A Feasibility Study for MPI over HDFS

W. Feng, D. Zhang, J. Zhang, K. Hou, S. Pumma, and H. Wang
Department of Computer Science, Virginia Tech
Blacksburg, VA USA
Email: {wfeng, daz3, zjing14, kaixihou, sarunya, hwang121}@vt.edu

***Abstract*—With the increasing prominence of integrating high-performance computing (HPC) with big-data (BIGDATA) processing, running MPI over the Hadoop Distributed File System (HDFS) offers a promising approach for delivering better scalability and fault tolerance to traditional HPC applications. However, it comes with challenges that discourage such an approach: (1) two-sided MPI communication to support intermediate data processing, (2) a focus on enabling N-1 writes that is subject to the default HDFS block-placement policy, and (3) a pipelined writing mode in HDFS that cannot fully utilize the underlying HPC hardware. So, while directly integrating MPI with HDFS may deliver better scalability and fault tolerance to MPI applications, it will fall short of delivering competitive performance.**

Consequently, we present a performance study to evaluate the feasibility of integrating MPI applications to run over HDFS. Specifically, we show that by aggregating and reordering intermediate data and coordinating computation and I/O when running MPI over HDFS, we can deliver up to 1.92× and 1.78× speedup over MPI I/O and HDFS pipelined-write implementations, respectively.

I. Introduction

Apache Hadoop [3] has become synonymous with big-data (BIGDATA) processing. Solutions built atop Hadoop include Mesos [10], Tachyon [17], YARN [24], and Spark [29], [30]. Moreover, many data-centric applications integrate with different parts of Hadoop, e.g., Hadoop Distributed File System (HDFS) and Hadoop Database (HBase), to provide distributed data services. In contrast, the high-performance computing (HPC) community typically uses traditional parallel programming models (e.g., MPI, OpenMP, and Pthreads) atop HPC parallel file systems (e.g., Lustre [1] and GPFS [22]) for compute-centric applications on HPC clusters.

With the increasing prominence of integrating HPC and BIGDATA infrastructures, including [8], [9], [12], [14], [15], [18], [20], [25], [26], [27], [28], running MPI applications over a commodity distributed file system, like HDFS, offers a promising cost-effective solution to support better scalability and fault tolerance for traditional HPC applications. For example, emerging deep-learning systems, e.g., the COTS system for image classification [6] and the Deep Speech 2 system for speech recognition [2], use MPI for frequent data communication and a distributed file system for large-scale and fault-tolerant data storage. However, several challenges inhibit their integration. First, MPI developers must invest significant effort to implement intermediate data processing, e.g., merge and sort, due to the lack of standard support in MPI. Second, most MPI-based applications use two-sided communication to process intermediate data, resulting in a lost opportunity to overlap computation and disk I/O. Third, previous studies that support MPI applications on HDFS, including [8], [27], [28], focus on translating I/O requests or modifying the lock mechanism of HDFS to enable the N-1 writes and are subject to the default HDFS block placement policy, resulting in poor data locality. Fourth, the pipelined-write mode used in HDFS cannot fully utilize the underlying HPC infrastructure.

In this paper, we tackle these challenges by proposing an underlying data communication and I/O middleware layer between an MPI application and HDFS that aggregates and reorders intermediate data and coordinates (ARC) them with computation and disk I/O via a set of user-friendly MPI-like interfaces for developers to manipulate intermediate data at will. In all, this paper makes the following contributions:

- A highly-optimized approach to merge and sort intermediate data to enrich (or supplement) MPI.
- Application of one-sided communication to expose more opportunities to overlap communication, computation, and disk I/O for intermediate data processing.
- A coordinator-selection mechanism that supports a set of I/O strategies that take into account data locality and communication pattern.
- A parallel-write mechanism for HDFS with delay-write mode to support fast disk I/O.
- An empirical study that illustrates how our performance-oriented approach can feasibly achieve between 1.31× and 1.92× speedups over MPI I/O and default HDFS (with pipelined write), respectively.

II. Background

The convergence of high-performance computing (HPC) and big-data (BIGDATA) processing has spawned two main approaches: (1) data-centric and (2) compute-centric. In turn, these approaches differ in both their *programming models* and *hardware/software configurations*.

In data-centric approaches, the programming models (e.g., MapReduce) partition data into blocks to enable distributed processing on a cluster. During processing, communication is minimal due to the independence between the blocks. In contrast, compute-centric approaches rely more heavily on communication and synchronization, and thus, the corresponding programming models (e.g., MPI) provide support for

This research was supported in part by the NSF XPS program via CCF-1337244. We also acknowledge Advanced Research Computing at Virginia Tech for access to their production computational resources.

collective communication (e.g., gather, scatter, all-to-all) and explicit synchronization (e.g., barrier, wait, wait-all).

For hardware and software configurations, a datacenter often consists of commodity compute nodes connected by an Ethernet network. Each node has its own local disk, which when aggregated with other disks forms a distributed file system (e.g., HDFS [3]) to provide persistent storage. In contrast, a supercomputing center connects nodes via a high-performance interconnect (e.g., Mellanox InfiniBand) to better serve the frequent communication and synchronization in compute-centric applications. Rather than using local storage, parallel file systems (e.g., Lustre [1], GPFS [22] and PVFS [21]) use separate storage to deliver fast I/O.

A. MPI One-sided Communication

MPI is the de facto standard to support interprocess communication for HPC applications. One-sided communication, i.e., remote memory access (RMA), supports fast and efficient communication. It only needs the communicating source to initiate API calls, e.g., `MPI_Put()` and `MPI_Get()`, and set communication parameters. One-sided communication also defines *window* objects and a *scope* to allow each process to register a memory region accessed by the permitted processes. To complete the issued RMA operations, two types of synchronization mechanisms are supported: (1) *active* target synchronization and (2) *passive* target synchronization.

B. HDFS Pipelined Write

The Hadoop Distributed File System (HDFS) is a popular distributed file system [3] that is designed to be highly fault-tolerant and support large-scale data storage. HDFS consists of a *NameNode* and multiple *DataNodes*. The NameNode manages a namespace of the entire file system and serves as a proxy between clients and DataNodes; DataNodes store data blocks and their replicas assigned by the NameNode.

In HDFS, a large input file is split into small blocks, where each block is replicated and distributed across DataNodes to support data resilience. Fig. 1 illustrates the workflow of a traditional HDFS block write with a single NameNode and three DataNodes, $disk0$, $disk1$, and $disk2$. The workflow consists of three main stages: (1) REQUEST (steps 1-3 in Fig. 1). A client requests to write a block $b0$ and the NameNode determines $disk0$ to store it and the other disks to store replicas based on placement policies. (2) WRITE (steps 4-5). The client transfers $b0$ to $disk0$, followed by a sequential replication to $disk1$ and then $disk2$. Asynchronously, other blocks from the client "flows" the same way from $disk0$ to $disk2$, causing a pipelined write. (3) CLOSING (steps 6-7). Every DataNode notifies the NameNode when it completes the block replication and the client is then aware of the completion of writing the first replica.

III. FEASIBILITY STUDY

To demonstrate the performance feasibility of running MPI over HDFS, we created proof-of-concept software called ARC, which supports the aggregation and reordering of intermediate

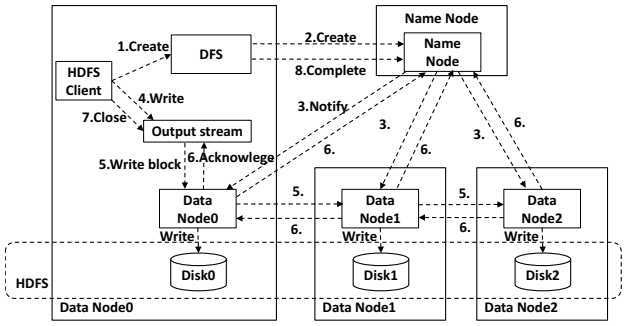


Fig. 1: HDFS pipelined write mode.

```
1  ARCInit(threshold, top, ARC_LEAST_RR);
2  ARCHandle handle = ARCRegister("/output_file");
3  while(true) {
4      size_t num_reads = load_queries(queries);
5      if(num_reads == 0)
6          break;
7      local_search(queries, num_reads, &count, metadata,
         ↪  realdata);
8      ARCWriteAfterSortWithDC((void*) metadata,
         ↪  sizeof(header), count, compare, desc, handle);
9  }
10 ARCFinalize();
```
Fig. 2: Code snippets: Using ARC interfaces.

data and the coordination of computation and I/O over HDFS. (Based on the outcomes of this study, as reported in §IV, we will subsume ARC's user-friendly MPI-like interface by simply "hijacking" an MPI application's I/O calls to run directly over HDFS but with our ARC functionality.)

A. Interfaces

Table I describes our user-friendly MPI-like interface called ARC. At a high level, ARC registers and buffers intermediate data, which generally contains metadata and real data. The metadata is further distributed amongst MPI processes for merging and sorting, and ARC uses the results to determine multiple coordinators via different rules. Then, the coordinators collect real data from all the MPI processes to construct the output data and flush to HDFS in a parallel manner.

Fig. 2 shows how easily users can write (or port) their application via the ARC API. *ARCInit* (line 1) allocates internal memory buffers and creates and configures the one-sided communication windows associated with these buffers. Users also need to set the *threshold* to trigger the aggregation, set the output element numbers *top*, and choose the coordinator-selecting strategy with *coord*. In this example, ARC selects the coordinators with the least local data (see §III-D for details). After that, *ARCRegister* registers a file with a *handle* (line 2) for the output data. After each round of local search, *ARCWriteAfterSortWithDC* (line 8) registers the intermediate data (i.e., *metadata* and *real data*) and leaves tedious data manipulation, communication, and I/O tasks to ARC. Finally, *ARCFinalize()* flushes the buffered output data to HDFS (line 10), finish all issued I/O requests, and free the *windows* and memory buffers. Users are also able to explicitly wait for the I/O requests against a specific file or just check it without being blocked by calling *ARCWait* or *ARCTest*, respectively.

TABLE I: Description of the messaging functions in ARC

int ARCInit(int threshold, int top, int coord) Allocate memory and initiate MPI one-sided communication configuration. ***threshold***: threshold to trigger aggregation. ***top***: number of sorted elements written to HDFS. ***coord***: coordinators selection strategy.
int ARCFinalize() Flush buffered output data to HDFS. Then deallocate memory and release MPI one-sided communication configuration.
ARCHandle ARCRegister(char *file) Register a HDFS file with a *ARCHandle* that can be used, later, to specify the output file or check the completion of HDFS I/O requests at the registered file. ***file***: HDFS file pathname. **return**: a *ARCHandle* link to the registered file.
int ARCWriteAfterSortWithDC(void *ptr, size_t size, size_t count, int (*compar) (const void *, const void *), struct deep_copy desc, ARCHandle handle) Write data to HDFS after sorting elements with deep copy. ***ptr***: pointer to the array of elements(on current MPI process). ***size***: size in bytes of each element. ***count***: number of elements. ***compar***: pointer to the user defined compare function used by sort. ***desc***: deep_copy variable to describe the number of pointers of deep copy, their offsets in user defined struct, and their data size. ***handle***: *ARCHanle* of the output file.
int ARCWriteAfterSort(const void *ptr, size_t size, size_t count, int (*compar) (const void *, const void *), ARChandle handle) Write data to HDFS after sorting elements without deep copy.
int ARCWrite(const void *ptr, size_t size, size_t count, ARCHandle handle) Write data to HDFS without sorting and deep copy. The order of elements are not guaranteed.
int ARCWait(ARChandle hanle) Wait for the completion of all issued I/O requests at the file refered by *handle*.
int ARCTest(ARChandle hanle) Check the status of all issued I/O requests at the file refered by *handle*.
int ARCFlush() Force to flush buffered output data to HDFS.

B. Aggregation of Intermediate Data

Prior to merge and sort, ARC needs to aggregate the metadata from all the MPI processes. Traditionally, the MPI-based merge and sort methods use a centralized solution. Such a solution assigns a master process to aggregate the metadata and then merges and sorts them locally, followed by propagating the result to all MPI processes with two-sided communication. However, because two-sided communication inhibits the realization of efficient aggregation and misses opportunities for overlapping communication, computation, and I/O to hide latency, we adopt a distributed strategy with one-sided communication.

In our distributed strategy, *ARCInit* allocates internal buffers and registers them with MPI window objects (*MPI_Win*). To reduce conflicts within the same window object, we define a dedicated MPI window object for each MPI process in ARC.

As shown in Fig. 3, ARC ensures that every MPI process updates its corresponding internal *count* and *data* buffers on each MPI process using *MPI_Accumulate* and *MPI_Put*, respectively, instead of *MPI_Get* in order to achieve better performance. To aggregate the real data (i.e., the internal *deep copy* buffer), ARC's coordinators use *MPI_Get* to avoid extra synchronization between the coordinators and other MPI processes. Then, every process merges and sorts the metadata in its local internal *data* buffer.

C. Reordering of Intermediate Data

To achieve better performance in the merge and sort step, as shown in Fig. 3, we leverage our highly optimized SIMD merge sort methods [11] to sort the elements in the data buffer.

While many compute-centric applications need sorting, such as muBLASTP [31], such functionality is not provided in the MPI standard and must be implemented by developers. To hide the details of data reordering and provide a highly-optimized sorting kernel, our ARC sort exploits not only thread-level parallelism but also data-level parallelism via carefully designed SIMD operations. We extend the idea of parallelization [11] to support the metadata structure rather than the built-in variables (e.g., float, int).

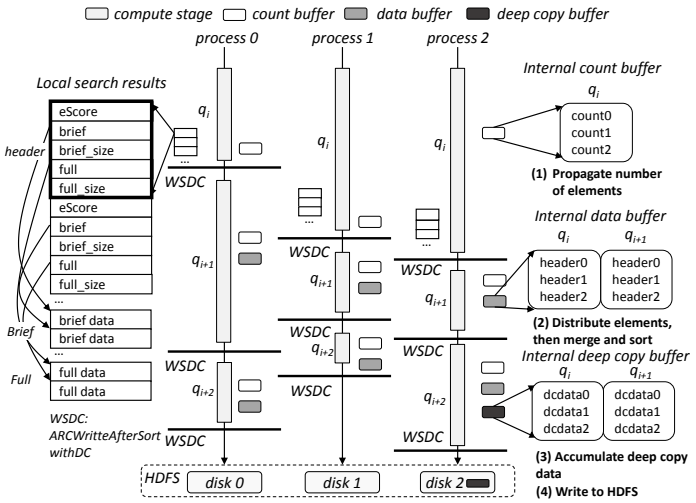


Fig. 3: Design of coordinating communication and disk I/O with computation.

D. Selecting Coordinators

ARC uses coordinators to aggregate and flush the output data, resulting in additional overhead for the coordinators, which might delay the completion time of the application. In ARC, we provide four coordinator-selecting strategies: (1) most local data, *ARC_Most*; (2) least local data, *ARC_LEAST*; (3) most local data with round robin, *ARC_MOST_RR*; and (4) least local data with round robin, *ARC_LEAST_RR*. *ARC_Most* chooses coordinators with the most local data contributing to the output data; it may produce less communication overhead because less data is being transferred. Thus, it might fit MPI applications with large unbalanced local data. In contrast,

ARC_LEAST might implicitly suggest a faster workload and lead to better overlapping for an unbalanced workload. Moreover, due to the nature that the runtime of a MPI application is determined by the slowest process, *ARC_LEAST* might help avoid the extra overhead on the slower process and reduce delay on the completion time of the application.

ARC_MOST_RR and *ARC_LEAST_RR* with round robin can help balance data-block distribution among DataNodes of HDFS. In addition, round robin might benefit *ARC_Most* from reducing extra overhead on the slower processes and lead to less communication overhead for *ARC_Least*. We also need to consider the availability of free space, due to the limited space of local disk (configured to be part of HDFS).

E. Parallel Block Write on HDFS

In contrast to the pipelined write in HDFS, ARC realizes an HDFS parallel write that supports writing replicas to DataNodes in a *parallel* manner. Specifically, ARC enables multiple MPI processes to be coordinators to write replicas to HDFS in parallel. Due to the importance of data locality in selecting writers, ARC maintains a table that maps DataNodes to running MPI processes. Then, during ARC's selection of coordinators (cf. III-D), ARC can select the top MPI processes with sufficient disk space and execute them on different DataNodes. The number of selected MPI processes equals the the number of replicas configured for HDFS. These coordinator processes aggregate output from all MPI processes and then write the final data locally in parallel.

Fig. 4 shows our parallel block writes on HDFS. The processes on three DataNodes are selected as coordinators to write the three replicas of one data block. One of the three MPI processes is assigned as a leader to communicate with the NameNode. To execute a parallel block write, (1) the leader sends a request to the NameNode to write a data block; (2) the NameNode generates a local file path for the block and adds metadata of the block into the namespace; (3) the NameNode returns the local path to the leader process; (4) the leader process sends the local path to the two other coordinators via one-sided MPI communication; (5) upon receipt of the local path, all coordinators write data to the local disk in parallel and notify the leader; the leader; and (7) the leader notifies the Namenode that all blocks are successfully written. In case of failure, all previous operations are retired, and another coordinator is chosen as the new leader. For client-NameNode communication, ARC uses remote procedure call (RPC); for writer-writer communication, ARC uses MPI.

In addition, because HDFS does not support N-1 write, which allows multiple writers to write in the same file concurrently, many research efforts have proposed N-1 write on HDFS [4], [8], [27], [28]. In ARC, we enable N-1 write on HDFS via a two-step process: (1) a N-M write (M is the number of replicas), where M coordinators aggregate the data from all processes via MPI one-sided communication, and (2) the coordinators perform a M-M write to write the data to HDFS via ARC's parallel write.

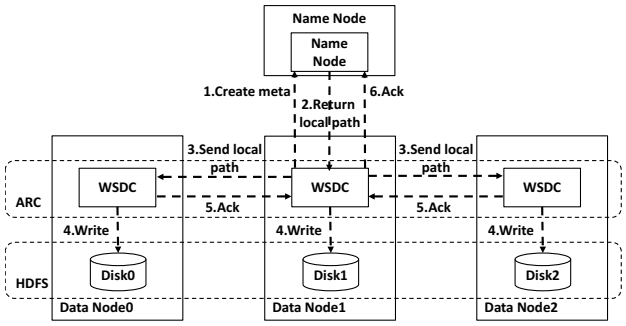


Fig. 4: HDFS parallel write mode in ARC.

IV. EXPERIMENTS

Here we present the empirical results of our feasibility study for MPI over HDFS via our ARC middleware. The study uses a 17-node Hadoop cluster using Hadoop 2.7.2, where one node is the NameNode and the other 16 are DataNodes. Each node has two Intel oct-core Sandy Bridge Xeon CPUs (E5-2670) with 64-GB main memory and 205-GB local disk. The communication network is QDR InfiniBand while the storage network is 10-Gbps Ethernet. We set up Hadoop with the default configuration, where the nblock size is 128 MB and the number of replicas is three. All programs are compiled with the Intel C/C++ compiler 13.1 with flags `-O3` and MPI-3.2.

A. Test Applications

To test the efficacy of ARC and, in turn, the feasibility of MPI over HDFS, we use two applications: (1) *pBWA* [19], short for parallel Burrow-Wheeler Aligner, a popular short-read alignment tool for mapping short DNA reads to a long reference genome [16] and (2) *pDIAMOND*, an MPI-parallelized version of the DIAMOND sequence alignment tool [5].

For *pBWA*, we realized three MPI implementations: (1) *pBWA*, the original pBWA implementation using MPI two-sided communication and MPI I/O interfaces to write the results to the underlying parallel file system; (2) *pBWA with HDFS*, the original pBWA using the default HDFS pipelined write to write the result on HDFS; and (3) *pBWA with ARC*, our modified pBWA with ARC to communicate and write the result to HDFS. For inputs, we use two real-world datasets from the *1000 Genomes Project* [7], as shown in Table II. The reference genome is the 3-GB `hg19.fa` from [23].

TABLE II: pBWA input datasets

Name	Number of reads	Size(GB)
SRR096576_1	95,661,734	18
SRR077475_1	22,493,783	5.6

For *pDIAMOND*, we also realized three MPI versions: (1) *pDIAMOND*, an MPI implementation of DIAMOND using MPI two-sided communication and MPI I/O to flush the data to the underlying parallel file system; (2) *pDIAMOND with HDFS*, i.e., *pDIAMOND* but using HDFS pipelined write to flush data to HDFS; and (3) *pDIAMOND with ARC*, our modified *pDIAMOND* with ARC. We use two sequence databases:

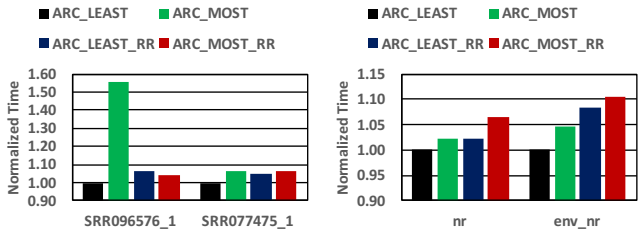
nr and *env_nr*, as shown in Table III. We randomly sample 10,000 sequences from each database as inputs, respectively.

TABLE III: DIAMOND reference databases

Name	Number of sequences	Size(GB)
nr	85,107,862	53
env_nr	6,865,992	1.7

B. Results via Different Strategies for Selecting Coordinators

As shown in Fig. 5, we tested the performance of our four strategies for selecting coordinators: (1) *ARC_LEAST*, choosing nodes with the least local data; (2) *ARC_MOST*, choosing nodes with the most local data; (3) *ARC_LEAST_RR*, choosing nodes with the least local data and round-robin method; and (4) *ARC_MOST_RR*, choosing nodes with the most local data and round-robin method. The execution times are normalized to the total execution time of using *ARC_LEAST*.



(a) *pBWA* with ARC (b) *MPI DIAMOND* with ARC

Fig. 5: ARC using different coordinators selecting strategies for *pBWA* and *pDIAMOND* (or *MPI DIAMOND*).

In all of our experiments, the *ARC_LEAST* strategy achieved the best performance. Selecting these "least local data" nodes can better exploit overlapping communication and disk I/O with computation and avoid extra overhead on the slowest nodes. In Fig. 5(a), the experimental results for *pBWA* with ARC on dataset *SRR096576_1* show a skewed workload. *ARC_MOST* continually selected the slowest nodes to flush the output data, resulting in heavy delay on its completion time. The round-robin method (e.g., *ARC_MOST_RR*) can be used to balance block distribution. However, there are cases that round-robin methods add extra overhead and slightly degrade performance, as indicated in Fig. 5(b).

C. Results via Different Implementations of Applications

1) ***pBWA****:* Fig. 6 shows the normalized execution times of our three *pBWA* implementations for different datasets on 8 and 16 nodes, respectively. We normalize the performance relative to the total execution time of *pBWA* with ARC, which achieves the fastest execution time in all cases. (Note: Lower normalized execution time means better performance.) In Figs. 6(a) and 6(b) for dataset SRR096576_1, *pBWA with ARC* delivers 1.28× and 1.31× improvement over *pBWA* on 8 and 16 nodes, respectively, and 1.34× and 1.75× improvement over *pBWA with HDFS*, on 8 and 16 nodes, respectively. In Figs. 6(c) and 6(d) for the dataset SRR077475_1, *pBWA with ARC* achieves 1.30× and 1.22× speedup over *pBWA* and 1.36× and 1.92× speedup over *pBWA with HDFS*, on 8 and 16 nodes, respectively.

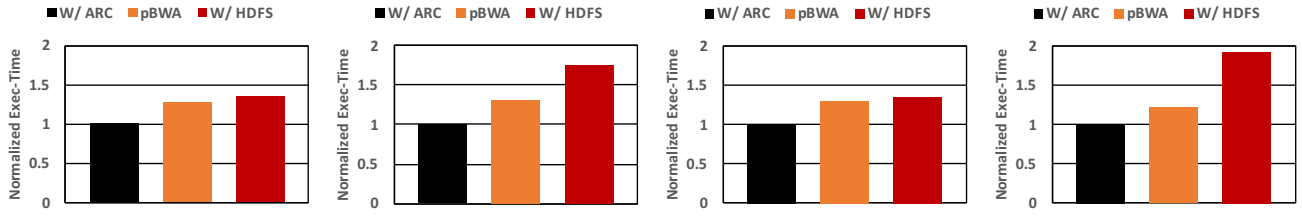
Why did our ARC-optimized *pBWA* implementation achieve superior performance to its *pBWA* counterparts? First, the disk I/O on a traditional file system consumes a large portion of the overall execution time for *pBWA* – 37%. In contrast, our HDFS parallel write provides fast disk I/O by flushing the output data to local disk, resulting in only 18% I/O overhead. By leveraging MPI one-sided communication in ARC to aggressively overlap disk I/O with communication and computation, this I/O overhead is reduced even further. In addition, in contrast to HDFS pipelined write, ARC buffers the output data to fit a block instead of flushing them to HDFS directly, thus avoiding many small write operations.

Fig. 7 shows the *scalability* of the three implementations over different datasets. We vary the number of nodes from 4 to 16 and use the performance on four nodes as the baseline. As shown in Fig. 7(a) for dataset SRR096576_1, *pBWA* with ARC delivers up to 3.58× improvement while *pBWA* with MPI I/O and with HDFS pipelined write delivers up to 2.83× and 2.49× improvement, respectively, on 16 nodes. In Fig. 7(b) for dataset SRR077475_1, we observe up to 3.28×, 2.97×, and 2.07× improvement for *pBWA* with ARC, MPI I/O, and HDFS pipelined write, respectively, on 16 nodes.

2) ***pDIAMOND****:* Fig. 8 presents the normalized execution times for our three *pDIAMOND* implementations against two databases. We normalize the performance relative to the total execution time of *pDIAMOND* with ARC. In Figs. 8(a) and 8(b) for database *nr*, *pDIAMOND* with ARC achieves 1.08× and 1.17× speedup over the baseline *pDIAMOND* with two-sided MPI communication and MPI I/O (i.e., the DIAMOND legend in Figs. 8(a) and 8(b)) on 8 and 16 nodes, respectively. It also delivers 1.18× and 1.20× improvement over *pDIAMOND* with two-sided MPI communication and HDFS pipelined write, on 8 and 16 nodes, respectively.

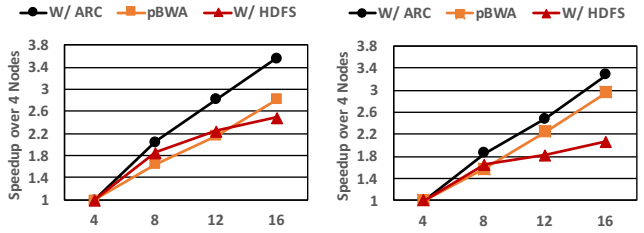
Figs. 8(c) and 8(d) show our experimental results running over the *env_nr* database. Our *pDIAMOND* with ARC achieves 1.43× and 1.78× speedup over *pDIAMOND* with two-sided MPI communication and MPI I/O, on 8 and 16 nodes, respectively. Comparing our *pDIAMOND* with ARC implementation to *pDIAMOND* with MPI two-sided communication and HDFS pipelined write, our implementation achieves 1.63× and 1.75× speedup, on 8 and 16 nodes, respectively. On very large databases like the *nr* database, disk I/O amounts to relatively small overhead (up to 15% across the three implementations) and results in less opportunity to overlap disk I/O with computation and communication. In contrast, searching against a small database, i.e., *env_nr*, can result in up to 36% disk I/O overhead. Our *pDIAMOND* with ARC delivers more speedup than the other two parallelized implementations. In addition, *pDIAMOND* needs the intermediate data to be merged and sorted, which ARC can further improve the performance of via our optimized SIMD methods.

Fig. 9 shows the *scalability* of the three *pDIAMOND* implementations over the two databases. We vary the number of nodes from 4 to 16 and use the performance with four nodes



(a) SRR096576_1 (8 nodes) (b) SRR096576_1 (16 nodes) (c) SRR077475_1 (8 nodes) (d) SRR077475_1 (16 nodes)

Fig. 6: Normalized execution time of pBWA with MPI I/O interfaces, ARC interfaces and HDFS pipelined write interfaces. Times are normalized to the total execution time of pBWA with ARC interfaces over SRR096576_1 and SRR077475_1 dataset respectively.



(a) SRR096576_1 (b) SRR077475_1

Fig. 7: Scalability of pBWA with ARC interfaces, compared to pBWA with MPI I/O interfaces and HDFS pipelined write interfaces for SRR096576_1 dataset and SRR077475_1 dataset.

as our reference baseline. In Fig. 9(a) for the *nr* database, *pDIAMOND* with ARC achieves up to 3.33× speedup over four nodes when running over the nr database while *pDIAMOND* with MPI I/O and *pDIAMOND* with HDFS pipelined write achieve up to 2.90× and 3.23× speedup on 16 nodes, respectively. In Fig. 9(b) for the *env_nr* database, we observe up to 2.13×, 1.63×, and 1.87× speedups for our *pDIAMOND* implementations with ARC, MPI I/O, and HDFS pipelined write, respectively, on 16 nodes versus the four-node baseline.

V. Related Work

There has been much research on taking advantage of distributed file systems for high-performance computing (HPC) applications, especially in the direction of enabling and optimizing "N-1 write." SDAFT [28] translates parallel I/O requests to distributed I/O requests for the HDFS system, which realizes the functionality of the N-1 write. SCALER [27] modifies the lock mechanism of HDFS NameNode to allow each worker to obtain the file write token, thereby supporting concurrent write on multiple blocks of a HDFS file. In particular, it can select multiple workers as aggregators to collect data with MPI and such data is written to the same data block of a HDFS file. PLFS [4], originally designed to optimize N-1 write on parallel file systems, has been extended on HDFS [8] to handle concurrent write checkpoint workloads. Our work differs by not only focusing on N-1 write, but also by delivering further performance optimizations, including optimized functionality, communication, and parallel disk I/O.

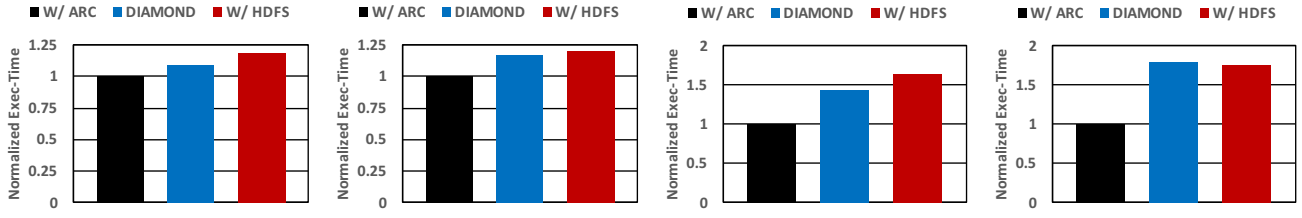
For parallel write on HDFS, Islam et al. [13] extend the socket-based HDFS in the Apache Hadoop and RDMA-based HDFS [14] to enable parallel replication with multiple output streams. They compare their parallel replication design with the pipelined replication design and claim that parallel replication can benefit HDFS for high-performance interconnects and protocols such as IPoIB and RDMA. Compared to their work, our proposed HDFS parallel write can deliver fast and efficient parallel I/O by using optimized MPI one-sided communication for data aggregation and flushing output data with the consideration of locality.

VI. Conclusions

In this paper, we present a feasibility study for running MPI applications over the commodity HDFS. Towards that end, we created ARC, a middleware prototype that aggregates and reorders intermediate data and coordinates computation and I/O. As a communication and I/O middleware layer, ARC enables MPI applications to efficiently execute on HDFS. End users can register the intermediate data with our user-friendly interfaces in a similar fashion to writing MPI programs. ARC then takes care of the data processing, i.e., using optimized merge and sort methods with MPI one-sided communication for data aggregation. With the selected coordinators, ARC can boost the performance by overlapping the computation, communication, and disk I/O. Moreover, we design an HDFS parallel write mechanism in ARC to take advantage of fast disk I/O. We demonstrate the efficacy of ARC and, in turn, the feasibility for running MPI applications over the commodity HDFS via two real-world applications: *pBWA* and *pDIAMOND*. These case studies demonstrate that ARC can achieve significant performance improvement over traditional MPI I/O and HDFS pipelined write.

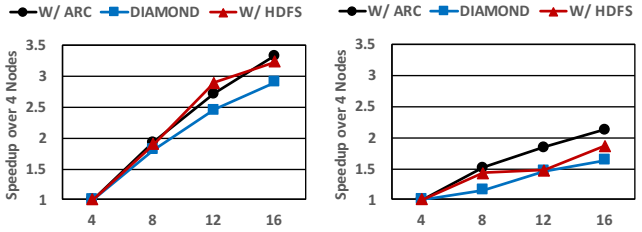
References

[1] Lustre: A scalable, high-performance file system. Whitepaper, 2003.
[2] D. Amodei, R. Anubhai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, J. Chen, M. Chrzanowski, A. Coates, G. Diamos, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *Proc. of the Int'l Conf. on Machine Learning*, 2016.
[3] Apache. Hadoop. http://hadoop.apache.org/.
[4] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. Plfs: a checkpoint filesystem for parallel applications. In *Proc. of the ACM Conf. on High Performance Computing Networking, Storage and Analysis*, 2009.
[5] B. Buchfink, C. Xie, and D. H. Huson. Fast and sensitive protein alignment using diamond. *Nat Meth*, 2015.
[6] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In *Proc. of the 30th Int'l Conf. on Machine Learning*, 2013.
[7] G. P. Consortium et al. A global reference for human genetic variation, 2015.



(a) nr (8 nodes) (b) nr (16 nodes) (c) env_nr (8 nodes) (d) env_nr (16 nodes)

Fig. 8: Normalized execution time of *(p)DIAMOND* with MPI two-sided communication and MPI I/O and *(p)DIAMOND* with HDFS pipelined write with respect to *(p)DIAMOND* with ARC. Times are normalized to the total execution time of *(p)DIAMOND with ARC* running over the nr and env_nr databases, respectively.



(a) nr database (b) env_nr database

Fig. 9: Scalability of *pDIAMOND* with ARC, compared to *pDIAMOND* with MPI two-sided communication and MPI I/O and *pDIAMOND* with MPI two-sided communication and HDFS pipelined write over the nr and env_nr databases.

[8] C. Cranor, M. Polte, and G. Gibson. Hpc computation on hadoop storage with plfs. *Parallel Data Laboratory at Carnegie Mellon University*, 2012.

[9] Y. Guo, W. Bland, P. Balaji, and X. Zhou. Fault tolerant mapreduce-mpi for hpc clusters. In *Proc. of the ACM Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2015.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[11] K. Hou, H. Wang, and W.-c. Feng. Aspas: A framework for automatic simdization of parallel sorting on x86-based many-core processors. In *Proc. of the ACM on Int'l Conf. on Supercomputing (ICS)*, 2015.

[12] J. Huang, X. Ouyang, J. Jose, M. Wasi-ur Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. K. Panda. High-performance design of hbase with rdma over infiniband. In *IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*, 2012.

[13] N. S. Islam, X. Lu, M. W. ur Rahman, and D. K. Panda. Can parallel replication benefit hadoop distributed file system for high performance interconnects? In *IEEE Symp. on High-Performance Interconnects*, 2013.

[14] N. S. Islam, M. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of hdfs over infiniband. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012.

[15] W. Jiang, V. T. Ravi, and G. Agrawal. A map-reduce system with an alternate api for multi-core environments. In *Proc. of the IEEE/ACM Int'l Conf. on Cluster, Cloud and Grid Computing*, 2010.

[16] H. Li and R. Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 2009.

[17] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proc. of the ACM Symp. on Cloud Computing*, 2014.

[18] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda. High-performance design of hadoop rpc with rdma over infiniband. In *IEEE Int'l Conf. on Parallel Processing*, 2013.

[19] D. Peters, X. Luo, K. Qiu, and P. Liang. Speeding up large-scale next generation sequencing data analysis with pbwa. *J. Appl Bioinform Comput Biol 1:1*, 2012.

[20] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *IEEE Int'l Symp. on High Performance Computer Arch. (HPCA)*, 2007.

[21] R. B. Ross, R. Thakur, et al. Pvfs: A parallel file system for linux clusters. In *Proc. of the Linux Showcase and Conf.*, 2000.

[22] F. B. Schmuck and R. L. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, 2002.

[23] M. L. Speir, A. S. Zweig, K. R. Rosenbloom, B. J. Raney, B. Paten, P. Nejad, B. T. Lee, K. Learned, D. Karolchik, A. S. Hinrichs, et al. The ucsc genome browser database: 2016 update. *Nucleic acids research*, 2016.

[24] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop Yarn: Yet Another Resource Negotiator. In *Proc. of the ACM Symp. on Cloud Computing*, 2013.

[25] E. H. Wilson, M. T. Kandemir, and G. Gibson. Will they blend?: Exploring big data computation atop traditional hpc nas storage. In *IEEE Int'l Conf. on Distributed Computing Systems (ICDCS)*, 2014.

[26] C. Xu, R. Goldstone, Z. Liu, H. Chen, B. Neitzel, and W. Yu. Exploiting analytics shipping with virtualized mapreduce on hpc backend storage servers. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 2016.

[27] X. Yang, Y. Yin, H. Jin, and X.-H. Sun. Scaler: Scalable Parallel File Write in HDFS. In *IEEE Int'l Conf. on Cluster Computing (CLUSTER)*, 2014.

[28] J. Yin, J. Zhang, J. Wang, and W.-c. Feng. SDAFT: A novel scalable data access framework for parallel BLAST. *Parallel Computing*, 2014.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of the USENIX conf. on Networked Systems Design and Impl. (NSDI)*, 2012.

[30] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, 2013.

[31] J. Zhang, S. Misra, H. Wang, and W.-c. Feng. mublastp: database-indexed protein sequence search on multicore cpus. *BMC Bioinformatics*, 2016.