Topology-Aware Data Aggregation for Intensive I/O on Large-Scale Supercomputers

François Tessier, Preeti Malakar, Venkatram Vishwanath, Emmanuel Jeannot, Florin Isaila

Topology-Aware Data Aggregation for Intensive I/O on Large-Scale Supercomputers

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

Emmanuel Jeannot
Inria Bordeaux Sud-Ouest
Talence, France
Email: emmanuel.jeannot@inria.fr

Florin Isaila
University Carlos III
Madrid, Spain
Email: florin@arcos.inf.uc3m.es

Abstract

Reading and writing data efficiently from storage systems is critical for high performance data-centric applications. These I/O systems are being increasingly characterized by complex topologies and deeper memory hierarchies. Effective parallel I/O solutions are needed to scale applications on current and future supercomputers. Data aggregation is an efficient approach consisting of electing some processes in charge of aggregating data from a set of neighbors and writing the aggregated data into storage. Thus, the bandwidth use can be optimized while the contention is reduced. In this work, we take into account the network topology for mapping aggregators and we propose an optimized buffering system in order to reduce the aggregation cost. We validate our approach using micro-benchmarks and the I/O kernel of a large-scale cosmology simulation. We show improvements up to 15× faster for I/O operations compared to a standard implementation of MPI I/O.

1 Introduction

Optimizing data movement is critical for improved performance in high performance computing (HPC). We are witnessing the computational capability of HPC systems growing rapidly and exascale is now within reach. These systems are enabling large-scale simulations with higher fidelity and resolutions, among others, to model more complex phenomena. These simulations are generating and accessing increasing amounts of data. Often, it is more costly to access, move or allocate data than to actually process data. During the data lifetime, efficient access to the storage Input/Output system[1] is becoming increasingly critical. The I/O requirements can be extremely important (as depicted by simulations estimates given in Table 1), however, the current I/O middleware and

[1]Called I/O for the remaining of the paper. Even if an application performs other kind of I/O (to local disk or the network), here we consider only I/O to the storage/parallel file system

system face several challenges with respect to scalability, contention, latency and diverse application patterns. Also, given the limited scaling of I/O bandwidth in comparison to that of the computational capability of HPC systems and the current expectation that this will be even dire on future HPC systems, scalable I/O mechanisms that fully exploit the platform characteristics will be critical.

Thus, to improve the overall efficiency of a high-end parallel system, novel solutions to cope with efficient and optimized data access to the I/O system are needed.

Table 1: I/O requirements of diverse large-scale applications

Scientific domain	**Simulation**	**Data size**
Cosmology	Q Continuum	**2 PB / simulation**
High-Energy Physics	Higgs Boson	**10 PB / year**
Climate / Weather	Hurricane	**240 TB / simulation**

The current large-scale computing infrastructures are often characterized by network interconnects with complex topologies (e.g., multidimensional tori, dragonfly). Additionally, these systems are architected to have a separation of computation and I/O networks to avoid I/O interference and for functional decoupling. In these systems, I/O accesses require data movements along several hops of various networks. Hence, optimizing the data movement requires not only staging the data within these networks, but also to adapt I/O access pattern of the applications to the characteristics of the filesystems and the system topology.

To reduce latency of access and contention to the I/O system while improving its scalability, a common strategy (called two-phase I/O) is to aggregate data to a set of compute nodes (called aggregators) and have only the aggregators communicate with the I/O system. This approach poses several challenges such as: where to map aggregators among the various compute nodes, or, how to optimize communications to and from these aggregators? In this work, we explore the two-phase I/O (and specifically the write access) by carefully placing aggregators taking into account the application's communication needs (I/O access patterns), the topology of the underlying interconnect, and effective pipelining of communications to the aggregators and to the storage. The goal is to balance the aggregation phase cost with the I/O phase cost so as to minimize the overall time the application spends in writing the data to the storage system.

The main contribution of this paper is a novel approach to optimizing I/O data aggregation on large scale HPC infrastructures. First, we present a novel aggregator placement optimization framework and this framework is used to evaluate various approaches for data movement, including our topology-aware method. Next, we discuss a holistic end-to-end approach for I/O that goes beyond aggregator placement to also include pipelined aggregation buffering, file system awareness, and efficient inter-node communication (one-sided) for both the aggregation and I/O. Finally, we evaluate our approaches at scale on supercomputers and demonstrate that our approach significantly outperforms state-of-the-art techniques and represents a promising approach for scalable I/O on HPC systems.

2 Context and Motivation

In this section we start by describing the I/O subsystems of current and expected large-scale supercomputers. Next, we discuss the two-phase I/O algorithm used in MPI. Finally, we highlight the limitations of the current two-phase approach.

2.1 Storage systems on large-scale supercomputers

The current high-performance system architectures have undergone several improvements in order to tackle the I/O challenges. The networks topologies, despite being more complex, tend to reduce the distance between the data and the storage. While the compute nodes and the I/O infrastructures are commonly partitioned to avoid I/O interference, the interconnect networks bring these two entities closer. This partitioning is a characteristic of the IBM BG/Q supercomputers where the I/O nodes are dedicated to the I/O tasks and separated from the 5D-torus topology [6]. Similarly, Cray has also chosen a similar strategy for its supercomputers wherein the system has a subset of nodes called LNET nodes to manage I/O. To optimize the I/O bandwidth, a dragonfly network has been implemented reducing the number of hops from a compute node to the LNET node.

However, the amount of data produced by the applications remains extremely high and this sole architectural solution is not sufficient. Writing data out for future analysis suffers an I/O bandwidth limitation whereas storing data in memory for *in situ* analysis is bounded by the amount of available memory. Similar to solutions to overcome the memory bottleneck by adding more levels of hierarchy with faster but smaller memory close to the computing units, an approach to improve I/O performance is to create new tiers of storage between the main memory of the compute nodes and the storage system. Some supercomputers made the choice of allocatable DRAM by embedding Intel Knights Landing processors. Others chose to add NVRAM (on-node SSD for instance) to have a trade-off between cost, bandwidth and capacity. Burst buffer nodes as the ones used on the Cray Cori infrastructure [2] are also a method to achieve high I/O performance. In this case, nodes similar to I/O nodes contain SSDs for data staging.

2.2 MPI Two-phase I/O

The MPI-2 standard [9] introduced the notion of *two-phase I/O* [7] for collective I/O operations. The goal of this improvement is to optimize the I/O performance by reducing the network contention, increasing the I/O bandwidth and simplifying the data access pattern. In two-phase I/O, a subset of processes called *aggregators* is responsible for the I/O phase. These aggregators are elected during the MPI collective I/O calls. Each aggregator manages a chunk of contiguous data in file from a subset of processes. For read access, the aggregators load a part of the file and distribute smaller chunks of data to a subset of processes. For write access, an aggregator gathers data from a subset of processes in a contiguous way and writes the aggregated data to the file system (through an I/O node if necessary). A toy example of the two-phase I/O mechanism is shown in Figure 1. In this example, four processes need to write non-contiguous pieces of data to a shared file. During the aggregation phase, two processes elected as aggregators gather these pieces of data into contiguous chunks in memory (X then Y then Z). Once this phase is finished, the aggregators effectively write the data in file (I/O phase).

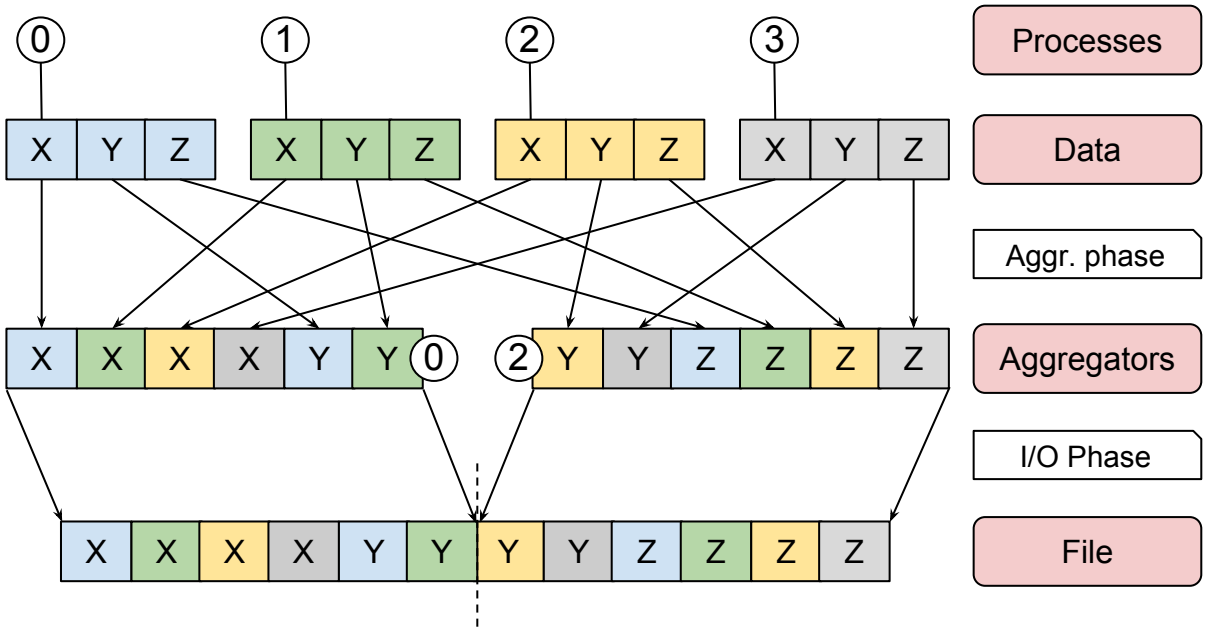

Figure 1: Two-phase I/O mechanism

2.3 Limitations

The two-phase I/O technique has several limitations. First, this two-phase approach works well for large messages but performs poorly with processes writing small data pieces (e.g., less than 1 MB). Secondly, some MPI I/O implementations of two-phase I/O take into account the topology of the machine to partition and elect aggregators (e.g. ROMIO considers the BG/Q topology to partition the number of elected aggregators). However, they do not use the topology information to propose an efficient aggregator placement policy. Moreover, beyond the characteristics of the underlying topology, the *a priori* details of the application's I/O patterns also could help to compute the placement, however, this is not currently considered. This is critical for emerging application I/O patterns in multi-physics applications, analysis outputs, etc. Therefore, in this paper we address the problem of mapping aggregators taking into account both the topology and the I/O access pattern of the application while optimizing performance for short messages.

3 Related Work

Parallel I/O is an active research topic due to the increasing requirements of applications for data movement to memory or storage. From a filesystem perspective, GPFS [12] or Lustre [1] 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 [9] standard, on top of parallel filesystems is commonly deployed. In these, collective I/O allows to achieve improved performance. For this, Chaarawi et al. [4] 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 [19] or collective I/O autotuning with machine learning [11]. One of the de facto

collective I/O algorithm is called *two-phase I/O* [7]. 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 2.2). ROMIO [13] is a popular implementation of MPI I/O using two-phase I/O included in the widely-used MPICH library [10]. There have been a number of approaches to improve this library and the two-phase I/O algorithm [14, 15, 16]. Approaches based on multi-threading to overlap aggregation and I/O phases using double buffering have been studied in [18, 17]. The number of aggregators or the buffer size needed in collective I/O remains still an open topic [5]. Finally, 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 [8]. Others concentrated their efforts on the specific problem of sparse data patterns on BG/Q by offering an algorithm to take paths on the network topology into account [3]. 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 [20]. Our approach differs from the above solutions by attempting to combine both an optimized buffering system and a topology-aware quantitative aggregators mapping strategy targeting any kind of architecture and being extensible to address new tiers of storage. It does so while also accounting for the application's I/O pattern.

4 Our Approach

Our approach consists of optimizing the data aggregation by taking into account critical parameters including the topology of the underlying architecture, the filesystem block size, and the double buffering with pipelined data movement achieved via one-sided communications. Thus, we developed a new data movement optimization library implemented in C++. This provides two simple functions to the user of parallel I/O while hiding all complexities of the underlying system. In fact, from the application developer point of view, using this aggregation mechanism comes down to describing the upcoming I/O operations (data sizes and offset in file) through an initialization function and commit the data instead of directly calling `MPI_File_write` in the application. This section first describes the parameters we identified to optimize the two-phase I/O method, and, next, discusses implementation details. Finally, we describe the challenges we address for scalable performance.

4.1 Aggregator Placement

The various implementations of the MPI-2 standard propose a couple of algorithms for two-phase I/O and particularly for aggregators mapping. In MPICH a strategy consists in choosing, for n aggregators, the first one on the bridge node (the compute node connected to the I/O node) and select the $n-1$ remaining ones on different nodes based only on their rank. For instance, let's consider 4 aggregators. The first one will be assigned on the bridge node. For the next three, the ranks of all processes are sorted and the aggregators are picked in this order such that no two aggregators are on the same node. Depending on the process mapping strategy, this placement can easily choose aggregators on neighboring nodes and thus create contention, or these could be selected far from the storage system leading to a large additional latency cost. Our strategy consists in electing the aggregators whose number is based on a fixed ratio (i.e. n aggregators per k nodes) in order to find a compromise between the cost of aggregating data and the cost of sending data

to the storage system (we can extend this to other types of memory: burst buffers, NVRAM, ...). To achieve this, we create a level of partitioning where a partition, which is subset of processes, contains processes that write a contiguous piece of data. Inside each partition, a process is elected as aggregator according to the topology and the amount of data to manage. Thus, the number of partitions corresponds to the number of aggregators. It has to be noted that a process can be involved in more than one partition. In this case we consider only the amount of data related to a given partition.

Figure 2 shows a simple example of partitioning and aggregator election on a grid according to our approach. In this figure, two more strategies are depicted. Indeed, to evaluate our topology-aware placement, we implemented three other methods. These simpler strategies can be described as follow:

- Shortest path: a rank hosted on the node with the smallest distance to the storage system is elected as an aggregator;
- Longest path: same as the previous strategy except that the longest distance to the storage system is considered.
- Greedy: the lowest rank in partition is the aggregator is selected;

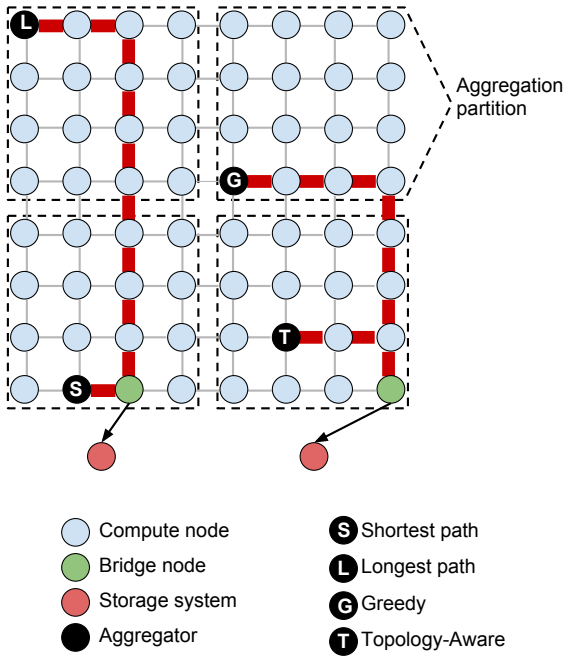

Figure 2: Data aggregation for I/O: simple partitioning and aggregator election on a grid with one different strategy per partition.

As explained before, our topology-aware strategy aims at taking into account the network topology and the amount of data exchanged between ranks and their aggregator. More precisely, we defined an objective function to minimize the time to perform the I/O and find an efficient aggregator placement.

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
- $\omega(u, v)$: The data size exchanged between nodes u and v
- $d(u, v)$: The number of hops between nodes u and v
- IO: The storage system (I/O node) of the partition.
- l: The interconnect latency
- $B_{i \to j}$: The bandwidth from node i to node j.

For the first step of our strategy, each process involved in an aggregation partition computes the cost of aggregating data from all other ranks if it were chosen as the aggregator. This can be done in a distributed way once all the processes know the amount of data produced by each of them. We then keep the maximum cost as all the others will be bounded by this one. Formally, each process A computes the cost C_1:

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

The second step consists in computing the cost of sending the aggregated data to the storage system. The first version of our model took into account the total amount of aggregated data to compute this cost. However, while cost C_1 considers a small amount of data (the data sent by the rank with the maximal cost), C_2 considers the sum of the aggregated data. This creates an imbalance between these two cost and can make C_1 negligible compared to C_2. To avoid this, in C_2, we normalize the aggregated data with the number of processes involved in the aggregation phase to have C_1 and C_2 of the same order of magnitude. For each process A, we define the cost C_2 as:

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

Our topology-aware strategy minimizes the objective function defined as the sum of the costs C_1 and C_2. More generally, this placement policy can be formulated as the solution for each partition of this objective function:

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

This sum is computed by each process independently in $O(n), n = |V_C|$. Indeed the topology characteristics d, l, B can be precomputed at start time and the data exchange between processes depends on the data distribution and is accessible in constant time after a preliminary call to `MPI_Allgather`. Hence, finding the process having the minimum cost is done through a `MPI_Allreduce` with the `MPI_MINLOC` operation. All these MPI calls involve meta-data of very small size compared to the actual data and hence have a negligible cost compared to sending the application data.

4.2 File System Block Size

The block size of a filesystem corresponds to an indivisible block of memory on disk requested for each read or write operation, no matter the size of data involved. Thus, writing a piece of data smaller than the block size implies a lock on a piece of memory of size *blocksize*. In the context of parallel I/O, the multiplication of locks on disks creates an important bottleneck. To evaluate the impact of the block size, we wrote a simple benchmark and ran it on a BG/Q supercomputer and its GPFS high-performance filesystem. This benchmark works as follow: one process per node writes the same amount of data to a single shared file at the corresponding offset. The results are depicted in Figure 3. The purple curve shows the bandwidth achieved while writing a chunk of data which is a multiple of the filesystem block size (8 MB in this case). The green curve shows this bandwidth with the same chunk of data plus 1 kB. We see that writing a multiple of the block size can perform up to 3x better than the non-aware case.

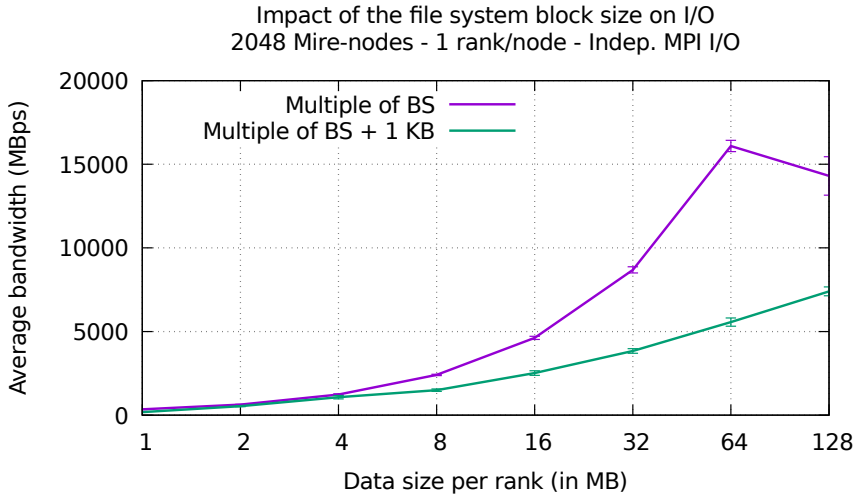

Figure 3: Benchmark measuring the impact of the filesystem block size for write operations.

In order to reduce this penalty, our algorithm has been designed to aggregate and write on an aggregator an amount of data which is a multiple of the filesystem block size. In other words, the buffers used by the aggregators to stage data during the I/O phase are allocated as a multiple of the filesystem block size. The default buffer size used in MPI two-phase I/O implementations has this property as well.

4.3 Pipelined Aggregation Buffers

In order to optimize both the aggregation phase and the I/O phase, each aggregator manages two buffers and overlaps the communications. In fact, as the aggregation phase is performed with RMA operations (one-sided communication), no synchronization is needed between the processes sending data to the aggregators and the aggregators themselves. Moreover, the aggregators perform non-blocking independent writes to the storage system making themselves available for other operations. In this way the aggregators are able to flush a full buffer while receiving data in the second one. This loop is performed as many times as necessary to process the data. Each instance of buffer

filling and flushing is called a *round*. A global round is equivalent to the same buffer of all the aggregators filled (if applicable) and flushed.

4.4 Algorithm

The Algorithm 1 describes the core function of our aggregation process; that is, the `Commit` function called by the processes to write a piece of data. This function is recursive in order to split a piece of data among different buffers and/or aggregators. As we initialize our aggregation step with the upcoming writes, each process knows for each piece of data the target tuple {round, aggregator, buffer}. The lines 2 to 5 return these values. The chunk size corresponds to the amount of data to be written. If the piece of data fits in one buffer, this amount is equal to the data size given as parameter. In case of data splitting, this amount is smaller than the data size and an extra round is necessary. 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 `Commit` function again while updating the parameters.

Algorithm 1: Data Aggregation

```
1  Function Commit (data, size, offset)
2      round ← GetRound();
3      aggr ← GetAggregatorRank();
4      chunkSize ← GetRoundSize(round);
5      bufferId ← globalRound % 2;
7
8      while round ≠ globalRound do
9          Fence ();
10         if I am an aggregator then
11             Write_Buffer (bufferId);
12         globalRound ← globalRound + 1;
13         bufferId ← globalRound % 2;
15
16     Put (data, chunkSize, offset, aggr, bufferId);
18
19     if chunkSize = size then
20         while round ≠ m_round do
21             Fence ();
22             if I am an aggregator then
23                 Write_Buffer (bufferId);
24             round ← round + 1;
25     else
26         Commit (data + roundSize, size − chunkSize, offset + chunkSize);
```

4.5 Challenges at scale

When bringing these optimizations at scale, we needed to address two challenges in our model. Firstly, the partitioning phase can appear costly for certain cases. In particular, if one needs to partition a large MPI communicator, the fact that a process can be involved with several aggregators limits the parallelization of this part of the algorithm. In MPI, a communicator splitting cannot produce sub-communicators with an intersection. Put it in another way a process cannot belong to more than one sub-communicator resulting from a `MPI_Comm_split`. It implies that we have to perform the aggregator mapping partly sequentially. Moreover, some `MPI_Comm_split` implementations can be very unoptimized (e..g., on BG/Q) increasing the computation time. However, even if in most cases this limitation is absent, it has to be noted that we compute the aggregators mapping once for the whole application lifetime reducing its impact. Secondly, it is clear that to avoid contention our implementation needs to keep at most one aggregator per node. As a consequence, we have to update a list of nodes already selected as aggregators. This list is shared among at least a subset of processes. Because of this, some synchronization steps are necessary.

5 Evaluation

To validate our approach, we carried out experiments with both micro-benchmarks as well as with a real application called HACC from which we extracted the I/O kernel. We conducted our experiments on the Mira supercomputer, an IBM BG/Q supercomputer located at Argonne National Laboratory.

5.1 Targeted supercomputer

Mira is an IBM BG/Q supercomputer hosted at Argonne National Laboratory with 49,152 compute nodes each with 16 hyper-threaded PowerPC A2 cores (1600 MHz). Each node has 16 GB of main memory. These nodes are interconnected through a 5D torus high-speed network giving a theoretical bandwidth of 1.8 GBps per link. The BG/Q architecture splits the nodes into *Psets*. A *Pset* is a subset of nodes sharing the same I/O node. On Mira, this subset contains 128 nodes including two bridge nodes. A bridge node is a regular compute node connected to an I/O node (so each I/O node is connected to two bridge nodes). The Figure 4 shows this architecture. Note that for all the experiments using our aggregation strategy, we set the bandwidth between two nodes to 1.8 GBps (theoretical bandwidth) and the latency to 20 ms (measured). We compiled the test applications and our library with GCC v4.4.7 and used the default MPI installation on Mira, which is based on MPICH2 v1.5.

The MPI I/O part is based on ROMIO, an open-source high-performance implementation. MPI I/O has a highly configurable set of parameters; the default number of aggregators on Mira is set to 16 per *Pset* and the size of the buffer employed to aggregate the data is set to 16 MB. The default aggregator mapping strategy used by ROMIO on BG/Q is the one described in Section 4.1. The version of MPI-2 hosted on Mira has been optimized for the BG/Q platform.

5.2 Evaluation of the placement strategies

We evaluate our topology-aware strategy with a micro-benchmark and compare this approach to the other approaches described in section 4.1, namely, greedy, shortest path and longest path. The

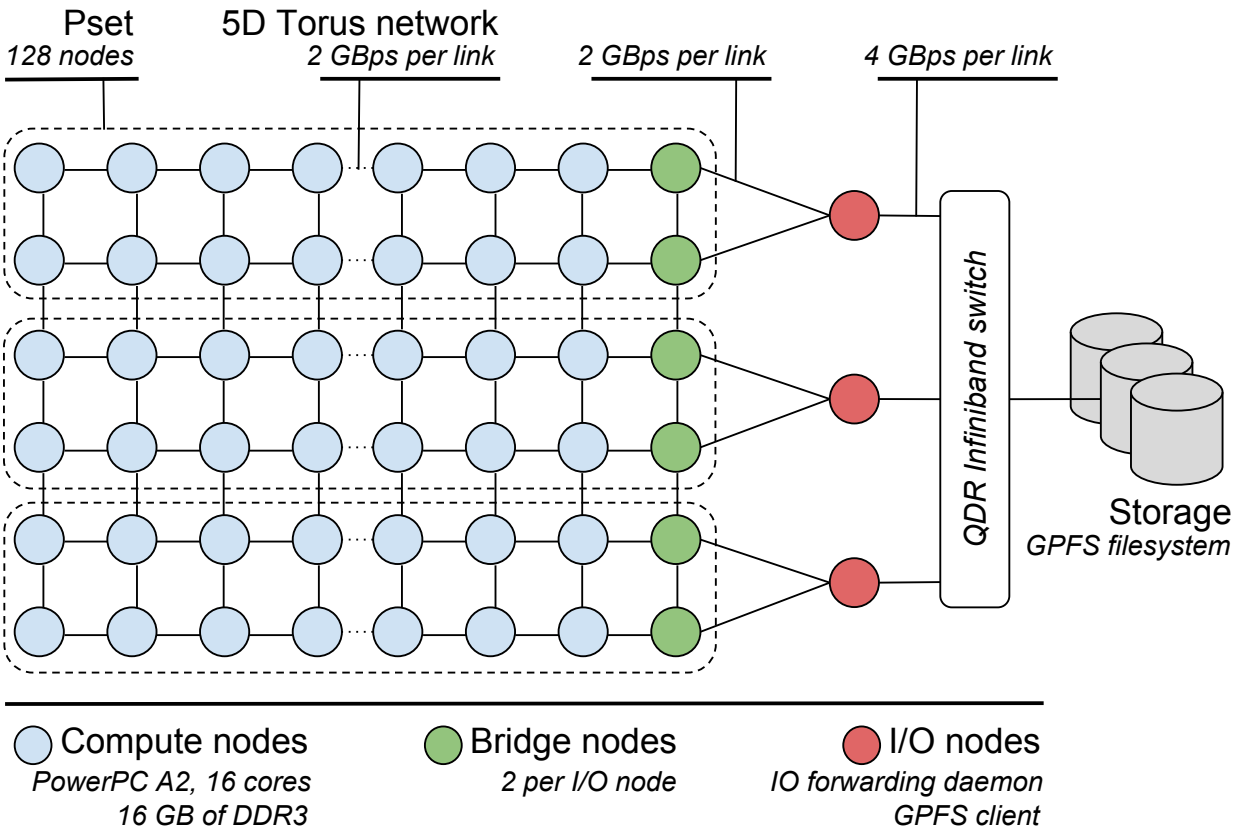

Figure 4: BG/Q architecture

micro-benchmark works as follow: Each rank produces an amount of data distributed randomly between 0 and 2 MB. In order to reduce the file system noise, the data were sent to the null device of the I/O nodes instead of a real file. This effectively measures the performance of just aggregating the data and moving this out to the storage system. It eliminates any I/O contention, and thus, performance degradation at the storage system. The experiments were carried out on 512 Mira-nodes with 16 ranks per node (8192 ranks). We set the number of aggregators to 16 for the `MPI_COMM_WORLD` communicator (i.e. 4 aggregators per *Pset*) to intensify the impact of the placement policy. The aggregator's buffer size was set to 16 MB. The results are shown in Table 2 and are calculated from 20 runs.

Table 2: Impact of aggregators placement strategy

Strategy	**I/O Bandwidth (MBps)**	**Aggr. Time/round (ms)**
Topology-Aware	2638.40	310.46
Shortest path	2484.39	327.08
Longest path	2202.91	370.40
Greedy	1927.45	421.33

These results illustrate the I/O bandwidth achieved and the time needed per round for the two-phase I/O step. We notice that the topology-aware strategy gives the best bandwidth compared to the other methods. The shortest-path strategy offers the second best performance and greedy is the worst.

5.3 Impact of the number of aggregators and the buffer size

Finding the tradeoff between the number of aggregators and the buffer size is an open problem [5]. Even though it is not the main purpose of this paper, it is important to understand the behavior of our topology-aware aggregation algorithm with these parameters. For this, we carried out experiments with a micro-benchmark where we vary the number of aggregators and the buffer size. We ran these experiments on 1024 Mira-nodes, with 16 ranks per node, and with 1 MB produced by each process. The data is again written to the null device of the I/O nodes to understand the performance of the aggregation and data movement. Table 3 presents these results.

Table 3: I/O Bandwidth (in MBps) achieved by a simple benchmark with a topology-aware aggregator placement while varying the number of aggregators and the buffer size.

#Aggr/Pset	**Buffer size**		
	8 MB	**16 MB**	**32 MB**
8	7652.49	8848.28	9050.71
16	7318.15	8774.58	9331.84
32	6329.95	7797.12	8134.41

In these experiments, the third column and row corresponds to the default parameters set in ROMIO on Mira: 8 aggregators per *Pset* and 16 MB of buffer size. With our algorithm, these settings produce a good I/O bandwidth. Moreover, the best I/O bandwidth is reached with larger buffers sizes. These observations are useful for future improvements and tuning.

5.4 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 is: 25,000 particles require approximately 1 MB.

5.4.1 Impact of data layout

During our experiments, we observed that ROMIO performs very poorly in case of an array of structure (AoS) data layout. This layout is the default one used by HACC. To have a good overview of the performance with the different standard strategies, we implemented an alternative data layout based on structures of array (SoA). Figure 5 describes the differences between these

two layouts in the context of HACC. For all the next experiments with this application, results for both these layouts are shown (except for POSIX I/O which performs independent I/O).

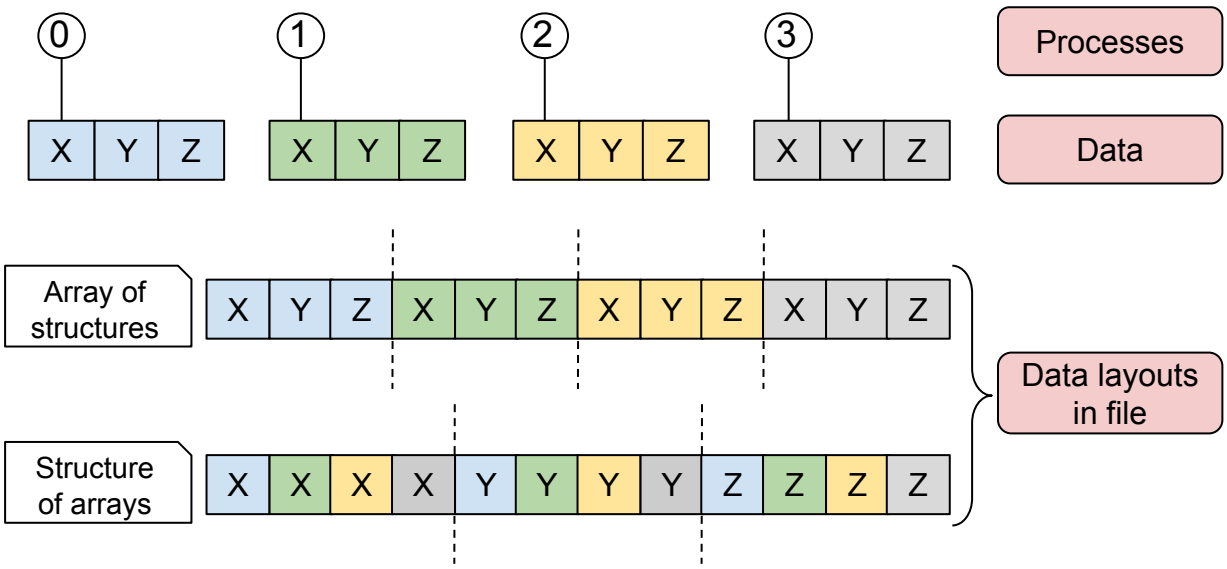

Figure 5: Data layouts implemented in HACC

The difference between the two types of organizations in terms of I/O can be noted. In the case of HACC, where the particles are written by variables, an AoS design needs more time to fill an aggregator buffer as it is bounded by the time necessary to write all the variables from one process. With a SoA layout, all the processes write the same variables at the same time. The time required to fill a buffer gets closer to the time needed to write one variable from one process. Except in few cases, we observed this impact in our results. It also explains the performance of ROMIO.

5.4.2 Results

The Figures 6 and 7 show respectively the results when writing to a single file shared by all the processes and to one file per *Pset*. These experiments were run on 1024 Mira-nodes and 16 ranks per node. For each strategy, we vary the number of particles per rank. To conduct fair experiments, we set the number of aggregators to 16 per *Pset* and the buffer size to 16 MB, as the default settings of ROMIO on Mira.

Writing to a single shared file (Figure 6) results in poor performance in general. On 1024 nodes, the effective peak I/O bandwidth is estimated to be 22.4 GBps (while the theoretical bandwidth is 28.8 GBps). The best performance we can achieve in this case does not exceed 5 GBps. Nevertheless, only our topology-aware strategy is able to achieve this. We can also notice that our approach outperforms both POSIX I/O and MPI I/O regardless of the data size or the data layout. This difference is particularly substantial on small messages and tends to decrease while the data size per rank increases. When 5000 particles are written by a process (∼200 KB), in the case of an array of structures data layout, our method provides an I/O bandwidth 15× higher than MPI I/O. This factor is 4× larger with a different data layout (SoA). As explained before, we can clearly see the poor performance obtained with MPI I/O on a specific data layout.

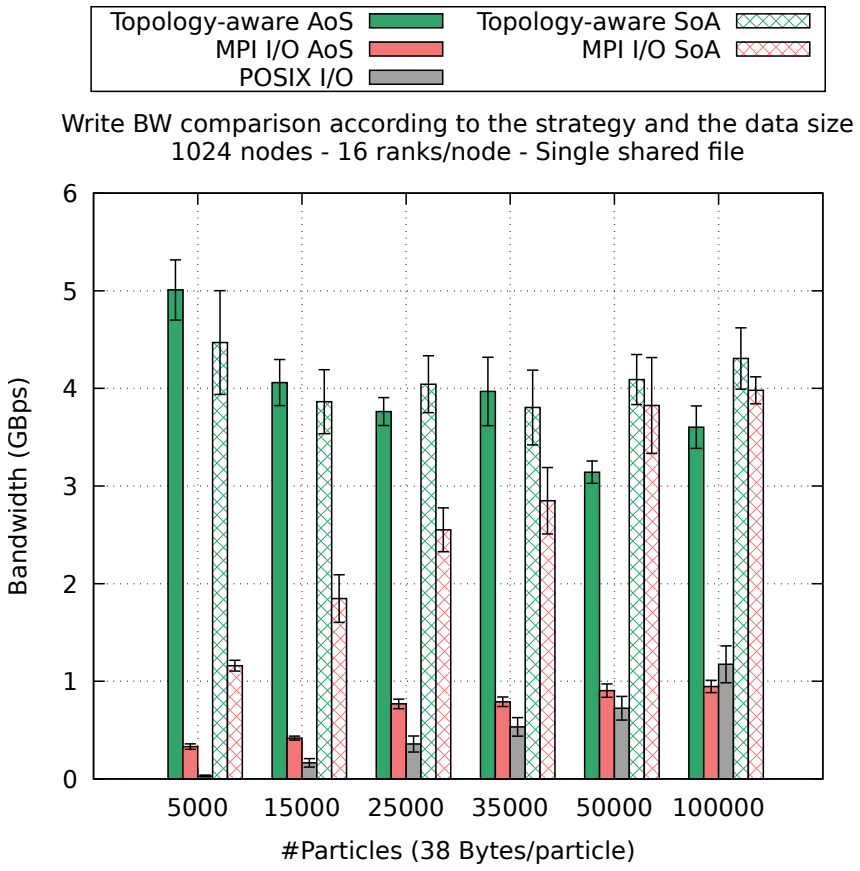

Figure 6: Single shared file from 1024 nodes (16 ranks/node)

Dividing the output data into several files, also known as sub-filing, appears to be an attractive solution. The results presented in Figure 7 are a relevant illustration with I/O bandwidth close to the peak value. Like the previous results, we observe that our strategy based on topology-aware placement and optimized buffers (block size awareness and pipelining) achieves much better performance than the standard approaches. Again, this gap decreases slightly as the data size increases. However, the I/O bandwidth remains higher with our method, even with large messages (2 MB per process).

Figure 8 presents results from the same experiments as we perform weak scaling to 4096 nodes (64K MPI processes). These results show very similar behavior at scale. As on 1024 nodes, we are able to significantly improve the I/O bandwidth, irrespective of the data layout in file and particularly on messages smaller than 2 MB.

6 Conclusions and Future Work

In this paper, we have demonstrated the importance of the data movement optimizations for intensive I/O operations. In particular, we developed an I/O library on top of MPI I/O based on the two-phase I/O scheme and this takes into account the topology of the infrastructure, an efficient buffering system and the access patterns of applications. This model achieves very good

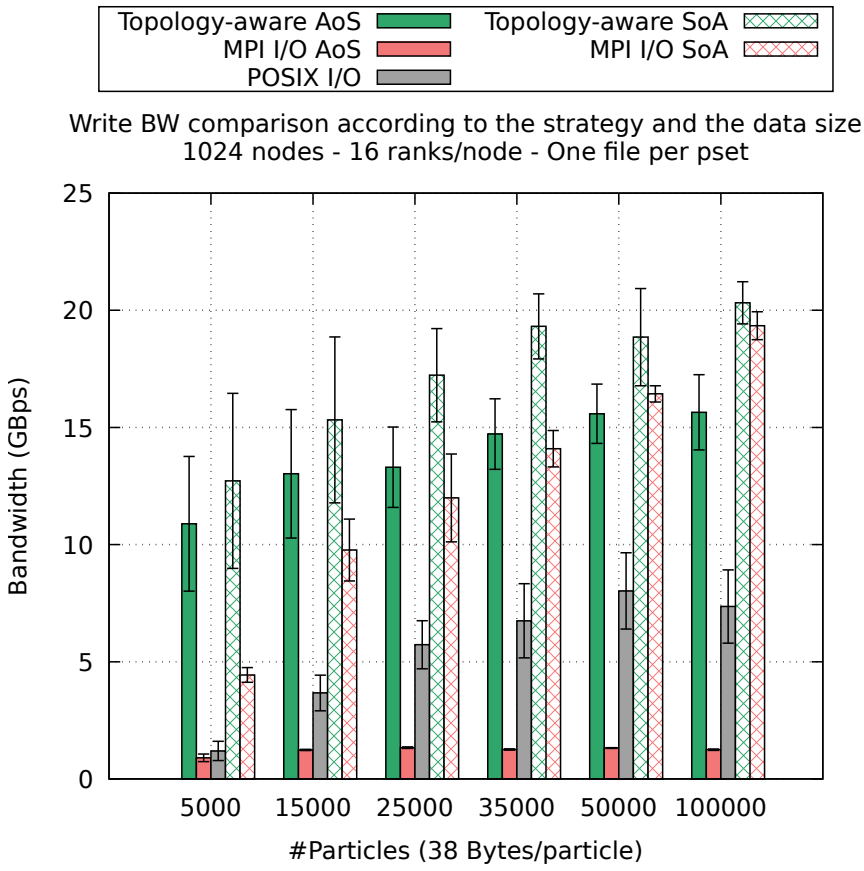

Figure 7: One file per *Pset* from 1024 nodes (16 ranks/node)

performance at scale and outperforms standard approaches. For the I/O kernel of a cosmological application, our solution was able to achieve a 15 × improvement over default parallel I/O implementation. Additionally, we demonstrated the needs to design an algorithm capable of dealing with different data layout by overlapping communications. We demonstrate the necessity to fully exploit the architecture characteristics in order to achieve performance at scale and meet the expectations of large-scale simulations. As part of our future work, we plan to consider routing strategies in our algorithm. This information could be extremely useful to reduce performance degradation due to network contention. Another research track is to extend this aggregation method to a larger varieties of data patterns (2D or 3D arrays for example). Finally, we will need to adapt our model to the various expected tiers of storage and memory.

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 DOE 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

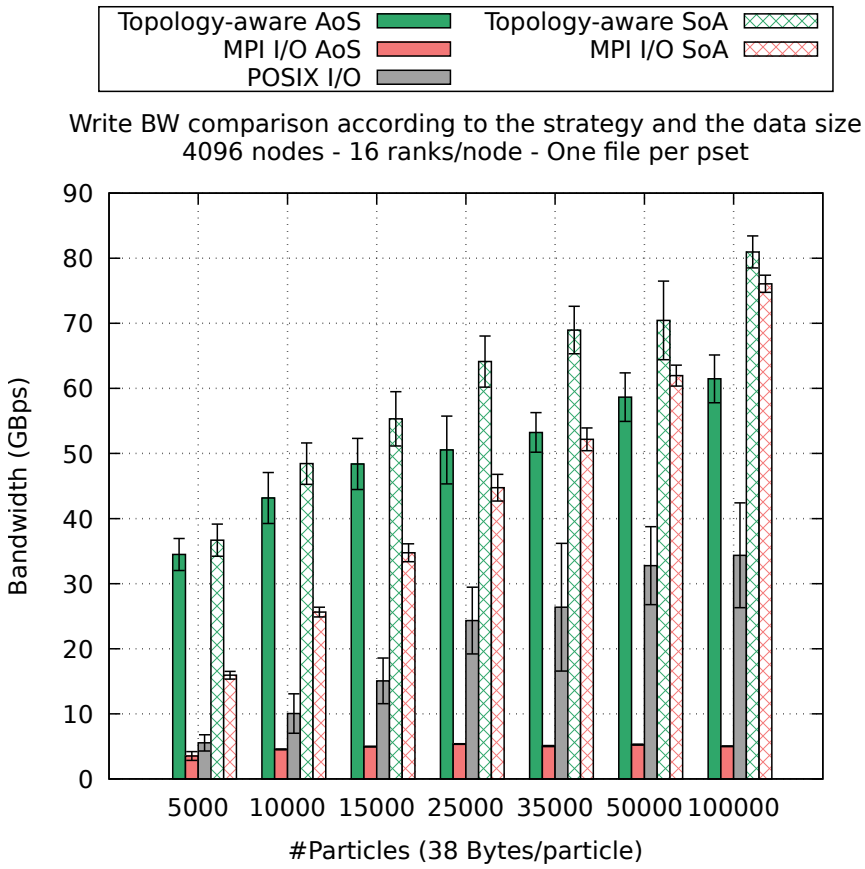

Figure 8: One file per *Pset* from 4096 nodes (16 ranks/node)

on Extreme Scale Computing (JLESC). The research leading to these results has been partially supported by funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 328582.

References

[1] Lustre filesystem website. `http://lustre.org/`.

[2] Wahid Bhimji, Debbie Bard, Melissa Romanus, David Paul, Andrey Ovsyannikov, Brian Friesen, Matt Bryson, Joaquin Correa, Glenn K. Lockwood, Vakho Tsulaia, Suren Byna, Steve Farrell, Doga Gursoy, Chris Daley, Vince Beckner, Brian Van Straalen, David Trebotich, Craig Tull, Gunther H. Weber, Nicholas J. Wright, Katie Antypas, and Prabhat. Accelerating science with the NERSC burst buffer early user program. In *CUG2016 Proceedings*, 2016. Best paper award, in press.

[3] Huy Bui, Jason Leigh, Eun-Sung Jung, Venkatram Vishwanath, and Michael E. Papka. Improving data movement performance for sparse data patterns on the blue gene/q supercomputer. In *ICPP Workshops*, pages 302–311. IEEE Computer Society, 2014.

[4] Mohamad Chaarawi, Suneet Chandok, and Edgar Gabriel. *Performance Evaluation of Collective Write Algorithms in MPI I/O*, pages 185–194. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[5] Mohamad Chaarawi and Edgar 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*, pages 428–437, 2011.

[6] Dong Chen, Noel A. Eisley, Philip Heidelberger, Robert M. Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L. Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J. Parker. The ibm blue gene/q interconnection network and message unit. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 26:1–26:10, New York, NY, USA, 2011. ACM.

[7] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved parallel i/o via a two-phase run-time access strategy. *SIGARCH Comput. Archit. News*, 21(5):31–38, December 1993.

[8] Rosa Filgueira, David E. Singh, Juan Carlos Pichel, Florin Isaila, and Jesús 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*, pages 137–149, 2008.

[9] Message Passing Interface Forum. Mpi-2: Extensions to the message-passing interface, July 1997. http://www.mpi-forum.org/docs/docs.html.

[10] William 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*, pages 7–, London, UK, UK, 2002. Springer-Verlag.

[11] F. Isaila, Prasanna Balaprakash, S. M. Wild, D. Kimpe, R. Latham, R. Ross, and Paul D. Hovland. Collective i/o tuning using analytical and machine-learning models. In *IEEE Cluster 2015*, Chicago, IL, 09/2015 2015. IEEE, IEEE.

[12] 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 '02, Berkeley, CA, USA, 2002. USENIX Association.

[13] Rajeev Thakur, William Gropp, and Ewing 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.

[14] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective i/o in romio. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, pages 182–, Washington, DC, USA, 1999. IEEE Computer Society.

[15] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, IOPADS '99, pages 23–32, New York, NY, USA, 1999. ACM.

[16] Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in mpi i/o. *Parallel Comput.*, 28(1):83–105, January 2002.

[17] 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*, pages 232–235, Feb 2014.

[18] Yuichi Tsujita, Hidetaka Muguruma, Kazumi Yoshinaga, Atsushi Hori, Mitaro Namiki, and Yutaka Ishikawa. Improving collective i/o performance using pipelined two-phase i/o. In *Proceedings of the 2012 Symposium on High Performance Computing*, HPC '12, pages 7:1–7:8, San Diego, CA, USA, 2012. Society for Computer Simulation International.

[19] Vishwanath Venkatesan, Rakhi Anand, Jaspal Subhlok, and Edgar Gabriel. Optimized process placement for collective i/o operations. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 31–36, New York, NY, USA, 2013. ACM.

[20] Venkatram Vishwanath, Mark Hereld, Vitali Morozov, and Michael 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*, SC '11, pages 19:1–19:11, New York, NY, USA, 2011. ACM.