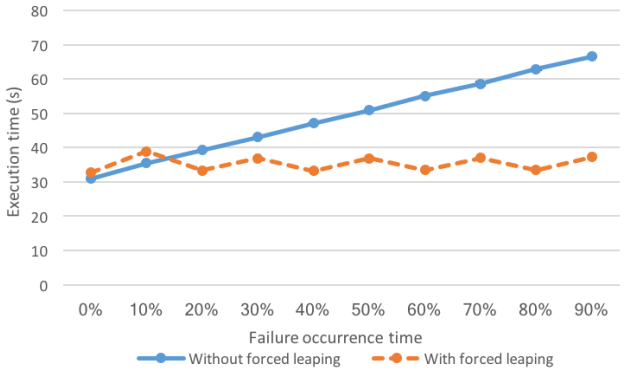


Figure 6: Execution time of HPCCG with a single injected failure. Collocation ratio is 2 for srMPI.

Every time we set our failure injector to randomly pick a process to inject a failure, and the failure is scheduled to occur at certain point during the execution. Corresponding to the x-axis, the scheduled failure time varies from 10% to 90% of the application's execution. For example, 10% means the application completes 15 iterations for a total of 150 iterations.

As expected, without forced leaping the execution time increases with the failure occurrence time, as reflected by the blue line in Figure 6. The reason is that failure recovery

time for srMPI is proportional to the amount of divergence between mains and shadows, and the divergence grows as the execution proceeds. On the other hand, forced leaping can effectively reduce the divergence by leaping the shadow forward to the state of its associated main, similar to the idea that checkpointing can reduce the amount of wasted work due to a failure by saving the execution state. To prove the effectiveness of leaping, we insert 4 forced leaping at 20%, 40%, 60% and 80% of the execution. The red line in Figure 6 clears show that the divergence effect is bounded due to periodic leaping, regardless of the failure occurrence time.

Next, we compare rsMPI with checkpointing for multiple failures. To run the same number of application-visible processes, rsMPI needs more nodes than checkpointing to host the shadow processes. For fairness, we take into account the extra hardware cost when comparing srMPI to checkpointing, by defining the following metric:

$$\text{Efficiency} = \frac{T_f \times N}{T_e \times M}$$

, where T_f and N are the execution time and number of nodes without failures, and T_e and M are the actual execution time and required number of nodes for a specific fault tolerance mechanism. Intuitively, $T_f \times N$ represents the total amount of workload required by the application, and $T_e \times M$ is the actual amount of work carried out. The efficiency will be in the range 0 to 1, inclusive, and the higher is the better.

The forced leaping interval for an application is selected such that no buffer overflow at the shadows would take place. Therefore, the interval should vary from system to system and also depends on the application patterns. We assume checkpointing/restart has the same buffer pressure as it needs to perform message logging, so its checkpointing interval is selected based on the same metric as rsMPI. We evaluated rsMPI with 2 different collocation ratios, i.e., 2 and 4. When collocation ratio is 2, rsMPI uses 50% more nodes than checkpointing, and the execution rate of each shadow is roughly 50% of the processor rate. Therefore, we set the checkpointing interval to be the same as the forced leaping interval for srMPI. When collocation ratio is 4, rsMPI needs 25% more nodes, and each shadow's rate is roughly 25% of the processor rate. As a result, we loose the checkpointing interval to be twice of the forced leaping interval.

With the checkpointing and forced leaping inserted to the application code, we randomly injected up to 10 failures into the execution. Figure 7 shows the comparison between checkpointing and srMPI (collocation ratio of 4) for both execution time and efficiency defined above. Although the failure-free execution time of srMPI is slightly larger than that of checkpointing, which results from srMPI's consistency protocol, the failure recovery time of checkpointing immediately overwhelms that of srMPI as failures occur. With 10 failures, the execution time of checkpointing is

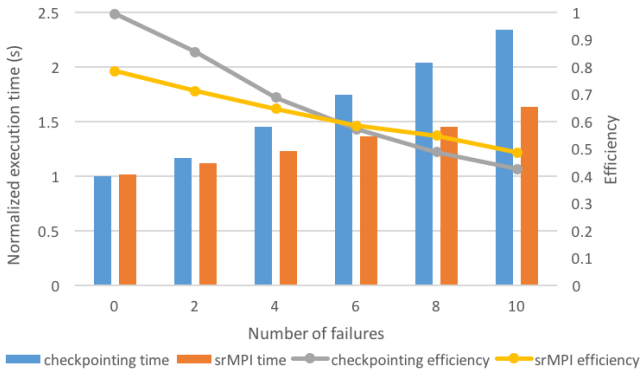


Figure 7: Execution time of HPCCG with multiple injected failures. Collocation ratio is 4 for srMPI.

42.8% more than that of srMPI. Considering hardware overhead, the efficiency of checkpointing is also worse than that of srMPI when the number of failures reaches 6.

Between srMPI with collocation ratio of 2 and srMPI with collocation ratio of 4, srMPI with collocation ratio of 2 wins in execution time, while srMPI with collocation ratio of 4 wins in efficiency.

VI. CONCLUSION AND FUTURE WORK

ACKNOWLEDGMENT

This research is based in part upon work supported by the Department of Energy under contract DE-SC0014376.

REFERENCES

[1] S. Ashby, B. Pete, C. Jackie, and C. Phil, "The opportunities and challenges of exascale computing," 2010.

[2] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson *et al.*, "Addressing failures in exascale computing," *International Journal of High Performance Computing Applications*, p. 1094342014522573, 2014.

[3] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[4] E. Elnozahy and et. al., "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[5] S. Kalaiselvi and V. Rajaraman, "A survey of checkpointing algorithms for parallel and distributed computers," *Sadhana*, vol. 25, no. 5, pp. 489–510, 2000. [Online]. Available: http://dx.doi.org/10.1007/BF02703630

[6] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985. [Online]. Available: http://doi.acm.org/10.1145/214451.214456

[7] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *Int. J. High Perform. Comput. Appl.*, vol. 23, no. 4, pp. 374–388, Nov. 2009. [Online]. Available: http://dx.doi.org/10.1177/1094342009347767

[8] S. Gao, B. He, and J. Xu, "Real-time in-memory checkpointing for future hybrid memory systems," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751212

[9] J. F. Bartlett, "A nonstop kernel," in *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, ser. SOSP '81. New York, NY, USA: ACM, 1981, pp. 22–29. [Online]. Available: http://doi.acm.org/10.1145/800216.806587

[10] W.-T. Tsai, P. Zhong, J. Elston, X. Bai, and Y. Chen, "Service replication strategies with mapreduce in clouds," in *Autonomous Decentralized Systems, 10th Int. Symp. on*, 2011, pp. 381–388.

[11] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, 2011, pp. 44:1–44:12.

[12] B. Mills, "Power-aware resilience for exascale computing," Ph.D. dissertation, University of Pittsburgh, 2014.

[13] X. Cui, T. Znati, and R. Melhem, "Adaptive and power-aware resilience for extreme-scale computing," in *16th IEEE International Conference on Scalable Computing and Communications*, July 18-21 2016.

[14] T. Hérault and Y. Robert, *Fault-Tolerance Techniques for High-Performance Computing*. Springer, 2015.

[15] S. Chakravorty, C. L. Mendes, and L. V. Kalé, "Proactive fault tolerance in mpi applications via task migration," in *International Conference on High-Performance Computing*. Springer, 2006, pp. 485–496.

[16] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, "Fault prediction under the microscope: A closer look into hpc systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 77.

[17] K. Chandy and C. Ramamoorthy, "Rollback and recovery strategies for computer programs," *Computers, IEEE Transactions on*, vol. C-21, no. 6, pp. 546–556, June 1972.

[18] P. Hargrove and J. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1, 2006, p. 494.

[19] J. Plank and M. Thomason, "The average availability of parallel checkpointing systems and its importance in selecting runtime parameters," in *Fault-Tolerant Computing*, 1999, pp. 250 –257.

[20] A. Guermouche and et. al., "Uncoordinated checkpointing without domino effect for send-deterministic mpi applications," in *IPDPS*, May 2011, pp. 989–1000.

[21] B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM, 1975.

[22] S. Agarwal and et. al., "Adaptive incremental checkpointing for massively parallel systems," in *ICS 04*, St. Malo, France.

[23] J. Plank and K. Li, "Faster checkpointing with n+1 parity," in *Fault-Tolerant Computing*, June 1994, pp. 288–297.

[24] E. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit," *TC*, vol. 41, pp. 526–531, 1992.

[25] G. Zheng, L. Shi, and L. V. Kalé, "Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi," in *Cluster Computing, 2004 IEEE International Conference on*. IEEE, 2004, pp. 93–103.

[26] X. Ni, E. Meneses, and L. V. Kalé, "Hiding checkpoint overhead in hpc applications with a semi-blocking algorithm," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*. IEEE, 2012, pp. 364–372.

[27] A. Moody, G. Bronevetsky, K. Mohror, and B. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC*, 2010, pp. 1–11.

[28] D. Hakkarinen and Z. Chen, "Multilevel diskless checkpointing," *Computers, IEEE Transactions on*, vol. 62, no. 4, pp. 772–783, April 2013.

[29] R. Riesen, K. Ferreira, J. R. Stearley, R. Oldfield, J. H. L. III, K. T. Pedretti, and R. Brightwell, "Redundant computing for exascale systems," December 2010.

[30] F. Cappello, "Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212–226, 2009.

[31] J. Stearley and et. al., "Does partial replication pay off?" in *DSN-W*, June 2012, pp. 1–6.

[32] J. Elliott and et. al., "Combining partial redundancy and checkpointing for HPC," in *ICDCS '12*, Washington, DC, US.

[33] X. Ni, E. Meneses, N. Jain, and L. V. Kalé, "Acr: Automatic checkpoint/restart for soft and hard error protection," ser. SC. New York, NY, USA: ACM, 2013, pp. 7:1–7:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503266

[34] D. Fiala and et. al., "Detection and correction of silent data corruption for large-scale high-performance computing," ser. SC, Los Alamitos, CA, USA, 2012.

[35] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The lam/mpi checkpoint/restart framework: System-initiated checkpointing," *The International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, 2005.

[36] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette *et al.*, "Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes," in *Supercomputing, ACM/IEEE 2002 Conference*. IEEE, 2002, pp. 29–29.

[37] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring automatic, online failure recovery for scientific applications at extreme scales," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 895–906.

[38] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for open mpi," in *2007 IEEE International Parallel and Distributed Processing Symposium*, March 2007, pp. 1–8.

[39] C. Engelmann and S. Böhm, "Redundant execution of hpc applications with mr-mpi," in *PDCN*, 2011, pp. 15–17.

[40] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and

J. J. Dongarra, "An evaluation of user-level failure mitigation support in mpi," in *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 193–203.

[41] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Trans. Comput. Syst.*, vol. 1, no. 3, pp. 222–238, Aug. 1983. [Online]. Available: http://doi.acm.org/10.1145/357369.357371

[42] G. Bronevetsky, D. Marques, K. Pingali, S. McKee, and R. Rugina, "Compiler-enhanced incremental checkpointing for openmp applications," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–12.