

An Idleness- and Failure-Tolerant Pipeline Broadcast Protocol for MPI

Xiuyi Qin

terryqqy@gmail.com

Independent Researcher

San Jose, California, USA

Abstract

As large-scale distributed computing continues to evolve with the rise of artificial intelligence and scientific applications, system resilience has become a critical concern, especially under high node count and complex synchronization. This paper presents a lightweight fault-tolerant protocol tailored for the pipeline-based broadcast model in MPI, addressing both node failure and idleness. Built on the User-level Fault Mitigation (ULFM) library, the protocol introduces a finite-state machine design to manage recovery at the application level, allowing nodes to dynamically reconfigure the broadcast topology upon abnormal events. A central node acts as a sentinel, coordinating timeout detection and recovery signaling across the pipeline. The protocol is tested in a high-performance computing (HPC) production environment, demonstrating minimal overhead even under multiple fault or timeout scenarios. Our results show that the approach effectively restores broadcast operations with scalable performance.

CCS Concepts

• **Computing methodologies** → **MapReduce algorithms**; • **Computer systems organization** → **Reliability**; **Fault-tolerant network topologies**.

Keywords

Broadcast, Collective operations, Fault tolerance, MPI

ACM Reference Format:

Xiuyi Qin. 2025. An Idleness- and Failure-Tolerant Pipeline Broadcast Protocol for MPI. In *Proceedings of (TBD)*. ACM, New York, NY, USA, 5 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

Scientific computing advances in domains such as quantum chemistry have largely followed two complementary directions. The first focuses on fundamental theoretical developments [18–20], while the second emphasizes data-driven numerical simulations, including statistical and machine learning approaches [5]. Both directions rely critically on large-scale distributed computing to achieve high accuracy and practical applicability. This paradigm demands a ever

greater number of CPU or GPU cores, which in turn increases the likelihood of node failures or idle states. As a result, resilience—the ability of an application to remain operational in the presence of faults—has become a critical challenge in high-performance systems [3, 4, 9, 21, 27, 28].

Large-scale data transfer among nodes is one of the most fundamental collective operations in HPC applications. Such operations often serve as the first indicators of hardware or software faults. Broadly, two main approaches exist to address resilience in distributed large-scale data. The first is using the distributed file system, which supports data storage and access across multiple nodes. This approach reduces inter-node dependency by focusing on hardware resilience and parallel I/O at the infrastructure layer [1, 13, 24–26]. The second approach is to message passing protocols, which use the underlying network infrastructure to transfer data between nodes. While this method eases the burden on centralized data storage, it increases inter-node dependencies and introduces complexity in synchronization and fault-tolerance.

MPI (Message Passing Interface) [6, 7] has emerged as the de facto communication standard in HPC and has recently introduced support for user-level fault tolerance [2, 15]. This extension enables recovery from failures at the application level, offering a promising path to resilience in collective operations. Among MPI’s collective operations, the broadcast subroutine plays a central role in synchronizing nodes by broadcasting data from one process to all others. Wadsworth and Chen [23] analyzed the theoretical bandwidth usage of various broadcast topologies, including round-robin, daisy chain, binomial tree, scatter-and-gather, and pipelining. Their benchmark results showed that pipelining outperformed other methods when the data size exceeded 1,000 double-precision values. In practice, MPICH [7] uses the binomial tree algorithm for short messages (less than 12,288 bytes), while larger messages are handled by scatter followed by allgather. In OpenMPI [6], the broadcast algorithm is selected based on message size and node scale, progressing through binomial tree, pipelining, split binary tree, and scatter-allgather. Algorithm selection may also be dynamically tuned based on communicator size.

Although the MPI_Bcast function utilizing ULFM library is able to detect a fault event and continue to work, its resilience is vulnerable to idle nodes, especially as communication time increases. Moreover, the broadcast phase might be customized to run with disk I/O or data preprocessing parallelly. These overlapping tasks can introduce delays, potentially stalling the entire broadcast operation.

Several fault-tolerant broadcast protocols have been proposed outside the ULFM framework, including tree-based FTCO [17], Fault-Aware MPI [8], tree-based ABFT [11, 12], message queue-based FT [14]. Hursey and Graham [10] demonstrated a fault-recovery pattern for ring communication using ULFM. Building

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
TBD,

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
<https://doi.org/XXXXXXX.XXXXXXX>

upon this, our work extends the recovery mechanism to handle idleness-induced delays, drawing inspiration from the acknowledgment mechanism proposed by Margolin [17].

In the following section, we describe the design of our fault-tolerant protocol. Benchmark results and performance analyses are presented in the final section.

2 Fault-tolerant Model

The pipeline broadcast model adopts a linear topology where all participating nodes are responsible for forwarding data sequentially. The first node splits the data into multiple chunks and begins transmission by sending each chunk to node 2. Assuming there are N worker nodes, nodes 2 through $N-1$ receive one chunk and pass it to their next neighbor. The final node, node N , only performs data reception. This model is functionally equivalent to applying `MPI_Sendrecv` at each intermediate node between the head and the tail of the pipeline. In our implementation, we use `MPI_Isend` and `MPI_Irecv` to achieve the same pipelined behavior, while also extending the model with fault-tolerant capabilities.

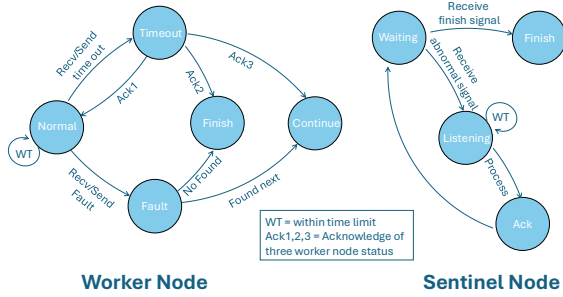


Figure 1: The status transition diagram for pipeline fault-tolerant protocol.

To handle faults, we employ a finite state machine (FSM) approach, inspired by similar concepts in networking [22]. Our fault-tolerant protocol can be viewed as a deterministic finite automaton, characterized by a finite set of states and deterministic transitions between them. The state transition diagram of our protocol is illustrated in Figure 1.

The protocol is divided into two functional perspectives: sentinel node and worker node.

On the worker node side, the default state is *Normal*, in which the worker performs receiving, any intermediate data processing, and sending. These operations may occur concurrently, particularly processing and sending. If an abnormal event is detected during a send or receive operation, the node transitions to one of two exceptional states: *Fault* or *Timeout*.

The *Fault* state is triggered by the ULFM signal handler when a node failure is detected. Upon entering this state, the affected node attempts to reconnect by locating the next or previous available node in the pipeline. Notably, compared to the timeout in the next paragraph, the *Fault* state is local in scope, minimizing its impact on the remaining nodes [10].

Algorithm 1 Idleness Detection Using Timed MPI Send and Receive

```

1: Input:    buffer,    count,    datatype,    peer-
   Rank,      tag,      communicator    ∈
   {MPI_SUCCESS, MPI_CUSTOM_TIMEOUT, MPI_CUSTOM_FAIL}
2: Initiate nonblocking communication:
3: if send operation then
4:   MPI_Isend(..., request)
5: else
6:   MPI_Irecv(..., request)
7: end if
8: if MPI call fails then
9:   return MPI_CUSTOM_FAIL
10: end if
11: start ← MPI_Wtime()
12: while MPI_Wtime() − start ≤ TIMEOUT do
13:   MPI_Test(request, flag)
14:   if MPI error detected then
15:     return MPI_CUSTOM_FAIL
16:   end if
17:   if flag == true then
18:     return MPI_SUCCESS
19:   end if
20: end while
21:   ▷ Timeout without communication completion
22: if send operation then
23:   return MPI_CUSTOM_TIMEOUT_SEND
24: else
25:   return MPI_CUSTOM_TIMEOUT_RECV
26: end if

```

In the practical HPC environments, node idleness or slowness is often as prevalent as explicit faults. Such issues can be triggered by memory exhaustion, disk idling, file system hangs, or network congestion. This condition leads to the *Timeout* state, which is activated when a chunk transfer exceeds a predefined time limit. Timeouts are detected using a combination of `MPI_Isend` or `MPI_Irecv` with `MPI_Test`, alongside a timing mechanism shown in the Algorithm 1. Once a timeout occurs, the node reports it to the sentinel and awaits acknowledgment indicating an alternative or the original neighbor. It is important to notice that when one worker reaches the *Timeout* state, other workers will timeout shortly afterward due to cascading delays. Without assistance from the sentinel, a worker node cannot distinguish between a genuine timeout and a propagated delay caused by upstream congestion. Upon receiving the acknowledgment from sentinel, the node transitions to the appropriate next state: *Normal* state if the neighbor node is healthy and the timeout was a false alarm; the *Finish* state if the destination is beyond the boundary of the pipeline; or *Continue* state if the neighbor is valid and capable of completing the remaining data transfer. The detail is listed in the Algorithm 2.

On the sentinel node side, it continuously monitors signals from worker nodes at the *Waiting* status. When it receives an abnormal event report, it enters a “listening” state and begins collecting timeout or fault notifications from all nodes. After the listening period expires, which is indicated by timeout triggered at the master node, it processes the signals to identify idle or failed nodes and

Algorithm 2 Worker-Side Pipeline Broadcast with Sentinel Recovery

```

1: Determine initial prevRank and nextRank
2: for each segment iteration do
3:   while receive not successful do
4:     rc  $\leftarrow$  MPI_Trecv(prevRank)
5:     if rc == MPI_CUSTOM_FAIL then
6:       prevRank  $\leftarrow$  prevRank - 1
7:     else if rc == MPI_CUSTOM_TIMEOUT_RECV then
8:       Notify sentinel
9:       Receive updated prevRank from sentinel
10:    end if
11:  end while
12:  if worker has successor then
13:    while send not successful do
14:      rc  $\leftarrow$  MPI_Tsend(nextRank)
15:      if rc == MPI_CUSTOM_FAIL then
16:        nextRank  $\leftarrow$  nextRank + 1
17:      else if rc == MPI_CUSTOM_TIMEOUT_SEND then
18:        Notify sentinel
19:        Receive updated nextRank from sentinel
20:      end if
21:    end while
22:  end if
23: end for
24: Notify sentinel of completion

```

replies back to each node with updated routing instructions—either a next or previous available neighbor. It then returns to the *Waiting* state, awaiting further event reports. The corresponding detail is illustrated in the Algorithm 3.

Upon completion of the pipeline phase, all nodes reach a synchronization barrier and send a success signal to the sentinel, which then concludes the broadcast phase. It is important to note that to ensure correctness of the timeout detection mechanism, a synchronization barrier must precede the broadcast stage. This guarantees that all nodes begin timing from a consistent reference point.

3 Results and Discussions

In this section, we present the performance evaluation and analysis of our fault-tolerant protocol. All tests were conducted in a production environment on the Amazon AWS multi-node cluster. The network is based on AWS EC2 Ethernet without EFA InfiniBand support. Since the protocol is implemented at the application level, the underlying network topology and hardware details are abstracted to represent a general-case scenario. In our setup, each node corresponds to a distinct physical machine equipped with multiple CPU or GPU cores. All nodes are interconnected via a full-duplex network. The OpenMPI version we tested is the latest v5.0.x based on PPRTE backend.

In the pipeline broadcast model, chunk size plays a critical role in determining the overall transmission time. For our experiments, we fixed the chunk size at 1 megabyte. The master node acts solely as a sentinel and does not participate in the actual data transmission, so the reported node counts exclude the master.

Algorithm 3 Sentinel Coordination and Fault Recovery

```

1: Initialize array slavelive[1..N]  $\leftarrow$  0
2: while broadcast not completed do
3:   Receive status message (code, rank) from any worker
4:   if code == MPI_SUCCESS then
5:     Broadcast termination condition
6:     break
7:   else
8:     slavelive[rank]  $\leftarrow$  code
9:   end if
10:  ▷ Collect additional timeout signals
11:  while true do
12:    Attempt timed receive of (code', rank')
13:    if timeout occurs then
14:      break
15:    else
16:      slavelive[rank']  $\leftarrow$  code'
17:    end if
18:  end while
19:  ▷ Compute recovery routing
20:  for each worker i = 1 to N do
21:    if slavelive[i] == TIMEOUT_SEND then
22:      Find next alive rank j > i
23:      Send j to worker i
24:    else if slavelive[i] == TIMEOUT_RECV then
25:      Find previous alive rank j < i
26:      Send j to worker i
27:    end if
28:  end for
29: end while

```

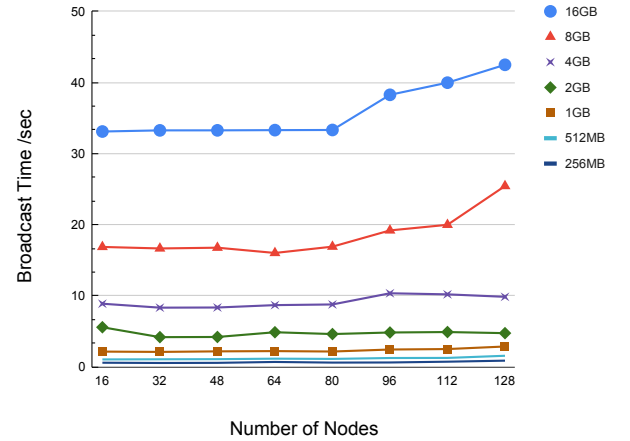


Figure 2: The broadcast time results changing with number of nodes and the amount of data transferred to each node.

Figure 2 shows the benchmark results for pipeline broadcast time as a function of both the number of nodes and total data size. As observed, the overall broadcast time increases linearly with the data size, while it grows only marginally with the number of nodes.

Table 1: Benchmark result of pipeline-based fault-tolerant broadcast protocol to transfer 16 GB data through 32 nodes under the circumstances of various abnormal events. The timeout limit is set to 5 seconds. The total 10 seconds is subtracted from timeout event to scale it aligning with the other events.

num. of event	1	2 random	2 serial	4 random	4 serial
fault event	33.21	33.24	33.26	33.31	34.26
timeout event	44.74	45.87	44.64	45.23	45.28
timeout scaled	34.74	35.87	34.64	35.23	35.28
mixed scaled	34.74	35.65	34.54	36.25	35.28

This behavior is consistent with theoretical expectations: the total broadcast time in a pipeline model includes both saturation time (pipeline filling and draining) and data transmission time, the latter of which dominates when the data volume is large relative to the number of nodes. The data transmission time can be approximated as the total data size divided by the effective bandwidth.

Table 1 presents the results of broadcasting 8 GB of data to 32 nodes under three conditions: (1) no fault, (2) with injected faults, and (3) with timeouts. As indicated in Figure 2, the average baseline time for the 8 GB broadcast is approximately 33 seconds. In the presence of faults or timeouts, the additional time required to detect and recover from events is negligible compared to the total broadcast duration. Notably, this overhead remains nearly constant even as we increase the number or alter the relative location of the events.

However, timeout scenarios inherently incur more overhead than explicit faults. This is expected, as timeouts require both the node and the master to independently reach timeout thresholds before corrective action can be taken. Consequently, timeout and mixed fault-timeout cases experience an overhead approximately equal to 2× the timeout limit, accounting for detection delays on both the node and the master sides. This overhead is comparatively lighter than the cost to restart from the checkpoint file [16].

4 Conclusion

We have presented a lightweight fault-tolerant protocol tailored for the pipeline-based broadcast model in MPI, built upon the ULFM library. The protocol enables local recovery of the broadcast pipeline in the event of node failures, leveraging ULFM’s fault-detection capabilities. In addition, it extends the fault model to include timeout detection, with a designated sentinel node to coordinate recovery and maintain pipeline integrity. This protocol could also be extended to support reduction operations and is not limited to the MPI framework.

Production-level testing on a HPC cluster demonstrated the protocol’s ability to reliably detect and recover from a variety of abnormal events—including both faults and timeouts—with minimal overhead added to the total execution time. This shows that the proposed approach is both practical and scalable, offering a viable solution for enhancing resilience in large-scale distributed systems using MPI.

Acknowledgments

The authors would like to thank Qiang Xu and Kyle Williams for insightful discussions and valuable feedback that helped improve this work.

References

- [1] Hammad Ather, Jean Luca Bez, Chen Wang, Hank Childs, Allen D. Malony, and Suren Byna. 2024. Parallel I/O Characterization and Optimization on Large-Scale HPC Systems: A 360-Degree Survey. arXiv:2501.00203 [cs.DC] <https://arxiv.org/abs/2501.00203>
- [2] Aurelien Bouteiller and George Bosilca. 2022. Implicit Actions and Non-blocking Failure Recovery with MPI. In *2022 IEEE/ACM 12th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE, 36–46. doi:10.1109/ftxs56515.2022.00009
- [3] Huangliang Dai, Shixun Wu, Hairui Zhao, Jiajun Huang, Zizhe Jian, Yue Zhu, Haiyang Hu, and Zizhong Chen. 2025. FT-Transformer: Resilient and Reliable Transformer with End-to-End Fault Tolerant Attention. arXiv:2504.02211 [cs.DC] <https://arxiv.org/abs/2504.02211>
- [4] John deVadoss and Matthias Artzt. 2025. A Byzantine Fault Tolerance Approach towards AI Safety. arXiv:2504.14668 [cs.DC] <https://arxiv.org/abs/2504.14668>
- [5] Pavlo O. Dral. 2020. Quantum Chemistry in the Age of Machine Learning. *The Journal of Physical Chemistry Letters* 11, 6 (2020), 2336–2347. doi:10.1021/acs.jpclett.9b03664
- [6] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users’ Group Meeting*. Budapest, Hungary, 97–104.
- [7] Yanfei Guo, Ken Raffanetti, Hui Zhou, Pavan Balaji, Min Si, Abdelhalim Amer, Shintaro Iwasaki, Sangmin Seo, Giuseppe Congiu, Robert Latham, Lena Oden, Thomas Gillis, Rohit Zambre, Kaimeing Ouyang, Charles Archer, Wesley Bland, Jithin Jose, Sayantan Sur, Hajime Fujita, Dmitry Durnov, Michael Chuzelev, Gengbin Zheng, Alex Brooks, Sagar Thapaliya, Taru Doodi, Maria Garazan, Steve Oyanagi, Marc Snir, and Rajeev Thakur. 2025. Preparing MPICH for exascale. *The International Journal of High Performance Computing Applications* 39, 2 (2025), 283–305. doi:10.1177/10943420241311608
- [8] Amin Hassani, Anthony Skjellum, and Ron Brightwell. 2014. Design and Evaluation of FA-MPI, a Transactional Resilience Scheme for Non-blocking MPI. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 750–755. doi:10.1109/DSN.2014.78
- [9] Jiajun Huang, Sheng Di, Xiaodong Yu, Yujia Zhai, Zhaorui Zhang, Jinyang Liu, Xiaoyi Lu, Ken Raffanetti, Hui Zhou, Kai Zhao, Zizhong Chen, Franck Cappello, Yanfei Guo, and Rajeev Thakur. 2024. An Optimized Error-controlled MPI Collective Framework Integrated with Lossy Compression. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 752–764. doi:10.1109/IPDPS57955.2024.00072
- [10] Joshua Hursey and Richard L. Graham. 2011. Building a Fault Tolerant MPI Application: A Ring Communication Example. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 1549–1556. doi:10.1109/IPDPS.2011.308
- [11] Joshua Hursey and Richard L. Graham. 2011. Preserving Collective Performance across Process Failure for a Fault Tolerant MPI. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 1208–1215. doi:10.1109/IPDPS.2011.274
- [12] Joshua Hursey and Richard L. Graham. 2012. Analyzing fault aware collective performance in a process fault tolerant MPI. *Parallel Comput.* 38, 1 (2012), 15–25. doi:10.1016/j.parco.2011.10.010 Extensions for Next-Generation Parallel Programming Models.
- [13] Olga Kogiou, Hariharan Devarajan, Chen Wang, Weikuan Yu, and Kathryn Mohror. 2023. I/O characterization and performance evaluation of large-scale storage architectures for heterogeneous workloads. In *2023 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*. 44–45. doi:10.1109/CLUSTERWorkshops61457.2023.00017
- [14] Chao LI, Changhai ZHAO, Haihua YAN, Chao LIU, Jiamin WEN, and Zengbo WANG. 2018. A fault tolerant high-performance reduction framework in complex environment. *Journal of Beijing University of Aeronautics and Astronautics (in Chinese)* 44, 10 (2018), 2115–2124.
- [15] Nuria Losada, Patricia González, María J. Martín, George Bosilca, Aurélien Bouteiller, and Keita Teranishi. 2020. Fault tolerance of MPI applications in exascale systems: The ULFM solution. *Future Generation Computer Systems* 106 (2020), 467–481. doi:10.1016/j.future.2020.01.026
- [16] Nuria Losada, María J. Martín, and Patricia González. 2017. Assessing resilient versus stop-and-restart fault-tolerant solutions in MPI applications. *The Journal of Supercomputing* 73, 1 (2017), 316–329. doi:10.1007/s11227-016-1863-z

- [17] Alexander Margolin and Amnon Barak. 2021. Tree-based fault-tolerant collective operations for MPI. *Concurrency and Computation: Practice and Experience* 33, 14 (2021), e5826. doi:10.1002/cpe.5826 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5826
- [18] Xiuyi Qin and So Hirata. 2020. Anharmonic Phonon Dispersion in Polyethylene. *The Journal of Physical Chemistry B* 124, 46 (2020), 10477–10485. doi:10.1021/acs.jpcc.0c08493
- [19] Xiuyi Qin and So Hirata. 2021. Finite-temperature vibrational full configuration interaction. *Molecular Physics* 119, 21–22 (2021), e1949503. doi:10.1080/00268976.2021.1949503 arXiv:https://doi.org/10.1080/00268976.2021.1949503
- [20] Xiuyi Qin and So Hirata. 2023. Finite-temperature many-body perturbation theory for anharmonic vibrations: Recursions, algebraic reduction, second-quantized reduction, diagrammatic rules, linked-diagram theorem, finite-temperature self-consistent field, and general-order algorithm. *The Journal of Chemical Physics* 159, 8 (08 2023), 084114. doi:10.1063/5.0164326
- [21] M. Raynal. 2022. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems*. Springer International Publishing. https://books.google.com/books?id=3YdyEAAQBAJ
- [22] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319. doi:10.1145/98163.98167
- [23] Daniel M. Wadsworth and Zizhong Chen. 2008. Performance of MPI broadcast algorithms. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–7. doi:10.1109/IPDPS.2008.4536478
- [24] Chen Wang, Kathryn Mohror, and Marc Snir. 2021. File System Semantics Requirements of HPC Applications. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing* (Virtual Event, Sweden) (HPDC '21). Association for Computing Machinery, New York, NY, USA, 19–30. doi:10.1145/3431379.3460637
- [25] Chen Wang, Houjun Tang, Jean Luca Bez, and Suren Byna. 2024. Object-Centric Data Management in HPC Workflows - A Case Study. In *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*. 104–108. doi:10.1109/CLUSTERWorkshops61563.2024.00024
- [26] Chen Wang, Ce Yu, Shanjia Tang, Jian Xiao, Jizhou Sun, and Xiangfei Meng. 2016. A general and fast distributed system for large-scale dynamic programming applications. *Parallel Comput.* 60 (2016), 1–21. doi:10.1016/j.parco.2016.09.002
- [27] Yujia Zhai, Elisabeth Gien, Quan Fan, Kai Zhao, Jinyang Liu, and Zizhong Chen. 2021. FT-BLAS: a high performance BLAS implementation with online fault tolerance. In *Proceedings of the 35th ACM International Conference on Supercomputing* (Virtual Event, USA) (ICS '21). Association for Computing Machinery, New York, NY, USA, 127–138. doi:10.1145/3447818.3460364
- [28] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. 2021. FT-CNN: Algorithm-Based Fault Tolerance for Convolutional Neural Networks. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2021), 1677–1689. doi:10.1109/TPDS.2020.3043449