TAPIOCA: An I/O Library for Optimized Topology-Aware Data Aggregation on Large-Scale Supercomputers

François Tessier, Venkatram Vishwanath
Argonne Leadership Computing Facility
Argonne National Laboratory
Lemont, IL, USA
Email: {ftessier,venkat}@anl.gov

Emmanuel Jeannot
Inria, LaBRI, Univ. Bordeaux, CNRS, Bordeaux-INP
Talence, France
Email: emmanuel.jeannot@inria.fr

Abstract—**Reading and writing data efficiently from storage system is necessary for most scientific simulations to achieve good performance at scale. Many software solutions have been developed to decrease the I/O bottleneck. One well-known strategy, in the context of collective I/O operations, is the two-phase I/O scheme. This strategy consists of selecting a subset of processes to aggregate contiguous pieces of data before performing reads/writes. In this paper, we present TAPIOCA, an MPI-based library implementing an efficient topology-aware two-phase I/O algorithm. We show how TAPIOCA can take advantage of double-buffering and one-sided communication to reduce as much as possible the idle time during data aggregation. We also introduce our cost model leading to a topology-aware aggregator placement optimizing the movements of data. We validate our approach at large scale on two leadership-class supercomputers: Mira (IBM BG/Q) and Theta (Cray XC40). We present the results obtained with TAPIOCA on a micro-benchmark and the I/O kernel of a large-scale simulation. On both architectures, we show a substantial improvement of I/O performance compared with the default MPI I/O implementation. On BG/Q+GPFS, for instance, our algorithm leads to a performance improvement by a factor of twelve while on the Cray XC40 system associated with a Lustre filesystem, we achieve an improvement of four.**

I. Introduction

In many high-performance computing (HPC) applications, the way data is moved, allocated or accessed plays a critical role for the performance. Until recently, computation was, most of the time, the main performance bottleneck. Today, however, application developers are facing new challenges where data management is becoming more and more critical. Indeed, numerical simulations are generating and accessing an increasing amount of data. One fundamental part of the data lifetime is the access to the storage input/output system.[1] This I/O requirement can be extremely important: hundreds of petabytes are commonly accessed during a simulation campaign. However, current I/O systems and middleware are facing several problems related to latency, contention, scalability, and the diversity of I/O patterns. Moreover, the ratio between computational capability and I/O bandwidth is increasing: for the past 20 years, the ratio FLOPS/IOPS has increased by a factor of 100 for the top supercomputer in the Top500 ranking over the years. Therefore, developing scalable I/O mechanisms is essential in order to fully exploit the platform capabilities.

Today, HPC systems feature complex interconnect topologies (e.g., 5D torus, dragonfly) to enable fast communication between nodes. Moreover, these architectures have a separate network for accessing the storage system in order to avoid contention and traffic interference and to enable functional decoupling. Therefore, performing I/O requires data movement along several diverse networks and for several hops. Hence, it is important not only to stage the data within the network but also to adapt the I/O pattern to the underlying architecture topology.

Two-phase I/O is a common strategy to reduce contention and latency onto the I/O system: data are aggregated on a subset of compute nodes (called aggregator) and then sent to the storage system. The two-phase I/O approach poses several issues. On which compute node should a given aggregator be allocated? How should one optimize communication such that the aggregation allows for performance improvement (e.g., how to fill intermediate buffer, pipeline communication, etc.)? Moreover, the issue of performance potability is important. We do not want to design an ad-hoc aggregation framework for each target system (with its specific network topology and storage architecture), but we want to provide a generic solution that works effectively across diverse systems.

In this paper, we present a library called TAPIOCA (Topology-Aware Parallel I/O: Collective Algorithm), an MPI-based library for performing two-phase I/O. It consists of an algorithm to compute the aggregator selection and placement, and takes into account the system topology, as well as the application I/O pattern, and enables pipelining of the communication through the aggregators to the storage. Performance portability is ensured by a model describing the different cost of aggregation and transfer to the storage system as well as a simple yet well-defined API. We evaluate our solution and demonstrate its scalability by experimenting with several ap-

[1]We will call this I/O in the remaining of this paper even if an application performs another kind of I/O: disk, network, etc.

plications and benchmarks on two high-end high-performance computing (HPC) systems - one based on an IBM Blue Gene/Q system with a GPFS file system and a second system based on an Intel/Cray KNL-based XC40 system with a Lustre file system. On both systems we compare our approach with the default MPI I/O implementation, and we show substantial improvement of I/O access without modifying the code from one system to another.

The outline of this paper is the following. In Section II, we present the context of our study. Related work is detailed in Section III. Our approach and the TAPIOCA library are described in Section IV. In Section V experimental results are presented. The conclusion and future work are discussed in Section VI.

II. Context and Motivation

We present in this section an overview of the current and upcoming storage systems on large-scale supercomputers. Then, we describe the two-phase I/O scheme implemented in most of the common MPI I/O implementations to optimize collective I/O operations.

A. Storage Systems at Scale

Over the past several years, the time spent in I/O for large-scale simulations has been growing, leading to the need for improving data movement. With this in mind, efforts have been made on network topologies to reduce the distance between data and storage and to isolate the storage system from compute nodes in order to avoid interference. On the IBM BG/Q, for example, a 5D-torus network offers a limited number of hops between compute nodes and storage while providing different routes to distribute the load. In addition, a node's partitioning in blocks of 512 nodes linked to four I/O nodes reduces as much as possible the impact of I/O interference between jobs and ensure a good performance reproducibility. On Cray XC40, a dragonfly network topology is deployed. Thus, the minimal distance from one node to another is at most three hops. On this platform, a subset of nodes, called LNET routers, plays the role of a proxy to the storage system. Another network is then dedicated to send data to disk.

A complementary approach consists in multiplying tiers of memory and storage as data staging areas with various sizes and performances. For example, SSD-based burst buffers as intermediate nodes between compute nodes and storage system can supply a smaller storage capacity but a higher I/O bandwidth. Certain Intel Knights Landing (KNL) nodes distributed by Cray with the XC40 architecture come with a local SSD and a high bandwidth memory bank (MCDRAM) that can be used as a cache or allocatable memory.

At the same time, parallel file systems have been improved to support an increasing I/O load in terms of both throughput and available storage capacity. This software stack is accompanied by strong algorithms to balance the I/O load.

Despite these upgrades, however, room remains for improvement in parallel I/O and more generally in data movements. In particular, the runtime systems offer an interesting lever affecting the way data is moved across a large-scale system and through the memory and storage hierarchy. TAPIOCA falls within this scope by proposing an efficient approach for collective I/O operations.

B. MPI I/O and the Two-Phase I/O Scheme

MPI [1] is commonly used to implement large-scale distributed-memory applications on high-performance clusters. MPI I/O is a critical component of MPI for performing I/O. The collective I/O mechanism in MPI I/O helps applications effectively read and write data at scale. In collective I/O, all MPI tasks involved in the communicator call the I/O routine, typically in a bulk synchronous manner. This type of operation allows the MPI runtime system to optimize data movement based on knowledge from the application including parameters such as data size, and the layout in both memory and storage.

The two-phase I/O algorithm is a well-known and efficient optimization available in MPI I/O implementations such as ROMIO [2]. It consists in selecting a subset of processes to aggregate contiguous pieces of data (aggregation phase) before writing them to the storage system (I/O phase). Figure 1 depicts this technique using an illustrative example involving four processes, two of them chosen as aggregators. Thus, the network contention decreases around the storage system while the I/O bandwidth is substantially increased by the write of large chunks of contiguous data. However, this approach as it has been implemented suffers several limitations. First, even if the I/O performance is better compared with that of an unoptimized operation, it usually remains far from the peak I/O bandwidth achievable. Second, we noticed an inefficient aggregator placement policy even though a smart aggregator's mapping may have a real impact on performance. Third, the common implementations fail to take advantage of the data model, the data layout, and the memory and system hierarchy.

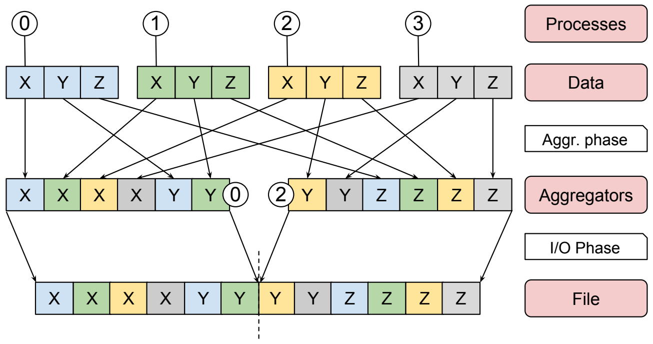

Fig. 1: Two-phase I/O mechanism

In this work, we focus on the two-phase I/O scheme and address several of these limitations. Specifically, we have developed an I/O library called TAPIOCA on top of MPI I/O implementing a topology-aware two-phase I/O scheme optimized for large-scale supercomputers. This library has been created following three key directions: an efficient implementation of the two-phase I/O algorithm, an improved aggregator placement taking into account system characteristics and a generic interface to query system topology information.

III. Related Work

Parallel I/O is an active research topic because of the increasing requirements of applications for data movement to memory or storage [3]. While I/O tuning is probably the first step to increase I/O bandwidth on new architectures [4], [5], [6], improvements at different layers of the I/O software stack are also necessary. From a file system perspective, GPFS [7] or Lustre [8] are examples of widespread highly scalable parallel file systems. At a library or application level, parallel I/O libraries such as MPI I/O, part of the MPI-2 [1] standard, on top of parallel file systems are commonly deployed. In these, collective I/O enables improved performance. For this, Chaarawi et al. [9] evaluate various collective I/O write algorithms. Other approaches to optimizing collective I/O have also been undertaken using techniques such as process placement based on the I/O pattern [10] or collective I/O autotuning with machine learning [11].

One of the de facto collective I/O algorithms is called *two-phase I/O* [12]. This method adds a level of hierarchy in collective I/O phases by aggregating data on a subset of processes before writing it onto the storage system (more details are given in Section II-B). ROMIO [2] is a popular implementation of MPI I/O using two-phase I/O; it is included in the widely used MPICH library [13]. A number of approaches have sought to improve this library and the two-phase I/O algorithm [14], [15], [16]. Approaches based on multithreading to overlap aggregation and I/O phases using double buffering also have been studied in [17], [18]. Nevertheless, the number of aggregators or the buffer size needed in collective I/O remains still an open topic [19]. The placement of aggregators is a well-known problem. Certain approaches focus on data locality and a polynomial time assignment algorithm (the Hungarian algorithm) to reduce the communication between compute nodes and aggregators [20]. Others have concentrated their efforts on the specific problem of sparse data patterns on the BG/Q, using an algorithm to take into account paths on the network topology [21]. A more general method designed to increase the I/O bandwidth of collective I/O for the previous version of IBM supercomputers BG/P has been proposed in [22]. In our previous work [23], we implemented an initial topology-aware data aggregation method for the BG/Q system.

Our current approach differs from all the above solutions by combining both an optimized buffering system and a topology-aware quantitative aggregator mapping strategy targeting any kind of architecture such as IBM BG/Q and Cray XC40 together with both GPFS and Lustre and being extensible to address new tiers of storage, while also accounting for the application's I/O pattern.

IV. Our Approach

TAPIOCA is an I/O library on top of MPI I/O implementing a topology-aware two-phase I/O scheme optimized for large-scale supercomputers. This library has been created following three key directions: an efficient implementation of the two-phase I/O algorithm, an improved aggregator placement taking into account system characteristics and a generic system topology interface allowing one to easily query a new architecture. In the rest of this paper, we describe how our implementation meets these three points and we present results obtained from our experiments on the Mira IBM Blue Gene/Q and the Theta Cray XC40 supercomputers at Argonne National Laboratory.

A. Implementation of the Two-Phase I/O Scheme

Compared with the MPI standard, our approach requires the description of the upcoming I/O operations before performing read or write calls. We extract from this information the data model (multidimensional arrays) and the data layout (array of structures, structure of arrays). The identification of these data patterns allows us to determine better I/O scheduling and to reduce the idle time for all the MPI tasks. Let us take as an example MPI processes writing three arrays in a file, each one describing a dimension of coordinates in (x,y,z), following an array of structures data layout. Algorithm 1 describes the collective MPI I/O calls needed for that operation. Each call to `MPI_File_write_at_all` is a collective operation independent of the next calls.

Algorithm 1: Collective MPI I/O writes.

```
$n \leftarrow 5$;
$x[n]$, $y[n]$, $z[n]$;
$offset \leftarrow rank \times 3 \times n$;

MPI_File_write_at_all ($f$, $offset$, $x$, $n$, $type$, $status$);
$offset \leftarrow offset + n$ ;

MPI_File_write_at_all ($f$, $offset$, $y$, $n$, $type$, $status$);
$offset \leftarrow offset + n$;

MPI_File_write_at_all ($f$, $offset$, $z$, $n$, $type$, $status$);
```

With TAPIOCA, users have to describe the upcoming writes. Algorithm 2 uses the previous example to show how to call TAPIOCA to write data in file. Since we have three variables to write, we declare arrays of size 3 describing the number of elements, the size of the data type, and the offset in file (*for* loop starting line 6). Then, we initialize TAPIOCA with this information. This phase allows our library to schedule the aggregation phase in order to completely fill an aggregator buffer before flushing it to the disk. Figure 2 gives another perspective of what happens when calling three independent MPI I/O collective writes and TAPIOCA. In our example, MPI I/O has to flush three almost empty buffers in file while TAPIOCA can aggregate all the data.

Additionally, we improved both the aggregation and I/O phases by overlapping them. With this aim in mind, we allocate two buffers per aggregator and supply them as a pipeline: while aggregating data in the first buffer, an aggregator can flush the second one to the storage system concurrently. This

Algorithm 2: Collective TAPIOCA writes.

```
1  $n \leftarrow 5$;
2  $x[n]$, $y[n]$, $z[n]$;
3  $offset \leftarrow rank \times 3 \times n$;
5
6  for $i \leftarrow 0,\ i < 3,\ i \leftarrow i+1$ do
7      $count[i] \leftarrow n$;
8      $type[i] \leftarrow$ sizeof ($type$);
9      $ofst[i] \leftarrow offset + i \times n$;
11
12 TAPIOCA_Init ($count$, $type$, $ofst$, 3);
14
15 TAPIOCA_Write ($f$, $offset$, $x$, $n$, $type$, $status$);
16 $offset \leftarrow offset + n$ ;
18
19 TAPIOCA_Write ($f$, $offset$, $y$, $n$, $type$, $status$);
20 $offset \leftarrow offset + n$;
22
23 TAPIOCA_Write ($f$, $offset$, $z$, $n$, $type$, $status$);
```

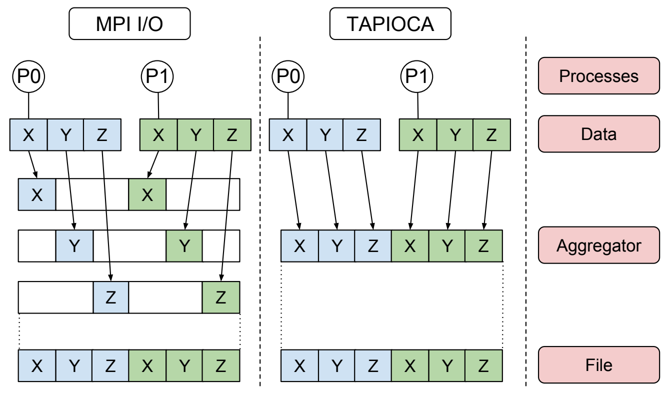

Fig. 2: Calling three independent MPI I/O collective writes and TAPIOCA.

can be done with one-sided MPI communication (Remote Memory Access) to aggregate data in the aggregators' buffers and, thanks to non-blocking MPI I/O functions, to effectively read or write buffers in file. Algorithm 3 details this part of our method. For each call to `TAPIOCA_Write`, we recover information computed during the initialization phase such as the round number, the target aggregator, the amount of data to write during this round and the aggregator buffer to put data in (lines 2 to 5). Then, the *while* loop starting from line 8 blocks the processes whose current round is different from the global round in a fence (barrier in the context of MPI one-sided communication). Only the processes with the matching round can lift the barrier. If a process passing this fence is an aggregator, it flushes the appropriate buffer into the file. Line 16 just puts the data into the target buffer. If the process has written all its data, it enters a portion of code similar to the one starting from line 8. Else, we recursively call this `TAPIOCA_Write` function again while updating the parameters.

Algorithm 3: Data aggregation with TAPIOCA.

```
1  Function TAPIOCA_Write ($f, offset, data, size, type, status$)
2      $round \leftarrow$ GetRound();
3      $aggr \leftarrow$ GetAggregatorRank();
4      $chunkSize \leftarrow$ GetRoundSize($round$);
5      $bufferId \leftarrow globalRound$ % 2;
7
8      while $round \neq globalRound$ do
9          Fence ();
10         if I am an aggregator then
11             iFlush_Buffer ($bufferId$);
12         $globalRound \leftarrow globalRound + 1$;
13         $bufferId \leftarrow globalRound$ % 2;
15
16     RMA_Put ($data$, $chunkSize$, $offset$, $aggr$,
        $bufferId$);
18
19     if $chunkSize = size$ then
20         while $globalRound \neq TotalRounds$ do
21             Fence ();
22             if I am an aggregator then
23                 iFlush_Buffer ($bufferId$);
24             $globalRound \leftarrow globalRound + 1$;
25             $bufferId \leftarrow globalRound$ % 2;
26     else
27         TAPIOCA_Write
            ($f, offset + chunkSize, data + roundSize, size - chunkSize, type, status$);
```

B. Topology-aware aggregators placement

The second main contribution of this work on data aggregation concerns the aggregators placement policy. The various implementations of the MPI-2 standard propose a couple of aggregators mapping strategies for two-phase I/O. For example, in MPICH [13] a strategy consists in selecting the bridge node (i.e. the node directly linked to the I/O node) as a first aggregator and the other aggregators following a rank order. This strategy takes into account neither the distance between the compute nodes and the storage system nor the amount of data exchanged. Moreover, the process mapping may severely impact the performance by selecting aggregators on neighboring nodes inevitably creating contention. Our strategy involves considering the topology of the architecture and the data access pattern in an objective function in order to determine a near-optimal aggregator placement optimizing data movements. For the rest of this paper, we call "partition" a subset of nodes hosting processes sharing a contiguous piece of data in file. The number of aggregators defines the partition size, each partition electing one aggregator among the processes.

Given, for each partition:

- V_C: The set of compute nodes performing aggregation in the partition;
- $A \in V_C$: An aggregator chosen among compute nodes;
- IO: The storage system (I/O node) of the partition;
- $\omega(u, v)$: The amount of data exchanged between nodes u and v;
- $d(u, v)$: The number of hops between nodes u and v;
- l: The interconnect latency;
- $B_{i \to j}$: The bandwidth from node i to node j.

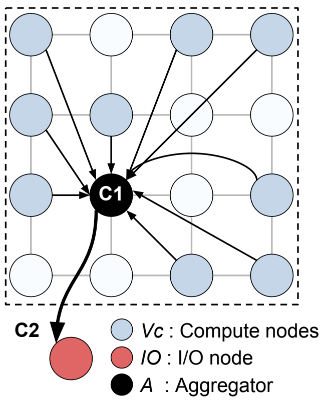

Fig. 3: Objective function minimizing the communication costs to and from an aggregator.

Figure 3 shows the two costs computed in our objective function for one partition. These two costs model the two phases of the algorithm.

The cost C_1 corresponds to the cost of aggregating data into aggregator buffers. Every process involved in the partition computes this cost in a distributed way as if it were chosen as the aggregator. From a candidate point of view, for each rank producing data, the aggregation cost is computed and summed up. Formally, each process A computes the cost C_1.

$$C_1 = \sum\nolimits_{i \in V_C, i \neq A} \left(l \times d(i, A) + \frac{\omega(i, A)}{B_{i \to A}} \right)$$

For the next step, the candidate computes the cost of sending the sum of aggregating data to the storage system (I/O node). For each process A, we define the cost C_2 as follows.

$$C_2 = l \times d(A, IO) + \frac{\omega(A, IO)}{B_{A \to IO}}$$

The objective function modeling our topology-aware aggregator placement strategy seeks to minimize the sum of the costs C_1 and C_2. Formally, each process in a partition computes this objective function.

$$TopoAware(A) = min\left(C_1 + C_2\right)$$

A call to `MPI_Allreduce` with the `MPI_MINLOC` enables our algorithm to choose as an aggregator the process with the minimal cost. Hence, for each partition an aggregator is elected. We note that on certain architectures, information about I/O nodes locality is missing. Theta, one of the two tested platforms in this paper, is such an architecture. In that case, the cost C_2 is set to 0.

C. Abstraction of the Topology

Proposing a generic approach able to target any architecture is a significant challenge. In our cost model, we based our method on variables easy to determine: bandwidth, latency, distance between nodes and gateways to the storage system. In order to tackle various topologies making our approach work on a diverse set of supercomputers, we developed a generic interface, using c++, to implement our data aggregation method for use on any system. Listing 1 presents the main function prototypes to implement to take advantage of aggregator placement on a high-performance architecture. Some of these values can be computed dynamically during the execution, while others, depending on the platform, need a one-time preliminary run of vendor tools to gather topology information. For example, on BG/Q+GPFS an hardware-specific MPI extension (MPIX library [24]) offers a set of functions providing information such as the distance to the I/O node (`MPIX_IO_distance`) while on Cray XC40 associated with a Lustre filesystem, more work is needed to gather the I/O nodes placement. Overall, the effort required to support a new architecture is quite low and is independent of the application.

Listing 1: Function prototypes for aggregators placement

```
int getBandwidth (int level);
int getLatency ();
int NetworkDimensions ();
void RankToCoordinates (int rank, int* coord);
int IONodesPerFile (char* filename, int *nodesList);
int DistanceToIONode (int rank, int IONode);
int DistanceBetweenRanks (int srcRank, int destRank);
```

V. Evaluation

We evaluated our approach with large-scale experiments on Mira and Theta, two leadership-class supercomputers at Argonne National Laboratory. We present our results in this section.

A. Targeted supercomputers

1) Mira: Mira is a 10 PetaFLOPS IBM BG/Q supercomputer ranked in the top ten of the Top500 ranking for years (see Figure 4). Mira contains 48K nodes interconnected with a 5D-torus high-speed network providing a theoretical bandwidth of 1.8 GBps per link. Each node hosts 16 hyperthreaded PowerPC A2 cores (1600 MHz) and 16 GB of main memory. Following the BG/Q architecture rules, Mira splits the nodes into *Psets*. A *Pset* is a subset of 128 nodes sharing the same I/O node. Two compute nodes of a *Pset* offer a 1.8 GBps link to the I/O node. These nodes are called the bridge nodes. GPFS [7] manages the 27 PB of storage. In terms of software, we compiled the test applications and our library with the IBM XL compiler, v12.1, and used the default MPI installation on Mira based on MPICH2 v1.5 (MPI-2 standard).

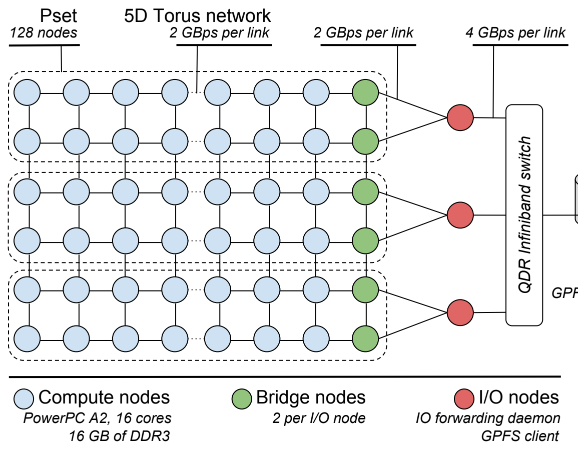

Fig. 4: IBM BG/Q architecture

2) Theta: Theta is a 10 PetaFLOPS Cray XC40 supercomputer. This architecture (see Figure 5) consists of more than 3600 nodes and 864 Aries routers interconnected with a dragonfly network. The routers are distributed in groups of 96 internally interconnected with 14 GBps electrical links, while 12.5 GBps optical links connect groups together. Each router hosts four Intel KNL 7250 nodes. A KNL node offers 68 1.60 GHz cores, 192 GB of main memory, a 128 GB SSD, and 16 GB of MCDRAM. The MCDRAM, also called high-bandwidth memory, can be used as an additional cache or as a high-speed allocatable memory (up to 400 GBps). On this platform, we compiled the test applications and TAPIOCA with the Cray wrapper invoking the Intel compiler (v17.0) optimized for this architecture. We used the default Cray MPI implementation based on MPICH and implementing the MPI-3 standard.

The storage system on Theta provides 9.2 PB of usable space managed by the Lustre file system [25], [8]. Figure 6 shows a simple example of Lustre on this supercomputer. Disks are hosted on OST (object storage target) and accessible through OSS (object storage server). Theta has 56 OST and OSS nodes (ratio 1:1). From an application point of view, each OSS is accessible through 7 LNET nodes allocated among the compute nodes. Unfortunately, the vendor does not currently provide a way to know how the data is distributed on LNET nodes. TAPIOCA offers an abstraction for Lustre that is not yet implementable. It explains why aggregators placement on this platform do not take the I/O phase into account for now.

B. Collective I/O Optimization with User-Defined Parameters

To achieve good performance on large-scale supercomputers with collective I/O operations, users often have to tune their environment to take advantage of certain optimizations. We listed the most common parameters that may have an impact on I/O performance and compared on both architectures a baseline I/O bandwidth with the default parameters and an optimized run with I/O tuning. This first study allows us to present a fair comparison between TAPIOCA and MPI I/O in the rest of this paper.

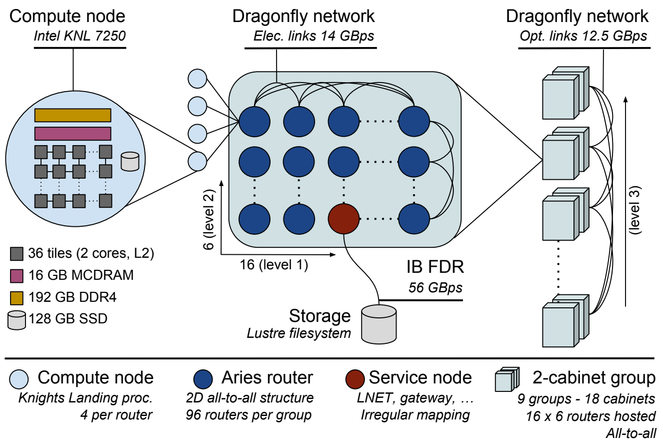

Fig. 5: Cray XC40 architecture.

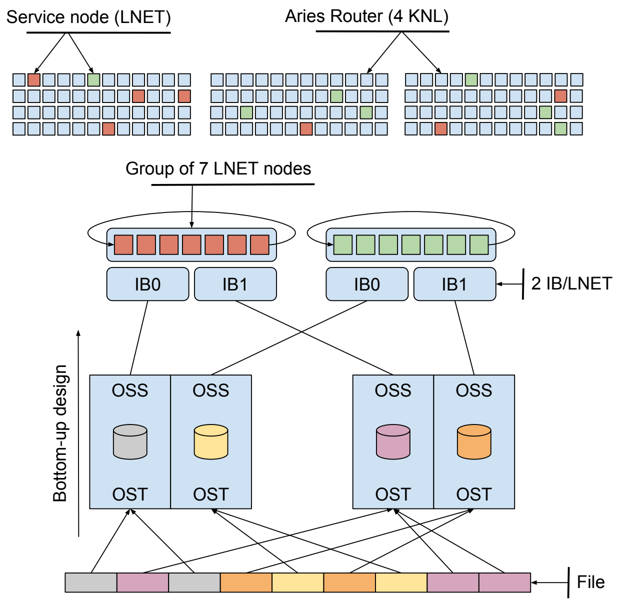

Fig. 6: Storage system on Theta managed by Lustre.

To evaluate I/O performance, we ran IOR, a popular I/O benchmark [26]. We varied the data size read and written per process from 200 KB to 4 MB. All the I/O calls were MPI I/O collective operations. A run was repeated 20 times, and the mean and the standard deviation were calculated. It has to be noted that we used a recommended subfiling technique on Mira (one file per *Pset*) for our experiments on this architecture.

On Mira (Figure 7), runs with the default parameters gave up to 7.3 GBps for read and around 2 GBps for write with a large variability. To increase this I/O bandwidth, we mainly set environment variables optimizing collective calls and reducing lock contention by sharing files locks. We note that the default number of aggregators and the aggregator buffer size set to their default values (i.e., 16 aggregators per *Pset* and 16 MB) offered the best performance. These settings were able to increase the read bandwidth by 13% on the best case, and the optimized write bandwidth outperformed three times the baseline case on 4 MB.

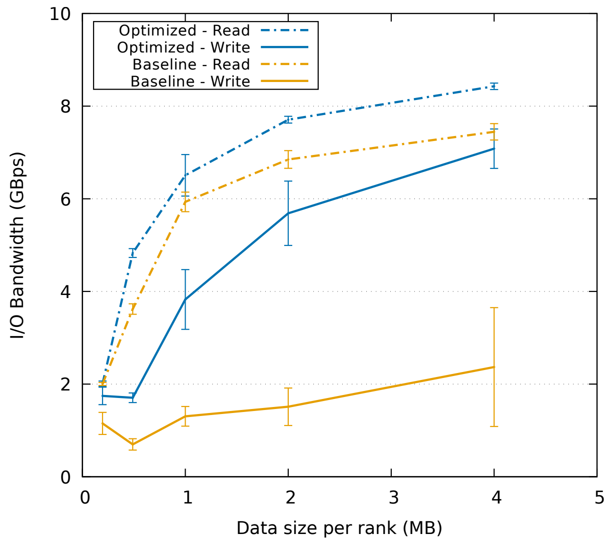

Fig. 7: I/O bandwidth achieved with IOR benchmark on 512 Mira nodes, 16 ranks per node, with and without user-defined optimizations.

Figure 8 depicts the same experiment on Theta. On this platform, IOR with the default parameters reveals a read bandwidth of approximately 800 MBps while up to 36 GBps can be reached with optimized parameters. The write bandwidth is increased from nearly 200 MBps to 10 GBps in the best case. The gap is substantial between these two scenarios. Indeed, by default on Theta, the number of OSTs (disks) is set to 1 and the stripe size (size of the chunks of data distributed among the OSTs) to 1 MB. Using 48 OSTs and a stripe size of 8 MB highly increases the I/O bandwidth. As on Mira, two locking modes are available. Lock sharing set for collective operations reduces the lock contention and takes part in the performance improvement. Another parameter is the number of aggregators per OST in MPI I/O. Our experiments showed that two aggregators per OST per set of 512 compute nodes (if 48 OSTs are used) gives a good increase. We also identified a routing algorithm (IN_ORDER) that provides better I/O bandwidth.

In general, writing data is much more expensive than reading it. For the rest of this paper, we focus our evaluation on primarily improving write bandwidth though our approach tackles both.

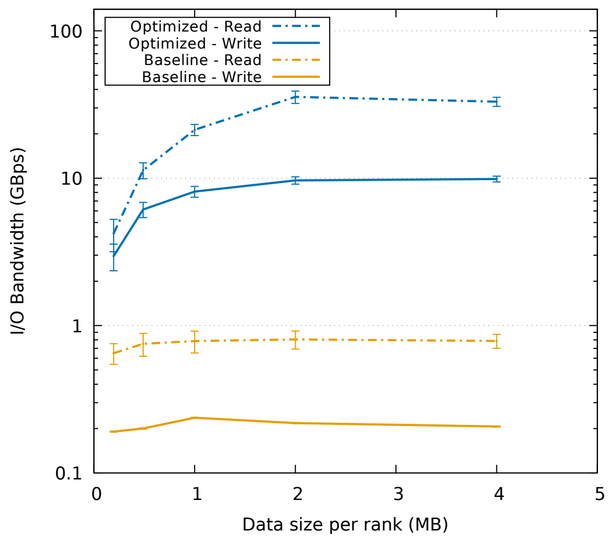

Fig. 8: I/O bandwidth achieved with IOR benchmark on 512 nodes on Theta, 16 ranks per node, with and without user-defined optimizations. Log scale on y-axis.

C. *Microbenchmark*

We first compared TAPIOCA with the MPI I/O implementations installed on the testbeds with a microbenchmark. In this code, every MPI process writes 1 MB as a contiguous piece of data in file during a collective call. For each experiment, we set the user-defined parameters to optimize the I/O bandwidth.

1) Mira: Mira has been studied for years leading to a well-optimized software stack. Particularly, MPICH2 v1.5, the default MPI implementation on Mira, has been improved to fit the BG/Q architecture and offer good performance at scale. Figure 9 shows the I/O bandwidth achieved with MPI I/O and TAPIOCA with our microbenchmark. We notice that both methods provide similar results. Since every process sends the same amount of data at the same time in one contiguous chunk, the benefit of a topology-aware aggregators placement is negligible as well as the advantage of the I/O scheduling computed in TAPIOCA.

2) Theta: Figure 10 depicts the I/O bandwidth obtained on 512 Theta-nodes with our micro-benchmark. TAPIOCA outperforms the default MPI I/O implementation (Cray MPI) for all message sizes. On the larger case (3.6 MB per rank), the write bandwidth we can achieve with TAPIOCA is two times higher than with MPI I/O. This performance improvement can be attributed to the topology-aware placement of aggregators, as well as as the use of pipelining of aggregation and I/O phases. These results point out a good portability of the I/O performance with TAPIOCA regardless of the architecture or the weakness of this or that MPI implementation.

This microbenchmark also allowed us to highlight a decisive correlation between aggregator buffer size set in TAPIOCA and stripe size of the Lustre file system. Table I shows the average I/O bandwidth achieved on 512 nodes and 16 ranks per node with various aggregator buffer sizes and stripe sizes.

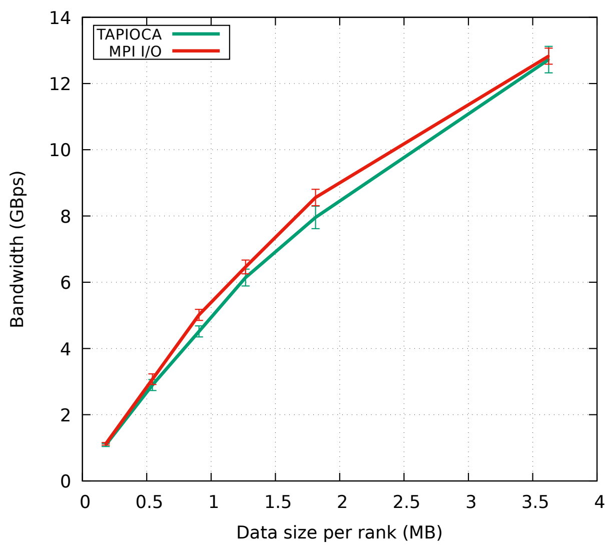

Fig. 9: I/O bandwidth achieved with a microbenchmark tested on 1,024 Mira nodes, 16 ranks per node. 32 aggregators per *Pset* set for TAPIOCA associated with a 32 MB aggregation buffer size. Average and standard deviation from 10 runs.

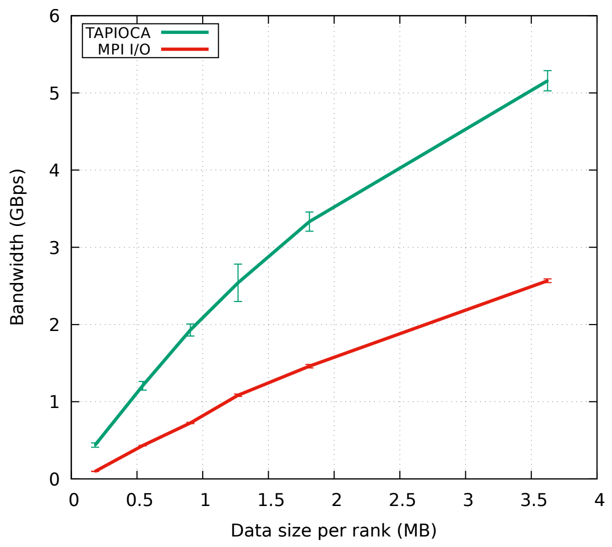

Fig. 10: I/O bandwidth achieved with a microbenchmark tested on 512 Theta nodes, 16 ranks per node. 48 aggregators set for TAPIOCA associated with a 8 MB aggregation buffer size. Stripe size: 8MB. Average and standard deviation from 10 runs.

Specifically, we set the buffer size in TAPIOCA to 4 MB, 8 MB, and 16 MB. For each case, we changed the stripe size in such a way that we can maintain a certain ratio. We observe that a 1:1 ratio—that is, an aggregator buffer size equal to the stripe size—gives the best performance. We took this analysis into account for the following experiments.

TABLE I: Ratio "Aggregator buffer size : Stripe size"

Ratio	1 : 8	1 : 4	1 : 2	1 : 1	2 : 1	4 : 1
I/O Bw (GBps)	0.36	0.64	0.91	1.57	1.08	1.14

D. HACC-IO

HACC-IO is the I/O kernel of HACC (Hardware Accelerated Cosmology Code). This large-scale cosmological application requires the massive compute power of supercomputers to simulate the mass evolution of the universe with particle-mesh techniques. In terms of I/O, every process of a HACC simulation manages a number of particles. Each particle is defined by nine variables—XX, YY, ZZ, VX, VY, VZ, phi, pid, and $mask$—corresponding to the coordinates, the velocity vector, and relevant physics properties. The size of a particle is 38 bytes. A useful base value of 25,000 particles requires approximately 1 MB. In the following, we first present our results on Mira with 1,024 and 4,096 nodes and 16 ranks per node (resp. 16K and 64K processes), then on Theta with 1,024 and 2,048 nodes and 16 ranks per node (resp. 16K and 32K processes). We compare our approach to MPI I/O with two data layouts: array of structures (AoS) and structure of arrays (SoA). For these experiments, we vary the data size per rank from 5K to 100K particles.

1) Mira: Figure 11 shows the results on 1,024 Mira nodes, with 16 ranks per node and one file per *Pset* as output. By varying the number of particles managed by each rank, we increase the data size on the x-axis. We note that subfiling is an efficient technique to improve I/O performance on the BG/Q since up to 90% of the peak I/O bandwidth is achieved by our topology-aware strategy. We also note that we outperform the default implementation even on large messages.

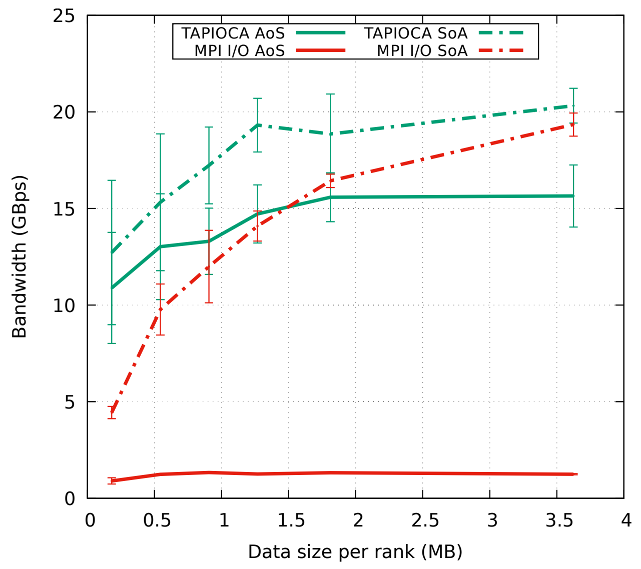

Fig. 11: I/O bandwidth achieved on Mira by writing one file per *Pset* from 1,024 nodes (16 ranks/node). TAPIOCA: 16 aggregators per *Pset*, 16 MB aggregator buffer size.

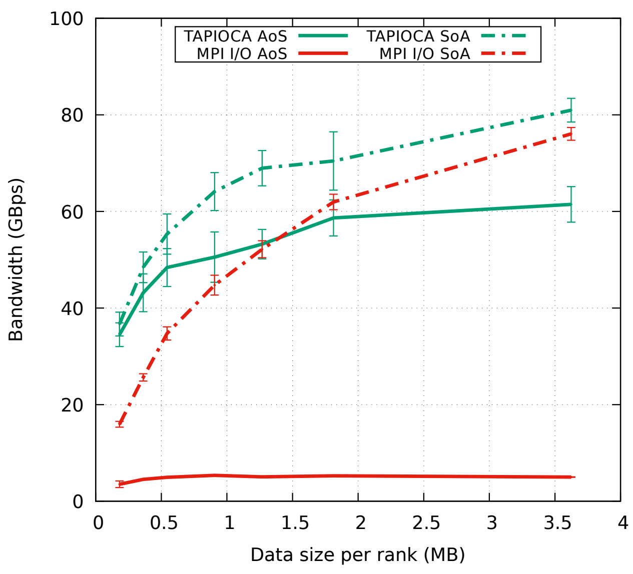

Fig. 12: I/O bandwidth achieved on Mira by writing one file per *Pset* from 4,096 nodes (16 ranks/node). TAPIOCA: 16 aggregators per *Pset*, 16 MB aggregator buffer size.

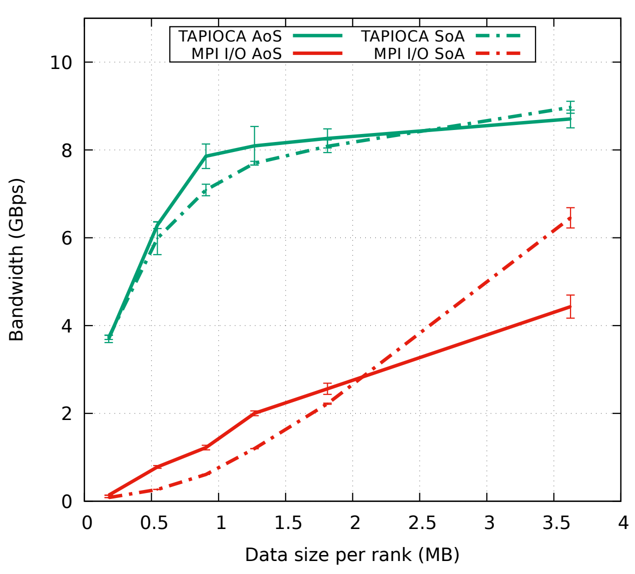

Fig. 13: I/O bandwidth achieved on Theta from 1,024 nodes (16 ranks/node). Lustre: 48 OSTs, 16 MB stripe size. TAPIOCA: 192 aggregators, 16 MB aggregator buffer size.

Figure 12 presents our experiments in the same configuration as the previous ones except that we ran it on 4,096 Mira nodes. The behavior is similar, with the peak I/O bandwidth almost reached (the peak is estimated to 89.6 GBps on this node count). As with experiments on 1,024 nodes, the gap with MPI I/O decreases as the data size increases. In any case, the I/O performance is substantially improved for both AoS and SoA layouts.

2) Theta: Our experiments on Theta show a good I/O performance gain as well. Figure 13 depicts the I/O bandwidth obtained with HACC-IO on 1,024 nodes with 16 ranks per node. The best results for both MPI I/O and TAPIOCA were obtained with 4 aggregators per OST (192 aggregators in total). The stripe size and stripe count were identical in both cases to make a fair comparison. However, TAPIOCA greatly surpasses the default MPI I/O implementation on this platform regardless of the data layout. With approximately 1 MB produced per rank, our approach is around 7 times faster. This difference tends to decrease when the data size is increased.

On 2,048 nodes (Figure 14), we observed similar results. The stripe size and stripe count were identical in both cases to make a fair comparison. To improve I/O performance, we set the number of aggregators to 8 per OST for both methods (384 in total). Again, a significant gap exists between TAPIOCA and the tested MPI I/O implementation. Even on the largest case (3.6 MB) and an array of structures data layout, our method is 4 times faster than MPI I/O.

VI. Conclusions and Future Work

In this paper, we have shown a technique to improve I/O performance at scale. In particular,, we have presented TAPIOCA, an I/O library on top of MPI I/O, implementing a topology-aware optimized version of the two-phase I/O scheme. We designed TAPIOCA with two pipelined aggregators buffers filled thanks to MPI one-sided communication and flushed with non-blocking calls to overlap aggregation and I/O phases. We also benefited from the architecture's characteristics to place aggregators and tune the algorithm (number of aggregators, buffer size, cost model). Thus, on a simple microbenchmark, we showed at least the same I/O performance as MPI I/O on a well-known platform and a good

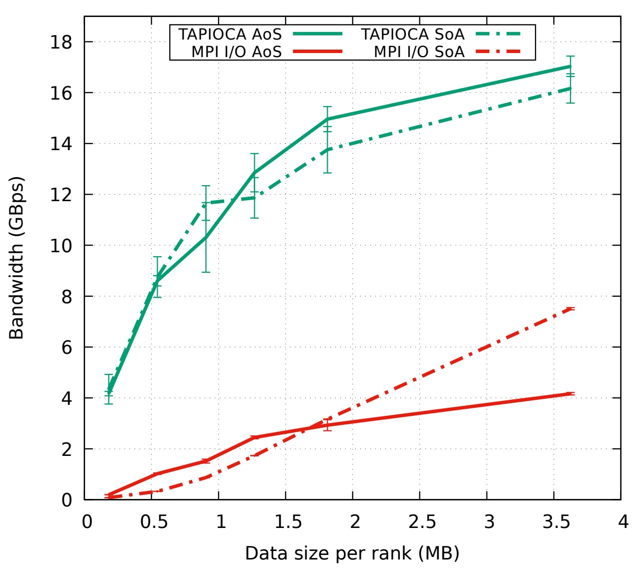

Fig. 14: I/O bandwidth achieved on Theta from 2048 nodes (16 ranks/node). Lustre: 48 OSTs, 16 MB stripe size. TAPIOCA: 384 aggregators, 16 MB aggregator buffer size.

improvement on a more recent architecture. On the I/O kernel of a large-scale cosmological simulation, TAPIOCA was up to 12 times faster than the MPI I/O implementation on the BG/Q with a specific data layout. On the Cray XC40 supercomputer, we also highly outperformed MPI I/O with the same code showing an excellent performance portability. Our large-scale experiments on two different architectures also highlighted the scalability of our algorithm as well as our generic approach.

We now plan to extend this library to one-to-many data movements from one level of memory hierarchy to another. For instance, one possibility is a method that efficiently aggregates data from the DRAM on the MCDRAM on KNL in order to move it to burst buffers in an optimized manner. Another research track is to develop new aggregators placement strategies taking into account more characteristics (routes, etc.) and data layouts (meshes, 2D and 3D arrays, etc.) A more short-term goal involves including our approach in widely used I/O libraries such as MPI I/O or HDF5.

Acknowledgment

This research has been funded in part and used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract no. DE-AC02-06CH11357. This work was supported in part by the U. Department of Energy, Office of Science, ASCR, under award numbers 57L32, 57L11, 57K50, and 5080500. This research is partially supported by the NCSA-Inria-ANL-BSC-JSC-Riken Joint-Laboratory on Extreme Scale Computing (JLESC).

References

[1] M. P. I. Forum, “MPI-2: Extensions to the Message-Passing Interface,” July 1997, http://www.mpi-forum.org/docs/docs.html.

[2] R. Thakur, W. Gropp, and E. Lusk, “A case for using MPIs derived datatypes to improve I/O performance,” in *Proceedings of SC98: High Performance Networking and Computing*. ACM Press, November 1998. [Online]. Available: http://www.mcs.anl.gov/ thakur/dtype/

[3] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, “A multiplatform study of I/O behavior on petascale supercomputers,” in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC ’15. New York, NY, USA: ACM, 2015, pp. 33–44. [Online]. Available: http://doi.acm.org/10.1145/2749246.2749269

[4] S. Byna, R. Sisneros, K. Chadalavada, and Q. Koziol, “Tuning parallel I/O on Blue Waters for writing 10 trillion particles,” in *”Cray User Group (CUG) meeting”*, Apr 2015. [Online]. Available: https://sdm.lbl.gov/ sbyna/research/papers/201504-CUG-VPICBW.pdf

[5] M. S. Breitenfeld, K. Chadalavada, R. Sisneros, S. Byna, Q. Koziol, N. Fortner, Prabhat, and V. Vishwanath, “Recent progress in tuning performance of large-scale I/O with parallel HDF5,” 11 2014. [Online]. Available: http://www.pdsw.org/pdsw14/wips/breitenfeld-wip-pdsw14.pdf

[6] H. Bui, H. Finkel, V. Vishwanath, S. Habib, K. Heitmann, J. Leigh, M. Papka, and K. Harms, “Scalable parallel I/O on a Blue Gene/Q supercomputer using compression, topology-aware data aggregation, and subfiling,” in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb 2014, pp. 107–111.

[7] F. Schmuck and R. Haskin, “GPFS: A shared-disk file system for large computing clusters,” in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST ’02. Berkeley, CA, USA: USENIX Association, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083323.1083349

[8] “Lustre filesystem website,” http://lustre.org/.

[9] M. Chaarawi, S. Chandok, and E. Gabriel, *Performance Evaluation of Collective Write Algorithms in MPI I/O*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 185–194.

[10] V. Venkatesan, R. Anand, J. Subhlok, and E. Gabriel, “Optimized process placement for collective I/O operations,” in *Proceedings of the 20th European MPI Users’ Group Meeting*, ser. EuroMPI ’13. New York, NY, USA: ACM, 2013, pp. 31–36. [Online]. Available: http://doi.acm.org/10.1145/2488551.2488567

[11] F. Isaila, P. Balaprakash, S. M. Wild, D. Kimpe, R. Latham, R. Ross, and P. D. Hovland, “Collective I/O tuning using analytical and machine-learning models,” in *IEEE Cluster 2015*, IEEE. Chicago, IL: IEEE, 09/2015 2015.

[12] J. M. del Rosario, R. Bordawekar, and A. Choudhary, “Improved parallel I/O via a two-phase run-time access strategy,” *SIGARCH Comput. Archit. News*, vol. 21, no. 5, pp. 31–38, Dec. 1993. [Online]. Available: http://doi.acm.org/10.1145/165660.165667

[13] W. Gropp, “MPICH2: A new start for MPI implementations,” in *Proceedings of the 9th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK, UK: Springer-Verlag, 2002, pp. 7–. [Online]. Available: http://dl.acm.org/citation.cfm?id=648139.749473

[14] R. Thakur, W. Gropp, and E. Lusk, “Data sieving and collective I/O in ROMIO,” in *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 182–. [Online]. Available: http://dl.acm.org/citation.cfm?id=795668.796733

[15] ——, “On implementing MPI-IO portably and with high performance,” in *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, ser. IOPADS ’99. New York, NY, USA: ACM, 1999, pp. 23–32. [Online]. Available: http://doi.acm.org/10.1145/301816.301826

[16] ——, “Optimizing noncontiguous accesses in MPI I/O,” *Parallel Comput.*, vol. 28, no. 1, pp. 83–105, Jan. 2002. [Online]. Available: http://dx.doi.org/10.1016/S0167-8191(01)00129-6

[17] Y. Tsujita, H. Muguruma, K. Yoshinaga, A. Hori, M. Namiki, and Y. Ishikawa, “Improving collective I/O performance using pipelined two-phase I/O,” in *Proceedings of the 2012 Symposium on High Performance Computing*, ser. HPC ’12. San Diego, CA, USA: Society for Computer Simulation International, 2012, pp. 7:1–7:8. [Online]. Available: http://dl.acm.org/citation.cfm?id=2338816.2338823

[18] Y. Tsujita, K. Yoshinaga, A. Hori, M. Sato, M. Namiki, and Y. Ishikawa, “Multithreaded two-phase I/O: Improving collective MPI-IO performance on a Lustre file system,” in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb 2014, pp. 232–235.

[19] M. Chaarawi and E. Gabriel, “Automatically selecting the number of aggregators for collective I/O operations,” in *2011 IEEE International Conference on Cluster Computing (CLUSTER), Austin, TX, USA, September 26-30, 2011*, 2011, pp. 428–437. [Online]. Available: http://dx.doi.org/10.1109/CLUSTER.2011.79

[20] R. Filgueira, D. E. Singh, J. C. Pichel, F. Isaila, and J. Carretero, “Data locality aware strategy for two-phase collective I/O,” in *High Performance Computing for Computational Science - VECPAR 2008, 8th International Conference, Toulouse, France, June 24-27, 2008. Revised Selected Papers*, 2008, pp. 137–149. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92859-1_14

[21] H. Bui, J. Leigh, E.-S. Jung, V. Vishwanath, and M. E. Papka, “Improving data movement performance for sparse data patterns on the Blue Gene/Q supercomputer.” in *ICPP Workshops*. IEEE Computer Society, 2014, pp. 302–311. [Online]. Available: http://dblp.uni-trier.de/db/conf/icppw/icppw2014.html#BuiLJVP14

[22] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, “Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’11. New York, NY, USA: ACM, 2011, pp. 19:1–19:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063409

[23] F. Tessier, P. Malakar, V. Vishwanath, E. Jeannot, and F. Isaila, “Topology-aware data aggregation for intensive I/O on large-scale supercomputers,” in *Proceedings of the First Workshop on Optimization of Communication in HPC*, ser. COM-HPC ’16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 73–81. [Online]. Available: https://doi.org/10.1109/COM-HPC.2016.13

[24] M. Gilge *et al.*, *IBM system blue gene solution - blue gene/Q application development*. IBM Redbooks, 2014.

[25] P. Schwan, “Lustre: Building a file system for 1,000-node clusters,” in *PROCEEDINGS OF THE LINUX SYMPOSIUM*, 2003, p. 9.

[26] “IOR: Parallel filesystem I/O benchmark,” https://github.com/LLNL/ior.