BAASH: Lightweight, Efficient, and Reliable Blockchain-As-A-Service for HPC Systems

Abdullah Al Mamun, Feng Yan, Dongfang Zhao
aalmamun@nevada.unr.edu,fyan@unr.edu,dzhao@unr.edu
University of Nevada, Reno
Reno, NV, USA

ABSTRACT

Distributed resiliency becomes paramount to alleviate the growing costs of data movement and I/Os while preserving the data accuracy in HPC systems. This paper proposes to adopt blockchain-like decentralized protocols to achieve such distributed resiliency. The key challenge for such an adoption lies in the mismatch between blockchain's targeting systems (e.g., shared-nothing, loosely-coupled, TCP/IP stack) and HPC's unique design on storage subsystems, resource allocation, and programming models. We present BAASH, Blockchain-As-A-Service for HPC, deployable in a plug-n-play fashion. BAASH bridges the HPC-blockchain gap with two key components: (i) Lightweight consensus protocols for the HPC's shared-storage architecture, (ii) A new fault-tolerant mechanism compensating for the MPI to guarantee the distributed resiliency. We have implemented a prototype system and evaluated it with more than two million transactions on a 500-core HPC cluster. Results show that the prototype of the proposed techniques significantly outperforms vanilla blockchain systems and exhibits strong reliability with MPI.

CCS CONCEPTS

• **Computer systems organization** → **Availability**; **Redundancy**; *Reliability*; • **Information systems** → **Distributed storage**; Data replication tools.

KEYWORDS

Blockchain, MPI, fault tolerance, resilience, reproducibility, HPC

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476155

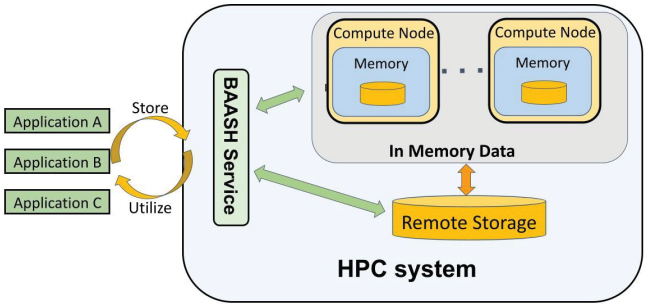


Figure 1: BAASH service is useful in a typical data movement workflow in an HPC environment.

ACM Reference Format:
Abdullah Al Mamun, Feng Yan, Dongfang Zhao. 2021. BAASH: Lightweight, Efficient, and Reliable Blockchain-As-A-Service for HPC Systems. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3458817.3476155

1 INTRODUCTION

1.1 Motivation

Distributed consistent caching. Exascale systems are likely to have extreme power constraints; yet, follow an expensive approach to move data anywhere, necessarily near the processors. However, the problem can be mitigated by replicating [16] or caching data distributedly through a distributed ledger. This is especially useful when the scientific workflows are coupled, as illustrated in Figure 1, such as multi-scale, multi-physics turbulent combustion application S3D [10], where the data moving cost often grows exponentially. Besides, performance in exascale systems will require enormous concurrency, requiring that data on each node be in the same state through synchronization leading to the eventual consistency for readers to see data from heterogeneous sources, which is desirable in modern large scale HPC systems [5].

Provenance tracking. On top of that, extreme-scale HPC systems (e.g., Cori [49]) rely on data provenance to reproduce and verify the scientific experimental results generated during the application executions. Essentially, data provenance is a well-structured log for efficient data storage along

with an easy-to-use query interface. Data provenance is conventionally implemented through file systems [47, 54] or relational databases [21, 31]. However, none of the current efforts exhibit the *immutability* property that guarantees the reliability of the provenance history.

Data fidelity. Over and above that, data fidelity is of prominent importance for scientific applications deployed to HPC systems, as the data upon which scientific discovery rests must be trustworthy and retain its veracity at every point in the scientific workflow. Silent data corruption has been a critical problem [14, 17] that leads to data loss or incorrect data, impacting the application performance at an extreme scale. Extreme-scale workflows are often long lasting and dependent on intermediate results collected from a data source or other applications, as shown in Figure 1. Therefore, those workflows typically require careful verification before caching the data in-memory or persisting them in remote storage. There have been noticeable incidents [20, 48] where the error rates grow exponentially as the system reaches extreme scale. Besides, there have been more than enough incidents about data falsification and fabrication, causing the withdrawal of scientific publications and other consequences. To this end, developing trustworthy data service for scientific computing and HPC systems has been recently incentivized by various federal funding agencies, such as the National Science Foundation [39] and the U.S. Department of Energy [15].

Blockchain-as-a-service for HPC. We argue that all of the aforementioned services could be built upon or improved by a decentralized consensus protocol—the very core technical innovation by blockchains. In essence, a blockchain can enable data synchronization in HPC systems with strong reliability in an autonomous fashion. The Oak Ridge National Laboratory has released a white paper [1] discussing a wide range of potential applications that can benefit from blockchains on the Oak Ridge Leadership Computing Facility. Recent studies [2–4, 12, 33, 34, 42, 43] showed that blockchains could be leveraged to provide a variety of services on HPC systems.

1.2 Challenges

While blockchains have drawn much research interest in many areas such as cryptocurrency [19, 22, 28] and smart government [40], the HPC and scientific computing communities, although regarding resilience as one of their top system design objectives, have not taken blockchains into their ecosystems due to various (both technical and administrative) reasons, but most notably on the following two challenges: the shared-storage system infrastructure of HPC systems and the MPI programming model for scientific applications. By contrast, all the mainstream blockchain systems and frameworks assume the underlying systems are shared-nothing clusters with the TCP/IP network stack (rather than MPI and ranks).

Specifically, although blockchain itself exhibits many research and application opportunities (comprehensive reviews available from [13, 53]), one of the most impelling challenges for employing blockchains into HPC systems lies in the consensus protocols for the unique I/O subsystem. Existing consensus protocols used in mainstream blockchains are either based on intensive computation (the so-called proof-of-work, or POW, for instance) or intensive network communication (e.g., practical Byzantine fault tolerance, PBFT), which are inappropriate for HPC in terms of both performance and cost as shown in Table 1.

Nonetheless, some of the efforts [3, 4], recently attempt to resolve some of the challenges to adopt blockchain or blockchain-like data caching in the HPC ecosystem. However, these systems either follow conventional POW [3]; hence, not appropriate for permissioned environment like HPC, or are not yet equipped with the most desired essential features as shown in Table 1, for instance, ***(i)*** extensive computation or communication free scalable consensus mechanism to establish trustworthiness among the distributed cached data, ***(ii)*** full-competent HPC protocol account for both the compute nodes and the remote storage accounting for the parallel file system (e.g., [45]) with reasonable overhead; ***(iii)*** parallel block processing, ***(iv)*** parallel consistency in the distributed ledger with lower latency, and lastly, ***(v)*** fault-tolerance support addressing MPI node failure that strengthens consistency in distributed ledger across the nodes. Hence, the state-of-the-art systems yet suffer from a lightweight strategy to deploy the blockchain-like resiliency in HPC infrastructure.

Although MPI brings a lot of opportunities to facilitate parallel processing in scientific computing, deploying blockchain with MPI is a very challenging job. In MPI, one rank failure brings down the entire communicator; hence, it raises a critical issue for a blockchain service: one single failure would crash the entire blockchain service. Several fault-tolerance approaches [8, 24] (mostly through software exception handlers) have been designed to address MPI failures. However, no work exists to address the unique requirement of blockchains: a crashed blockchain node has to restart from scratch and verify all the hash values between every adjacent pair of blocks (i.e., there is no such a concept of *checkpoint*, unfortunately).

1.3 Our Contributions

In this work, we design a new blockchain framework for HPC environments, deployed as a middleware, namely BAASH,

Table 1: Summary of limitations of present blockchain-based provenance systems.

Features	ProvChain [34]	SmartProv [42]	LineageChain [43]	IMB [3]	SciChain [4]	BAASH
Support for diskless nodes	×	×	×	✓	✓	✓
Lightweight computation	×	×	✓	×	✓	✓
Free of message broadcast	✓	✓	×	✓	✓	✓
Fully compatible to MPI	×	×	×	×	×	✓
Parallel block processing	×	×	×	×	×	✓
Parallel ledger resiliency	×	×	×	×	×	✓
Realtime fault monitoring	×	×	×	×	×	✓
Scalability on 500+ cores	×	×	×	×	×	✓

as illustrated in Figure 1. BAASH serves as a *distributed trustworthy ledger* fully compatible with the shared-storage infrastructure of HPC systems. The proposed framework overcomes the shortcomings of the state-of-the-art blockchain systems by completely *eliminating the resource-intensive requirements* through a set of specially crafted protocols. Moreover, BAASH is equipped with a *parallel processing layer* compatible with MPI, which enables BAASH to deliver low-overhead and scalable performance. The protocols assure *parallel resiliency* and account for both the compute nodes and the remote storage (e.g., [45]) with reasonable overhead. Therefore, a system implementing the proposed protocols can be smoothly deployed to an HPC infrastructure. On top of that, the *realtime fault monitor* co-designed with BAASH takes care of MPI's rank failures and thus supports the highly-desired *(eventual) consistent caching* across the compute nodes.

To summarize, this paper makes the following contributions:

- We design a set of HPC-specific scalable consensus protocols to facilitate a parallel block processing to provide a lightweight distributed in-memory and shared-storage resiliency support;
- We develop a reliable fault monitoring mechanism to ensure consistent BAASH service that can tolerate MPI rank failures to ensure the availability of the BAASH service;
- We implement a system prototype of the proposed consensus protocols and parallelization approaches with OpenMP and MPI; and
- We carry out an extensive evaluation of the system prototype with millions of transactions on an 500-core HPC platform and experimentally demonstrate the effectiveness of BAASH.

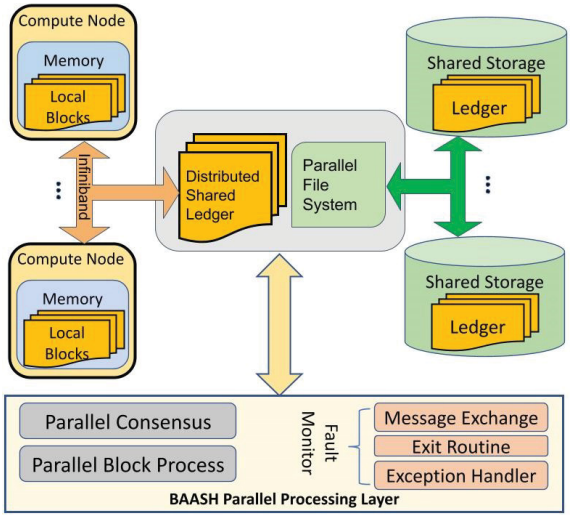


Figure 2: Overall architecture of BAASH on HPC systems.

2 BAASH DESIGN

2.1 Architecture

Figure 2 illustrates the high-level overview of our envisioned distributed BAASH system, which is deployed to a typical HPC system. Note that some customized HPC systems may have node-local disks, although this paper assumes that the compute nodes are diskless. For instance, a top-10 supercomputer called Cori [49] located at Lawrence Berkeley National Laboratory does have local SSD storage, namely, the burst buffer. However, the burst buffer is not a good candidate for ledgers because it is not designed for long-term data archival; its only purpose is to alleviate the I/O pressure for those I/O-intensive applications by caching the intermediate data locally. Besides, because scientific applications do not time-share the resources, the ledgers stored on the local storage (e.g., burst buffer) are usually purged after the job execution (for both performance and security reasons). In

other words, even if a ledger can be technically persisted to local disk in some scenarios, that persistence is not permanent, which motivates us to come up with a secondary ledger and validator on the remote storage. Specifically, four key modules of our envisioned system are highlighted in Figure 2: a resilient distributed ledger, a parallel consensus protocol, a parallel processing layer, and a fault monitor. We will discuss each of them in more detail in the following.

2.1.1 Resilient distributed ledger. The first module is a resilient distributed ledger implementation optimized for high-performance interconnects (e.g., InfiniBand) and protocols (RDMA) across compute nodes. Because all compute nodes fetch and update the ledger with few latest blocks only in volatile memory (or, in persistent local storage but with a short lifetime—purged after the job is complete) with minimum overhead, there has to be at least one permanent, persistent storage (which cannot be compromised) to store back the ledger replicas in case of catastrophes (e.g., more than 50% of compute nodes crash and lose their ledgers). Note that the memory-only compute nodes are not necessarily less reliable than those with persistent storage; yet, the data stored on memory would get lost if the process that initiates the memory is killed. It should be noted that remote storage is a cluster of nodes in a typical high-performance computing ecosystem that makes the shared blockchain highly reliable against any unexpected catastrophe.

2.1.2 Scalabale in-memory & shared-storage consensus protocol. The second module is a consensus protocol that supports attaining consensus in parallel (neither costly computational overhead nor extensive network communication) to achieve the highest possible throughput through in-memory blockchain support of the compute nodes and the resilient remote ledger. In BAASH, a block is validated with two consecutive steps. *First,* the block is validated through a parallel mechanism with in-memory blockchain support in each node. To minimize the block validation time, we avoid any traditional serialized block processing mechanism followed by state-of-the-art blockchain systems. *Second*, if the majority of the compute nodes (i.e., at least 51%) are unable to reach a consensus about the validity of a block, the remote storage then participates in the block validation process with reasonably minimum overhead. We discuss more details in Section 2.2.

2.1.3 Parallel processing layer. The third module is the parallel processing layer between the diskless compute nodes and the remote persistent storage. The layer has three purposes: (i) It injects the parallel distribution mechanism for disseminating independent blocks with transactions into the logically divided cluster of compute nodes, (ii) It facilitates parallel block processing and achieving consensus without a communication-intensive peer-to-peer mechanism, and (iii) The ledgers persisting process in the remote storage (i.e., shared-blockchain) as well as synchronization with the distributed nodes can continue independently in parallel, without interfering with the block validation. More details about this layer is explained in Section 3.4 and Section 3.5.

Protocol 1 Parallel transaction manager

Require: Clusters C where the i-th cluster is C^i; Transactions T where the i-th transaction is T^i; Entities E where the i-th entity is E^i; Remote storage R; a new block b.
Ensure: No duplicate processing of T^i.

```
1:  function Txn-Process(b, C, R, E)
2:      for T^i ∈ T do
3:          if T^i is not locked then
4:              Create hash H_T^i with timestamp s and entity-id E^i
5:              Push in a block b
6:          else
7:              Push T^i to transaction wait queue
8:          end if
9:          if time ≥ t then
10:             BAASH-Consensus(b, C, R, E)    ▷ Protocol 2
11:         end if
12:     end for
13: end function
```

2.1.4 Fault monitor. The fourth module is a real-time fault tolerance mechanism implemented with three inevitable checkpoints:

(1) Message exchange among peers: Check the communication status between the sender and receiver during each message exchange.
(2) Exit routine of each process: An exit routine that checks the status of each node when the node finishes a validation process.
(3) Exception handler: Set checkpoints around the node validation process to catch any unexpected exceptions.

Fault monitor also restores the BAASH service if any of the cases occur while handling the block validation process by the shared storage's ledger. We discuss more details about the fault monitor in Section 3.6.

2.2 Consensus Protocols

To overcome the limitations of the conventional protocols (i.e., PoW and PBFT), as a co-design of the proposed BAASH blockchain framework for scientific applications, we propose a set of protocols. Protocol 1 coordinates parallel processing of transactions of blocks. The Protocol 2 assists in managing consensus of blocks in parallel through the distributed consensus managers. The block validation process is managed

Protocol 2 BAASH consensus

Require: Clusters C where the i-th cluster is C^i; Sub-clusters N where the i-th sub-cluster is N^i managed by a coordinator N_c^i; Total nodes n in a sub-cluster; Entities E where the i-th entity is E^i; a new block b; hash list b_h^l for b; remote storage R.

Ensure: At least 50% compute node list *agreedNodes* who validate b both with local blockchain and with remote persistent ledger R_B.

```
1:  function BAASH-Consensus(b, C, R, E)
2:      while |agreedNodes| <= n/2 do
3:          for C^i ∈ C do                          ▷ In parallel
4:              Push a block b in C^i's local queue
5:              for N^i ∈ C^i do                    ▷ In parallel
6:                  N_c^i ← Validation(b, N^i, E)   ▷ Protocol 3
7:              end for
8:          end for
9:      end while
10:     if |agreedNodes| <= n/2 then
11:         if R validates b then
12:             Generate hash b_h
13:             Persist-in-Storage(b, C, R)         ▷ Call Protocol 4
14:             Release block b from active queue
15:         else
16:             Push the block to pending queue
17:         end if
18:     else
19:         Pick a hash b_h from b_h^l
20:         Persist-in-Storage(b, C, R)             ▷ Call Protocol 4
21:         Release block b from active queue
22:     end if
23:     N^i ← ∅                                     ▷ Deconstruct the sub-cluster
24: end function
```

Protocol 3 Block validation

Require: Sub-clusters N where the i-th sub-cluster is N^i; Nodes n where the i-th node is n^i. Entities E where the i-th entity is E^i; a new block b; hash list b_h^l for b where the b_h^i is the hash from n^i.

Ensure: Block b contains valid transactions T before persisted.

```
1:  function Validation(b, N^i, E)
2:      for n^i ∈ N^i do                            ▷ In parallel
3:          if n^i validates b with affected E then
4:              agreedNodes ← agreedNodes ∪ n^i
5:              Create hash b_h^i
6:              b_h^l ← b_h^l ∪ b_h^i
7:          else
8:              b_h^i ← NULL
9:          end if
10:     end for
11:     Return agreedNodes,b_h^l, b
12: end function
```

by the Protocol 3. Finally, the Protocol 4 helps in managing the resilient distributed ledger by storing the validated block in nodes' memory and persisting in the shared storage. We will discuss each protocol in detail in the following sections.

2.2.1 Parallel transaction manager. Protocol 1 aims to ensure the distinctive operation of transactions during the parallel process. The BAASH engine leverages a queue that keeps track of the individual active and pending transactions. All the active transactions are locked to prevent duplicate transaction processing by creating a unique hash with the timestamp and the entity's address (e.g., node) that issues the transactions. When a transaction arrives, it (Protocol 1) first checks (Line 3) if the transaction is unlocked (i.e., pending) and creates a unique hash (Line 4) for it with the entities addresses and the current timestamp before pushing to a block (Line 5). The transaction batching process in a block continues until a specific time (t). The time (t) is the minimum latency over a network. Finally, Protocol 1 forwards the block to start the consensus process (Line 10).

2.2.2 BAASH consensus. Protocol 2 steers the entire block validation and persisting process. The *parallel processing module* (more details in Section 3.4) leverages this protocol to manage the equal dissemination of blocks among the clusters. At first, the BAASH engine creates a set of coordinators and logically distributes the compute nodes into a group of clusters to manage the smart load-balance of the workload that facilitates parallel block processing. Each cluster is managed by a coordinator (Line 3). Each cluster is then further split into a set of sub-clusters (Line 5) where each sub-cluster processes a block (Line 6). A sub-cluster dynamically get constructed with $3f + 1 + c$ number of nodes, where f is the maximum faulty nodes allowed in a sub-cluster, and c is the maximum number of attempts of nodes adjustment from a parent cluster (more details in Section 3.4). To be more specific, at first, $3f + 1$ nodes in a sub-cluster attempt to validate a block. If more than 50% nodes in a sub-cluster fail, the coordinator attempts a maximum of c times to add an idle node from the parent cluster to continue the validation.

If a validation process exceeds the total limit (i.e., c), while attaining consensus from more than 50% nodes, the coordinator involves the remote storage to come forward to provide consensus (Line 11). Possible node failures would be either software exceptions, false validation, or crashes. If the block is valid the protocol picks a hash from the list of its (block) validators (Line 19) or create a hash within remote storage (Line 12) before persisting it (i.e., block) through the help of

Protocol 4 Persist in storage

Require: Compute nodes n where the i-th node is n^i; n^i_B the local blockchain on n^i; Clusters C where the i-th cluster is C^i; a newly mined block b; remote storage R; H_T the hash table that contains hashes of blocks stored in remote storage; R_B the blockchain copy on the storage;
Ensure: Validate b, store it to C, and persist it to R

```
1:  function PERSIST-IN-STORAGE(b, C, R)
2:      for C^i ∈ C do
3:          for n^i ∈ C^i do
4:              if b ∉ n^i_B then
5:                  n^i_B ← n^i_B ∪ b          ▷ store in local chain
6:              end if
7:          end for
8:      end for
9:      if b ∉ H_T then                ▷ Only one look-up is needed
10:         R_B ← R_B ∪ b                      ▷ store in shared chain
11:     end if
12: end function
```

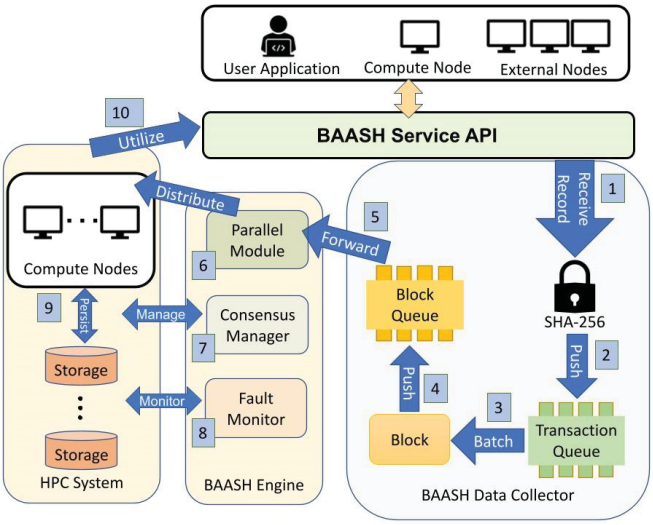


Figure 3: BAASH implementation with MPI and shared storage.

Protocol 4 (Line 13 and 20). Finally, the sub-cluster (N^i) is deconstructed again at the end of a block processing (Line 23) to leverage the free nodes for the next block processing. The time complexity of this protocol is $O(|C|)$, which is significantly lower than the existing PBFT algorithm (i.e., $O(|C|^2)$).

2.2.3 Block validation. Protocol 3 works on block validation. The validation process checks whether a block with transactions is valid (e.g., legitimate provenance operations). When a node in a cluster receives a block, it starts checking the transactions in the block against the entities that are affected, as shown in Line 3. The entities consist of any files, nodes, or any other external source, etc.

A node provides a vote after validating the block (Line 4). A validator (i.e., Node) creates a hash with its private key and the current timestamp after processing all the transactions in a block (Line 5). The validator sets the hash with *Null* value if a block is invalid (Line 8). Finally, the Protocol 3 takes advantage of *consensus manager* (more details in Section 3.5) to aggregate the votes (i.e., agreedNodes) and forwards the consensus along with the hash list (i.e., b^l_h) as well as the block with valid transactions (Line 11) to Protocol 2.

2.2.4 Persist-in-storage. Protocol 4 assists in managing the resilient distributed ledger by storing blocks in parallel to the nodes' local memory in all clusters (Line 2 and 3) while persisting blocks to the remote storage through a parallel file system (e.g., GPFS) (Line 10). Persisting in remote storage adds one more layer of reliability to the data on volatile memory. However, while attaining high reliability, the system should not exhibit significant overhead. We use a key-value model to store both the blocks and transactions both in the in-memory ledger and the remote storage. In the key-value data model, a hash represents a key. Therefore, when a node attempts to store a block in the storage node, it first looks up quickly in the storage to check whether the block is already stored. If the block hash is already in the storage, the respective node does not attempt to access the remote storage further. This is addressed in Line 9, where we only need one look-up to check in the storage. This prevents touching the shared storage more than once when other nodes try to store the same block again.

This replication process is independent and hence, does not interrupt the overall performance of BAASH during the block validation. The theoretical time complexity of this protocol is $O(|C|)$ and $|C|$ could be a fairly large number (e.g., tens of thousands of cores in leading-class supercomputers [49]). It should be noted that in HPC system remote storage is managed through a distributed cluster of storage nodes; hence, failure of a single storage node does not lead to any loss to the full ledger stored in the remote storage. We will demonstrate the effectiveness of the protocol in the evaluation section.

3 SYSTEM IMPLEMENTATION

We have implemented a prototype system of the proposed blockchain architecture in the user space and consensus protocols with Python-MPI. Although this paper presents BAASH as a user-library with a callable programming interface, BAASH can also be deployed at the system level through a wrapper. MPI is a message-passing application programming interface that allows us to develop our own

parallel processing layer. We use the mpi4py package [38] for leveraging MPI with Python. This MPI package for Python provides bindings of the MPI standard for the Python programming language, allowing any Python program to exploit multiple processors across machines. Typically, for maximum performance, each CPU (or core in a multi-core machine) will be assigned one single process or a distinct *rank*.

At this point, we only release the very core modules of the prototype. Some complementary components and plug-ins are still being tested with more edge cases and will be released once they are stable enough. Figure 3 illustrates the overview of the implemented architecture of the proposed parallel blockchain on MPI. The prototype system has been deployed to 500 cores on an HPC cluster.

As shown in Figure 3, new transactions received through BAASH service API from the nodes or any other sources (e.g., external HPC cluster or scientific workbench) are first hashed using SHA-256 [46] (step 1). Then, the transactions are pushed in a queue (step 2) to be batched or encapsulated in a block (step 3) and added to the block queue (step 4). The parallel processing module monitors both the clusters of nodes as well as the block queue. The parallel module receives a block (step 5) and distributes it whenever a cluster is available (step 6). Afterward, the blocks are validated first in compute nodes. The consensus manager collects the consensus of the block from the cluster (step 7), and if the block is validated successfully, BAASH decides to append the block both in the in-memory ledger of the compute nodes and in the remote storage through a parallel file system (i.e., GPFS) (step 9). During the entire block validation process, a fault monitor keeps track of each node and takes the necessary initiatives to recover any node failure and reinstate the BAASH service (step 8). Once the data from a source is validated and appended in the distributed ledger, other applications can access the data as a reliable source through BAASH API (step 10).

3.1 Data Models

The data structure for the proposed BAASH ledger is a linked list and stored in a hash table where each tuple corresponds to a block, which references a list of transaction records stored in another table. In each hash table, a corresponding hash acts as a primary key that helps in identifying a block or a transaction. All properties of a block (e.g., block ID, parent block hash, transactions list, and time stamp) and all properties of a transaction (e.g., transaction ID, transaction data, time stamp, and entities ID such as Node-ID, application-ID, file-ID) are encapsulated respectively in a `Block` object and in a `Transaction` object at runtime.

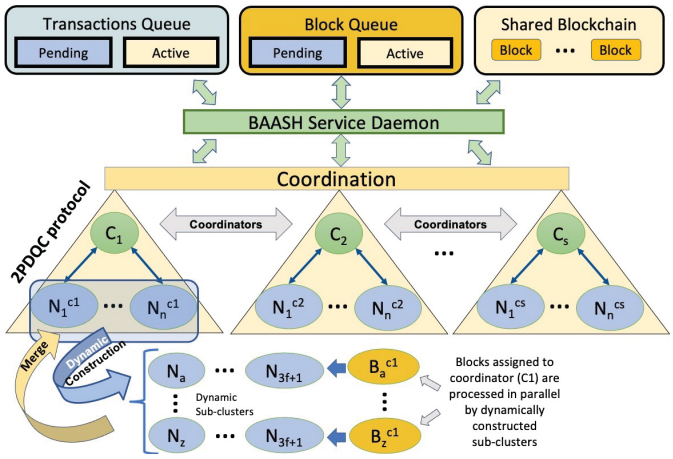


Figure 4: Parallel processing module in BAASH manages blocks in parallel through distributed coordinators or consensus managers.

3.2 Worker Nodes

In BAASH, transactions are first appended to the transaction queue, which is discussed in detail in §3.3. When the created transaction queue reaches a limit, the nodes encapsulate them (i.e., transactions) in a block. When a block is encapsulated with a number of transactions, it is then pushed into a queue (discussed in §3.3) before it is broadcasted in the network.

The nodes are responsible for validating the blocks and sending consensus to the respective coordinators (MPI communicators) through the *parallel processing module* and distributed *consensus manager*s (more details in Section §3.4 and §3.5). A coordinator will store a block both in nodes' local in-memory as well as in the remote storage after validation. The format of storing data in a compute node and in remote storage is explained in §3.1. Each node communicates with the respective coordinator through our parallel mechanism implemented with MPI discussed in §3.4. The communication between compute nodes and the remote storage is managed through a parallel file system (e.g., GPFS).

3.3 Block and Transaction Queue

The newly created or received transactions are pushed into a *pending transaction* queue. In BAASH, we can adjust the block size according to our demand. For instance, for all the experiments presented in this paper, each block consists of 4 transactions on average. Therefore, the block encapsulation process is triggered as soon as the queue size reaches 4. This is because we want to compare our experimental results with the benchmarks [13]. Once the pending transactions are encapsulated they are moved to the *active transaction* queue. When a block is ready to propagate, it is pushed

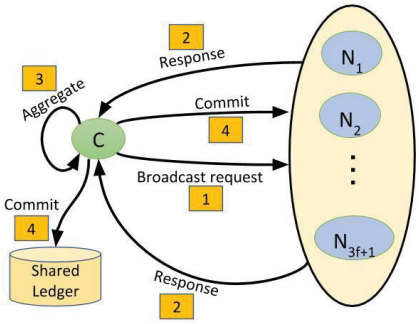


Figure 5: A consensus manager (i.e., a coordinator) guarantees the consistency in block management through the two-phase distributed quorum commit protocol.

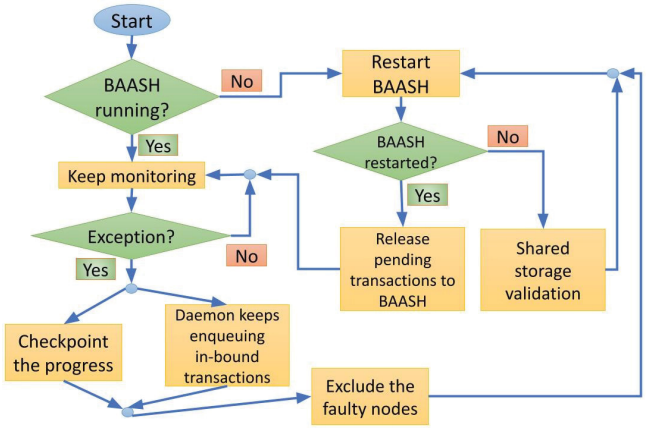


Figure 6: Workflow of BAASH's monitor to handle MPI failures. The monitor ensures a consistent BAASH service against arbitrary node failures.

to a *pending block* queue. At a fixed time interval, a block pops out from the queue periodically and is transferred to a coordinator. Once the block is transferred, it is then moved to the *active block* queue.

It could be argued that a queue might not deliver data at a sufficient rate to feed the network because a queue is a linear data structure that can hardly be parallelized for the sake of scalability. This is alleviated by the following two approaches in our implementation. First, we adjust the time interval larger than the latency for the queue to pop an element. In doing so, the overhead from the queue itself is masked completely. Second, we implement the queue using a loosely-coupled linked-lists such that the queue can be split and reconstructed arbitrarily.

3.4 Parallel Processing Module

As shown in Figure 4, the entire network is logically divided into s number of clusters. BAASH creates a set of dynamic coordinators (i.e., MPI communicators) to manage the clusters from the list of nodes. The selection process of coordinators is random, and a coordinator can not actively participate in block validation process. Each cluster is managed by a single coordinator. A cluster consists of n number of nodes from where the *parallel processing module* dynamically creates z number of of sub-clusters. Figure 4 shows how the parallel module in BAASH coordinates with the transaction queue, block queue, and the shared blockchain to distribute the blocks with transactions among a set of dynamically constructed sub-clusters of nodes from the parent cluster assigned to a coordinator (i.e., MPI communicator).

When a block arrives, BAASH starts looking for available $3f + 1$ nodes to construct a sub-cluster to start processing the block. f is the maximum faulty nodes allowed in a sub-cluster. If more than 50% nodes in a sub-cluster fail, BAASH attempts to add new nodes in the initial cluster. The initial cluster then becomes $3f + 1 + c$, where c is the maximum number of nodes adjustment from a parent cluster in case of more than 50% nodes failure in the sub-cluster. To be more specific, c denotes how many times the parallel module attempts to add a new node from the parent cluster if the majority (more than 50%) nodes in a sub-cluster fail. The cluster is flexible enough to deconstruct again to merge with the parent cluster when no blocks are available to process.

3.5 Consensus Manager

Distributed *consensus managers* (i.e., coordinators) assist in managing votes for a block processed in parallel by a set of nodes in the logically divided sub-clusters. It (i.e., consensus manager) guarantees consistency in parallel block processing through a newly designed protocol named *two-phase distributed quorum commit (2PDQC)* protocol.

Figure 5 illustrates the entire workflow of the proposed protocol. First, a coordinator broadcasts a block validation request among the nodes in a cluster. Second, the nodes forward their votes to the coordinator after the validation phase. Third, the coordinator aggregates the votes to prepare the final decision about the legitimacy of the block. Finally, the coordinator commits the block in the shared storage (shared blockchain) and synchronizes the nodes with the shared blockchain. It should be noted that all the distributed coordinators (Figure 4) keep monitoring the shared blockchain during the entire process to avoid any duplicate processing or double-spending in logically distributed clusters.

3.6 Fault Monitor

A real-time distributed monitoring process (daemon service) as shown in Figure 6 is designed to keep monitoring the

BAASH service to intercept the three aforementioned checkpoints (Section 2.1.4). If any of the checkpoints raises an alert during the block validation process, the daemon service stores (i.e., check pointing) the progress and attempts to restart the BAASH service. It (i.e., BAASH) excludes the faulty nodes during the construction of the logically distributed clusters.

The daemon keeps en-queuing the in-bound transactions and leverages the remote storage (i.e., shared blockchain) to continue the validation process while the BAASH service takes time to restart. Once the service restarts the daemon starts forwarding the pending transactions to BAASH. In the case of remote storage failure, the entire validation process is restarted automatically, which is the worst case and very rare to happen.

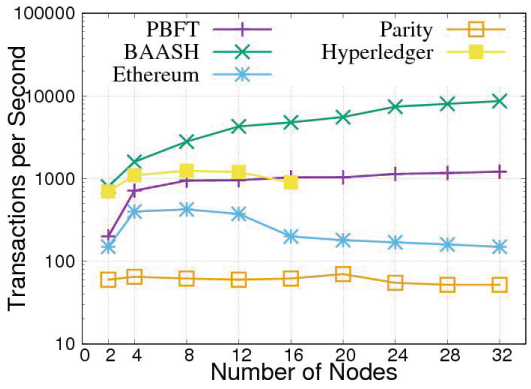
4 EVALUATION

4.1 Experimental Setup

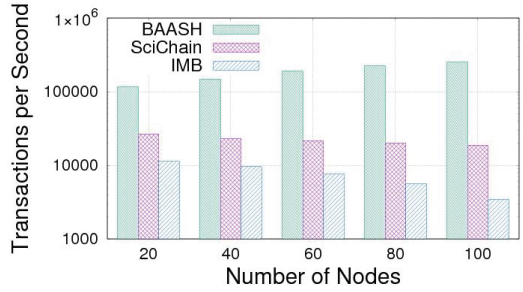
Testbed. All experiments are carried out on a high performance computing cluster comprised of 58 physical nodes interconnected with FDR InfiniBand. Each node is equipped with an Intel Core-i7 2.6 GHz 32-core CPU along with 296 GB 2400 MHz DDR4 memory; hence each node can be emulated with up to 32 nodes through user-level threads. There is no local disk on compute nodes, which is a typical configuration on HPC systems; a remote 2.1 PB storage system is available managed by GPFS [45]. Each node is installed with Ubuntu 16.04, Python 3.7.0, NumPy 1.15.4, mpi4py v2.0.0, and mpich2 v1.4.1. We deploy our system prototype on up to 500 cores and report the average results unless otherwise noted.

Workloads. We use YCSB [51], the common benchmark for evaluating blockchains [13]. The YCSB benchmark generates data in a standard format which is flexible to abstract the scientific data from arbitrary applications. There are many existing tools and frameworks (e.g., Myria [36]) for migrating scientific data sets into transactional format; we do not discuss the migration procedure in this paper and simply assume the scientific data are already in a format that can be dealt with blockchains and specifically, BAASH. In our system prototype, the default batch size of a block is set with 4, and we deploy more than two million transactions (2,013,590) in all of the experiments. It should be noted that BAASH is flexible enough to encapsulate any number of transactions in a block.

Systems for Comparison. We compare Ethereum [18], Parity [41], and Hyperledger Fabric [25] with system prototype of BAASH, whose original codename is HPC-blockchain. We also compare the state-of-the-art HPC blockchains SciChain [4], and In-memory blockchain (IMB) [3]. The sixth system is a *Conventional Blockchain* based on practical-byzantine-fault-tolerance (i.e., PBFT) protocol deployed to the same cluster (i.e., 58 nodes). We apply PBFT in each individual dynamically constructed cluster (i.e., with $3f+1$ nodes) instead of the entire blockchain network during the validation of a block to ensure the fairness of the comparison.



(a) Performance in Conventional blockchains (batch = 4)

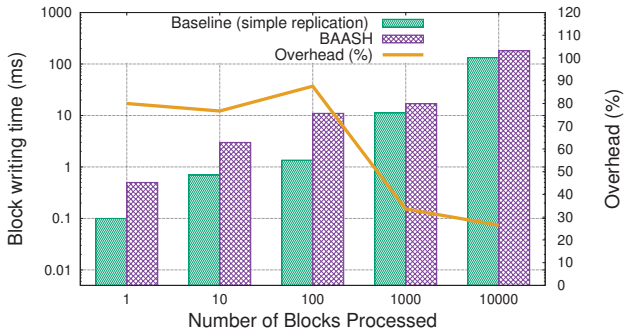


(b) Performance in HPC blockchains (batch = 250)

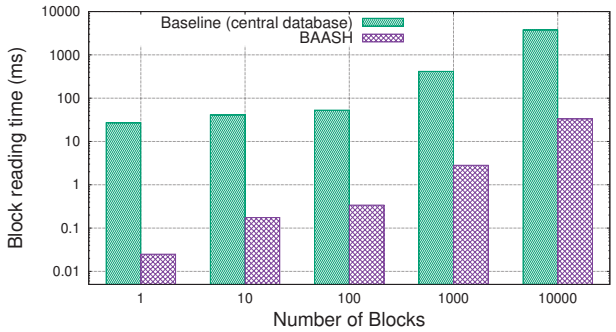
Figure 7: Transactions generated per second. BAASH outperforms the conventional blockchains. BAASH exhibits scalability compared to the state-of-the-art HPC blockchain systems.

4.2 Throughput

In this section, we measure the throughput by BAASH against the other blockchain systems. In the first experiment, we keep the batch size to BAASH's default (i.e., 4) while using 16 nodes as clients where each node issue 1024 transactions per second. Figure 7 shows the throughput with varying node scales. In terms of throughput, BAASH outperforms other conventional systems at all scales, as shown in Figure 7(a). Specifically, it has up to 6× higher throughput than Hyperledger and PBFT on 16 nodes, as well as 12×, and 75× higher throughput than Ethereum, and Parity, respectively, on 16 nodes. Thanks to the proposed consensus protocols that are optimized for HPC systems, BAASH's throughput is not degraded at larger scales beyond 16 nodes, while both



(a) Performance overhead comparison with replication across nodes.



(b) Performance comparison between BAASH and a central database.

Figure 8: Performance and overhead comparison.

Ethereum and Parity show some degradation. Worse yet, Hyperledger cannot scale beyond 16 nodes at all.

We also measure the performance of the BAASH against state-of-the-art HPC blockchains where each node issues 1024 transactions per second. In the experiment, we increase the batch size to 250 to analyze the robustness. As shown in Figure 7(b), it is noticeable that throughput in BAASH reasonably increases while scaling; whereas, the throughput tends to decrease in the other blockchain systems. It being the fact that the parallel block processing mechanism injected in BAASH assembles the consensus protocol scalable enough to generate an HPC-compatible blockchain-like resilient system. On the contrary, the IMB [3] follows the proof-of-work, although with a lower nonce and the SciChain [4] does not offer a parallel processing mechanism that could leverage the distributed nature of the HPC infrastructure.

4.3 Performance and Memory Overhead

We report the performance overhead incurred by BAASH while providing distributed reliable resiliency on a 100-node cluster with increasing workloads. That is, we increase the workload from 1 to 10,000 blocks of transactions with the default batch size 4 to measure the overhead. First, we compare the block writing performance of BAASH against the baseline that only provides resiliency through in-memory simple data replication across the nodes without any reliable verification through a decentralized protocol. We choose the simple replication as the baseline because it provides in-memory resiliency support with the minimal writing effort. As shown in Figure 8(a), although the performance overhead is around 80% while processing smaller workloads, what is more interesting and more important is the fact that the overhead starts to decline as the workload increasing, e.g., the overhead is only around 20% while processing 10,000 blocks, thanks to the parallel processing layer in BAASH which *amortizes* the cost across the participating nodes.

Table 2: Maximum memory requirement per node in BAASH at different scales.

Number of nodes	Max memory per node (MB)
20	1.4
40	2.8
60	4.1
80	5.5
100	6.9

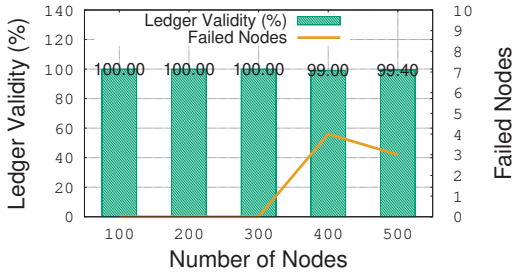
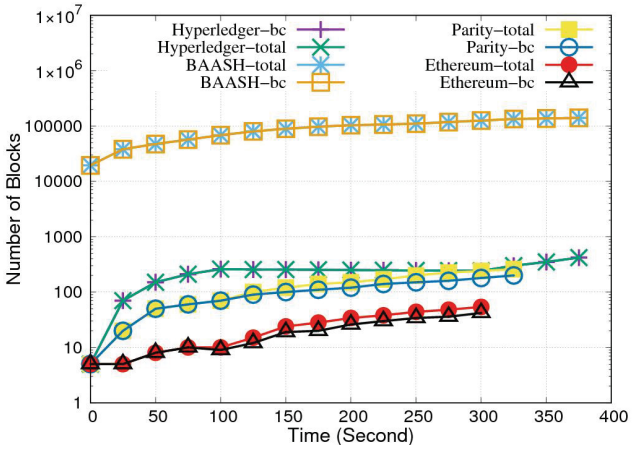


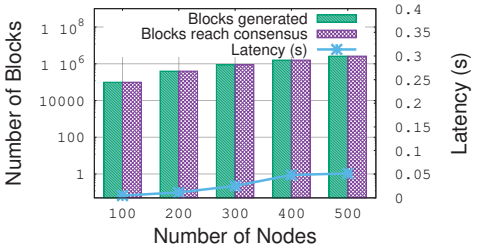
Figure 9: Ledger validity in BAASH (in all scales at-least 99% nodes hold consistent ledger).

Second, we compare the block reading performance of BAASH against the baseline that provides resiliency through the central database structured in a simple *SQLite* database in the remote storage instead of distributed resiliency through in-memory support across the nodes. We do not note the reading performance against the other baseline (i.e., simple replication without blockchain service) because both BAASH and the other baseline exhibit identical performance while reading blocks from in-memory. Figure 8(b) illustrates that the block reading performance of BAASH is significantly faster compared to the centralized database. For example, BAASH can perform almost 112× quicker compared to the baseline while reading 10,000 blocks.

We further assess the memory footprint requirement per node incurred by the BAASH service with varying node scales up to 100 nodes, where each node issues 1,024 transactions per second. Table 2 illustrates that with the increment



(a) All of the blocks in BAASH safely get persisted after successful validation: the BAASH-total line is completely covered by the BAASH-bc line.



(b) 99%+ blocks reach consensus at a large scale while incurring negligible latency.

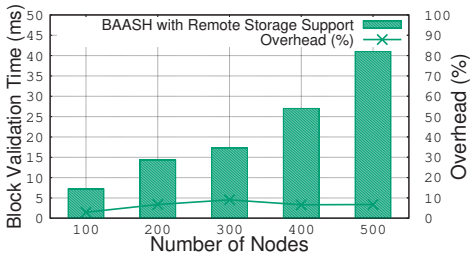
Figure 10: Reliability in BAASH resilience support.

of nodes, the workload increases; yet, each node requires a reasonable amount of memory (e.g., less than 7 MB at 100 nodes) to provide reliable distributed resiliency support across the nodes through BAASH.

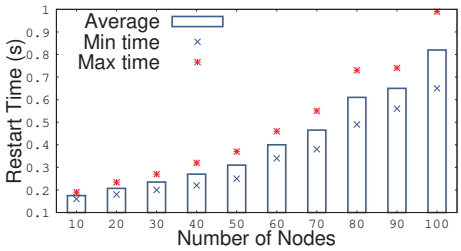
4.4 Reliability and Fault Tolerance

We measure the reliability of BAASH on up to 500 nodes and check how many nodes can hold 100% valid ledgers. We also keep track of how much node failure occurs during this experiment. Figure 9 shows that in all scales, at least 99% nodes keep the valid ledger. We find 100% nodes hold valid ledger on up to 300 nodes. BAASH guarantees the resiliency as long as more than 50% nodes hold consistent ledger at all scale.

To further measure the reliability of BAASH, we investigate how many blocks reach consensus in all the systems on 16 nodes, where 8 concurrent nodes issue 1024 transactions per second. The reason for choosing 16 nodes is to keep the result comparable to [13]. Figure 10(a) reports the number of blocks that reach the consensus and are appended to the blockchain. Both in Ethereum and Parity, some blocks



(a) Validation through remote storage handled by the fault monitor during BAASH recovery exhibits reasonable performance overhead.



(b) BAASH's recovery controlled by the fault monitor lies within decent limit while the remote storage handling the validation.

Figure 11: BAASH recovery overhead by fault monitor.

are unable to reach consensus due to double spending or selfish mining attacks, and the difference increases as time passes. Though Hyperledger is not vulnerable to those attacks because of not allowing any fork and 100% blocks tend to reach consensus, it is significantly slower than BAASH. In BAASH, almost 100× more blocks are generated compared to Hyperledger, and 100% blocks reach consensus. This is because, BAASH implementation relies on the specially crafted parallel transaction processing mechanism and the newly designed two-phase distributed quorum commit protocol to ensure consistent block processing in a parallel manner. Figure 10(b) reports the number of blocks that reach consensus in BAASH at different scales ranging from 100 to 500 nodes. We notice that in all scales, more than 99% generated blocks can reach consensus with negligible latency.

The entire MPI network fails if a node crashes; hence, could hinder the entire block validation process. Therefore, we keep the remote storage to continue the validation. To further study the fault tolerance, we check the block validation time by BAASH with remote storage support at different scales by switching off a random node in the middle of a block validation process and compare the overhead. During the experiment, each node issues 1024 transactions per second. The goal of this experiment is to see how much is the overhead if we leverage the remote storage during

the entire MPI network failure while restoring the BAASH service. As shown in Figure 11(a), even with failures of the compute node, the block validation process continues successfully with the support of the remote storage. Besides, BAASH with remote ledger incurs negligible overhead even at a large scale (i.e., roughly 5% at 500 nodes cluster).

In case of a severe catastrophe (e.g., compute nodes crash), the distributed fault monitor in BAASH simply restarts the BAASH service excluding the faulty nodes while continuing the validation process with the support of the shared storage. During the restarting phase, the BAASH daemon needs to restore the nodes with the latest block, so that the nodes can join the validation process. To measure the BAASH service restoration time on up to 500 nodes cluster, we switch off a random node in the middle of a block validation process and restore the service (i.e., BAASH) excluding the failed node. We conduct the experiment several times at different scales to measure the minimum and maximum restart time. Figure 11(b) exhibits BAASH service requires reasonable time during the restoration phase. However, the good news is, the block validation is not stopped during this restoration phase, because BAASH leverages shared storage's blockchain to continue the successful block validation process.

5 RELATED WORK

At present, blockchain research focuses on various system perspectives. Algorand [22] proposes a technique to avoid the Sybil attack and targeted attack. Bitcoin-NG [19] increases throughput through a leader selection from each epoch who posts multiple blocks. Monoxide [50] distributes the workload for computation, storage, and memory for state representation among multiple zones to minimize the responsibility of full nodes. Sharding protocols [29, 52] have been proposed to scale-out distributed ledger. Hawk [30] is proposed to achieve transactional privacy from public blockchains. A blockchain protocol based on proof-of-stake called Ouroboros [27]is proposed to provide stern security guarantees in blockchains and better efficiency than those blockchains based on proof-of-work (i.e., PoW). Some recent works [23, 28, 37] propose techniques to optimize Byzantine Fault Tolerance (BFT).

Being inspired by Hyperledger [25], Inkchain [26] is designed as another permissioned blockchain solution that enables customization and enhancement in arbitrary scenarios. To improve reliability and achieve better fault tolerance, BigchainDB [7] is built upon the Practical Byzantine Fault Tolerant (PBFT) and comes with blockchain properties (e.g., decentralization, immutability, owner-controlled assets) as well as database properties (e.g., high transaction rate, low latency, indexing and querying of structured data). A 2-layer blockchain architecture (2LBC) is proposed in [6] to achieve stronger data integrity in distributed database systems based on a leader-rotation approach and proof-of-work. Unfortunately, none of the aforementioned work addresses the underlying platform architecture other than shared-nothing clusters assumed by existing blockchain systems that could help us to bridge the gap between the HPC and blockchain technology. The proposed blockchain framework presented by this paper, therefore, for the first time, showcases a practical parallel blockchain-like framework developed with MPI that allows us to leverage the decentralized mechanism in HPC systems.

We are well aware of recent advances in MPI. Various approaches [8, 9, 11, 24, 32, 35, 44] were proposed to improve or characterize the MPI features in order to design various solutions. However, all of these works are orthogonal to our work, and none of these aim to develop a lightweight blockchain framework with MPI in order to facilitate the parallel processing in managing distributed ledgers. Therefore, the aforementioned works can be merged into our work for further improvement in MPI specific packages.

6 CONCLUSION AND FUTURE WORK

This paper proposes two key techniques to enable a blockchain service in HPC systems. First, a much-needed lightweight set of scalable consensus protocols is designed to account for both the diskless compute nodes and the remote shared storage in HPC. Second, in an HPC environment, a must-have property of blockchain, i.e., the reliability in front of MPI, is guaranteed by multiple layers of subsystems, from background daemon services to callable routines to applications. The HPC-aware consensus protocols and MPI-compatible reliability, under the framework coined as BAASH, collectively enable a blockchain service for HPC systems. A system prototype of the proposed techniques is implemented and evaluated with more than two million transactions on up to 500 cores, which demonstrates both high performance and high reliability compared to state-of-the-art systems.

Our future research will (i) incorporate a *parallel data integration* mechanism for the cross-blockchains or heterogeneous data sources, and (ii) support *eventual* consistency for readers to see data from the remote writers in the exascale system. Our hope is that BAASH could serve as a starting point for a new line of system research on decentralized services crafted to HPC systems.

7 ACKNOWLEDGEMENTS

This work is supported in part by the following grants: National Science Foundation CCF-1756013, IIS-1838024, and CAREER-2048044.

REFERENCES

[1] Advancing the Science and Impact of Blockchain Technology at Oak Ridge National Laboratory. https://info.ornl.gov/sites/publications/Files/Pub118487.pdf, Accessed 2019.

[2] Abdullah Al-Mamun, Tonglin Li, Mohammad Sadoghi, Linhua Jiang, Haoting Shen, and Dongfang Zhao. Poster: An mpi-based blockchain framework for data fidelity in high-performance computing systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2019.

[3] Abdullah Al-Mamun, Tonglin Li, Mohammad Sadoghi, and Dongfang Zhao. In-memory blockchain: Toward efficient and trustworthy data provenance for hpc systems. In *IEEE International Conference on Big Data (BigData)*, 2018.

[4] Abdullah Al-Mamun, Feng Yan, and Dongfang Zhao. SciChain: Blockchain-enabled lightweight and efficient data provenance for reproducible scientific computing. In *IEEE 37th International Conference on Data Engineering (ICDE)*, 2021.

[5] Benjamin S Allen, Matthew A Ezell, Paul Peltz, Doug Jacobsen, Eric Roman, Cory Lueninghoener, and J Lowell Wofford. Modernizing the hpc system software stack. *arXiv preprint arXiv:2007.10290*, 2020.

[6] L. Aniello, R. Baldoni, E. Gaetani, F. Lombardi, A. Margheri, and V. Sassone. A prototype evaluation of a tamper-resistant high performance blockchain-based transaction log for a distributed database. In *13th European Dependable Computing Conference (EDCC)*, 2017.

[7] BigchainDB. https://github.com/bigchaindb/bigchaindb, Accessed 2018.

[8] Darius Buntinas. Scalable distributed consensus to support mpi fault tolerance. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 1240–1249. IEEE, 2012.

[9] Lu Chao, Chundian Li, Fan Liang, Xiaoyi Lu, and Zhiwei Xu. Accelerating apache hive with mpi for data warehouse systems. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 664–673. IEEE, 2015.

[10] Jacqueline H Chen, Alok Choudhary, Bronis De Supinski, Matthew DeVries, Evatt R Hawkes, Scott Klasky, Wei-Keng Liao, Kwan-Liu Ma, John Mellor-Crummey, Norbert Podhorszki, et al. Terascale direct numerical simulations of turbulent combustion using s3d. *Computational Science & Discovery*, 2(1):015001, 2009.

[11] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. Characterization of mpi usage on a production supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 30:1–30:15. IEEE Press, 2018.

[12] Hao Dai, H Patrick Young, Thomas JS Durant, Guannan Gong, Mingming Kang, Harlan M Krumholz, Wade L Schulz, and Lixin Jiang. Trialchain: A blockchain-based platform to validate data integrity in large, biomedical research studies. *arXiv preprint arXiv:1807.03662*, 2018.

[13] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. Blockbench: A framework for analyzing private blockchains. In *ACM International Conference on Management of Data (SIGMOD)*, 2017.

[14] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale. *CoRR*, abs/2102.11245, 2021.

[15] DOE SBIR. https://www.sbir.gov/sbirsearch/detail/1307745, Accessed 2017.

[16] Shaohua Duan, Pradeep Subedi, Philip Davis, Keita Teranishi, Hemanth Kolla, Marc Gamell, and Manish Parashar. Corec: Scalable and resilient in-memory data staging for in-situ workflows. *ACM Transactions on Parallel Computing (TOPC)*, 7(2):1–29, 2020.

[17] Shaohua Duan, Pradeep Subedi, Philip E Davis, and Manish Parashar. Addressing data resiliency for staging based scientific workflows. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–22, 2019.

[18] Ethereum. https://www.ethereum.org/, Accessed 2018.

[19] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 45–59, Berkeley, CA, USA, 2016. USENIX Association.

[20] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2012.

[21] Ashish Gehani and Dawood Tariq. SPADE: Support for Provenance Auditing in Distributed Environments. In *Proceedings of the 13th International Middleware Conference (Middleware)*, 2012.

[22] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM.

[23] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.

[24] Yanfei Guo, Wesley Bland, Pavan Balaji, and Xiaobo Zhou. Fault tolerant mapreduce-mpi for hpc clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 34:1–34:12, 2015.

[25] Hyperledger. https://www.hyperledger.org/, Accessed 2018.

[26] Inkchain. https://github.com/inklabsfoundation/inkchain, Accessed 2018.

[27] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Advances in Cryptology (CRYPTO)*, 2017.

[28] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 279–296, 2016.

[29] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[30] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[31] S. Lee, S. Kohler, B. Ludascher, and B. Glavic. A sql-middleware unifying why and why-not provenance for first-order queries. In *IEEE International Conference on Data Engineering (ICDE)*, 2017.

[32] Mingzhe Li, Khaled Hamidouche, Xiaoyi Lu, Hari Subramoni, Jie Zhang, and Dhabaleswar K Panda. Designing mpi library with on-demand paging (odp) of infiniband: challenges and benefits. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 433–443. IEEE, 2016.

[33] X. Liang, S. Shetty, D. Tosh, C. Kamhoua, K. Kwiat, and L. Njilla. Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017.

[34] X. Liang, S. Shetty, D. Tosh, C. Kamhoua, K. Kwiat, and L. Njilla. Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017.

[35] Xiaoyi Lu, Fan Liang, Bing Wang, Li Zha, and Zhiwei Xu. Datampi: extending mpi to hadoop-like big data computing. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 829–838. IEEE, 2014.

[36] Parmita Mehta, Sven Dorkenwald, Dongfang Zhao, Tomer Kaftan, Alvin Cheung, Magdalena Balazinska, Ariel Rokem, Andrew Connolly, Jacob Vanderplas, and Yusra AlSayyad. Comparative evaluation of big-data systems on scientific image analytics workloads. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*, 2017.

[37] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016.

[38] MPI4PY. https://mpi4py.readthedocs.io/en/stable/intro.html, Accessed 2019.

[39] NSF CICI. https://www.nsf.gov/pubs/2018/nsf18547/nsf18547.htm, Accessed 2018.

[40] Svein Ølnes. Beyond bitcoin enabling smart government using blockchain technology. In *International Conference on Electronic Government*, pages 253–264. Springer, 2016.

[41] Parity. https://ethcore.io/parity.html/, Accessed 2018.

[42] Aravind Ramachandran and Murat Kantarcioglu. Smartprovenance: A distributed, blockchain based data provenance system. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pages 35–42, 2018.

[43] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. Fine-grained, secure and efficient data provenance on blockchain systems. *Proceedings of the VLDB Endowment*, 12(9):975–988, 2019.

[44] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. Static/dynamic validation of mpi collective communications in multi-threaded context. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 279–280. ACM, 2015.

[45] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[46] SHA-256. https://en.bitcoin.it/wiki/SHA-256, Accessed 2018.

[47] Chen Shou, Dongfang Zhao, Tanu Malik, and Ioan Raicu. Towards a provenance-aware distributed filesystem. In *TaPP Workshop, USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[48] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. *ACM SIGARCH Computer Architecture News*, 43(1):297–310, 2015.

[49] The Cori Supercomputer. http://www.nersc.gov/users/computational-systems/cori, Accessed 2018.

[50] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchain with asynchronized consensus zones. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019. USENIX Association.

[51] YCSB. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads, Accessed 2018.

[52] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 931–948. ACM, 2018.

[53] Kaiwen Zhang and Hans-Arno Jacobsen. Towards dependable, scalable, and pervasive distributed ledgers with blockchains. In *38th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2018.

[54] Dongfang Zhao, Chen Shou, Tanu Malik, and Ioan Raicu. Distributed data provenance for large-scale data-intensive computing. In *IEEE International Conference on Cluster Computing (CLUSTER)*, 2013.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

All experiments are carried out on a high-performance computing cluster comprised of 58 nodes interconnected with FDR InfiniBand. Each node is equipped with an Intel Core-i7 2.6 GHz 32-core CPU along with 296 GB 2400 MHz DDR4 memory; hence each node can be emulated with up to 32 nodes through user-level threads.

Author-Created or Modified Artifacts:

```
Persistent ID: https://github.com/mamunbond07/beast
Artifact name: BAASH
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Intel Core-i7 2.6 GHz 32 core CPU

Operating systems and versions: Ubuntu 18.04

Libraries and versions: Python 3.7.0, NumPy 1.15.4, mpi4py v2.0.0, and mpich2 v1.4.1