VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale

Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, Franck Cappello

HAL Id: hal-02184203
https://hal.science/hal-02184203

Submitted on 15 Jul 2019

VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale

Bogdan Nicolae*, Adam Moody‡, Elsa Gonsiorowski‡, Kathryn Mohror‡, Franck Cappello*

*Argonne National Laboratory, USA
Email: {bnicolae,cappello}@anl.gov
‡Lawrence Livermore National Laboratory, USA
Email: {moody20,gonsiorowski1,kathryn}@llnl.gov

Abstract—Global checkpointing to external storage (e.g., a parallel file system) is a common I/O pattern of many HPC applications. However, given the limited I/O throughput of external storage, global checkpointing can often lead to I/O bottlenecks. To address this issue, a shift from synchronous checkpointing (i.e., blocking until writes have finished) to asynchronous checkpointing (i.e., writing to faster local storage and flushing to external storage in the background) is increasingly being adopted. However, with rising core count per node and heterogeneity of both local and external storage, it is non-trivial to design efficient asynchronous checkpointing mechanisms due to the complex interplay between high concurrency and I/O performance variability at both the node-local and global levels. This problem is not well understood but highly important for modern supercomputing infrastructures. This paper proposes a versatile asynchronous checkpointing solution that addresses this problem. To this end, we introduce a concurrency-optimized technique that combines performance modeling with lightweight monitoring to make informed decisions about what local storage devices to use in order to dynamically adapt to background flushes and reduce the checkpointing overhead. We illustrate this technique using the VeloC prototype. Extensive experiments on a pre-Exascale supercomputing system show significant benefits.

Index Terms—parallel I/O; checkpoint-restart; immutable data; adaptive multilevel asynchronous I/O

I. INTRODUCTION

High performance computing (HPC) applications can produce massive amounts of output data during their run times. The output data is often used for *defensive* purposes, i.e., to survive failures that can occur frequently at large scale using fault tolerance strategies based on *checkpoint-restart*. Output data can also be *productive*: the data is needed to describe the state of the application at key moments during run time to facilitate post-analytics or to revisit previous states as part of the computational model (e.g. adjoint computations).

Regardless whether defensive or productive, the output data typically needs to be coordinated, i.e., all processes that make up the HPC application contribute with a piece of data to form a *global checkpoint*. Therefore, persisting the global checkpoint to external storage (typically a parallel file system such as Lustre) involves a large number of concurrent writers that either write to individual files or to one or several shared files. In either case, they compete for limited I/O resources, which leads to poor scalability and large performance overhead.

In the quest to reach Exascale, many architectural trade-offs are necessary. One such trade-off involves the growing gap between the compute and I/O capabilities, which means less I/O bandwidth will be available per compute unit. Therefore, straightforward approaches that write checkpoints directly to external storage in a blocking fashion (*synchronous checkpointing*) are not scalable and lead to unacceptable overhead.

In order to overcome the high overhead, a popular solution is *asynchronous checkpointing*: nodes are increasingly equipped with local storage (e.g., HDDs, SSDs, NVMs, etc.) that can be used to cache the checkpoints while they are being flushed to external storage in the background. The advantage of this solution lies in the fact that the application is blocked only for the duration of the writes to the local storage, which is assumed to be much faster than the external storage.

However, as many-core architectures become more popular, increasing high node-level parallelism, asynchronous checkpointing involves many concurrent writers per node (e.g., several MPI ranks using one or more I/O threads per rank) that compete for the I/O bandwidth of the local storage. Under these circumstances, I/O bottlenecks at node-level are increasingly problematic, as they are detrimental both directly (i.e., they block the application for longer) and indirectly (i.e., they lead to longer background flushes to external storage that can cause prolonged interference with the application).

Furthermore, in the design of modern supercomputing infrastructure, heterogeneity plays an increasingly important role. This trend is visible at the level of external storage, as parallel file systems are complemented with burst buffers, key-value stores, etc. Despite significant improvement over parallel file systems, such alternatives do not fully solve the problem of I/O bottlenecks under concurrent I/O [1]. Therefore, heterogeneity is adopted also at node-level in the form of heterogeneous local storage (HDDs, SSDs, storage class memory devices, etc.). As a consequence, there is significant I/O performance variability between the various options, both at the level of local storage and external storage.

Under these circumstances, it is non-trivial to design an asynchronous checkpointing strategy that can take advantage of the heterogeneity of local storage to cache checkpoints without causing local I/O bottlenecks.

In this paper, we propose an adaptive asynchronous checkpointing strategy that is specifically designed to manage het-

erogeneous local storage in such a way that it avoids local I/O bottlenecks. Our proposal is based on the key idea that performance variability introduced by heterogeneity can be exploited make better decisions regarding where to store the local checkpoints. Specifically, it may be better to wait until asynchronous writes to external storage free up space on fast local devices rather that directly write to slow local devices. To this end, we introduce a lightweight technique that facilitates such decisions efficiently under high write concurrency. We summarize our contributions as follows:

- We present a series of design principles that enable efficient asynchronous checkpointing with heterogeneous local storage. In particular, we emphasize the importance of: (1) hiding the complexity of heterogeneous storage; (2) consolidating asynchronous writes to external storage on an active back end to enable elastic control of the I/O parallelism; (3) fine-grained chunking of the checkpoints to exploit node-local devices better; (4) adaptive chunk placement using performance modeling (Section IV-A).
- We demonstrate these design principles in practice through a series of algorithmic descriptions that are implemented on top of the *VeloC* (Very Low Overhead Checkpoint-Restart) runtime (Section IV-D) using a reference implementation (Section IV-E).
- We evaluate our approach in a series of experiments conducted on Theta, a pre-Exascale system hosted at Argonne National Laboratory (ANL). We use both a synthetic benchmark and an HPC application. The results show significant performance and scalability improvements for our proposal when compared with state-of-art approaches (Section V).

II. RELATED WORK

Cache management for many-core processors and deep hierarchies have been explored in various contexts: location-awareness [2], latency-power trade-offs [3], locality-aware data access control [4], software-defined cache hierarchies [5]. However, the aspect of using cache hierarchies for multi-core nodes in the context of producer-consumer scenarios received relatively little attention in the literature.

Exploiting local storage as a write cache before flushing application data to external storage asynchronously has been proposed before in the context of node-level I/O aggregation from multiple cores [6], or I/O forwarding [7]. However, such efforts use a single level of cache, placing the emphasis on aggregation. Other efforts such as [8] focus on smart ordering of asynchronous flushes from memory to local storage. Again, a single level of local storage is considered, but such approaches eliminate the need to perform blocking local writes, which is an interesting complement to our own work.

Multilevel checkpoint-restart is a popular approach to leverage multiple storage levels in the context of checkpointing. One example in this direction is the Scalable Checkpoint/Restart Library (SCR) [9], which supports local storage, partner replication, and XOR encoding on remote nodes in addition to the parallel file system. The Fault Tolerant Interface

(FTI) [10] is another related effort that offers similar support while adding Reed-Solomon (RS) encoding [11]. Both offer limited support for asynchronous checkpoint flushes between the levels and there is no coordination between the levels as in our approach.

An alternative to asynchronous checkpointing that can be used to reduce I/O bottlenecks to both local and external storage is reducing the checkpoint sizes. In this context, *incremental checkpointing* was proposed: it is based on the idea that checkpoint data does not fully change from one checkpoint to another, thus storing only incremental differences is enough to reconstruct the original checkpoint. Incremental approaches can be broadly classified into two categories: *page-based* and *deduplication-based*. Page-based approaches [12], [13] trap writes to memory in order to track all changes and build a set of dirty pages that need to be saved. De-duplication based approaches [14], on the other hand, identify differences by means of computation (most often hashing). Furthermore, de-duplication can be extended beyond the scope of a single process by identifying memory pages with identical content across groups of processes [15], [16]. In either case, incremental checkpointing can be complemented with compression techniques [17] to further reduce the checkpoint sizes.

Several other efforts address related issues. CRUISE [18] is a user-space file system that stores data in main memory and transparently spills over to other storage. However, unlike our approach, it does not have a dynamic mechanism that exploits performance variability introduced by heterogeneity to optimize placement decisions. Triple-H [19] is a HDFS-based file system that uses heterogeneous devices and proposes optimized data placement decisions. It aims to be an all-around solution for generic I/O patterns, which is different to the write-intensive I/O pattern under concurrency we target in this work. Intermediate data placement and shuffle strategies for MapReduce [20] have used online profiling before to take decisions dynamically. Our approach is different in that it combines performance modeling for local storage with online profiling of external storage to minimize the overhead of dynamic decisions.

To the best of our knowledge, *we are the first to explore the benefits of adaptive asynchronous checkpointing based on performance modeling of hybrid local storage*.

III. PROBLEM SPECIFICATION AND CHALLENGES

A distributed application that needs to persist a global checkpoint is composed of n processes distributed across N nodes, with each node hosting p processes (which we refer to as writers). For simplicity, we assume two types of node-local heterogeneous storage: fast main memory and slower secondary storage (e.g., an SSD). This scenario is easy to generalize to an arbitrary number of local storage devices.

We assume the application allocates a majority of the main memory to hold the data structures needed during the computation, and only a small number k of checkpoint pieces can be cached in memory. On the other hand, we assume the secondary storage has enough capacity to cache the remaining
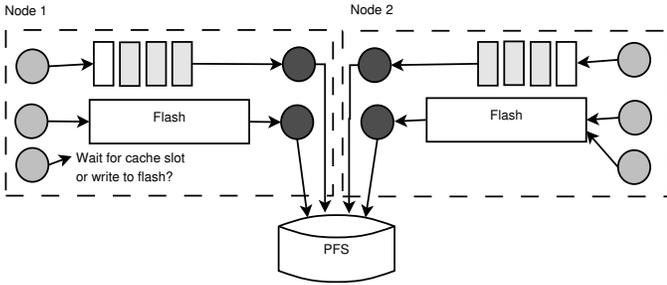
Fig. 1. An application distributed on $N = 2$ nodes with $p = 3$ writers per node (light circles), each writing to either a cache of capacity $k = 4$ slots or flash storage. Each node has $c = 2$ I/O threads (dark circles) that flush to a shared parallel file system in the background. There is only one empty (colorless) cache slot on Node 1 and it is mapped to the top-left writer. The unmapped writer (bottom-left) has two options: either wait for another cache slot to free up or proceed to write to flash storage.

checkpoint pieces until they are flushed to the external storage. This assumption is reasonable since the application typically does not checkpoint more data than it can keep in main memory.

Under these circumstances, the problem of asynchronous checkpointing can be modeled using a producer-consumer scenario: the p writers hosted on each node are the producers of checkpoint pieces, while there are c I/O threads per node that act as the consumers and flush the checkpoint pieces to the external storage. Note that they compete for I/O bandwidth at two levels: (1) on each node, the producers compete for the combined I/O bandwidth of the two local storage types; (2) globally all consumers compete for the I/O bandwidth of the external storage. An example of this is provided in Figure 1.

At first, it may seem that the producers are decoupled from the consumers. A naive strategy to handle this scenario could simply write to the cache as long as there is enough room and fall back to the secondary storage otherwise. However, adopting such a strategy may not always be optimal. This can happen when the background flushes are faster than the local writes to secondary storage, in which case it is better to wait until enough room is available in the cache.

Such a situation is non-trivial and can happen for several reasons: (1) modern supercomputing infrastructure is equipped with a large number of cores per node, therefore p is large and the competition for local I/O bandwidth is significant; (2) external storage is increasingly heterogeneous (e.g., parallel file systems, object stores, key-value stores, etc.) and may be shared by multiple applications, which leads to high performance variability; (3) there is significant interference between the local writes and the flushes to external storage.

The last point is particularly interesting to develop. To illustrate the effects of the interference, we refer the reader to Figure 5, where the ssd−only curve corresponds to an increasing number of p producers writing the same amount of total checkpointing data (64 GB) to an SSD while it is being flushed asynchronously in the background to a parallel file system. Details about exact hardware configuration can be found in Section V-A. As can be observed, there is a high

performance variability despite the fact that the amount of written data is the same.

Therefore, the design of efficient asynchronous checkpointing strategies for modern supercomputing infrastructure needs to address the complexity of the interplay between the producers and the consumers to avoid I/O bottlenecks both at the level of the node and at the level of the external storage.

IV. System design

A. Design principles

Our proposal is based on the following general design principles:

Hidden complexity of heterogeneous storage: Given the increasing complexity of heterogeneous storage both at node-local and global levels, it becomes unfeasible for applications to manage their checkpoints directly. Many users are simply unaware of the various types of storage available on the nodes where they run their applications. Even when they are aware of them, the users report difficulty in using complicated vendor-specific APIs and understanding the performance characteristics. To address this issue, we propose a transparent checkpointing runtime that exposes a single, simple checkpointing API to the users similar to FTI [10]. Specifically, each application process designates the memory regions that it needs to include in a checkpoint. Then, when global checkpointing is required, the processes collectively call a checkpoint primitive. From this moment, it is the responsibility of the runtime to decide where and how to save the protected memory regions into checkpoints. Therefore, if the user is not aware of the existence of a certain type of local storage, the runtime will take advantage of it automatically.

Aggregation of asynchronous I/O using an active backend: With increasing core count per node, asynchronous checkpointing involves a significant coordination overhead needed to manage the producers and consumers. Therefore, it is difficult to design a strategy that is embedded directly into the application processes. To address this issue, we propose to consolidate the management of the consumers into a dedicated process we call the *active backend*. Using this approach, the I/O threads responsible for flushing the local checkpoints to external storage can be spawned elastically on-demand in response to notifications from the producers, which enables better control over the interference introduced by the background flushes. Furthermore, since the I/O threads share the same memory space, monitoring the performance of the background flushes to external storage is trivial and incurs minimal overhead.

I/O load-balancing using fine-grained chunking: The size of the checkpoint each producer needs to write can be large. Therefore, if each producer needs to write the whole checkpoint to a single local storage device, it may happen that only some producers can find room on fast but low capacity devices, while others need wait or use slow but higher capacity devices. Under such circumstances, some producers will finish faster at the expense of other producers. This unfairness is a problem for two reasons. First, an application process

can continue running only after its producer has finished. Therefore, stragglers may cause unacceptable delays in some processes. Second, if a producer started writing the whole checkpoint to a slow device but a fast device becomes available meanwhile (due to consumers making room on it), it will lose the opportunity to speed up the writing, which causes unnecessary delays that slow down the entire group. To address this issue, we propose to split each checkpoint into fixed-sized chunks, which are then written to local storage and flushed independently. Such an approach improves the utilization of the fast but low capacity devices, therefore improving load balancing and reducing the checkpointing overhead.

Adaptive chunk placement strategy using performance modeling: Since the local storage devices may exhibit non-linear performance characteristics under concurrency, it is not enough to decide about the placement of the chunks on local storage based on instantaneous utilization alone. To address this issue, we introduce performance models to predict the write throughput on each local storage device as a function of how many producers are already writing to it. Based on the predictions, we propose the following placement strategy: select the local device that has enough free space to hold a new chunk and is predicted to be the fastest. If this device is faster than the external storage, then write the chunk to it, otherwise wait until a flush has finished (thereby releasing space on a local device) and then try again.

B. Algorithms

In this section, we briefly introduce a series of high-level algorithms that show how to implement the design principles introduced in Section IV-A for an arbitrary number of local storage devices (which form a set denoted $Local$).

Algorithm 1 Each producer invokes PROTECT to declare the memory regions that are part of the checkpoint, then invokes CHECKPOINT to serialize them to local storage

---
1: **procedure** PROTECT(*Addr, Size*)
2:   $MemRegions \leftarrow MemRegions \cup (Addr, Size)$
3: **end procedure**
4: **procedure** CHECKPOINT
5:   **for** $\forall Chunk \in MemRegions$ **do**
6:    enqueue P in Q
7:    $Dest \leftarrow$ wait for notification
8:    write $Chunk$ to $Dest$
9:    $Dest_w \leftarrow Dest_w - 1$
10:    notify active backend of new $Chunk$
11:   **end for**
12: **end procedure**

Specifically, each producer P acts according to Algorithm 1. The memory regions that are part of the checkpoint are declared using PROTECT, which accumulates them into the $MemRegions$ set. Checkpointing is performed by the CHECKPOINT primitive: the memory regions are split into chunks, each of which is written to a local storage device $Dest$ that is assigned by the active backend. $Dest_w$ keeps track of

the number of concurrent writers to $Dest$ and is incremented by the active backend before notifying P. Once P has finished writing $Chunk$ to $Dest$, it decrements $Dest_w$ and notifies the active backend to flush $Chunk$ to the external storage.

Algorithm 2 Active backend: assignment of local devices to producers

---
1: **procedure** ASSIGN_DEVICES
2:   **while true do**
3:    dequeue P from Q
4:    $Dest \leftarrow null$
5:    **while** $Dest = null$ **do**
6:     $Max_{BW} \leftarrow AvgFlushBW$
7:     **for** $\forall S \in Local | S_c < S_{max}$ **do**
8:      $S_{BW} \leftarrow$ MODEL($S, S_w + 1$)
9:      **if** $S_{BW} > Max_{BW}$ **then**
10:       $Max_{BW} \leftarrow S_{BW}$
11:       $Dest \leftarrow S$
12:      **end if**
13:     **end for**
14:     **if** $Dest = null$ **then**
15:      wait for any flush to finish
16:     **else**
17:      $Dest_w \leftarrow Dest_w + 1$
18:      $Dest_c \leftarrow Dest_c + 1$
19:      notify P to use $Dest$
20:     **end if**
21:    **end while**
22:   **end while**
23: **end procedure**

The active backend assigns a local storage device $Dest$ to a producer P according to Algorithm 2. First, it looks for all local storage devices S that have enough free space to hold a chunk (S_c is the number of chunks waiting to be flushed, S_{max} is the maximum number of chunks that S can hold) and that are predicted to be faster than external storage (according to the performance model MODEL). If no S satisfies this condition, then the active backend waits for any flush to finish and tries again. Otherwise, $Dest$ holds the fastest S that satisfies the condition. Then, both $Dest_w$ and $Dest_c$ are incremented (to claim a slot on $Dest$) and P is notified that it can write to $Dest$.

Since Q is a FIFO queue, this approach guarantees fairness: a producer that is ahead of another producer in the queue will always claim the best local device unless a flush has finished meanwhile and the conditions have changed. Also, note that we assume that there is at least one local device that is faster than the external storage (otherwise there is no need for asynchronous checkpointing). Therefore, P will never wait indefinitely to be assigned to $Dest$.

The active backend flushes the chunks according to Algorithm 3: it waits for notifications from the application processes in an infinite loop. Once a chunk was written to local storage, it begins flushing it to external storage using an elastic I/O thread pool. Each flush is handled by the FLUSH primitive,

Algorithm 3 Active backend: flushes to external storage

```
 1: procedure FLUSH(S, Chunk)
 2:     write Chunk to ExtStore
 3:     S_c ← S_c − 1
 4:     update AvgFlushBW based on moving average
 5: end procedure
 6: procedure PROCESS_CHECKPOINTS
 7:     while true do
 8:         wait for notification of new Chunk
 9:         execute FLUSH(S, Chunk) as async I/O
10:     end while
11: end procedure
```


Fig. 2. Architecture of VeloC (Very Low Overhead Checkpoint-Restart)

which is responsible for performing the write, releasing the space on the local storage device by decrementing S_c and finally updating $AvgFlushBW$ based on a moving average of the observed flush throughput.

C. Performance Model

We model the performance of local storage based on a *calibration* that runs on a single representative node of the system for each device type (assuming all nodes have similar access performance to the same device types). This is an infrequent process that needs to be performed only in exceptional circumstances (i.e., first-time installation, new devices added to nodes, device re-configuration or degradation due to wear).

Specifically, the calibration involves a series of benchmarks that measures the average write throughput for an increasing number of concurrent writers. Only a small set of representative samples (less than 10% of the maximum possible write concurrency) is needed. These samples are then interpolated using cubic B-spline, which is numerically stable as it uses compactly supported basis functions constructed via iterative convolution. One advantage of cubic B-spline is that it is known to be fast and accurate for samples that are equally spaced, which makes the gathering of samples much easier.

The interpolation function is then used during run time to predict the throughput under concurrency. The only dynamic information needed during run time is the number of concurrent writers, which can be maintained with minimal overhead using a single atomic inter-process counter per device. The interpolation function itself can be evaluated in any point in O(1), which means the call to MODEL in Algorithm 2 is trivial and incurs minimal overhead.

D. VeloC: Very Low Overhead Checkpoint-Restart

We have developed *VeloC*, a low overhead checkpoint-restart runtime specifically designed to deliver high performance and scalability for complex heterogeneous storage hierarchies without sacrificing ease of use and flexibility. VeloC is part of the Exascale Computing Project [21] and serves the needs of the future Exascale applications.

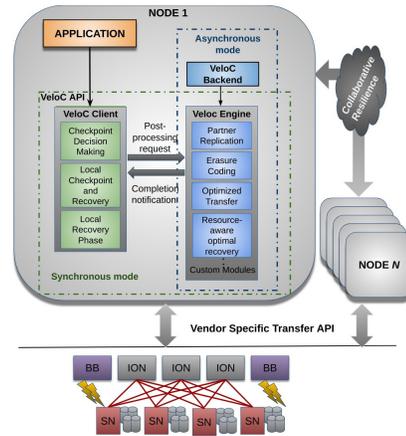The architecture of VeloC is depicted in Figure 2. The *client* is responsible for exposing the checkpointing API (mentioned in Section IV-A) to the application and to manage the local checkpoints. The *engine* is responsible to perform post-processing on the local checkpoints, which includes optimized transfer support to heterogeneous external storage (e.g. parallel file systems, burst buffers, key-value stores, etc.) using vendor APIs (where applicable). The *active backend* runs an instance of the engine in the background to asynchronously mask the overhead of post-processing from the application.

VeloC also supports additional post-processing needed to implement *multilevel checkpointing*: the local checkpoints can be persisted on other nodes (using techniques such as replication or erasure coding), which enables them to survive a majority of failures, thereby reducing the frequency of checkpointing to external storage. In a strict sense, multilevel checkpointing is outside the scope of this paper. However, it is worthwhile mentioning it because the notion of external storage can be extended to include the local storage available on other nodes too.

E. Implementation details

We implemented VeloC using a modular design that facilitates easy extensions. Specifically, the notifications sent by the clients on the control plane are forwarded to each post-processing module, which then decides locally how to handle it. The order in which the modules are notified can be controlled such that the effects of one module can change the behavior of another module. For the purpose of this work, we activated only the transfer module that is responsible for optimized background flushes to external storage.

To implement the algorithms presented in Section IV-B, we extended the control plane of VeloC to include a shared memory space that holds S_w and S_c and $AvgFlushBW$ using atomic integers to enable lock-free read/update operations.

The active backend implements an elastic I/O threading model to parallelize asynchronous flushes to external storage using dedicated C++ language constructs (in particular, std::async). Using this approach, VeloC avoids explicit thread management, which enables compiler-level optimizations that are specific to the platform where VeloC is running (e.g. lightweight user-space threads). The monitoring of the asyn-

chronous flushes to external storage and computation of the moving average is implemented using an optimized circular buffer available in the Boost C++ collection of libraries [22].

V. EVALUATION

A. Experimental Setup

The experiments were performed on *Theta*, a 12 PetaFlops pre-Exascale Cray XC40 system based on KNL Intel Xeon Phi 7230 SKU. The system is equipped with 4392 nodes, each containing a 64 core processor (256 hardware threads) with 16 gigabytes (GB) of high-bandwidth in-package memory (MCDRAM), 192 GB of DDR4 RAM (20 GB/s), and a 128 GB SSD (700 MB/s). The interconnect topology is based on Dragonfly with a total bisection bandwidth of 7.2 TB/sec. In terms of software, the system is using the Cray ecosystem (Linux-based), on top of which we added the Boost C++ collection of libraries v1.66.

For the purpose of this work, the node-local storage is composed of a fraction of the main memory (DDR4 RAM) acting as the cache and the SSD acting as the secondary storage. In both cases, the memory regions are serialized into checkpoints that are split into 64 MB large chunks. Each chunk is stored locally as an independent file. The file system used for the cache is tmpfs, which is provided by default under the /dev/shm mount point. The file system corresponding to the SSD is ext4. The KNL nodes were configured to boot in the flat mode (MCDRAM and DDR4 RAM managed separately). The external storage is provided a Lustre PFS deployment, which is mounted using POSIX.

B. Methodology

We compare several approaches throughout our evaluation.

Cache-only and SSD-only: These two approaches are used as baselines for comparison. Specifically, cache−only corresponds to the ideal case when enough cache space is available to store all chunks, which means there is no need for the SSD. This is the fastest possible solution. Therefore, the goal of any other approach is to perform as close as possible to cache−only. On the other hand, ssd−only corresponds to the worst case scenario when no cache is available and all local checkpoints need to be stored on the SSD. This is the slowest possible solution. Any other approach needs to be faster than ssd−only to be viable.

Hybrid-Naive: This approach implements a standard multi-tier caching strategy (write to the cache if space available, otherwise to the SSD). It is a good reference for comparison because it is not aware of the background flushing and therefore lacks the adaptability dimension but implements everything else (chunking and I/O flushing using an elastic thread pool). This emphasizes the limitations of using standard multi-tier caching for asynchronous checkpointing.

Hybrid-Opt: This approach implements our adaptive asynchronous checkpointing strategy for two-tier hybrid local storage as detailed in Section IV-B. Specifically, it uses the cache (which is always faster than the PFS) when possible,

otherwise it either waits for the cache or uses the SSD if the predicted throughput is higher than the PFS flush rate.

These approaches are compared using an asynchronous checkpointing benchmark and an HPC application. The scenario we consider is coordinated checkpointing at regular intervals. Both applications consist of a set of distributed processes that use MPI to communicate and synchronize. Specifically:

Asynchronous Checkpointing Benchmark: Each MPI process allocates a fixed-sized array, fills it with random data and protects it using the VeloC PROTECT primitive. After all processes have finished this step (by synchronizing using a barrier), they checkpoint the array by invoking CHECKPOINT concurrently. Each process reports individually the time required to write to local storage, then waits on a barrier until all processes have finished checkpointing. At this point, the first rank reports the total time to checkpoint to local storage. Then, all processes wait for the asynchronous flushes to the parallel file system to finish (using a dedicated WAIT primitive offered by VeloC) and then synchronize again using a barrier. Finally, the first rank reports the overall completion time until all asynchronous flushes have finished.

HPC application - HACC: HACC [23] is a complex framework that simulates the mass evolution of the universe using particle-mesh techniques. HACC splits the force calculation into a specially designed grid-based long/medium range spectral particle-mesh (PM) component that is common to all architectures, and an architecture-specific short-range solver. Parallelization of the long-range force calculation uses MPI. Short-range force algorithms, which depend on the architecture, are expressed in the appropriate programming model (OpenMP, CUDA, OpenCL) and complement the use of MPI. HACC uses OpenMP on Theta. HACC features an in-situ analytics framework called CosmoTools, which is called by HACC at the end of certain time steps (specified either explicitly or implicitly as a stride). CosmoTools has a modular architecture, enabling the user to write a custom module that can access the information about the particles at the specified time steps. For the purpose of this work, we developed a VeloC module that checkpoints the information about the particles. Specifically, at initialization it protects the critical data structures, then it initiates asynchronous checkpoints every time it is called by CosmoTools. Note that all MPI ranks are synchronized with a barrier before HACC calls CosmoTools, which means all ranks checkpoint simultaneously through the VeloC module.

C. Accuracy of the Performance Model

We begin by evaluating the accuracy of the performance model introduced in Section IV-C. To this end, we run an experiment that compares the predicted vs. actual write throughput on the local SSD of the compute nodes under increasing write concurrency.

First, we run the calibration using a size of 64 MB (i.e., the default chunk size) for each writer. We start with one writer and increase the number of concurrent writers in steps of 10
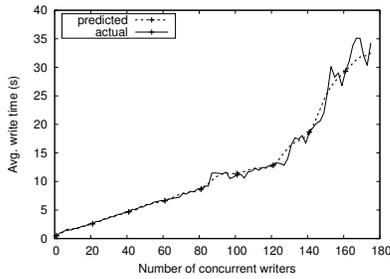
Fig. 3. Accuracy of the performance model: prediction based on cubic B-spline interpolation vs. actual results for a local SSD.

up to 180. Each time we record the average throughput. The samples are then interpolated to obtain predicted throughput for each number of concurrent writers. Next, we repeat the same experiment (using the same data size for each writer) but we increment the number of concurrent writers by one each time to obtain a direct measurement for each possible concurrency level.

The results are depicted in Figure 3. As can be observed, our approach based on interpolation has very high accuracy (the predicted curve almost overlaps with the actual curve) despite using a fast calibration (less than 30 mins) that is based on 10x fewer measurements than actual. Thus, the calibration is lightweight enough to create a negligible overhead.

D. Vertical Scalability of Checkpointing

In this section we evaluate the performance and scalability of our approach on a single node. To this end, we use the asynchronous checkpointing benchmark described in Section V-B.

First, we study the weak scalability. This involves an increasing number of concurrent writers that need to checkpoint each a fixed amount of data. We chose 256 MB per writer, because it represents approx. 1/3 of the total memory available per unit of parallelism (192 GB divided by 256), which is a typical proportion of data that needs to be checkpointed. We gradually increase the number of benchmark processes from 64 up to 256, which is the maximum number of hardware threads. We fix the cache size to 2 GB, which corresponds to a minimal overhead that is acceptable for most users. We study the impact of higher cache sizes in Section V-E.

We focus on two metrics. The duration of the *local checkpointing phase* refers to the total time required for all concurrent writers to finish writing the checkpoints to local storage. This metric is important because it directly contributes to the increase in application run time, as the application is blocked during the local checkpointing phase. The *flush completion time* refers to the additional time required after the local checkpointing phase to finish writing all local checkpoints to the parallel file system. This metric is important because the extra time required for asynchronous flushing indirectly contributes to the increase in application run time due to background interference (shared CPU and network bandwidth).

Figure 4(a) shows the total time to finish the local checkpointing phase for all four approaches. As expected, the overhead of cache−only is negligible, as the I/O bandwidth

of the main memory cannot be saturated by the writers. The opposite is seen for ssd−only: due to its limited I/O bandwidth, the local checkpointing phase grows with increasing number of writers. It is interesting to notice the fact that the growth is not linear, which means a high degree of I/O concurrency negatively impacts the performance of the SSD. Using a 2 GB cache can dramatically lower the duration of the local checkpointing phase, as demonstrated by both hybrid−naive and hybrid−opt. However, hybrid−opt leverages the cache much more efficiently, as it is up to 40% faster than hybrid−naive, which itself is up to 30% faster than ssd−only.

For flush completion time, depicted in Figure 4(b), the gap between hybrid−opt and the rest of the approaches is even higher: it is 2x faster than hybrid−naive and 2.5x faster than ssd−only. Furthermore, it can be observed that hybrid−opt is very close to cache−only, which represents the ideal case. Thus, it can be concluded that hybrid−opt is not reducing the local checkpointing phase at the expense of increasing the flush completion time, effectively becoming a double winner.

To understand these results better, in Figure 4(c) we show how many chunks were written to SSD using each of the approaches, with chunk size of 64 MB and each checkpoint consisting of 4 chunks. The ssd−only approach acts as a baseline, showing the total number of chunks. Since cache−only does not write to the SSD at all, it is omitted. As can be observed, hybrid−naive eagerly uses the SSD since the cache is always full. However, this is suboptimal since it is trapped by the low I/O throughput of the SSD. By contrast, hybrid−opt shows high flexibility in adapting to the parallel file system, effectively avoiding the SSD when it becomes a bottleneck.

Next, we study the strong scalability. This involves an increasing number of concurrent writers (from 1 to 256) that need to checkpoint a fixed total amount of data. We fix the total amount of data to 64 GB. Therefore, each writer needs to checkpoint an increasingly smaller amount of data. The cache size is 2 GB, same as in the previous case.

Figure 5 shows the total time to finish the local checkpointing phase. Since the overhead is negligible for cache−only, we omit it. Interesting to observe is the behavior of ssd−only: with less than 16 concurrent writers, the write performance to the SSD is very poor. In this case, caching seems to dramatically help alleviate this effect, as both hybrid−naive and hybrid−opt are up to an order of magnitude faster, which happens due to the better parallelization opportunities between the local writes and the asynchronous flushes to the parallel file system. After 16 concurrent writers, the write performance to the SSD starts dropping again due to contention. Nevertheless, hybrid−opt manages to outperform hybrid−naive in all configurations, ranging from 15%-60%. In the optimal case (16 concurrent writers), hybrid−opt is 38% faster than hybrid−naive and 6.5x faster than ssd−only.

E. Impact of Cache Size

In the previous sections, we evaluated the performance of our approach for a variable number of writers using a fixed

(a) Total time to write to local storage (lower is better).

(b) Total time to write to local storage and flush to the parallel file system (lower is better).
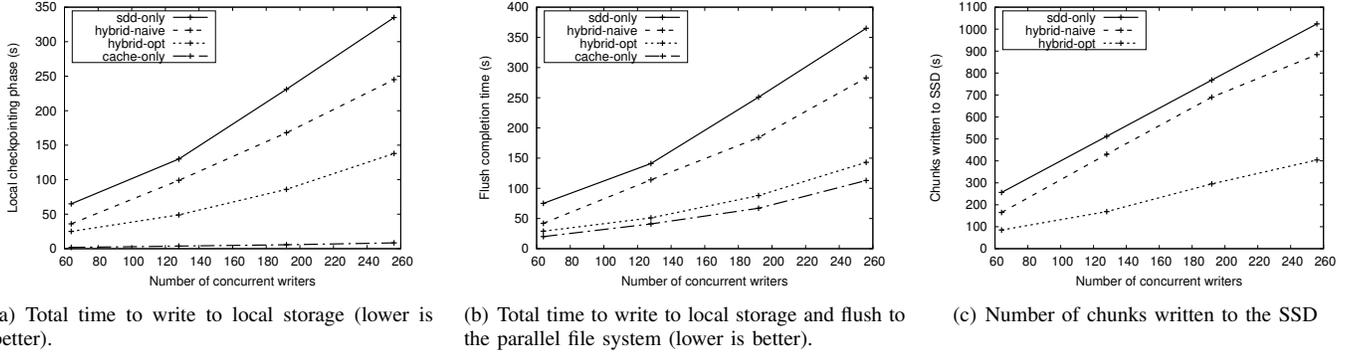
(c) Number of chunks written to the SSD

Fig. 4. Vertical weak scalability: an increasing number of concurrent writers (256 MB per writer) checkpoint on a single node.

cache size of 2 GB. In this section, we fix the total checkpoint size and the number of writers for two representative concurrency scenarios, while increasing the cache size. Our goal is to study the impact of the cache size on the local write performance. Note that 0% corresponds to ssd−only, while 100% corresponds to cache−only. Since these were discussed in the previous section, they are not included here. We compare the performance of our approach with hybrid−naive.

Figure 6(a) shows the results for 16 ranks per node, which is a common case when a large number of threads per rank is desired (up to 16 threads per rank for KNL). In this case, each rank writes a checkpoint that is 4 GB large, for a total size of 64 GB. The cache size varies from 2 GB (1% of the total RAM) to 8 GB (4% of the total RAM). We have chosen to use a cache size below 5% because this amount of spare RAM is within the tolerance level of most HPC applications. As expected, both hybrid−naive and hybrid−opt benefit from a larger cache size. However, the impact of an increasing cache size is more pronounced for hybrid−naive, in which case a 4x increase in the cache size leads to a 30% faster local checkpointing phase. By contrast, hybrid−opt is already efficient with a small cache size: a 4x increase in the cache size leads to only 5% faster local checkpointing phase. Interesting to observe is that hybrid−opt is at least 25% faster than hybrid−naive, regardless of cache size. Thus, our proposal is both faster and more memory-efficient.

Figure 6(b) shows the results for 64 ranks per node, which is common when a small degree of parallelism is desired in each rank (up to 4 threads). In this case, each rank writes a checkpoint that is 1 GB large, for a total size of 64 GB. The cache size varies like in the previous case from 2 GB to 8 GB. In this case, the higher write concurrency leads to a large I/O bottleneck on the SSD in the case of hybrid−naive, as it is more than 2x slower that hybrid−opt for 2 GB and 4 GB. Doubling the cache size seems to have little effect. Only starting from 6 GB does the gap between hybrid−naive and hybrid−opt get smaller. By avoiding the SSD under concurrency, our approach can maintain a good performance for all cache sizes. Again, it is both faster and more memory-efficient.

F. Horizontal Weak Scalability

In this section we evaluate the performance and scalability of our approach for an increasing number of nodes. To this end, we use the asynchronous checkpointing benchmark described in Section V-B.

We fix the number of concurrent writers per node to 16, with each checkpointing 2 GB of memory for a total of 32 GB per node. Then, we gradually increase the number of nodes from 64 to 256. We use a smaller total size per node than in the previous experiments because of the quota limit on the parallel file system (10 TB). The cache size is fixed at 2 GB per node, same as in the previous experiments described in Section V-D.

Figure 7(a) shows the total time to finish the local checkpointing phase. As expected, ssd−only maintains a stable trend, since the configuration is fixed for each node and the overhead of writing to the SSD is not dependent on the number of nodes. By contrast, the overhead of hybrid−naive and hybrid−opt is growing with increasing number of nodes. This is also expected, since it puts more I/O pressure on the parallel file system, therefore flushes are slower and the chunks spend more time in the cache. However, interesting to note is the that hybrid−opt keeps a steady advantage over hybrid−naive (up to 1.5x speedup), despite increasing I/O pressure and slower flushes. This can be explained by the fact that the parallel file system is behaving more dynamically with increasing number of nodes, therefore creating more opportunities to adapt to the variability of the flushes.

The same pattern is also visible in the flush completion time, depicted in Figure 7(b). This time, the gap between hybrid−opt and hybrid−async is even more amplified. This shows that the increasing I/O pressure on the file system is not enough to offset the overhead of writing locally on the SSD. This will eventually happen at much larger scale, in which case the gap between hybrid−naive, hybrid−opt and ssd−only will gradually close.

G. HACC: Real-life HPC application

In this section, we evaluate our proposal for HACC, an HPC application that simulates the evolution of the universe using particle-mesh techniques, as described in Section V-B.

We obtained two representative problems from the HACC team for 8 and 128 Theta nodes (8 MPI ranks per node × 16
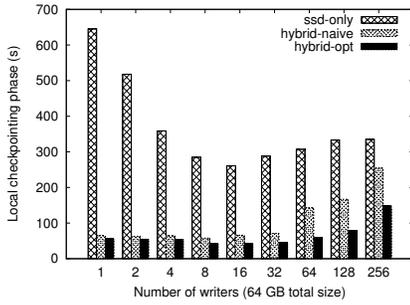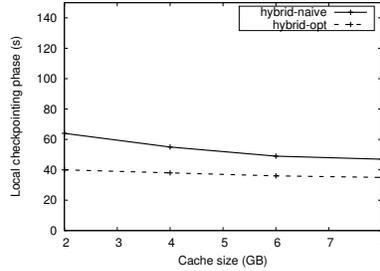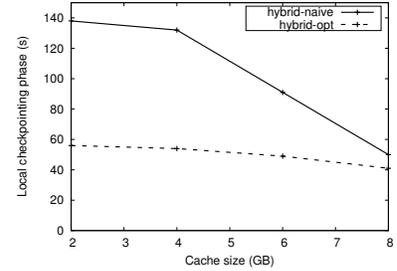
Fig. 5. Total time to checkpoint locally for an increasing number of writers (lower is better). Total size of the checkpoints is fixed at 64 GB.
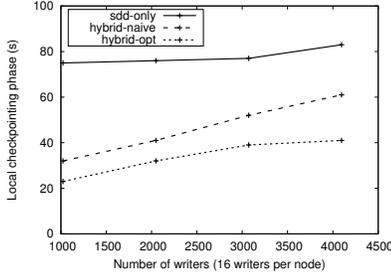

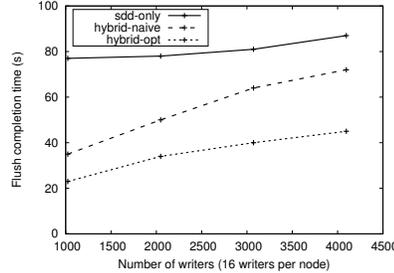(a) 16 concurrent writers (lower is better).   (b) 64 concurrent writers (lower is better).

Fig. 6. Total time to checkpoint locally for an increasing cache size. Total size of the checkpoints is fixed at 64 GB.

(a) Total time to write to local storage (lower is better).

(b) Total time to write to local storage and flush to the parallel file system (lower is better).

Fig. 7. Horizontal weak scalability: fixed number of concurrent writers per node (16), fixed checkpoint size per writer (2 GB), increasing number of nodes.
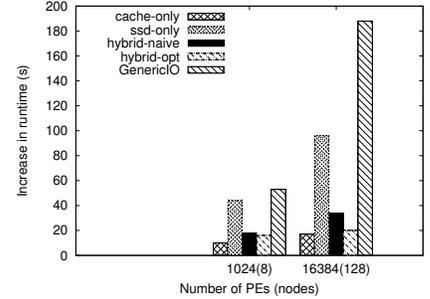

Fig. 8. HACC: Large-scale particle mesh simulation of the universe (128 PEs/node: 8 MPI ranks x 16 OpenMP threads per rank).

OpenMP threads per rank), which involves 128 PEs (parallel processing elements) per node. We run the problem for 10 iterations and initiate an explicit checkpoint at iterations 2, 5, 8. The total number of particles is fixed at 960 (8 nodes) and 2560 (128 nodes), which corresponds to an overall checkpoint size of 40 GB and 1.4 TB.

Since we are using asynchronous checkpointing, the metric that is the most relevant from the application perspective is the *increase in run time* due to checkpointing when compared with the baseline (no checkpointing). This metric captures both the local checkpointing phase (during which the application is blocked) as well as the indirect overhead due to background interference.

HACC's default checkpointing mechanism is a highly optimized synchronous strategy using the *GenericIO* library [24]. Specifically, the MPI ranks are partitioned in a system-specific manner, with one partition per I/O node and each partition writing data in a custom self-describing file format. Each rank writes its data into a distinct region of the file in order to reduce contention for file system level page locks. Furthermore, this approach also reduces contention to the metadata servers due to reduced number of files.

Therefore, in addition to the four approaches we used throughout our experiments so far, we also include GenericIO in our comparison. The cache size for hybrid−naive and hybrid−opt is fixed at 2 GB per node. The results are depicted in Figure 8. As can be observed, at small scale (8 nodes, 1024 PEs), ssd−only is slightly faster than *GenericIO*, while

hybrid−naive and hybrid−opt are 2.9x and 3.3x faster. At much larger scale (128 nodes, 16384 PEs), the gap between *GenericIO* and the asynchronous approaches increases significantly: ssd−only is 2x faster, hybrid−naive 5.5x faster, hybrid−opt 9.4x faster and cache−only 11x faster. This shows excellent performance and scalability potential for our approach, despite using a small cache size.

VI. CONCLUSIONS

This paper has proposed an adaptive asynchronous checkpointing strategy that introduces several novel principles: hidden complexity of leveraging heterogeneous storage, active backend to aggregate I/O requests, fine-grained chunking, smart placement decisions on hybrid local storage based on performance modeling.

We have illustrated the benefits of our proposal using both an asynchronous checkpointing benchmark and an HPC application (*HACC*). We have performed extensive experiments that show the benefits of our proposal for a variety of scenarios. Despite using only a fraction of the total RAM as cache (less than 5%), we have shown ≈10x speed-up over state-of-art synchronous approaches, up to 5x improvement over asynchronous approaches that use a single local tier and up to 40% improvement over flush-agnostic asynchronous approaches that combine multiple local tiers.

An important aspect that requires further discussion is how to tune the number of concurrent writers. Based on our experiments, there is a sweet spot and users can run similar

experiments to ours at small scale to find the best configuration for their application. However, many HPC applications cannot afford to change the number of ranks per node just to optimize checkpointing. However, regardless whether users can optimize the number of writers or not, our approach shows significant improvement in all configurations.

Overall, we conclude that leveraging hybrid local storage efficiently greatly improves the benefits of asynchronous checkpointing and minimizes the background interference with the application run time, as demonstrated by the results obtained for a real-life HPC application.

Encouraged by these results, we plan to broaden the scope of our work in future efforts. We see asynchronous checkpointing as a promising general direction. In this regard, we plan as a next step to increase the scale of the experiments, in order to further study the effects of I/O variability of the external storage under I/O pressure. Another interesting direction is to study how to leverage periodicity in the behavior of the application (e.g., CPU, network, local storage utilization) to run the asynchronous checkpointing in "work stealing" mode that further minimizes interference.

REFERENCES

[1] L. Cao, B. W. Settlemyer, and J. Bent, "To share or not to share: Comparing burst buffer architectures," in *HPC '17: The 25th High Performance Computing Symposium*, Virginia Beach, USA, 2017, pp. 4:1–4:10.

[2] J. Park, R. M. Yoo, D. S. Khudia, C. J. Hughes, and D. Kim, "Location-aware cache management for many-core processors with deep cache hierarchy," in *SC '13: The 2013 ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, Denver, USA, 2013, pp. 20:1–20:12.

[3] D. H. Yoon, T. Gonzalez, P. Ranganathan, and R. S. Schreiber, "Exploring latency-power tradeoffs in deep nonvolatile memory hierarchies," in *CF '12: The 9th Conference on Computing Frontiers*, 2012, pp. 95–102.

[4] Q. Shi, G. Kurian, F. Hijaz, S. Devadas, and O. Khan, "Ldac: Locality-aware data access control for large-scale multicore cache hierarchies," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, pp. 37:1–37:28, 2016.

[5] P.-A. Tsai, N. Beckmann, and D. Sanchez, "Jenga: Software-defined cache hierarchies," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 652–665, 2017.

[6] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O," in *CLUSTER '12 - Proceedings of the 2012 IEEE International Conference on Cluster Computing*, Beijing, China, 2012, pp. 155–163.

[7] T. Ilsche, J. Schuchart, J. Cope, D. Kimpe, T. Jones, A. Knüpfer, K. Iskra, R. Ross, W. E. Nagel, and S. Poole, "Optimizing i/o forwarding techniques for extreme-scale event tracing," *Cluster Computing*, vol. 17, no. 1, pp. 1–18, Mar. 2014.

[8] B. Nicolae and F. Cappello, "AI-Ckpt: Leveraging Memory Access Patterns for Adaptive Asynchronous Incremental Checkpointing," in *HPDC '13: 22th International ACM Symposium on High-Performance Parallel and Distributed Computing*, New York, USA, 2013, pp. 155–166. [Online]. Available: http://hal.inria.fr/hal-00809847

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

[10] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: High performance fault tolerance interface for hybrid systems," in *SC '11: The 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, USA, 2011, pp. 32:1–32:32.

[11] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[12] M. Vasavada, F. Mueller, P. H. Hargrove, and E. Roman, "Comparing different approaches for incremental checkpointing: The showdown," in *Linux'11: The 13th Annual Linux Symposium*, 2011, pp. 69–79.

[13] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *SC '05: Proc of 18th International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, USA, 2005, pp. 9:1–9:14.

[14] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira, "Adaptive incremental checkpointing for massively parallel systems," in *ICS '04: Proceedings of the 18th Annual International Conference on Supercomputing*. St. Malo, France: ACM, 2004, pp. 277–286.

[15] B. Nicolae, "Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal," in *IPDPS '13: The 27th IEEE International Parallel and Distributed Processing Symposium*, Boston, USA, 2013, pp. 1–10.

[16] ——, "Leveraging naturally distributed data redundancy to reduce collective I/O replication overhead," in *IPDPS '15: 29th IEEE International Parallel and Distributed Processing Symposium*, Hyderabad, India, 2015, pp. 1023–1032.

[17] ——, "On the Benefits of Transparent Compression for Cost-Effective Cloud Data Storage," *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, vol. 3, pp. 167–184, 2011.

[18] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. D. Panda, "A 1 pb/s file system to checkpoint three million mpi tasks," in *HPDC '13: 22nd International Symposium on High-performance Parallel and Distributed Computing*, New York, USA, 2013, pp. 143–154.

[19] N. S. Islam, X. Lu, M. W. ur Rahman, D. Shankar, and D. K. Panda, "Triple-h: A hybrid approach to accelerate hdfs on hpc clusters with heterogeneous storage architecture," in *CCGrid'15: 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015, pp. 101–110.

[20] M. W.-u. Rahman, N. S. Islam, X. Lu, and D. K. D. Panda, "A comprehensive study of mapreduce over lustre for intermediate data placement and shuffle strategies on hpc clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 633–646, 2017.

[21] "Ecp project," https://www.exascaleproject.org/.

[22] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[23] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, V. Vishwanath, T. Peterka, J. Insley, D. Daniel, P. Fasel, and Z. Lukić, "Hacc: Extreme scaling and performance across diverse architectures," *Commun. ACM*, vol. 60, no. 1, pp. 97–104, 2017.

[24] S. H. et al., "Hacc: Simulating sky surveys on state-of-the-art supercomputing architectures," *New Astronomy*, vol. 42, pp. 49 – 65, 2016.