Understanding the Effects of Communication and Coordination on Checkpointing at Scale

Kurt B. Ferreira and Patrick Widener
Scalable System Software
Sandia National Laboratories Albuquerque, NM
{kbferre,pwidene}@sandia.gov

Scott Levy and Dorian Arnold
Department of Computer Science
University of New Mexico
Albuquerque, NM
{slevy,darnold}@cs.unm.edu

Torsten Hoefler
Computer Science Department
ETH Zürich
Switzerland
htor@inf.ethz.ch

***Abstract*—Fault-tolerance poses a major challenge for future large-scale systems. Active research into coordinated, uncoordinated, and hybrid checkpointing systems has explored how the introduction of asynchrony can address anticipated scalability issues. However, few insights into selection and tuning of these protocols for applications at scale have emerged. In this paper, we use a simulation-based approach to show that local checkpoint activity in resilience mechanisms can significantly affect the performance of key workloads, even when less than 1% of a local node's compute time is allocated to resilience mechanisms (a very generous assumption). Specifically, we show that even though much work on uncoordinated checkpointing has focused on optimizing message log volumes, local checkpointing activity may dominate the overheads of this technique at scale. Our study shows that local checkpoints lead to process delays that can propagate through messaging relations to other processes causing a cascading series of delays. We demonstrate how to tune hierarchical uncoordinated checkpointing protocols designed to reduce log volumes to significantly reduce these synchronization overheads at scale. Our work provides a critical analysis and comparison of coordinated and uncoordinated checkpointing and enables users and system administrators to fine-tune the checkpointing scheme to the application and system characteristics.**

I. Introduction

In response to alarming projections of high failure rates due to increasing scale and complexity of high-performance computing (HPC) systems [1], many researchers have focused on methods and techniques for resilient extreme-scale HPC systems and applications. Considering non-algorithm-specific resilience approaches, researchers have studied both coordinated checkpoint/restart (cCR) and uncoordinated checkpoint/restart (uCR) protocols, with cCR having emerged as the de facto standard.

cCR protocols preempt all application processes to record a snapshot of the application's global state. cCR is attractive for several reasons. Its coordination protocol guarantees that the most recent global checkpoint captures a consistent global view, removing the need to store multiple checkpoints, sent messages, or other additional state information and thereby minimizing storage requirements. cCR also admits a relatively simple recovery procedure that does not suffer from *rollback propagation*, a scenario in which the most recent checkpoints from each application process do not comprise a consistent global state [2]. cCR does suffer from I/O contention issues since all processes checkpoint simultaneously, and with cCR protocols, upon a failure, even the surviving processes are perturbed as they must rollback to their most recent checkpoint. The rework executed by surviving processes also results in potentially unnecessary energy expenditures.

uCR protocols, in which each process in an application makes independent decisions about when to checkpoint, can mitigate cCR's I/O contention problem since processes are not forced to take checkpoints simultaneously. Additionally, when uCR is coupled with message logging, when failures occur, surviving processes are not forced to rollback to their most recent checkpoint and therefore can *run ahead* in their execution—unless and until they depend on a message from a failed process.

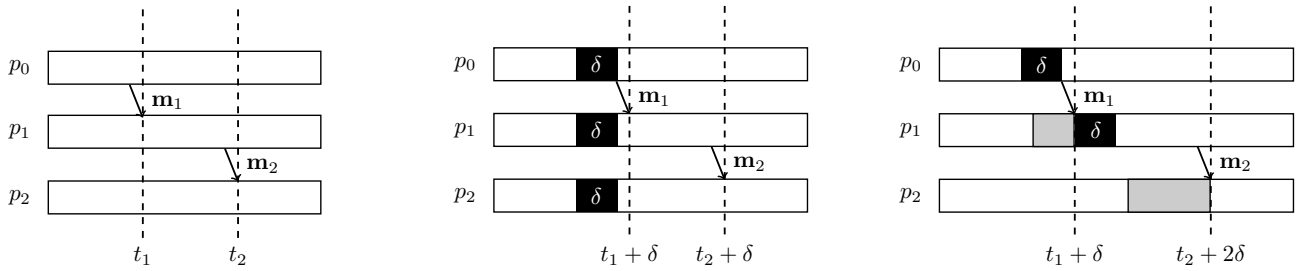
In order to better understand the potential benefits of uCR, in this paper we present in-depth analyses that reveal how the interplay between application and uCR protocol activities affects performance. We anticipate that this interplay will become increasingly important since many cCR implementations reduce their coordination costs by leveraging the application-level synchronization in programs that use the Bulk Synchronous Programming (BSP [3]) model, and future HPC designs are trending away from BSP. More specifically, using a wide array of applications, we demonstrate that when using uCR protocols with message logging[1] an application's communication activity coupled with local checkpoint overhead and the degree of synchronization amongst checkpoints from different processes can have a significant impact on overall application performance, in that local checkpointing activity introduces overheads that can be amplified or absorbed depending on an application's communication activities. We use profiles of these communication activities as well as micro-benchmark studies to attribute more precisely each application's coarse performance to its finer composition of communication operations. Lastly, we demonstrate the potential for tuning application performance by varying the degree of checkpoint synchronization by clustering checkpointing groups.

The possibility of uCR protocol activities inducing delays amongst processes, including processes that do not communicate directly with each other, is analogous to the manner in

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

[1] In this paper, we assume message logging is always used with uCR.

SC14, November 16-21, 2014, New Orleans, LA, USA
978-1-4799-5500-8/14/$31.00



(a) w/o checkpointing (b) coordinated checkpointing (c) uncoordinated checkpointing

Fig. 1. Propagation of uncoordinated checkpointing delay through application communication dependencies. The processes p_1, p_2, and p_3 exchange two messages m_1 and m_2 in each of the three scenarios. The black regions denote coordinated (b) and uncoordinated (c) checkpoint delays marked with δ.

which operating system noise can affect HPC applications [4], [5]. Figure I illustrates this phenomenon. Subfigure 1(a) shows a simple application running across three processes (p_0, p_1, and p_2). These three processes exchange two messages, $\mathbf{m}_1$ and $\mathbf{m}_2$. We assume here that these messages represent strict dependencies: any delay in the arrival of a message requires the recipient to stall until the message is received. Subfigure 1(b) shows the impact of coordinated checkpoint/restart (cCR). Because all of the checkpointing activity is coordinated across processes, the relative progress of the processes is unperturbed and all of the dependencies are satisfied at the appropriate time. Subfigure 1(c) illustrates the potential impact of relaxing the coordination requirement in uCR. If p_0 initiates a checkpoint at the instant before it would have otherwise sent $\mathbf{m}_1$, then p_1 is forced to wait (the waiting period is shown in grey) until the message arrives. If p_1 subsequently initiates a checkpoint before sending $\mathbf{m}_2$, then p_2 is forced to wait. Part of the time that p_2 spends waiting is due to a delay that was originated by p_0. The key point is that without coordination, checkpointing delays can propagate based on communication dependencies in the application.

Based on the studies presented in this paper, we make the following contributions:

- we demonstrate that even with very low resilience overheads, the cost and degree of synchronization amongst local checkpoints can have a greater performance impact than message logging overheads;
- we analyze how an application's communication pattern dictates whether local checkpoint overhead is amplified or absorbed by other processes;
- we show that CR protocols which leverage process clustering can be used to tune an application's performance sensitivity to local checkpointing activities; and
- we show that, in uCR protocols, when failures occur collective application-level communication can limit the extent to which the execution run-ahead of surviving processes actually improves overall application execution time for many practical workloads.

We proceed by describing the modeling and simulation tools that we use to execute these studies. Then, we present the results from our characterization of the performance of several application workloads with uCR (Section III). Next, we break down, using application profiles and microbenchmarks, how an application's communication activities can impact overall application performance with uCR (Section IV). Based on these results, we show how changing the degree of checkpoint synchronization can potentially impact performance (Section V). Finally, after an overview of related research works, we conclude by discussing the anticipated impacts of our results.

II. Predicting Checkpoint/Restart Performance

In this section, we describe our simulation-based approach to predicting the behavior of extreme scale applications with uCR. We begin by discussing how cCR overheads are currently modeled and why those techniques cannot be applied directly to uCR.

A. Modeling Sequential Checkpoint/Restart

For a *serial* computation, we can model the execution time of an application using CR by:

$$T = \mathrm{T_{app}} + \mathrm{T_c} + \mathrm{T_{rework}} + \mathrm{T_r} \tag{1}$$

Here $\mathrm{T_{app}}$ is the uninterupted application execution time (without any resiliency mechanisms or failures), $\mathrm{T_c}$ and $\mathrm{T_r}$ are the cumulative costs of all checkpoints and restarts, respectively, and $\mathrm{T_{rework}}$ is the cumulative cost of all lost work, that is work performed after the last checkpoint and before each failure. Let M, the Mean Time To Interrupt (MTTI), be the mean of a Poisson distributed random variable that models component failures in a system and δ the time to perform a checkpoint. Daly [6] shows that the serial application execution time using CR can be estimated as

$$T(\tau) = \mathrm{T_{app}} + (k-1)\delta + k\left(\frac{\tau+\delta}{2} + R\right)\left(\frac{\tau+\delta}{M}\right) \tag{2}$$

for a recovery time[2] R and a checkpoint interval τ (assuming $\mathrm{T_{app}} = k\tau, k \in \mathbb{N}$).

Assuming $\delta < 2M$, this formula can be used to derive the optimal checkpointing interval of a serial application, $\hat{\tau}$ [6].

$$\hat{\tau} = \sqrt{2\delta M}\left[1 + \frac{1}{3}\sqrt{\frac{\delta}{2M}} + \frac{1}{9}\left(\frac{\delta}{2M}\right)\right] \tag{3}$$

B. Modeling Distributed Checkpoint/Restart

If we assume that all component failures are identical, then $\hat{\tau}$ is also the optimal checkpointing interval for each node

[2]"Recovery time" is the time required before an application is able to resume real computational work.

of a parallel computer. We next discuss how to extend the serial time model of Equation (2) to model coordinated and uncoordinated checkpoint protocols in parallel machines.

a) Coordinated Checkpoint/Restart: We can model cCR by assuming a BSP-style execution model with additional checkpoint supersteps (cf. phases) that are performed after a communication superstep ends (when the network is quiet, cf. δ in Figure 1(b)). The synchronous nature of all supersteps (phases) allows us to model the runtime of a parallel application identical to a sequential execution as $T_{cCR}(\hat{\tau}) = T(\hat{\tau})$ using Equation (2).

b) Uncoordinated Checkpoint/Restart: We cannot model uCR with the simple BSP model because in uCR protocols each process takes checkpoints independently, and the checkpoints are not necessarily synchronized. Therfore, uCR requires a more elaborate model than cCR that considers detailed dependency chains amongst processes.

If processes were fully independent, then $T(\hat{\tau})$ using Equation (2) would be sufficient to predict the runtime of the optimal uCR strategy. However, the execution time of a process may depend on other processes through *happens-before* relations [7], a simple concept of causality (cf. Figure 1(c)). For example, a process that waits to receive a message depends on the progress of the sender process. If the sender process is delayed, then so is the receiver. Similarly, if a rendezvous protocol is used for the communication, the sender will also wait for the receiver before it can send the message.

Generally, happens-before relationships are a function of the application's *communication structure*. The communication structure or topology is the set of message matching relations and their relative timings. The timing is important since delay chains can be formed as transitive relations between happens-before relationships as shown in Figure 1(c): process p_2 does not communicate with process p_0 directly but is still delayed by a checkpoint event on process p_0.

C. Simulating Distributed Checkpoint/Restart

In general, the communication structure of Message Passing Interface (MPI) programs cannot be determined offline because message matches cannot generally be established statically [8]. Even if all parameters such as the complete communication structure and all relative timings are known, modeling the interactions analytically is challenging. Thus, we choose to use discrete event simulation to assess the overheads of uCR for real applications via their message traces.

Our simulator framework comprises `LogGOPSim` [9] and the tool chain developed by Levy et al. [10]. `LogGOPSim` uses the LogGOPS model, an extension of the well known LogP model [11], to simulate application traces that contain all exchanged messages and group operations. In this way, `LogGOPSim` reproduces all happens-before dependencies and the transitive closures of all delay chains of the application execution. It can also extrapolate traces from small application runs with p processes to application runs with $k \cdot p$ processes. The extrapolation produces exact communication patterns for all collective communications and approximates point-to-point communications [9]. `LogGOPSim` and its trace extrapolation features have been validated in [4], [9]. Levy et al.'s tool chain adds the capability to simulate uCR (including message logging) and cCR to `LogGOPSim`. The tool chain has been validated against experiments and established models in [10], [12].

LogGOPS parameter	Cray XE6	Cray XC30m
Latency	$2\mu s$	$1.8\mu s$
overhead per message	$15.7\mu s$	$12.4\mu s$
gap per message	$3.9\mu s$	$2.6\mu s$
Gap per byte	$2ns$	$1ns$
Overhead per byte	$0ns$	$0ns$
S: rendezvous threshold	65,536 bytes	65,536 bytes

TABLE I. LogGOPS parameters used in out study to represent the Cray XE6 (LAMMPS, CTH, and HPCCG) and the Cray cascade (MCCK, LULESH, and miniFE) architectures.

D. Simulation Setup and Reproducibility

To generate the data presented in this paper, we collected execution traces of each application with each of its inputs running on a 128-node system. We simulated the collected traces with `LogGOPSim` and verified that it accurately reproduces (within 6%) the execution time on the respective system. All used traces are available for download in our trace repository [13]. The full `LogGOPSim` infrastructure is available online so that all results can be reproduced and may form a basis of additional research efforts in this area.

Unless otherwise stated, we configure the simulator and make assumptions as follows. System parameters for the simulator are shown in TABLE I. These parameters were collected on our test systems using the Netgauge tool [14]. We assume local stable storage for each application process with a checkpoint commit bandwidth of 2GiB/sec. This commit rate is a value expected of future non-volatile storage devices, for example phase change memory [15]. To model independence of inter-process checkpoints, each process waits a random amount of time before taking its first checkpoint. Each subsequent checkpoint is taken periodically with the fixed checkpoint interval $\hat{\tau}$ (see Equation (3)). As we will show in Fig. III-C, as long as the checkpoint duration (δ) is significantly lower than the checkpoint interval, this value does not significantly contribute to the slowdown.

We model hardware failures as an exponential distribution with a probability density function $f(t, M) = \frac{1}{M} e^{\frac{-t}{M}}$. The simulator determines node failures and stalls the computation at the affected process for $\mathrm{T_r} + \mathrm{T_{rework}}$. The time $\mathrm{T_{rework}}$ is calculated as the time from the last local checkpoint and we use a restart value ($\mathrm{T_r}$) of 10 minutes, a value used in previous checkpointing work [6]. Processes that communicate with the recovering process are simply delayed until the recovery succeeds and the computation is virtually resumed. While more accurate failure models exist (e.g., [16]) and can be used in our simulator, we chose an exponential distribution as it has been shown to be reasonably accurate [17] and, more importantly, enables comparison with other significant works such as [6]. Lastly, in these studies we assume uCR protocol overhead is free and do not consider failures. Section III-B and III-C justify and quantitatively rationalize these latter assumptions.

III. Important uCR Considerations

Our simulation framework allows us to explore in depth the behavior of uCR protocols in applications. We have used these tools to develop insight into the nature of uCR at scale and assumptions surrounding its use. In this section, we show that a number of those assumptions should be re-thought in order to properly consider the use of uCR. We begin by introducing the suite of workloads that we used. We then demonstrate that, for applications using uCR, the ability of processes to make independent progress during node failure is limited; that the performance impact of local checkpointing is more significant than that of message logging; and that the duration of local checkpoints is more significant than their frequency. Via simulation, we also quantify the resilience overhead of uCR for our suite of workloads, and show how changing storage system parameters changes those overheads and thus the viability of uCR at scale.

A. Workloads

This section, as well as Sections IV and V, presents results from simulation experiments based on the behavior of a common set of workloads. These workloads were chosen to be representative of scientific applications that are currently in use and computational kernels thought to be important for future extreme-scale computational science. They include:

- CTH, a shock physics code [18].
- HPCCG, a conjugate gradient solver [19]
- LAMMPS, molecular dynamics code [20]. In this paper, we use the Lennard-Jones and SNAP potentials.
- MCCK, a neutronics proxy application [21].
- LULESH, an unstructured hydrodynamics benchmark [22]
- miniFE, a finite element benchmark [19]. We use two problem sizes: small (512MB/process) and large (1GB/process).

CTH, HPCCG, and LAMMPS are important US Department of Energy DOE applications which run for long periods of time in production modes and exhibit a range of different communication structures. MCCK, miniFE and LULESH are exascale application proxies from two of the DOE co-design centers [23], [24] and the Mantevo Project [25]. We note that these applications were designed for other purposes and the impact of uCR protocols was not necessarily considered. The results we present using these applications are not intended as feedback for their designers, but rather as indications that fault tolerance mechanisms must be considered during application design for future systems.

B. Independent Progress is Limited

A potential advantage of uCR protocols is that they can be combined with message logging to yield significant benefits when failures occur. When a process fails, message logging allows the failed process to recover from a saved checkpoint without requiring any other processes to roll back. The surviving processes can continue to execute unless and until they attempt to communicate with a failed process. If the surviving processes in the system are able to make progress while failed processes recover, the total runtime cost of the failure can be reduced. The magnitude of the runtime benefit depends on the inter-process dependencies in the application's communication pattern.

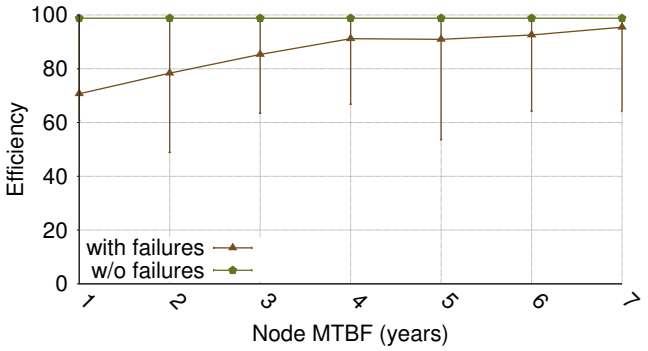


Fig. 2. The impact of failures on uncoordinated checkpointing for LAMMPS as a function of node MTBF. The simulator data for each system size represents the range of values produced over 16 runs of the simulator. This simulation models a ten hour run of LAMMPS using the SNAP potential, a node restart time of ten minutes and a checkpoint commit time of one second. Checkpoints are taken every 68 seconds based on the optimal checkpoint interval described in [6]. The whiskers on the plot represent the min/max range.

Using our simulator framework, we examined the extent to which surviving processes are able to make progress during failure recovery. For these experiments, we simulated the execution of LAMMPS with the quantum accurate SNAP potential across 65,536 nodes. Failures are simulated stochastically using an exponential distribution. We varied the mean of the distribution to represent a range of node mean times between failure (MTBF) from one to seven years [26], [27]. We used the SNAP potential for this experiment rather than the Lennard-Jones potential because it is more computationally intensive allowing us to simulate much longer periods of execution. Under realistic assumptions about node MTBF, longer periods of execution are necessary to simulate enough time that one or more failures are likely to occur. For the data presented here, we used a SNAP configuration that ran for 10.1 hours natively.

The results appear in Fig. 2. The upper dark green line shows the efficiency of LAMMPS/SNAP in the absence of failures. The lower brown line shows the mean efficiency with simulated failures. Based on the uCR with message logging failure recovery model described above, we expect that if surviving processes are able to make significant progress, there would be only small differences in efficiencies between the executions with and without failures. However, this figure shows that as failures become more likely (i.e., MTBF decreases) the efficiency penalty becomes significant. This data suggests that in this case, surviving processes are only able to make limited progress during failure recovery.

Another consideration is that the occurrence of a collective operation ends any independent progress (*run-ahead*) that a process may be able to make. The time between collective operations is therefore an upper-bound on the independent progress the surviving processes can make. These interarrival times necessarily vary from application to application; the types and frequencies of collective operations will depend

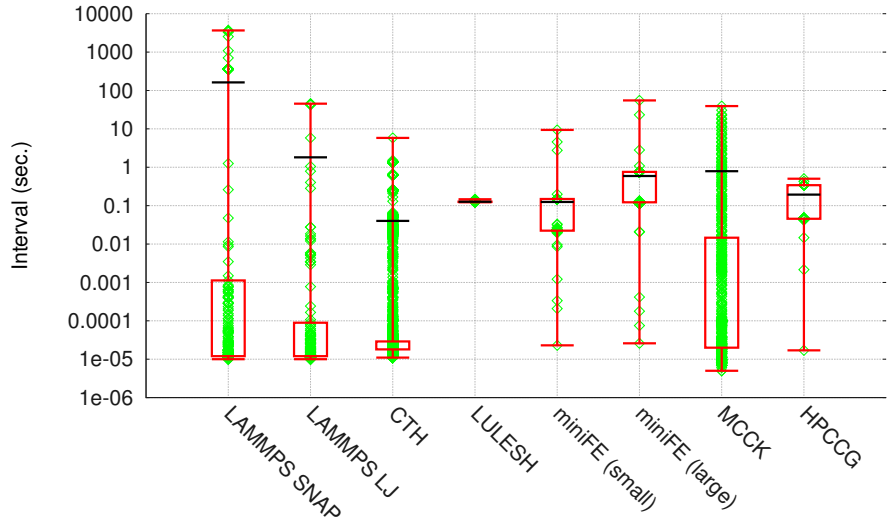


Fig. 3. Interarrival times for all MPI collective operations for LAMMPS SNAP and LJ, LULESH, HPCCG, miniFE, MCCK, and CTH. The candlesticks and whiskerbars show the minimum, 25th percentile, mean, 75th percentile and maximum interarrival times.

on the application and the problem input. Fig. 3 shows the interarrival time between MPI collective operations for several applications. Many of the applications have outliers indicating interarrival times for some collective operations in excess of 1 second. LAMMPS is extreme in this regard, with some collective operations experiencing extremely long intervals between occurrences. However, the 75th percentile for almost all of the applications is near or below 0.01 second.

From this figure, it is clear that significant run-ahead is possible in some applications. In general, however, surviving application processes that are unaffected by rollback activity will not be able to make much independent progress before a collective operation occurs, forcing them to wait while recovering processes catch up. While run-ahead may still yield an energy benefit, there appears to be little potential benefit available in terms of reducing solve time.

C. Message Logging Impact < Checkpoint Impact

Our next experiments set out to quantify the performance overheads of message logging protocols to application solve time for reasonable log write bandwidths. Many studies of uCR have stressed the importance of minimizing message log overheads, because saving message logs allows uCR protocols to avoid cascading rollbacks when failures occur. To make this viable, several research efforts have been directed toward dramatically reducing [28] or eliminating [29], [30] message log overhead. Our study reveals a more significant contributor to solve time that these efforts do not address: local checkpointing activity.

To support this contention, we analyzed the overheads of pessimistic message logging, where a communication operation cannot complete until the associated message has been committed to a form of stable storage. We modeled this cost by varying the CPU overhead per byte for send/receive operations (the O parameter of the `LogGOP` model). By using increasing values of O, we simulated pessimistic message logging to memory and NVRAM ($1ns/byte < O < 4ns/byte$) as well as to a parallel filesystem ($4ns/byte < O < 1024ns/byte$).

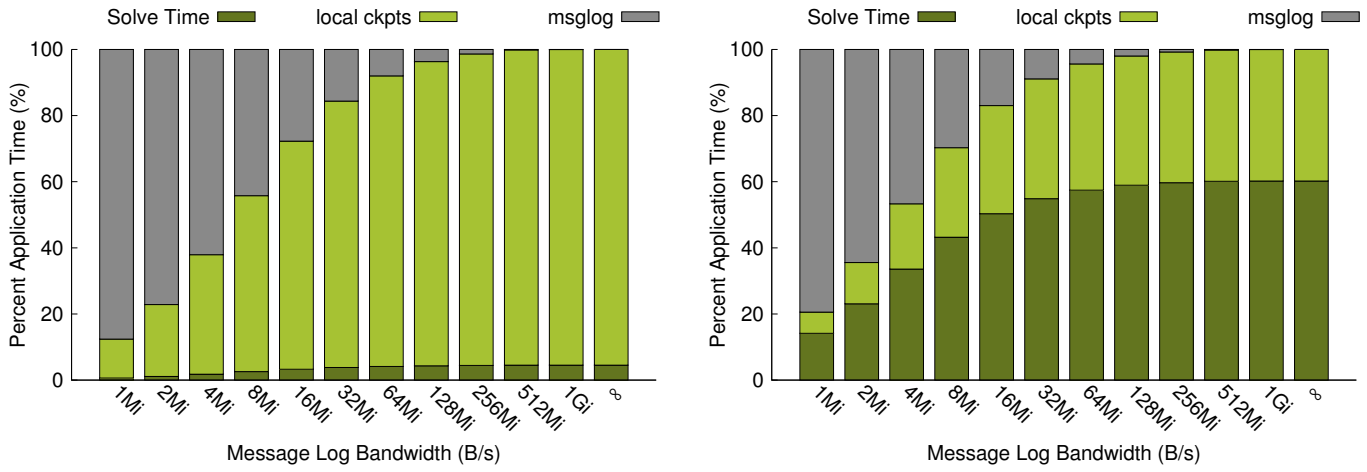
Fig. 4 shows the simulated performance of CTH and LAMMPS for the largest simulation node counts as the overhead per byte varies, broken down by operation—time spent committing message log, committing local checkpoints, and application solve time. As stable storage commit bandwidth increases, message logging is a steadily decreasing proportion of application run time for both CTH and LAMMPS. Overhead associated with local checkpointing increases under these same parameters. This result indicates that decreasing the overhead of local checkpointing will have more beneficial effect on applications at scale than will optimizing the storage of message logs.

D. Checkpoint Frequency Impact < Checkpoint Duration Impact

Research on OS noise has shown that the duration of CPU detours, rather than their frequency, has the greatest impact on application performance. In the context of resilience this suggests that checkpoint commit time rather than the checkpoint interval will have the greatest impact on application performance for uCR. Our results shown in Fig. III-C confirm this intuition. We first hold the checkpoint interval constant and vary the checkpoint commit time to satisfy Daly's equation for optimality (Fig. 5(a)). Fig. 5(b) shows the converse case, where the checkpoint commit time is fixed and the checkpoint interval varies to satisfy Daly's equation. In both cases, we measured simulated application completion time. These figures show that changes to the checkpoint interval have very little impact on application performance. On the other hand, increasing the checkpoint commit time can significantly increase time-to-solution.

E. Effect of Storage Bandwidth on uCR Efficiency

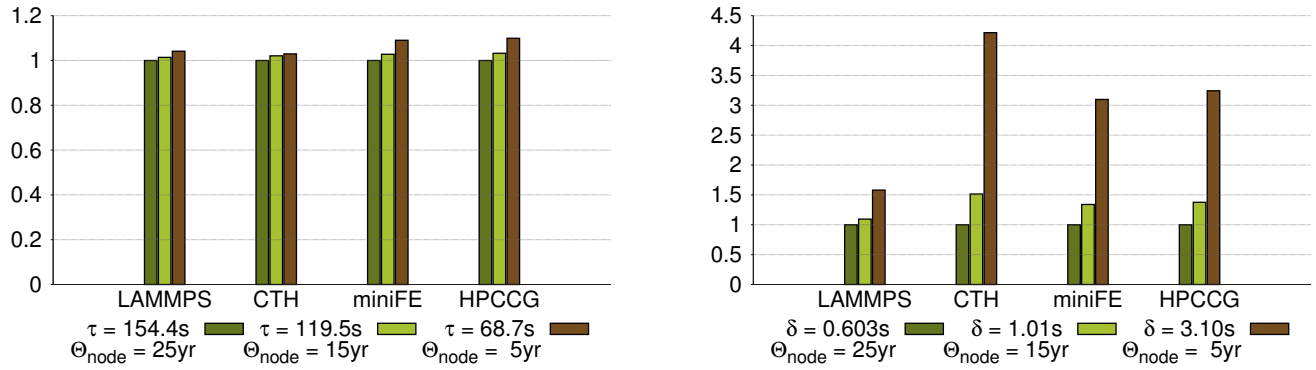
We also investigated the effect of varying storage bandwidth on uCR efficiency in our sample applications. To provide a baseline for this analysis, we simulated executions using uCR with a 120 second checkpoint interval, a 2 GiB/process checkpoint size, and a stable storage path with 2 GiB/second write



(a) CTH @ 65,536 processes

(b) LAMMPS LJ @ 65,536 processes

Fig. 4. Evaluating the percentage of time spent in solve, pessimistic message logging, and local checkpointing as stable storage commit bandwidth increases. This is simulated by varying the O (CPU overhead/byte) parameter of the simulator.



(a) Fixed Duration

(b) Fixed Frequency

Fig. 5. Checkpoint frequency vs. checkpoint duration for LAMMPS, CTH, miniFE and HPCCG.

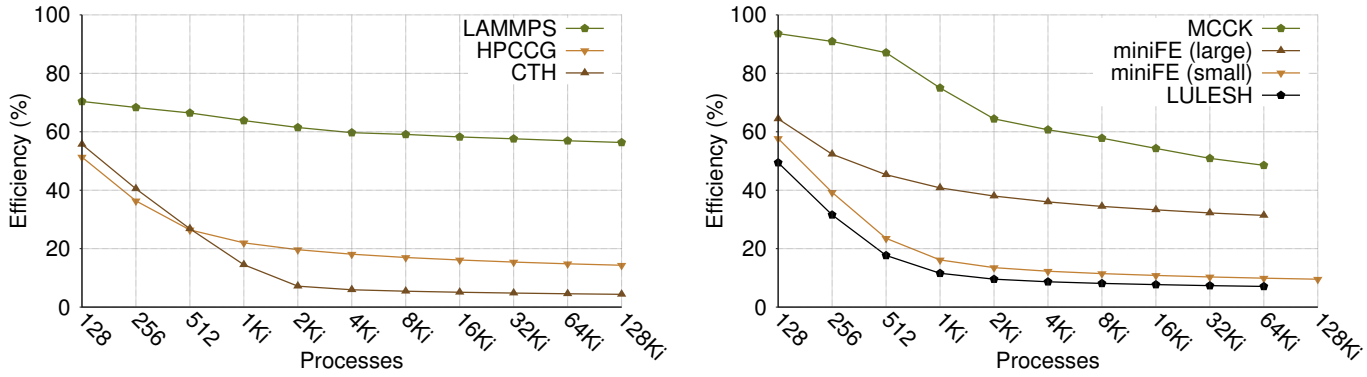
bandwidth. This equates to a 1 second checkpoint commit latency, or less than 1% of application execution time devoted to checkpointing. These results, shown in Fig. 6, indicate that LAMMPS degrades fairly gracefully with increasing application scale. CTH, HPCCG, LULESH, and miniFE all exhibit poor efficiency with dramatic degradation as scale increases.

A dominant performance factor for cCR is storage bandwidth, and a number of recent studies in the literature ([31] is a good example) have explored the effects of storage system configuration on cCR performance. We simulated executions of our sample applications with varying storage configurations for both cCR and uCR to observe any corresponding effects on uCR efficiency.

The first storage configuration approximates a current petascale HPC system where all node checkpoints are stored on a shared parallel file system, with 512 MiB/sec aggregate bandwidth to stable storage. Improvements on this type of design are now becoming common, with solid state drive (SSD) storage being incorporated per-node in order to alleviate contention for bandwidth to the shared file system. It has been proposed that checkpoints be written to the SSD on each node and "trickled" off-node to the filesystem as the application continues execution. These "burst-buffer" designs are contemplated for exascale machines by DOE and other organizations. Accordingly, our second storage configuration reflects such an arrangement with a two GiB/sec local storage path. Note that the per-node SSD effectively makes a coordinated checkpoint a parallel operation as contention for the shared storage is eliminated (at least from the application's perspective).

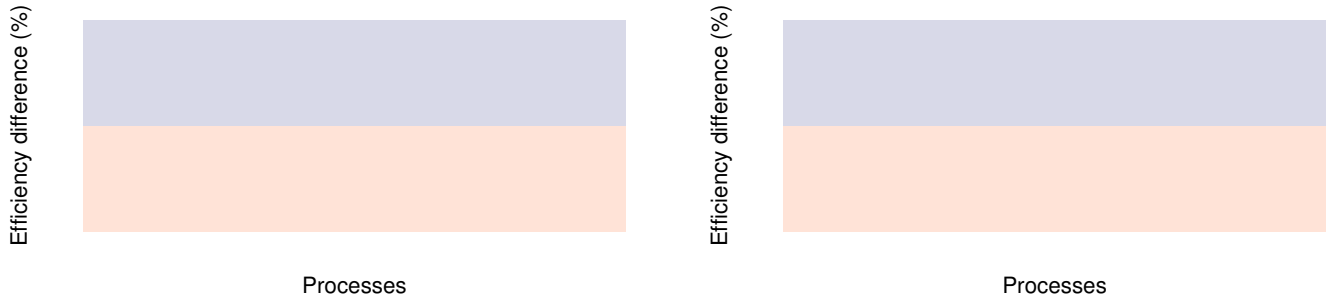
Fig. 7 presents this comparison, plotting the difference between uCR and cCR efficiencies (efficiency(uCR) − efficiency(cCR)) on its y-axis. For each application, a positive data point indicates better uCR efficiency at that scale and a negative data point indicates better cCR efficiency. Several observations are of interest:

- **Assuming unsaturated storage bandwidth, cCR may be preferable and scalable.** Higher effective storage bandwidth results in better efficiency for cCR across our application set, and increasing system scale does not affect efficiency. The results are not as consistent when scale, and therefore contention for storage bandwidth, increases in the parallel file system case.
- **Where dissemination-based operations dominate, cCR may be more efficient.** In both storage configurations,



(a) Apps (b) Proxies

Fig. 6. Simulated uCR efficiency (the ratio of time spent performing work for the application and not the resilience mechanism) using the simulator for our applications.



(a) Parallel File System (b) Local Stable Storage

Fig. 7. Comparing coordinated checkpointing to a shared 512 MiB/sec aggregate parallel filesystem and uncoordinated to a local two GiB/sec stable storage device and coordinated and uncoordinated both to that local storage.

the applications that are not completely compute-bound and which rely on dissemination algorithms (underlying `MPI_Barrier()` and `MPI_Allreduce()`) are more efficient in our simulations when using cCR (we explain this in greater detail in Section IV).

- **As contention for storage bandwidth grows, uCR may be more performance-competitive.** uCR shows better efficiency in our simulations for compute-bound apps as system scale increases when using a shared file system. The effect of dissemination algorithms is still clear in this case, but in the largest systems even applications using them trend toward higher efficiency with uCR.

IV. Breaking Down Application/uCR Performance

In the previous section, we made macro-level observations about the performance and scalability of several application workloads configured with uCR. We now perform more detailed analyses to understand better the reasons for their performance and scalability behaviors. In particular, we probe to understand how application's communication features (time spent in communication, communication frequency, specific communication operations) impact overall performance.

A. Communication/Computation Ratio

We experimented to determine the percentage of overall application time spent either executing application-level communication or computation operations. Our data shows that all of our applications are computation-bound. At a system size of 512 processes, for example, CTH spends about 40% and LULESH spends about 20% of their time in communication. At this system size, all other applications spend less than 10% of their time in communication, with LAMMPS and HPCCG spending less than 3% of their time in communication. Results from other tested system sizes are consistent with these observations. However, from Section III we know that **all** these applications show increased performance slowdowns when used with uCR. Therefore, we can conclude that, at least for the tested applications, **communication-boundness is not a significant determinant of performance for applications configured to use uCR**.

B. Communication Topology and uCR Performance

Next, we tried to understand precisely how an application's communication impacts its uCR-based performance. To examine this question, we used microbenchmarks that allow us to control the type of communication operations (for example, point-to-point or collective); the communication pattern (for

example, 3-D stencil); number of communication operations; and volume of data transferred. Each microbenchmark calls a particular MPI routine and then waits for a period, repeating this process until a given amount of wall-clock time has elapsed. The wait time is the average time step length for our applications (approx. 0.5 seconds).

To help identify the set of relevant communication characteristics to consider, we examined the percentage of time each application spends in MPI communication operations. We found that LAMMPS communication is dominated by `MPI_Send()` (a point-to-point operation). HPCCG, LULESH, and miniFE communication is dominated by `MPI_Allreduce()`. MCCK communication is dominated by `MPI_Allreduce()` and `MPI_Reduce()`. Finally, CTH uses a variety of operations, namely `MPI_Allreduce()`, `MPI_Bcast()` and `MPI_Send()`. Additionally, we study `MPI_Barrier()`, `MPI_Gather()`, `MPI_Scan()`, `MPI_Allgather()` and `MPI_Scatter()` since these operations are found commonly in other HPC applications.

Based on the results of this inquiry, we constructed three sets of microbenchmarks:

1) *stencil-based* microbenchmarks using point-to-point operations (like `MPI_Send()`). In these microbenchmarks, processes are organized into one to three dimensional Cartesian topologies, and each process communicates with its set of neighboring processes. For example, in a one dimensional, three point stencil (1D/3PT), processes are organized in a linear fashion and each process communicates with its two immediate neighbors in the line. We implement a variety of stencil patterns: 1D/3PT, 2D/5PT, 3D/7PT, and 3D/27PT.
2) *dissemination-based* microbenchmarks used to implement `MPI_Allreduce()`, `MPI_Barrier()`, `MPI_Gather()` and `MPI_Scan()`. In dissemination algorithms, processes exchange data in a butterfly-like pattern of logarithmic depth [32], [33].
3) *binomial-tree-based* microbenchmarks used to implement `MPI_Allgather()`, `MPI_Bcast()`, `MPI_Reduce()` and `MPI_Scatter()`. In these microbenchmarks, binomial trees rooted at process zero are used to exchange data.

Stencil-based communication patterns are found in many scientific applications including CTH. For the binomial-tree and dissemination-based microbenchmarks, the algorithm used to implement each operations corresponds to the method commonly used in many MPI implementations [34].

Fig. 8 shows how the efficiency for collective operations varies with scale for applications with uCR. For the stencil-based microbenchmarks (Fig. 8(c)), we observed very good efficiencies ($\sim$ 70%) which did not degrade with scale. Efficiencies remain constant since the number of communicating neighbors in stencil-based patterns is independent of scale. From Fig. 8(a), we see that for the dissemination-based microbenchmarks (`MPI_Allgather()`, `MPI_Allreduce()`, `MPI_Barrier()` and `MPI_Scan()`), even at small scales (128 nodes), efficiency is a modest 45%. Beyond these scales, efficiency continues to degrade drastically. Fig. 8(b) shows the results for the microbenchmarks that use a binomial tree topology. `MPI_Bcast()` and `MPI_Reduce()` maintain good performance independent of scale, while for `MPI_Gather()` and `MPI_Scatter()`, as scale increases beyond 2Ki nodes, declines. A surprising result to us is the difference in behavior between `MPI_Bcast()`/`MPI_Reduce()` and `MPI_Gather()`/`MPI_Scatter()`, particularly because they were all implemented using the binomial tree topology. We verified these performance behaviors on the Cray Cascade system using a CR library. All measurements between 128 and 1,024 nodes were within 6.7% of the simulation.

From these results, we were unable to draw definitive conclusions about macro-level application performance based on their operation composition. For instance, MCCK spends a significant portion of its communication time in `MPI_Allreduce()`; the `MPI_Allreduce()` microbenchmark exhibits increasingly poor performance as scale increases, but MCCK does not.

C. Communication Frequency and uCR

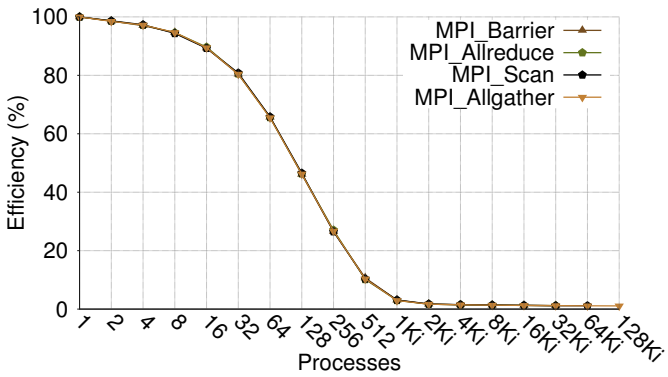
While communication topology does have an impact on uCR performance, the results of the previous section suggest that factors other than the communication topology matter as well. Reconsideration of Fig. I suggests an interplay between the checkpoint duration and the interarrival time of messages whose delays are likely propagated—in this case collective operations.

We now probe how the frequency of communication operations impacts uCR performance. Recall Fig. 3 that offers summary statistics for the interarrival rate of collective communication operations in our applications. In this data, the mean interarrival rate in seconds for collective communication operations were: LAMMPS: 1.81; MCCK: 0.79; miniFE (large): 0.59; HPCCG: 0.19; LULESH: 0.13; miniFE (small): 0.12; CTH: 0.04. Also, recall that for the application efficiency results from Fig. 6, the checkpoint commit time was one second. Based on our hypothesis, we would expect the applications' relative communication frequency to dictate their relative performances, and we would expect their absolute performances to be dictated by their communication frequency/checkpoint duration ratio. Fig. 6 corroborates these expectations: LAMMPS and MCCK exhibit relatively good performance efficiencies, miniFE (large) is in the middle, and HPCCG, LULESH, miniFE (small) and CTH exhibit very poor performance efficiencies as the application size scales.

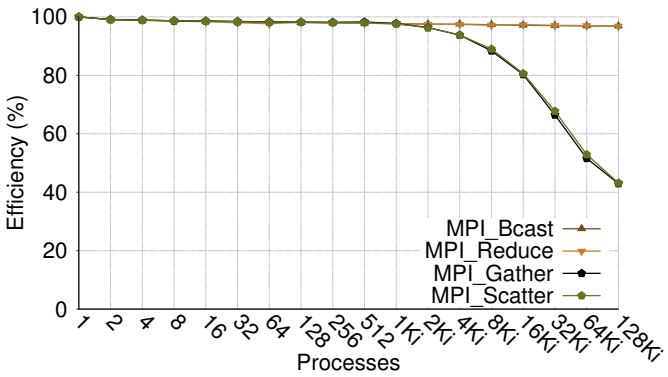
Finally, to corroborate further our hypothesis on the communication frequency/checkpoint duration ratio, we look at the effect of varying checkpoint duration with respect to collective interarrival times. Fig. 9 shows the results of an experiment in which two different checkpoint durations were used for each of LAMMPS/SNAP and LAMMPS/LJ. One duration of each pair was chosen to be close to the collective interarrival time of the problem, and the other was chosen to be far less than the interarrival time. Each application instance spends the same total local time on checkpointing, but their efficiency results are very different. Where the checkpoint duration is close to the collective interarrival time, efficiency for both LAMMPS potentials suffers noticeably as the application size scales.

V. Degree of Coordination & Performance

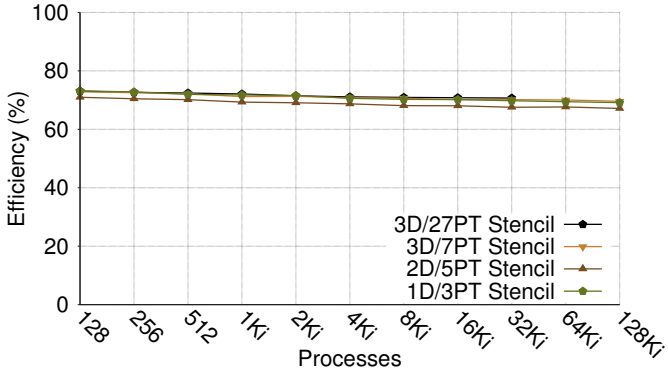
At this point, we have only considered the two extremes of coordination: complete coordination (cCR) and a complete



(a) Dissemination-based Microbenchmarks



(b) Binomial Tree-based Microbenchmarks



(c) Stencil-based Microbenchmarks

Fig. 8. Collective microbenchmark results: performance with uncoordinated checkpointing grouped by the used algorithm (dissemination, binomial tree and stencils). All message sizes are eight bytes, checkpoints are taken every 120 seconds and each checkpoint takes one second to complete. Note that we use SI metrics in that Ki means 2^{10} and K would mean 10^3.

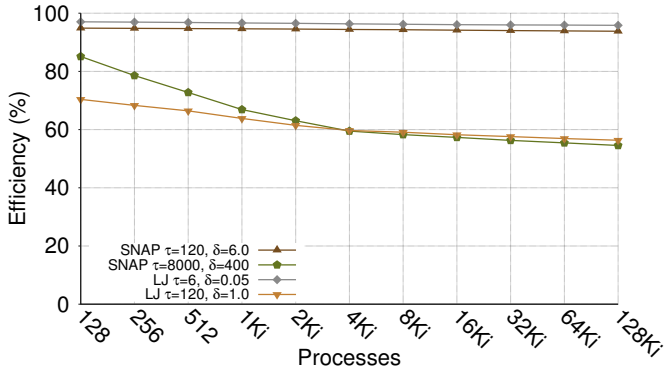


Fig. 9. Demonstrating the performance influence of checkpoint durations for LAMMPS/SNAP and LAMMPS/LJ with checkpoint durations both near and far from the collective (`MPI_Allreduce()`) interarrival rate of the problem (1 second for LAMMPS/LJ and 400 seconds for LAMMPS/SNAP). The same local total time is dedicated to checkpointing (5% and 1%, respectively) for each frequency/duration pair. Checkpoint durations nearer to the collective interarrival times produce greater efficiency impacts.

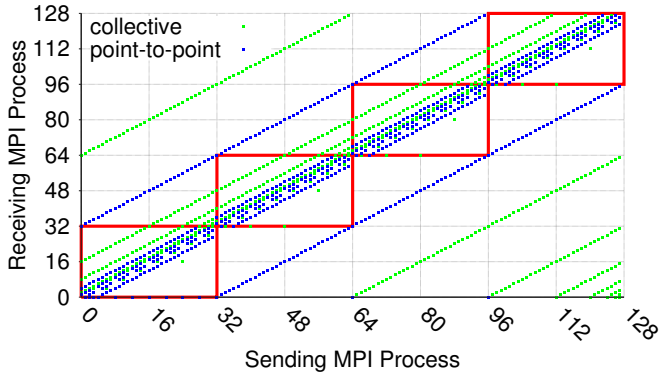


Fig. 10. An example of the naive clustering algorithm for CTH on 128 nodes. The cluster size in this example is 32 nodes. A dot at (x, y) represents one or more messages sent from the process whose MPI rank is x to the process whose MPI rank is y. The red rectangles represent cluster boundaries. Messages within the rectangles are intra-cluster messages that are not logged. Messages outside of the rectangles are inter-cluster messages that are logged.

lack of explicit coordination (uCR). In this section, we consider the implications of a third option: hierarchical checkpointing [28], [35]. Hierarchical checkpointing allows us to consider a mix of the two within a single application. This technique works by grouping application processes into *clusters*. The processes within a cluster collectively use cCR; messages between clusters are logged. When a failure occurs, all of the processes in the cluster containing the failed node rollback to the last checkpoint. Inter-cluster messages are replayed from message logs. Because intra-cluster messages need not be logged, one of the key benefits of hierarchical checkpointing is that the total volume of messages that must be logged is reduced.

In this section, we examine the impact of hierarchical checkpointing on application performance. For our experiments, we use a naive hierarchical checkpointing technique. Although more sophisticated clustering techniques exist (e.g., [29]), we observe that even our naive approach yields signifcant improvements in application performance. We group application processes by their MPI rank. If C is the number of clusters, then a process whose MPI rank is r will belong to cluster $\lfloor r/C \rfloor$. For example, consider Fig. 10. For our naive approach, this figure shows the relationship between clusters and application communication for CTH running on 128 processes. Each point (x, y) in the graph represents one or more messages sent from the process with MPI rank x to the process with MPI rank y. The blue points represent point-to-point messages and the green points represent collective messsages. The red rectangles represent cluster boundaries. Messages within the rectangles are intra-cluster messages and messages outside of the rectangles are inter-cluster messages. Because checkpoints are not coordinated between clusters delays can propagate and accumulate along inter-cluster dependencies. Within clusters, checkpoint coordination prevents delays from propagating. For the example shown in Fig. 10, the majority of the communication occurs within a cluster: 68.6% of process pairs that communicate with point-to-point messages are in the same cluster, and 71.7% of the process pairs that directly exchange messages during collective communication are in the same cluster. As the cluster size increases, the fraction of inter-cluster communication also increases.

Given this approach, we can control the degree of coordination by changing the number of clusters. Qualitatively, the number of clusters is inversely proportional to the degree of coordination. Including all of the processes in a single cluster reduces to cCR. When the number of clusters is equal to the total number of application processes, the approach reduces to uCR. As a result, we can explore the impact that the degree of coordination has on application performance by conducting a series of experiments where we vary the size of the clusters.

For our experiments, we examine the following scenario. We consider a system comprised of 65,536 nodes. Based on projections of future systems [36], we assume that the parallel filesystem will support a minimum per-process bandwidth of 32 MiB/second. We further assume that the I/O bandwidth on each node will support a maximum per-process bandwidth of 16 GiB/second. Finally, we assume that each process will produce a 2 GiB checkpoint.

One of the principal costs of cCR is filesystem contention. Because of the inter-process coordination, every application process is attempting to write their checkpoints to the parallel filesystem at the same time. As a result, contention for filesystem resources reduces per-process bandwidth. One potential advantage of hierarchical checkpointing is that checkpoints are not coordinated between clusters. Therefore, a smaller fraction of the application processes will be contending for filesystem resources while they are committing their checkpoints. For these experiments, we assume that, subject to the maximum per-process bandwidth stated in the previous paragraph, the reduction in per-process filesystem bandwidth due to contention for I/O resources is proportional to the number of simultaneous writers to the parallel filesystem.

Given these system characteristics, we performed several experiments to explore the relationship between cluster size and application performance. The results are shown in Fig. 11. For most of the applications that we considered, we observe that if we start with complete coordination (i.e., cCR) then relaxing the degree of coordination (and consequently improving per-process filesystem bandwdith) improves application performance until the number of clusters reaches 512. When there are more than 512 clusters, filesystem bandwidth is no longer the bottleneck; checkpoint commit time is dictated by local I/O bandwidth. As a result, dividing our system into more than 512 clusters provides no additional reduction in checkpoint commit time. Therefore, as the costs of accumulated checkpoint delays continue to increase, application performance begins to decrease. This figure demonstrates that, for miniFE, HPCCG, LULESH and CTH, when checkpoint commit time is dominated by the per-process share of parallel filesystem bandwidth, reducing the degree of coordination has the potential to significantly improve application performance. One exception to this trend is LAMMPS. As we demonstrated in Section III, LAMMPS is relatively insensitive to delays introduced by checkpointing. Therefore, performance improves significantly due to improved filesystem performance as the degree of coordination decreases. Moreover, for more than 512 clusters, the increased cost of checkpointing delays due to decreased coordination yields small decreases in application performance.

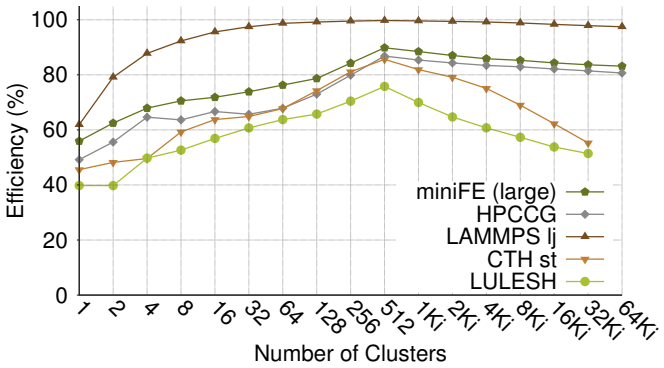


Fig. 11. The performance impact of clustering when the cost of file contention is considered. Based on recent projections [36], we assume that the minimum per-process share of the filesystem bandwidth is 32 MiB/s and that per-process bandwidth to the parallel file system is proportional to the number of simultaneous writers. Further, we assume a 2 GiB checkpoint and a maximum per-process bandwidth of 16 GiB/s. The knee that appears at 512 clusters is due to the maximum per-process bandwidth in our model of filesystem performance.

VI. Related Work

In this work, we study how the interplay between uCR protocol activity and application communication affects overall application performance. Here, we provide an overview of related projects that consider how application communication affects the performance of resilience mechanisms. We also discuss analogous OS noise research that shows how system activity can perturb distributed application performance.

CR Background. Checkpoint/restart protocols in HPC systems have been studied for some time. There are many descriptions of the foundations of both coordinated and uncoordinated CR protocols available in the literature [37]–[39].

Application Communication and CR Protocols. Alvisi et al. examined the impact of coarse-grained communication patterns on the performance of three communication-induced checkpoint/restart (ciCR) algorithms [40]. ciCR uses the application's communication patterns to avoid checkpoints that cannot be used to recover a consistent global state. They showed that such protocols perform best when the total volume of communication is low and the pattern of communication is random. In addition, they demonstrated that performance decreases for applications that rely on large volumes of communication and regular access patterns.

We are unaware of other works that study the impact of communication on checkpointing protocol performance. However, researchers have used applications' communication patterns to optimize CR protocols. Cappello, Guermouche and Snir observed that many HPC applications are *send-deterministic*: for a given set of inputs, the sequence of sent messages is deterministic [41]. Based on this observation, Guermouche et al. proposed a technique for reducing the volume of message log data that must be stored by excluding messages that will be deterministically replayed [28]. Monnet, Morin and Badrinath observed that for many applications the volume of inter-process communication is unevenly distributed [35] and proposed a hierarchical CR approach in which processes that communicate frequently use cCR and processes that communicate infrequently use ciCR. In our work, we study the impact of delay propagation uCR protocol activity via application communication.

Operating System Noise. Our study has antecedents in previously published work [4], [5] that characterizes application behavior in the presence of OS noise. Those works show that the pattern of the OS noise determines the effect that noise will have on application performance. Similarly, we show how the noise pattern caused by uCR protocols can impact application performance and show how strategies that can change this noise pattern can be used to improve application performance.

VII. Conclusion

Using a simulation-based approach, we showed that with sufficiently high ratios of communication frequency to checkpoint duration, uCR can yield good performance. Also, application communication activity can propagate and amplify even extremely low local checkpoint overheads, resulting in prohibitively poor overall application performance. We also show that increasing the degree of local checkpointing coordination can mitigate overhead amplification and improve overall application behavior. Lastly, we show that (1) for many useful applications, in practice message logging overhead likely contributes little to overall application performance; (2) with uCR, the frequency of checkpoint activity matters less than each checkpoint's duration; and (3) while uCR protocols allow surviving processes to make independent progress in execution, application communication patterns likely will limit any consequent benefits for overall application performance.

As researchers continue to explore the software/hardware design space for exascale HPC systems, these results can help to inform their decision processes. Additionally, these results lend to the resilience community new insights about the behavior and performance of applications using uCR protocols. Finally, our simulation framework is available for others to use or extend to perform other studies on application performance at extreme scale using different fault-tolerance strategies.

References

[1] K. Bergman *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep., Sep. 2008.

[2] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[3] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181

[4] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the Influence of System Noise on Large-Scale Applications by Simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010.

[5] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to os interference using kernel-level noise injection," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 19.

[6] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 303–312, 2006.

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[8] G. Bronevetsky, "Communication-sensitive static dataflow for parallel message passing applications," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2009, pp. 1–12.

[9] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, Jun. 2010, pp. 597–604.

[10] S. Levy, B. Topp, K. B. Ferreira, D. Arnold, T. Hoefler, and P. Widener, "Using simulation to evaluate the performance of resilience strategies at scale," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2013 SC Companion:*. IEEE, 2013.

[11] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "Logp: towards a realistic model of parallel computation," *SIGPLAN Not.*, vol. 28, no. 7, pp. 1–12, Jul. 1993.

[12] S. Levy, B. Topp, K. B. Ferreira, D. Arnold, P. Widener, and T. Hoefler, "Using simulation to evaluate the performance of resilience strategies and process failures," Sandia National Laboratories, Technical Report SAND2014-0688, 2014.

[13] "Trace repository," http://htor.inf.ethz.ch:8888/, retrieved 16 Jan 2014.

[14] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm, "Netgauge: A Network Performance Measurement Framework," in *Proceedings of High Performance Computing and Communications, HPCC'07*, vol. 4782. Springer, Sep. 2007, pp. 659–671.

[15] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2009.

[16] M.-S. Bouguerra, T. Gautier, D. Trystram, and J.-M. Vincent, "A flexible checkpoint/restart model in distributed systems," in *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I*, ser. PPAM'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 206–215. [Online]. Available: http://dl.acm.org/citation.cfm?id=1882792.1882818

[17] G. Gibson, B. Schroeder, and J. Digney, "Failure tolerance in petascale computers," *CTWatch Quarterly*, vol. 3, no. 4, November 2007. [Online]. Available: http://www.ctwatch.org/quarterly/articles/2007/11/failure-tolerance-in-petascale-computers/

[18] J. E. S. Hertel, R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. PetneY, S. A. Silling, P. A. Taylor, and L. Yarrington, "CTH: A software family for multi-dimensional shock physics analysis," in *Proceedings of the 19th Intl. Symp. on Shock Waves*, Jul. 1993, pp. 377–382.

[19] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratory, Tech. Rep. SAND2009-5574, 2009.

[20] S. J. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal Computation Physics*, vol. 117, pp. 1–19, 1995.

[21] Argonne National Laboratory, "Proxy-apps for neutronics," https://cesar.mcs.anl.gov/content/software/neutronics, retrieved 17 Jan 2014.

[22] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong, "Lulesh programming model and performance ports overview," Tech. Rep. LLNL-TR-608824, December 2012.

[23] Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx), http://exmatex.lanl.gov/, retrieved 16 Jan 2014.

[24] Center for Exascale Simulation of Advanced Reactors (CESAR), https://cesar.mcs.anl.gov/, retrieved 16 Jan 2014.

[25] Sandia National Laboratory, "Mantevo project home page," http://mantevo.org, Jan. 10 2014.

[26] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012022, 2007.

[27] G. Gibson, B. Schroeder, and J. Digney, "Failure tolerance in petascale computers," *CTWatch Quarterly*, vol. 3, 2007.

[28] A. Guermouche, T. Ropars, E. Brunet, M. Snir, and F. Cappello, "Uncoordinated checkpointing without domino effect for send-deterministic MPI applications," in *International Parallel Distributed Processing Symposium (IPDPS)*, May 2011, pp. 989–1000.

[29] T. Ropars, A. Guermouche, B. Uçar, E. Meneses, L. V. Kalé, and F. Cappello, "On the use of cluster-based partial message logging to improve fault tolerance for mpi hpc applications," in *Euro-Par (1)*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds., vol. 6852. Springer, 2011, pp. 567–578.

[30] A. Guermouche, T. Ropars, M. Snir, and F. Cappello, "HydEE: Failure containment without event logging for large scale send-deterministic mpi applications," in *IPDPS*. IEEE Computer Society, 2012, pp. 1216–1227.

[31] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*, 2010, pp. 1–11.

[32] D. Hensgen, R. Finkel, and U. Manber, "Two algorithms for barrier synchronization," *Int. J. Parallel Program.*, vol. 17, no. 1, pp. 1–17, Feb. 1988. [Online]. Available: http://dx.doi.org/10.1007/BF01379320

[33] B. Prisacari, G. Rodriguez, C. Minkenberg, and T. Hoefler, "Bandwidth-optimal all-to-all exchanges in fat tree networks," in *Proceedings of the 27th International ACM International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 139–148. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465434

[34] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[35] S. Monnet, C. Morin, and R. Badrinath, "A hierarchical checkpointing protocol for parallel applications in cluster federations," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE, 2004, p. 211.

[36] S. G. Challenges, "Architectures and technology for extreme scale computing," in *US Department of Energy Workshop Report*, 2009.

[37] A. Maloney and A. Goscinski, "A survey and review of the current state of rollback-recovery for cluster systems," *Concurrency and Computation: Practice and Experience*, Apr. 2009.

[38] D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using asynchronous message logging and checkpointing," in *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, 1988, pp. 171–181.

[39] L. Alvisi and K. Marzullo, "Message logging: Pessimistic, optimistic, causal, and optimal," *IEEE Trans. Softw. Eng.*, vol. 24, no. 2, pp. 149–159, Feb. 1998.

[40] L. Alvisi, E. Elnozahy, S. Rao, S. Husain, and A. de Mel, "An analysis of communication induced checkpointing," in *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, 1999, pp. 242–249.

[41] F. Cappello, A. Guermouche, and M. Snir, "On communication determinism in parallel HPC applications," in *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*. IEEE, 2010, pp. 1–8.