Copper: Cooperative Caching Layer for Scalable Data Loading in Exascale Supercomputers

1st Noah Lewis
Argonne Leadership Computing Facility
Argonne National Laboratory
Argonne, IL 60439
The Ohio State University
Columbus, Ohio, USA
lewis.3621@osu.edu

2nd Kaushik Velusamy
Argonne Leadership Computing Facility
Argonne National Laboratory
Argonne, IL 60439
kaushik.v@anl.gov

3rd Kevin Harms
Argonne Leadership Computing Facility
Argonne National Laboratory
Argonne, IL 60439
kharms@anl.gov

4th Huihuo Zheng
Argonne Leadership Computing Facility
Argonne National Laboratory
Argonne, IL 60439
huihuo.zheng@anl.gov

***Abstract*—Job initialization time of dynamic executables increases as HPC jobs launch on a larger number of nodes and processes. This is due to the processes flooding the storage system with a tremendous number of I/O requests for the same files leading to significant performance degradation causing the nodes to remain idle for an extended period of time waiting for I/O resources wasting valuable CPU cycles. In order to remove the I/O bottleneck occurring at job initialization time, a data loader capable of reducing storage-side congestion is necessary. In this paper, we introduce Copper, a read-only cooperative caching layer aimed to enable scalable data loading on a large number of nodes. We evaluate our data loading solution on the Aurora supercomputer located at Argonne National Laboratory. Our experiments show that Copper is able to load data in near constant time when scaling from 32 to $8,300$ nodes.**

***Index Terms*—HPC, scalability, hierarchical data-management**

I. Introduction

The number of nodes in high-performance computing (HPC) systems is rapidly increasing to deliver immense parallel computing power for scientists. To facilitate high-throughput data movement from storage to compute, HPC systems often use a parallel file system (PFS) such as Lustre [3] to minimize the amount of time nodes remain idle waiting for I/O resources. Lustre excels at providing high I/O throughput with large continuous file reads and writes. However, its performance has been shown to suffer when processing a large number of small file data and metadata requests [5]. The flooding of the underlying storage system with small I/O requests commonly occurs at job initialization time. This is because modern software requires numerous dependencies, such as shared libraries or Python bytecode, before the core computation can start. While initial load times may be amortized in long-running jobs, they can significantly reduce developer productivity in frequently restarted short-running jobs, which are common during interactive sessions.

In this paper we introduce Copper, a cooperative caching layer aimed to enable scalable data loading in exascale supercomputers. Copper functions as a read-only file system providing a highly generalizable and scalable data loading solution. Copper is built on the concept of cooperative caching based on work by Intel and Argonne as part of the CORAL Non-Recurring Engineering (NRE) project. The design document is no longer publicly available, but Copper takes the spirit of the design if not the technical details of its high-level design. We benchmark Copper's performance by using Python's module loader and measuring its ability to load modules when scaling to large node counts. Python is a highly popular programming language commonly used by HPC workloads due to its ease of use and large variety of popular machine learning (ML) tools and frameworks. While we evaluate Copper's performance by importing Python modules, similar performance improvements are expected in other workloads with comparable process initialization requirements, such as those with numerous dependencies consisting of a large number of small files.

This paper makes the following contributions:

- Provides an empirical analysis of the extended time required for job initialization to load program dependencies when using a large number of nodes
- Details the current state of the art solutions aimed to facilitate scalable data loading at job initialization
- Describes Copper's innovative system architecture that enables it to scale data loading efficiently across a vast number of nodes
- Measures the overall speedup and resource overhead found when utilizing Copper at scale

The structure of the paper is as follows: Section II presents an empirical analysis of the extended time required for job initialization time when using many nodes, Section III discusses related work and the gaps in the currently existing solutions, Section IV details the system architecture of Copper allowing it to scale to a large number of processes and nodes, Section V presents the results and resource overhead measured when using Copper at scale, and Section VI and VII concludes the work and details future R&D necessary to further improve the performance and reliability of Copper.

II. Problem Statement

As previously noted, when jobs utilize a large number of nodes, it is common for all processes on each node to load the same dependencies. This results in the storage system being flooded with a massive amount of I/O requests for the same set of files at job initialization time. The performance degradation of the storage system causes nodes to remain idle for extended periods of time. Since many process dependencies are often required before core computation can begin, they lie on the critical path. Therefore, minimizing process initialization time is crucial for speeding up overall job execution time.

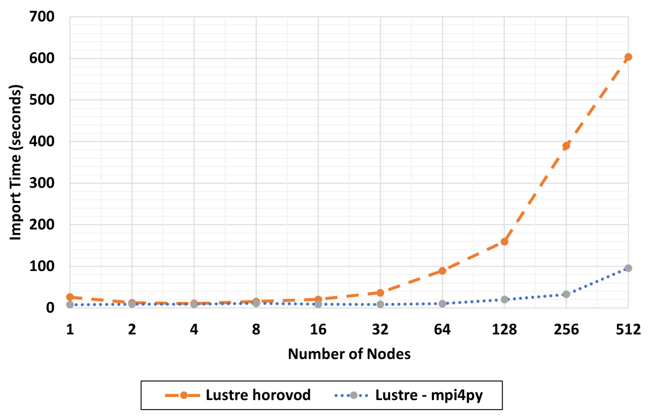

Fig. 1. Import times of the horovod with PyTorch and mpi4py Python modules with respect to the number of nodes when using the Lustre file system and 12 Python processes per node.

Figure 1 shows how the import times of the horovod with PyTorch and mpi4py Python modules increase with respect to the number of nodes used. There are 12 Python processes per node and the import times can be seen to begin increasing when using as few as 32 nodes. The time needed to load the modules increases rapidly, and when using 512 nodes, import times increase to about 603 seconds when importing horovod and about 100 seconds when importing mpi4py. Since user applications may rely on hundreds of Python modules at job initialization time, an effective data loading solution is necessary to mitigate the scalability issues demonstrated.

III. Related Work

Copper is not the first to address the inadequate job initialization times found when using a large number of nodes. In this section, we describe various systems and techniques that attempt to address this problem, each of which has its own critical shortcomings.

A. Spindle

Spindle [1] speeds up the dynamic linking and loading of shared object libraries and Python byte code loading by overriding the dynamic loader. A custom communication network is used which acts as a tree overlay network connecting nodes. Spindle hooks into the dynamic loader and redirects shared library requests through the tree overlay network to reduce the congestion that would occur if all nodes freely interacted with the underlying storage system. Copper takes inspiration from Spindle, however, Copper provides a general solution which is not specific to the application.

B. Storing Dependencies in a RAM disk

A common technique to reduce job initialization time is to place process dependencies into a RAM disk on each node. While this method demonstrates good performance, it often necessitates a complete system reset when files/data need to be modified. This solution is also limited to site provided software as users can not deploy modified system images. Additionally, since memory is a highly limited resource for many workloads, this technique further reduces the available memory for all applications.

C. Compressing Dependencies and Sharing with High-Speed Network

Another common technique to reduce job initialization time is to compress process dependencies into a single file and then distribute the compressed file to all nodes using the high-speed network. This can be done using containers or a simple tar file. Each node then decompresses the file into higher-tier storage such as a RAM disk. This method requires active management by the users to rebuild the tarball or container if dependencies are modified. In contrast, Copper provides on demand loading and caching of dependencies.

D. I/O Forwarding

Copper also shares a heritage with I/O forwarding frameworks such as IOFSL [11]. I/O forwarding frameworks are designed to forward and replay I/O functional calls on designated I/O forwarding/gateway nodes in order to concentrate the number of I/O calls from fewer clients thereby reducing the overall simultaneous I/O load on the storage system.

IV. Design

Copper reduces storage congestion at job initialization by limiting storage I/O to a single node and then distributing the data to all other requesting nodes via the high-speed network. Copper is composed of two distinct layers that run within the same background process on each node: a tree overlay network layer responsible for sharing file data, file metadata, and directory to child-entries between nodes, and an I/O interception layer responsible for intercepting I/O requests and determining how they should be resolved.

Each Copper background process includes three distinct caches for storing file data, file metadata, and directory-to-child entries. The caches are created using C++'s `std::unordered_map` [4] that has, on average, constant time removal, insertion, and search. Currently, Copper lacks a cache eviction strategy, cache size limitations, and configuration options for managing various aspects of the cache. In the remainder of the paper, the term 'node local cache' will be used to refer to all three caches collectively, and an 'I/O resource' will denote data that may be stored in any of these three caches unless specification is necessary.

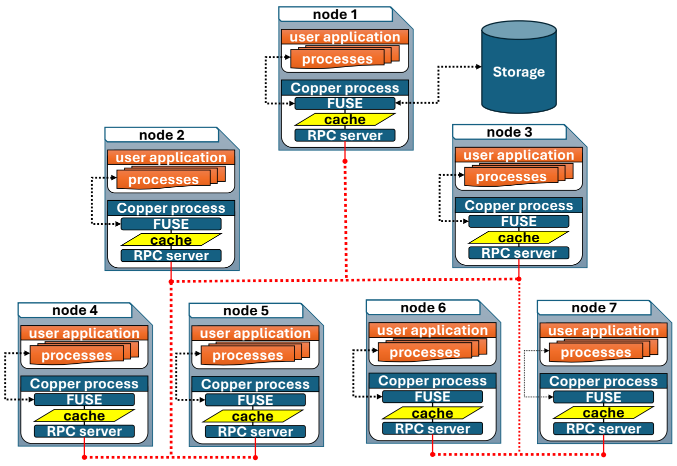

Fig. 2. RPC overlay network layer working in conjunction with the I/O interception layer. Only the root node performs I/O against the underlying storage. I/O requests, file data, and metadata are shared via the overlay network through RPC communication paths represented by the red dotted lines.

Figure 2 shows the tree overlay network layer working in conjunction with the I/O interception layer. A logical n-ary tree overlay network is constructed and storage I/O is restricted to the root node. Since both layers operate within the same background process on each node, fast access to the shared node local cache is possible without expensive communication primitives such as pipes or sockets.

A. Tree Overlay Network Layer

The tree overlay network layer ensures the fast transfer of I/O resources through the high-speed network. In addition, the overlay is responsible for routing I/O requests between the parents and children within the tree. Any node which is not the root of the tree will forward a request to its parent. The parent will check its cache to see if it can fulfill the request, if not, the request is then forwarded again to its parent until a cache hit occurs or the root of the tree is reached which performs the I/O using the underlying file system. Currently, the root node is chosen somewhat arbitrarily and is simply the first address in the nodelist file, however, future work discussed in section VII would more intelligently select the root node and take into account the machine topology when constructing the RPC tree.

Figure 3 shows the round trip time of the RPC framework for different payload sizes being sent from a node to the root node and back. The total size of the tree shown in the figure is 31 nodes. For payload sizes less than or equal to $131,072$ bytes

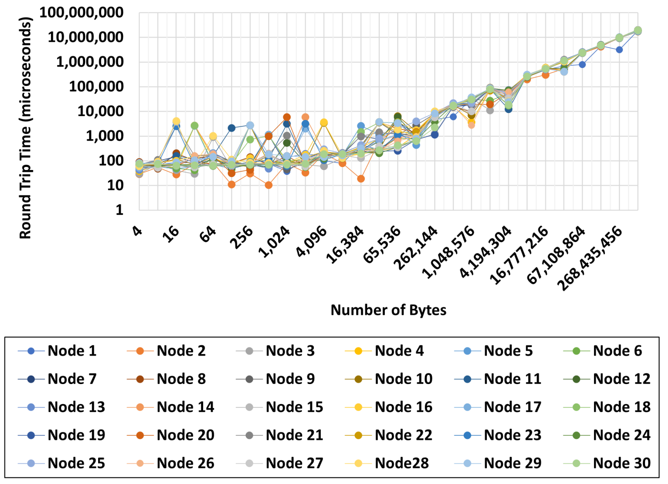

Fig. 3. Round trip time to get from a given node to the root and back when using a total of 31 nodes. As can be seen, sub-millisecond times are observed when sending $131,072$ bytes and less. The majority of file metadata payloads are within this range resulting in fast responses for I/O metadata operations.

we observed most round trips to be in the sub-millisecond range. This is significant because the majority of file metadata I/O requests will be within this byte range resulting in fast metadata response times.

1) Stopping Redundant RPCs: An optimization to reduce total communication traffic in the overlay network is to prevent forwarding redundant RPCs. If a given node receives a request for an I/O resource that is not present in the local cache and the node has an outstanding request to its parent node for the same I/O resource, the requesting thread will wait on a condition variable. Once the forwarded request returns from the parent and the I/O resource is put into the node local cache, the waiting thread is notified, at which point it can simply read from the node local cache to retrieve the I/O resource. This eliminates redundant RPCs from overwhelming the RPC engines higher up the tree which would be subject to servicing more requests.

B. I/O Service Layer

The I/O service layer uses the file system in userspace (FUSE) to receive I/O requests. When an I/O request is received by the Copper FUSE implementation, it decides whether the I/O request should be fulfilled (i.e., read I/O operations) and, if so, what steps are necessary to complete it. Copper supports the critical FUSE operations necessary for read-only workloads: `init`, `destroy`, `open`, `opendir`, `readlink`, `getattr`, `read`, `readdir` and `ioctl`.

The `init` and `destroy` operations are invoked on file system mount and unmount. These operations are responsible for starting and stopping the RPC server. The `ioctl` operation can be used to clear the internal cache or retrieve various file system metrics. The `lock` operation returns a 'fake' success status, because otherwise, the Python module importer would fail due to the error, even if it does not affect the imported module. Faking success for this operation is acceptable because Copper is a read-only file system, meaning mutually exclusive access to files is not required.

1) Entire File Read-ahead: Copper assumes that the majority of files will be requested in their entirety. All requests for file data will result in the requesting file being read in its entirety regardless of whether the user application requested the entire file. While this may cause the initial read for a subset of the file to take longer, subsequent requests for any other parts of the file can be satisfied quickly by the node local cache. In addition, this reduces the complexity of having to store partial segments of the file data in the node local cache and avoids a large number of RPCs being sent for small subsets of the file data.

2) Skipping Large Files: By default, if any node receives a request for file data in which the size of the file data exceeds 1 MB, the request is sent directly to the underlying storage system instead of going through the RPC overlay network. The default value of 1 MB is arbitrary and should be adjusted by the user to suit their specific workload. In our experiments we set the value to 10 MB. The large majority of files are under this size and skipping them still allows Copper to significantly reduce storage congestion by eliminating file data and metadata requests for small files.

C. End User Perspective

Copper aims to be highly generalized in addition to being easy to integrate into workloads without change to the user's code. When launching Copper a mount point is specified by the user which will be the directory FUSE is mounted at. This mount point will then be an alternate view for root, or /. For example if a user wants to use Copper to access data in the path `/example_data` and the Copper mount point is `/copper_mnt` then the user can simply access the data by appending the data path to the Copper mount point: `/copper_mnt/example_data`. Any I/O requests occurring in this directory will go through Copper. It should be noted that any directory can be accessed through Copper meaning it is NOT restricted to a single directory on the system.

For many users, we envision they will install program dependencies such as Python modules or shared object libraries and simply configure their environment variables (e.g., `LD_LIBRARY_PATH`, `PYTHONPATH`, or `RPATHS` etc.) to use the Copper mount point. As an example, if a user has all their Python modules in a folder named `/python_modules` and the Copper mount point is `/copper_mnt`, they would simply need to prepend their `PYTHONPATH` with `/copper_mnt/python_modules`. This way, the Python module loader will attempt to load modules through this path, which will be managed by Copper. Once this is done, Copper is being utilized without any change to the user application code. The user application can access the files and folders within the Copper mount point if it would be beneficial for their use case (i.e., many process needing read-only access to a large number of small files at the same time). They would simply prepend the Copper mount point path to the path of the data they would like to be cached.

D. Software Description

Copper has several dependencies it uses to provide high-speed data loading. It uses Mochi [1] [8] data services to create and destroy RPC servers as well as send and receive RPC requests. In particular, Thallium is used which acts as a C++ wrapper for Mochi's underlying C based services. The underlying Mochi services used by Thallium are Margo, Mercury [9], and Argobots [10] which are used to define RPCs, expose segments of memory for RDMA, and lookup RPC addresses.

The I/O service layer uses the file system in userspace (FUSE) software interface to effectively receive I/O requests. Developing the file system in userspace, as opposed to kernel space, allows Copper to leverage powerful userspace tools and libraries, such as the aforementioned Mochi services. This approach significantly accelerates development and greatly simplifies debugging. Operating strictly in userspace allows for the service to be run by the user with only the users credentials meaning no escalated privileges are required. The only requirement is for the HPC system to provide FUSE. Currently, Copper is built using the high-level FUSE API, however this is subject to change with further details available in Section VII. Because FUSE receives I/O system calls, it is broadly compatible with most major I/O interfaces including but not limited to MPI-IO, POSIX, and STDIO.

V. Results

The results presented are obtained with the Aurora supercomputer located at Argonne National Laboratory. The Aurora supercomputer has $10,624$ nodes with a total of 166 racks with $21,248$ CPUs and $63,744$ GPUs. The nodes each have 2 Intel Xeon CPU Max Series equipped with 52 physical cores supported with 2 hardware threads per core. Each CPU has 512 GB of DDR5. The system is connected in a dragonfly topology and each node has 8 HPE Slingshot-11 NICs. Copper utilizes a single NIC per node and, by default, uses 4 CPU cores per node, although the number of cores can be configured. During the testing period, Aurora was in a pre-production state and under various types of testing and configuration by vendor staff. As such, the storage system was not always operating within the most stable conditions as can be seen with the baseline results.

The storage system is the Grand Lustre file system which resides on an HPE ClusterStor E1000 platform with 100 PB of capacity across $8,480$ disk drives. The file system consists of 160 object storage targets and 40 metadata targets with an aggregate data transfer rate of 650 GB/s. Due to limited time and machine accessibility, the results presented are based on single sample sizes. In addition, as part of this testing, the launch time of `mpiexec` was evaluated to understand the cost of the most basic launch and was found to be between 0.1 and 0.3 seconds at the node counts tested.

[1]https://mochi.readthedocs.io/

The first test case was run in order to evaluate importing a complex python module, PyTorch. The test launches `python -c 'import pytorch'` via `mpiexec` and measures the entire runtime for the execution. Figure 4 shows the times observed when importing PyTorch on 32 to $8,300$ nodes while running 12 Python processes per node. As can be seen, Copper, represented by the orange solid line, provides a significant performance improvement compared to Lustre, represented by the black dotted line, decreasing import times by 60 seconds when using $2,048$ nodes. This effectively halves the time needed to import the PyTorch module. Getting the import times for PyTorch when not using Copper at $8,300$ nodes was not accomplished due to the risk of destabilizing the system.

We also measure the times for importing PyTorch a second time to evaluate the impact of reading from either the Copper cache or possibly the page cache. The second run consist of a second `mpiexec` of the exact same command within the same job. This should demonstrate the cost of reading from cache only. Since the page cache is managed by the operating system, we cannot guarantee that data accessed during the import of the Python modules remains in the cache. In contrast, Copper does not currently implement any cache eviction strategy. This means that any I/O resource requested a second time on a given node is guaranteed to be served from Copper's internal cache unless the file data is larger than the max cacheable file size. Figure 5 shows the time measured when importing PyTorch a second time. As can be seen, Copper has no increase in import times when scaling from 32 to $2,048$ nodes once the module is cached. In contrast, the second import time without Copper has significant performance degradation when scaling to large node counts. At $2,048$ nodes, the second import of PyTorch with Lustre takes about 180 seconds while Copper reduces this time to approximately 2 seconds. The expectation is that Lustre should perform better if the requested data was in the page cache, but likely overall system stability issues were effecting its performance. The increase to 30 seconds when using Copper at $8,300$ nodes was likely due to the storage system being congested for files that were above the 10 MB size and therefore not cached. We suspect increasing the max cacheable file size would result in lower import times.

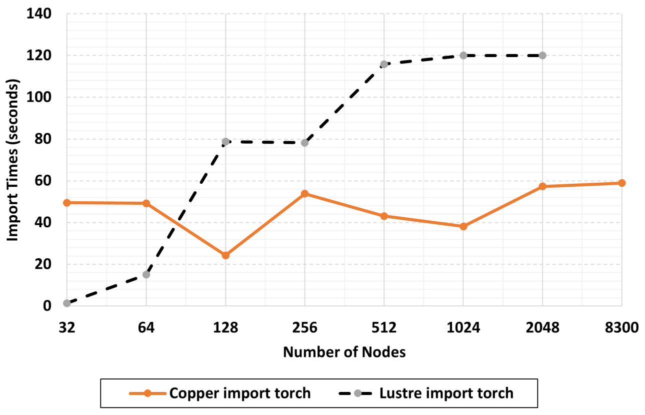

Fig. 4. Cold start time to import torch when using 12 Python processes per node on Lustre versus Copper. As can be seen Copper significantly decreases the time required to import modules and is observed to have constant load times when scaling from 32 to $8,300$ nodes.

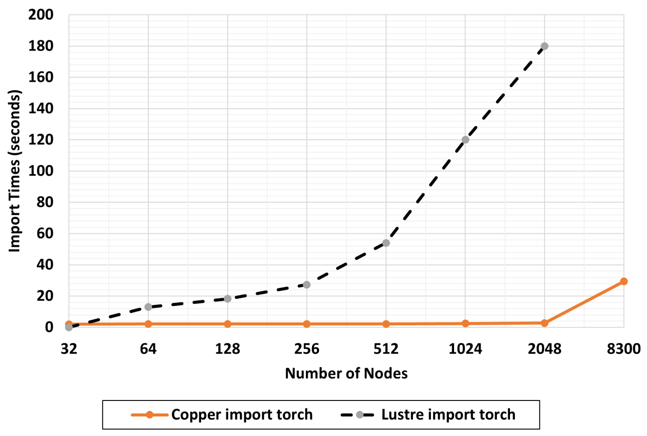

Fig. 5. Second run time to import torch using 12 Python processes per node on Lustre versus Copper. There is significant performance degradation when using Lustre at large node counts taking minutes of time while Copper is observed to have a low constant time loading of modules once cached.

As previously mentioned, Copper utilizes three caches on each node. For the presented results, files that were larger than 10 MB were skipped, resulting in Copper's per node memory overhead for PyTorch to be 36.6 MB. The data cache represented the largest portion of memory overhead, totaling 36.3 MB; the metadata cache occupied 0.2 MB, and the directory-to-child-entry cache occupied 0.01 MB. The data cache contained a total of 728 files, of which 690 were Python bytecode files, 27 were shared libraries, and the remaining 11 were Python source code files.

VI. Conclusion

As HPC jobs scale to utilize more nodes, job initialization times increase drastically, leading to extended periods where nodes remain idle while waiting for I/O to complete. Previous solutions addressed this issue with application specific solutions while Copper provides a generic solution enabling a broader range of use cases.

Copper significantly decreases job initialization time when used with jobs that use a large number of nodes by restricting the I/O against the underlying storage system to a single node effectively reducing the amount of congestion the underlying storage system incurs and utilizing the high-speed internode network to share file data and metadata. By servicing I/O requests at the system call level, Copper maintains high generalizability, requiring no changes to user application code, only accessing data via an alternate file path.

We found that Copper is capable of speeding up job initialization time by providing near constant time data loading when scaling from 32 to $8,300$ nodes. By acting as a highly generalize data loading solution, Copper effectively reduces the amount of time nodes remain idle waiting for I/O resources. Ultimately, this acceleration facilitates a faster pace of scientific discovery by significantly decreasing overall job execution time.

VII. Future Work

While Copper shows a significant speed up in job initialization time, there are many areas requiring further R&D that could enhance its overall performance, robustness, and adaptability to diverse workloads. In this section, we introduce several key features which could be added to expand Copper's capabilities.

A. Low Level FUSE API

Currently, the I/O interception layer is built using the high-level FUSE API. While this greatly simplified and accelerated the development process, we expect porting Copper to use the low-level FUSE API would improve overall performance. Vangoor et. al. [6] provide a thorough performance analysis of FUSE and the ramifications for using the high-level versus low-level FUSE API.

B. Multiple Root Nodes

Currently, I/O requests against the underlying storage system are restricted to a single node: the root node. However, PFSs offer high I/O throughput and using one node to perform I/O may underutilize the underlying storage system. One possible improvement would be to create several logical RPC network trees each with their own root node capable of performing I/O against the underlying storage system. This would have two key benefits: reducing the height of the RPC network tree and enabling multiple nodes to perform I/O, thereby properly utilizing the underlying storage system's capabilities.

C. Failure and Recovery

Currently if a node failure occurs within Copper, the failed node and its descendants will no longer be able to reach the root node. This would often result in the user application no longer being able to access their data through Copper possibly causing the user application to crash. In HPC systems, node failures are not uncommon especially for jobs which utilize a large number of nodes [7]. Long running jobs may take up to weeks or months to complete so failure recovery mechanisms are necessary. Future work could include patching the overlay network to remove the failed node allowing the previously stranded nodes to have a new path.

D. Network Security

Currently, the RPC network endpoints running on each server have no form of security. On HPE's Slingshot, the existing VNI system provides isolation from other jobs, but future work could add authentication to each endpoint to ensure that only authenticated users can access the endpoints, improving the overall security of the system.

Acknowledgment

This work was done on a pre-production supercomputer with early versions of the Aurora software development kit. This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357

References

[1] Frings, Wolfgang, et al. "Massively Parallel Loading." *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, 2013, pp. 389–398. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2464996.2465020

[2] Ross, R.B., Amvrosiadis, G., Carns, P. et al. Mochi: Composing Data Services for High-Performance Computing Environments. J. Comput. Sci. Technol. 35, 121–144 (2020). https://doi.org/10.1007/s11390-020-9802-0

[3] "Lustre : A Scalable , High-Performance File System Cluster." (2003).

[4] "std::unordered_map." *cppreference.com*, 14 Nov. 2023, https://en.cppreference.com/w/cpp/container/unordered_map. Accessed 24 July 2024.

[5] Han, Jungsoo, Dongwoo Kim, and Heon Y. Eom. "Improving the Performance of Lustre File System in HPC Environments." 2016 IEEE 1st International Workshops on Foundations and Applications of Self Systems (FASW), Augsburg, Germany, 2016, pp. 84-89. IEEE, doi: 10.1109/FAS-W.2016.29.

[6] Vangoor, Bharath Kumar Reddy, Vasily Tarasov, and Erez Zadok. "To FUSE or Not to FUSE: Performance of User-Space File Systems." 15th USENIX Conference on File and Storage Technologies (FAST 17), USENIX Association, Feb. 2017, Santa Clara, CA, pp. 59-72. ISBN 978-1-931971-36-2. USENIX, www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor.

[7] Das, A., F. Mueller, and B. Rountree. "Systemic Assessment of Node Failures in HPC Production Platforms." 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Portland, OR, USA, 2021, pp. 267-276, doi: 10.1109/IPDPS49936.2021.00035.

[8] Robert B. Ross et al. "Mochi: Composing Data Services for High-Performance Computing Environments", Journal of Computer Science and Technology. 35, 121–144 (2020). https://doi.org/10.1007/s11390-020-9802-0

[9] Soumagne J, Kimpe D, Zounmevo J, Chaarawi M, Koziol Q, Afsahi A, Ross R. Mercury: Enabling remote procedure call for high-performance computing. In Proc. the 2013 IEEE International Conference on Cluster Computing, September 2013, Article No. 50.

[10] Seo S, Amer A, Balaji P et al. Argobots: A lightweight lowlevel 1threading and tasking framework. IEEE Transactions on Parallel and Distributed Systems, 2018, 29(3): 512-526.

[11] Fitzpatrick B. Distributed caching with Memcached. Linux Journal, 2004, 2004(124): 72-76.