2023 22nd International Symposium on Parallel and Distributed Computing (ISPDC) | 979-8-3503-4127-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISPDC59212.2023.00021

Modeling Multi-Threaded Aggregated I/O for Asynchronous Checkpointing on HPC Systems

Mikaila J. Gossman[*], Bogdan Nicolae[†], Jon C. Calhoun[*]
[*]Holcombe Department of Electrical and Computing Engineering, Clemson University, Clemson, SC 29634
[†]Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439
Email: {mikailg, jonccal}@clemson.edu and {bnicolae}@anl.gov

Abstract—**HPC systems encompass more components with each new generation. As a result, the process of interacting with stable storage systems like parallel file systems (PFS) becomes increasingly difficult. Larger systems often result in more frequent failures, increasing the need and frequency to incorporate fault-tolerant mechanisms. One example is checkpoint-restart (C/R), where applications or systems save their data to non-volatile storage devices, such as a PFS. On failure, the system or application is restored to a saved state and computation continues. Today, asynchronous C/R is gaining traction for its ability to checkpoint data to permanent storage concurrently with the application. However, asynchronous C/R brings about many new challenges. For starters, asynchronous C/R introduces complex resource contention between the application and the C/R implementation. Additionally, some implementations adopt file-per-process writing strategies, which overwhelm PFS' at high core counts. In this work, we explore how multi-threaded POSIX I/O impacts aggregated throughput. To this extent we characterize the influence of different I/O parameters, such as the number of writer threads and how they access storage devices, has on aggregated I/O. We use the information gathered in this study to identify best practices when performing aggregated I/O as a first step in designing an efficient I/O aggregation scheme for asynchronous C/R.**

Index Terms—**Checkpoint-Restart, C/R, Multi-Threaded I/O, I/O Optimization, Fault Tolerance, I/O Aggregation, Asynchronous Checkpoint-Restart**

I. Introduction

Modern scientific applications are composed of high-performance computing (HPC) workflows that leverage the convergence between traditional bulk-synchronous simulations, big data analytics, and artificial intelligence (AI). With ever-increasing computational and data processing capabilities, the push towards Exascale has resulted in HPC systems made of thousands of compute nodes, each equipped with many-core CPUs and GPUs, complemented by a heterogeneous storage stack that includes deep local memory hierarchies (e.g., high bandwidth memory, volatile host memory, persistent memory, NVMe-enabled flash storage) and external data repositories (e.g., parallel file systems).

Under such circumstances, it is not possible to leverage the full I/O bandwidth of the HPC system, neither at the level of a single compute node, nor globally across many compute nodes by simply serializing I/O operations. Similar to how parallel and distributed programming paradigms like MPI or OpenMP [1] have allowed HPC applications to abstract and optimize bulk-synchronous communication patterns by taking advantage of high-end networking infrastructures, optimizing parallel and distributed I/O patterns requires dedicated abstractions that combine expert knowledge of the node architecture, network topology, and heterogeneous storage [2].

One fundamental I/O pattern in HPC is checkpointing: it involves a large number of processes, distributed in groups over a large number of compute nodes, that need to simultaneously capture important data structures at key moments during runtime and save persistent checkpoints of these data structures to an external data repository. Checkpointing is used in a wide range of scenarios: fault tolerance based on checkpoint-restart, batch job preemption (e.g., to make room for higher priority on-demand jobs without losing progress), job migration, revisiting intermediate states (e.g. adjoint computations or AI training of large models), exploring alternative computational paths (e.g., sensitivity analytics of AI models), etc.

In this context, checkpointing is traditionally performed *synchronously* by direct I/O interaction with the data repository. In this case, the application is blocked during the I/O and experiences a delay equal to the I/O overhead. On the flip side though, all resources are available for checkpointing and therefore can be dedicated to minimize the I/O overhead. However, with increasing heterogeneity of the compute nodes, I/O can be overlapped with computations by performing them *asynchronously* to reduce the impact on end-to-end runtime. In this case, a common strategy is to capture the checkpoints on fast local storage, then flush them in the background to the data repository. This is challenging because it involves competition for resources at all levels: CPU cores, memory bandwidth, network bandwidth, etc. [3].

Asynchronous checkpointing has matured over time. For example, Exascale-ready checkpointing libraries exist for use in production [4]. However, while they focus on minimizing the end-to-end impact on application runtime, they often do so at the cost of utilizing data layouts that are not easy to manage at user level. For example, VELOC [4] writes one file per process to the parallel file system. While this is not a problem if checkpoints are needed transparently (e.g. in fault tolerance techniques based on checkpoint-restart), users often need to change the data layout to make it easier to manipulate and reuse checkpoints later. In this context, a common technique is I/O aggregation that results a smaller number of files (e.g. N-1 or N-M, with $M << N$).

Unfortunately, while I/O aggregation has been extensively

DOI 10.1109/ISPDC59212.2023.00021

studied in the context of synchronous checkpointing (e.g., MPI-IO, HDF5), how to apply it for asynchronous checkpointing remains largely unexplored. In this paper, we perform an initial study of what I/O parameters impact the performance of asynchronous I/O aggregation. Specifically, given a set of checkpoints captured as a set of local files on each compute node and that need to be flushed to a parallel file system concurrently, we vary (1) the number of writer threads, (2) contiguity and alignment of data, and (3) number of files used for aggregation. Our goal is to evaluate, identify and explain configurations that maximize the overall I/O throughput. We summarize our contributions as follows:

- We develop OpenMP benchmarks designed to act as a simple and convenient proxy that enables a fast evaluation of a large number of combinations of I/O parameters.
- We run extensive experiments using our benchmark on a high-end HPC system to obtain a comprehensive set of results that we study from multiple perspectives.
- From this study, we extract a set of best-practices that we believe can by leveraged to design new asynchronous I/O aggregation techniques and algorithms.

II. Related Work

A. Multi-threading in I/O

Exploiting multi-threading capabilities to improve performance of I/O is a well researched problem. In this context, ROMIO, a popular MPI-IO implementation, has relied on multi-threading to optimize a two-phase I/O protocol since 2014 [5]. ROMIO uses multiple threads to parallelize data exchange between *I/O leaders* (processes interacting with the PFS) and *non-leaders* (other processes), as well as actual file I/O instructions (e.g. read/write). They found that further overlapping the multi-threaded communication and write stages improved throughput by up to 60% in some cases.

As ROMIO has become well established in the HPC community, other works have explored extending multi-threaded capabilities. Kang et al. [2] introduced a two-layer aggregation method (TAM) that uses multi-threading to aggregate all intra-node I/O requests before redistributing I/O requests across all compute nodes. This directly reduces the amount of messages exchanged between compute nodes to complete I/O redistribution, and in some applications directly improves the collective I/O operations. However, if I/O requests are already sufficiently contiguous, TAM provides no improvement and introduces redundant overhead. Feki et al. [1] use multi-threading capabilities within MPI-IO to parallelize aggregating I/O requests (denoted *I/O build* phase). Overlapping the build phase with thread-initiated read/write operations (denoted *file access* phase) results in a 69% faster I/O.

B. Multi-threading in Checkpointing

Libraries such as SCR [6] and FTI [7] have adopted multi-level checkpointing for a long time. The focus on asynchronous I/O using multi-threading has been further refined by VELOC [4]. It maintains an active thread pool on each compute node, managed by an *active backend*. When application processes are ready, they first write their data as independent files to node-local storage, then the application resumes computation. At the same time, the active backend uses an optimized flushing strategy to persist the checkpoints for a variety of data repositories, including parallel file systems (PFS). However, as the number of processes increases, the lack of I/O aggregation support becomes burdensome [8].

A previous study [8] built on the asynchronous flushing strategy of VELOC to perform file aggregation via the original implementation's POSIX threads, and with MPI-IO. However, their results show that using POSIX threads to aggregate I/O by simply writing the checkpoint content at different offsets in the same file under concurrency suffers from *false sharing*, which increases contention and reduces I/O throughput. MPI-IO may improve the I/O throughput compared to the POSIX-based method, however, it is limited by the collective operations it uses for synchronization, which have poor support to address stragglers that are more likely to occur in asynchronous I/O due to resource contention.

III. Methodology

We design and develop an OpenMP micro-benchmark to characterize the expected performance of aggregated file I/O under different configurations. To imitate a simple distributed application ready to checkpoint, our micro-benchmark generates data for N compute nodes, each of which simulates K processes per compute node writing separate files of 1 GiB ($\pm 20\%$, to simulate a slight load imbalance) that need to be checkpointed. We refer to this subset of the data as *local data* throughout the rest of this manuscript. We fix $K = 8$, which is a typical number of processes running on an HPC compute node (because HPC systems typically employ 4-8 GPUs and one process per GPU).

Unlike the naive POSIX I/O aggregation strategy described in Section II, we assume a more advanced strategy that collects the local data from groups of *follower* compute nodes on the local storage of *leader* nodes, each of which is then responsible to flush both its own local and the received data as checkpoint files to the PFS. Note that a leader does not need to wait to collect all received data before flushing to the PFS. Instead, it can overlap I/O with the receive operations. We spawn M OpenMP writer threads that collaborate to parallelize these two concurrent stages as much as possible by balancing their load. To avoid over-utilization of the memory used for buffering received checkpoint files, data (whether from a local file or received) is transferred in 64 MiB chunks.

We focus our study on evaluating the impact of: (1) the number of writer threads, (2) the contiguity of the data, and (3) the number of files used by the leader for I/O aggregation.

A. Multi-threading Design

The local files of the leader are evenly distributed among the M OpenMP writer threads. Compared with evenly distributing a set number of bytes per thread, such a scheme avoids synchronization overheads associated with assigning

I/O operations at fine granularity. On the other hand, we do not have to consider such overheads with regards to the received data (from a write thread perspective), and thus it is divided up into an even number of bytes per thread.

B. Contiguous, Interleaved and Aligned I/O

Contiguity and alignment of I/O operations play a significant role in improving the aggregated I/O throughput of the PFS for various reasons: OS caching, alignment to memory pages, emphasis on performance optimizations (e.g., metadata) for a small number of writers per file and for large I/O operations that historically represented the majority in I/O patterns, etc. Since we have chosen to flush to the PFS in chunks of 64 MiB, our writes are always aligned. Thus, in this work we focus on the impact of contiguity on the performance and scalability of multi-threaded I/O. Since each write thread will write a local file fully to the PFS, the question of contiguity applies in our context for the received data. To handle the received data, we implement two alternative strategies for the writer threads:

Contiguous. We pre-assign a contiguous region in one of the checkpoint files written to the PFS by the leader. In this case, each writer thread needs to wait until it receives data that falls within this region. While this may introduce some delays, it guarantees contiguity and also benefits from the overlapping of receives with I/O, which means the delays have the potential to be masked. Specifically, each writer thread writes a contiguous region starting at the following offset:

$$offset = t_{id} * \frac{recv_{total}}{t_{count}} \tag{1}$$

Interleaved. We handle the received data on a first come, first served basis. In this case, the writer threads do not wait for specific data to be received, but at the expense of interleaving I/O operations with different offsets at fine granularity, which may lower I/O performance. Thus, each writer thread writes a 64 MiB chunk of the received data at an offset defined by:

$$offset = S * t_{id} + (n_{passes} * t_{count})) \tag{2}$$

where $S = 64$ is the maximum size of the buffer in MiB, t_{id} is the thread ID, n_{passes} is the number of times the writer thread has written a chunk of data to the PFS, and t_{count} is the total number of writer threads.

C. OST Load Balancing

We assume the PFS is deployed on P I/O servers (or OSTs in the Lustre [9] terminology, which is a popular PFS on HPC systems). To avoid over-subscribing the OSTs due to excessive I/O concurrency (at scale, $N >> P$), but at the same time allow better load balancing across the leaders to avoid OST stragglers, we assume each leader will interact with a small, controllable number of OSTs by deactivating striping. Thus we implement our micro-benchmark with a configurable number of non-striped aggregated checkpoint files per leader.

For this work, we assume we are not over-subscribing the OSTs by choosing a total number of aggregated checkpoint files $M = P * i$, where i is a small number checkpoint files assigned to each leader. However, we are also interested in determining if the leader's I/O bandwidth is limited by the minimum between the bandwidth of its own network interface and the aggregated bandwidth of using up to i OSTs.

IV. Experimental Evaluation

A. Setup

1) Hardware and Software: Our experiments were performed on Argonne's Theta, an HPC system comprised of Intel KNL 7230 compute nodes. Each node contains 64 cores and 4 hardware threads per core. In these experiments, we use a Lustre-based PFS with a peak aggregated bandwidth of 250 GB/s. The filesystem uses a total of 160 OSTs managed by 40 OSS'. Our benchmark is are written using OpenMP 4.5 and compiled with GCC 7.5.0.

2) Methodology: For the purposes of these experiments, we utilize a single compute node acting as a leader and simulate $K = 8$ MPI processes per follower compute node (meaning each "simulated" follower generates random data directly on the leader rather than sending data over the network). Both the local files of the leader and the simulated received data amount to ≈ 1 GiB per MPI rank (to account for the slight imbalance discussed previously). We vary the number of followers reporting to the leader $1 - 64$, which changes the proportion of received checkpoint data vs. local checkpoint data on the leader. Note that the writer threads will only interact with memory buffers that are already prepared, which eliminates potential delays in receiving data. This allows us to isolate the impact of contiguity from the perspective of I/O performance and scalability alone (rather than as a more complex trade-off that we plan to study in future work).

Furthermore, we vary the number of M OpenMP writer threads from $1-16$, and the number of files per leader from $1-8$. The files per leader are varied in multiple of 4 (since Lustre groups 4 OSTs into an OSS). Toggling the ratio between the number of writer threads and the number of non-leader nodes shows how the amount of work impacts throughput. The ratio between the number of threads to number of files per leader characterizes the impact of contention for the OST the file is housed on. Finally, toggling the ratio between the number of non-leaders to the number of files per leader clarifies the greatest contributor to degradation observed in both of the previous scenarios. We run each configuration 3 times and average the results presented in the graphs.

Since the local files are always written contiguously, we focus our study on the simulated received data, which is subject to the contiguous vs. the interleaved strategy. The metric we focus on is the aggregated I/O write throughput (measured by dividing the total received size by the time-to-completion of all writer threads).

B. Aggregating to One File Per Leader

Figure 1 illustrates the aggregated I/O write throughput when M OpenMP writer threads aggregate to a single file.

This corresponds to a scenario where multiple I/O writer threads interact with a single OST (since striping is disabled).

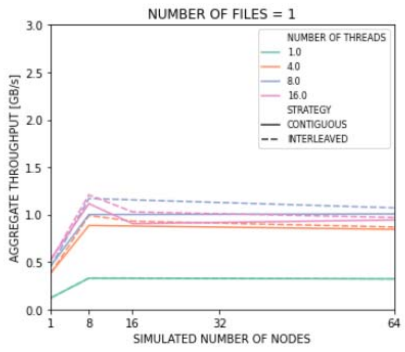

Fig. 1: Contiguous vs. interleaved aggregated throughput when number of files used for flushing = 1. Higher is better.

As can be observed, one I/O writer thread only reaches a throughput of a few hundred MiB/s. In this case, the leader engages a single CPU core, which is not enough to sustain a high I/O throughput in its interactions with the OST. On the other hand, as we increase the number of I/O writer threads, we can see a dramatic increase in aggregated I/O throughput, which reaches beyond 1 GiB/s for 8 I/O writer threads. This is actually the optimal number of I/O writer threads. Beyond that, when using 16 I/O writer threads, the aggregated I/O throughput begins do see a significant drop, which means the benefit of the leader engaging more CPU cores to avoid a client-side CPU bottleneck is offset by excessive contention to the OST, which needs to handle more concurrent writers and therefore becomes a server-side bottleneck.

Regarding the scalability of increasing received data from follower processes (8 GiB/node), we observe an interesting effect. The aggregate I/O throughput is the highest for a small number of followers, then drops slightly and stabilizes as the number of followers increases. This can be explained by the fact that the client-side OS cache on the leader absorbs write I/O overheads slightly better for a small number of followers, but, as expected, it begins to see diminishing returns.

Finally, when we compare the contiguous vs. the interleaved strategy, we observe that the interleaved writing just barely outperforms contiguous writing in all cases except for the case of one writer (because in this case there is no difference between the two strategies). As we continue to scale the amount of data, the interleaving starts to converge to the contiguous throughput. This is surprising because we initially expected that issuing contiguous writes to the PFS would obtain higher aggregated I/O throughput based on results from works like TAM [2], and especially considering that a PFS is tuned for large I/O operations. However, it seems that writing in chunks of moderate sizes like 64 MiB is enough to mitigate such considerations. While we could experiments with smaller chunk sizes to see what sizes cause visible differences, this would not have practical implications since HPC compute nodes can easily spare 64 MiB buffer space for a small number of I/O writer threads (which is 8 for our testbed).

C. Aggregating to More Files Per Leader

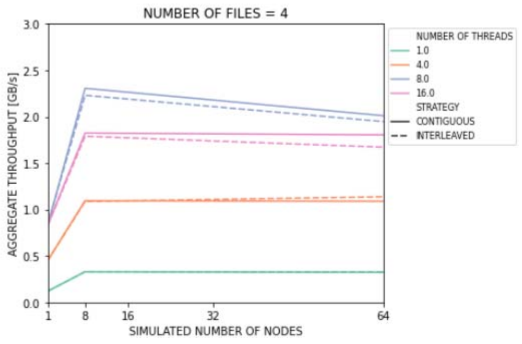

Fig. 2: Contiguous vs. interleaved aggregated throughput when number of files used for flushing = 4. Higher is better

Figure 2 shows the aggregated I/O throughout for the case in which the leader aggregates the checkpoining data to 4 files. We choose to examine this scenario to further explore the relationship between the amount of data per node to the number of files on the PFS. Furthermore, this gives us the baseline performance of a single OSS (since each OSS manages four OSTs, this allows us to saturate an OSS).

We observe in these experiments that 8 I/O writer threads, which was the optimal configuration for one aggregated file, continues to be the optimal configuration in this case too, reaching up to ≈ 2.5 GiB/s aggregated I/O throughput. The difference between the other number of I/O threads is much more dramatic: 16 I/O writer threads reach just a little over 1.5 GiB/s, while 1 and 4 I/O writer threads reach a significantly lower I/O throughout. Furthermore, we observe the same scalability trend as for one aggregated file: more followers slightly reduces the aggregated I/O throughput, which is especially noticeable for the optimal configuration of 8 I/O threads.

The contiguous vs. interleaved strategies show an interesting reversal in this case compared with the case of a single aggregated file shown in Figure 1. Specifically, the contiguous strategy now obtains a marginally better aggregated I/O throughput. In this case, a possible explanation is that interleaved writes that involve concurrent interactions with more OSTs amplify the client-side cache consolidation overheads on the leader, which slightly reduces the aggregated I/O throughput. However, this effect needs to be studied in greater detail to confirm this explanation.

Lastly, we discuss the results when each leader aggregates the checkpointing data to 8 files on the PFS. The results, depicted in figure 3 shows virtually identical results compared with the case of 4 aggregated files per leader that was discussed in figure 2. This is an important finding, because it shows that once client-side CPU bottlenecks are resolved by using the optimal number of I/O threads, increasing the number of aggregated files only helps alleviating the server-side OST bottlenecks up to a point, after which the network

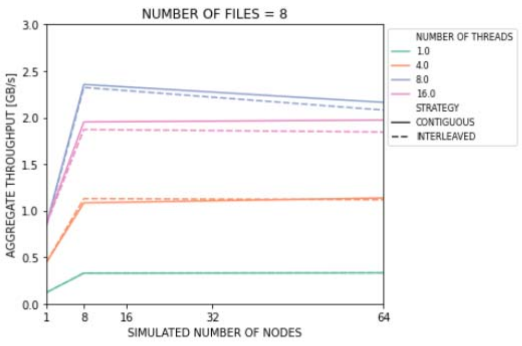

Fig. 3: Contiguous vs. interleaved aggregated throughput when number of files used for flushing = 8. Higher is better

bandwidth shared by the I/O writer threads in the interactions with the PFS becomes a client-side bottleneck again.

At scale this has important implications: since we have a small number of OSTs compared with number of compute nodes ($N >> P$), it is important to limit the over-subscription of OSTs to avoid the risk of some OSTs becoming stragglers.

V. Conclusion and Future Work

Over the last decade, numerous efforts have shown the potential for multi-threaded I/O to boost I/O write throughput in large HPC systems. In this work we study various parameters and strategies that can be applied to checkpointing I/O patterns: number of writer threads, interleaving of I/O under concurrency, and the number of output files. Such findings are important in the context of designing new I/O aggregation strategies for asynchronous flushing using background I/O writer threads, which is a gap in current state of art.

We specifically focus on leader-follower aggregation strategies that collect checkpointing data on a subset of compute nodes in order to limit the degree of contention on the I/O servers of data repositories such as parallel file systems, which typically come in much smaller numbers than the compute nodes. In this case, we find that contiguous or interleaved writing to the PFS results in roughly the same aggregated I/O throughput. Specifically, if all I/O threads are writing to a single file, it is slightly better to adopt an interleaving write strategy. Otherwise, if I/O leaders aggregate into a subset of files, it is better to assign contiguous regions of data to minimize the number of OSTs each thread accesses. However, given the difference between the two is minimal for modern parallel file systems, our opinion is that future works should focus on ensuring discrete, individual access to resources like OSS', OSTs, and storage devices, rather than guaranteeing contiguity for each thread (which is harder to coordinate efficiently and may have other impacts like not being able to absorb delays in receiving checkpointing data efficiently).

Finally, we look at tuning the number of aggregated files such that we can maximize utilization of the PFS without overwhelming it. Our experiments show that we achieve a maximum aggregated throughput of ≈ 2.5 GiB/s when writing to 4 or more files. Based on these results, we conclude that performance of multi-threaded I/O aggregation is most significantly bound by how threads interact with OSTs. Thus, future multi-threaded aggregation methods should focus on minimizing the number of OSTs threads access. In general, this is expected since other users on large distributed systems are also accessing the limited set of OSTs. Thus, the more OSTs an application or library utilizes, the greater likelihood it has of being negatively impacted by other jobs. However, it is somewhat non-intuitive, as file-per-process strategies (which by nature access an increasing number of OSTs), typically provide the highest throughput. Therefore, future multi-threaded I/O aggregation strategies should also focus on minimizing not only the number of OSTs they access, but the underlying storage devices on those OSTs as well.

In the future we plan to run these experiments across other compute platforms in order to validate our experimental procedure across various systems. Furthermore, we use the knowledge gained here to design an optimized POSIX-based I/O aggregation strategy for asynchronous checkpointing. Later, we will characterize how our aggregation strategy impacts both C/R libraries and concurrently running scientific applications.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. SHF-1910197 and SHF-1943114. The material was supported by U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11.357.

References

[1] R. Feki and E. Gabriel, "Design and evaluation of multi-threaded optimizations for individual mpi i/o operations," in *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2022, pp. 122–126.
[2] Q. Kang, S. Lee, K. Hou, R. Ross, A. Agrawal, A. Choudhary, and W.-k. Liao, "Improving mpi collective i/o for high volume non-contiguous requests with intra-node aggregation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2682–2695, 2020.
[3] S.-M. Tseng, B. Nicolae, F. Cappello, and A. Chandramowlishwaran, "Demystifying asynchronous i/o interference in hpc applications," *The International Journal of High Performance Computing Applications*, vol. 35, 2021.
[4] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, and F. Cappello, "Veloc: Towards high performance adaptive asynchronous checkpointing at large scale," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 911–920.
[5] Y. Tsujita, K. Yoshinaga, A. Hori, M. Sato, M. Namiki, and Y. Ishikawa, "Multithreaded two-phase i/o: Improving collective mpi-io performance on a lustre file system," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2014, pp. 232–235.
[6] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. USA: IEEE Computer Society, 2010, p. 1–11. [Online]. Available: https://doi.org/10.1109/SC.2010.18
[7] "Fault tolerance interface," https://github.com/leobago/fti, accessed: 2022-12-16.
[8] M. J. Gossman, B. Nicolae, J. C. Calhoun, F. Cappello, and M. C. Smith, "Towards aggregated asynchronous checkpointing," *ArXiv*, vol. abs/2112.02289, 2021.
[9] "Lustre : A scalable , high-performance file system cluster," 2003.