Adding topology and memory awareness in data aggregation algorithms

François Tessier, Venkatram Vishwanath, Emmanuel Jeannot

Adding Topology and Memory Awareness in Data Aggregation Algorithms

François Tessier[a], Venkatram Vishwanath[b], Emmanuel Jeannot[c]

[a]*Inria, University of Rennes, CNRS, IRISA Rennes, Rennes, France*
[b]*Argonne National Laboratory, Lemont, IL, USA*
[c]*Inria, LaBRI, Univ. Bordeaux, CNRS, Bordeaux-INP, Bordeaux, France*

Abstract

With the growing gap between computing power and the ability of large-scale systems to ingest data, I/O is becoming the bottleneck for many scientific applications. Improving read and write performance thus becomes decisive, and requires consideration of the complexity of architectures. In this paper, we introduce TAPIOCA, an architecture-aware data aggregation library. TAPIOCA offers an optimized implementation of the two-phase I/O scheme for collective I/O operations, taking advantage of the many levels of memory and storage that populate modern HPC systems, and leveraging network topology. We show that TAPIOCA can significantly improve the I/O bandwidth of synthetic benchmarks and I/O kernels of scientific applications running on leading supercomputers. For example, on HACC-IO, a cosmology code, TAPIOCA improves data writing by a factor of 13 on nearly a third of the target supercomputer.

Keywords: Data movement, I/O, data aggregation, deep memory and storage hierarchy, architecture-aware placement
2000 MSC: 68W15, 68W10

1. Introduction

In the domain of large-scale simulations, driven by the demand for reliability and precision, the generation of tera- or petabytes of data has become increasingly prevalent. However, a growing disparity between compute power and I/O performance on supercomputers has emerged. Over the past decade, the ratio of I/O bandwidth to computing power for the first three systems on the top500 [1] list has decreased by a factor of ten, as illustrated in Figure 1. In that context, efficiently moving data between the applications and the storage system within high-performance computing (HPC) machines is crucial.

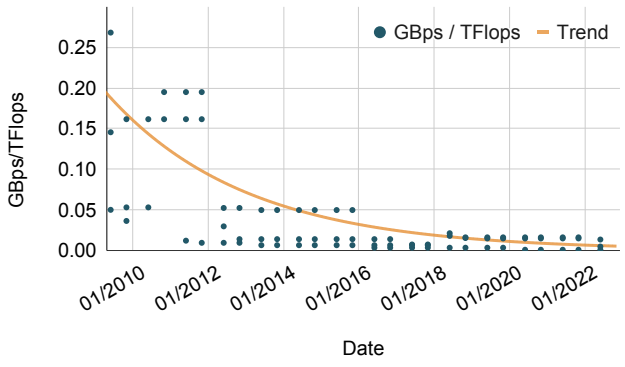

Figure 1: Ratio of I/O bandwidth (GBps) to computing power (TFlops) of the top 3 supercomputers of the Top500 over the past 10 years

[1] `https://www.top500.org/`

On the application side, managing I/O is complicated by the diverse data structures employed. For instance, particle-based applications often require writing multiple variables in multidimensional arrays distributed among processing entities, while adaptive mesh refinement (AMR) applications must handle varying I/O sizes depending on input parameters. The popularity of deep learning algorithms has introduced new workloads demanding vast quantities of input data. Additionally, complex workflows like in-situ visualization and analysis further exacerbate this complexity. Consequently, optimizing data movement is of paramount importance for the foreseeable future for scaling science.

From a hardware perspective, there has been a growing disparity between the amount of data that needs to be transferred and the memory or storage capabilities in terms of both capacity and performance. To address this issue, hardware vendors have introduced intermediate tiers of memory and storage, which must be utilized effectively to alleviate the I/O bottleneck. However, these memory hierarchy levels come with unique characteristics and sometimes require a dedicated software stack, making efficient use challenging. In addition, the process of moving data necessitates traversing intricate network topologies that must be considered, such as an all-to-all, 5D-torus or dragonfly.

In this landscape, harnessing these architecture and application characteristics are key to making optimized decisions. Among these techniques, data aggregation plays a central role for mitigating data movement bottlenecks. It involves aggregating data at various points in the architecture to optimize expensive data access operations. In collective I/O operations, for instance, data aggregation accumulates contiguous data chunks

in memory before writing them to the storage system. This approach is called "two-phase I/O". However, the current implementations of the two-phase I/O scheme suffer several limitations, especially with regard to the complexity of modern architectures. A reevaluation of this algorithm that fully leverages the potential of new technologies such as RDMA (Remote Direct Memory Access) and asynchronous operations can highly improve I/O performance. Furthermore, an approach that is agnostic to the network topology and the memory is necessary to effectively handle the deepening complexity of memory and topology hierarchies.

In this paper, we introduce TAPIOCA, an I/O library designed to perform architecture-aware data aggregation at scale. TAPIOCA targets applications using collective I/O operations and can be extended to intricate workflows such as in-situ or in-transit analysis that may require temporarily persistent data. With an abstraction layer for the network interconnect and deep memory hierarchy, this library can execute data aggregation on any memory or storage system in current and forthcoming large-scale systems, offering seamless portability across various supercomputers. To determine the most suitable location for data aggregation, we also provide a detailed cost model minimizing data movement. To validate our approach, we demonstrate how TAPIOCA outperforms traditional I/O calls on a synthetic benchmark and the I/O kernels of two real applications. We run our experiments on two leadership-class supercomputers and a visualization cluster at Argonne National Laboratory, USA, all featuring characteristics we are seeing on emerging exascale architectures.

2. Context and Motivation

2.1. Large-Scale Simulations

Large-scale simulations can be categorized into various groups, and among them, certain applications are heavily reliant on I/O operations, resulting in substantial time spent accessing the storage system. There are several factors contributing to this behavior. For instance, some applications involve extensive reading of input data that must be processed during the simulation. Conversely, other applications generate significant amounts of data that require subsequent processing after generation. Additionally, certain applications frequently access the file system for checkpointing purposes. It is worth noting that these categories are not mutually exclusive, and an application can fall into multiple categories. Therefore, optimizing the I/O access of such applications holds paramount significance, as it directly impacts the overall execution time.

2.2. Accessing Data at Scale

In recent years, the ratio between compute and I/O performance of supercomputers has been constantly degrading. Nowadays, in many applications, the I/O is becoming a bottleneck, requiring to improve data movement. In order to reduce this gap, effort has been carried out at the hardware level and especially for the topology of the machine. Indeed, the networks topologies, despite being more complex, tend to reduce the distance between the data and the storage. Many supercomputers features I/O nodes that are embedded within racks to serve as a proxy to the parallel file-system. This architecture helps to avoid I/O interference by decoupling the compute network and the I/O network. 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 [1]. 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 (although the routing strategy can transmit packets through more links). 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 using on-node memory and storage to reduce the I/O pressure of each application on the parallel file-system. This allows several optimizations. For example, SSD-based burst buffers (as the ones used on the Cray Cori infrastructure [2]) are intermediate nodes between compute nodes and storage system that can supply a smaller storage capacity but a higher I/O bandwidth. These storage tiers are designed to absorb the burst and accelerate the I/O phase of applications. While writing data out for future analysis is costly, another technique involves storing data in memory for *in situ* analysis. Although bandwidth-efficient, this approach is limited by the amount of memory available.

An Ad-Hoc file systems [3, 4] is an application-level file systems, deployed at launch-time that intercept I/O accesses to store data locally, on-nodes, hiding and abstracting the local memory or burst buffers present in the machine. With such system, the question if staging data at some point (when the SSD are full) or at the end of the application remains a key problem.

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, an abstraction layer is necessary for taking network topologies, local memory and disks into account. The goal is to use in a simple way the hardware features of the supercomputer at their full potential to optimize I/O operations.

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

MPI [5] is widely utilized for the development of large-scale distributed-memory applications on high-performance clusters. Within MPI, the MPI I/O component plays a critical role in facilitating input and output operations. One significant aspect of MPI I/O is the collective I/O mechanism, which enables efficient reading and writing of data at scale. In collective I/O, all MPI tasks involved in the communication invoke the I/O routine in a synchronized manner, allowing the MPI runtime system to optimize data movement based on various application

parameters, including data size, memory layout, and storage arrangement.

The two-phase I/O algorithm, present in MPI I/O implementations like ROMIO [6], is a well-established and efficient optimization technique. It involves selecting a subset of processes to aggregate contiguous data segments (aggregation phase) before writing them to the storage system (I/O phase). The primary objective of this approach is to minimize latency and enhance parallel file-system accesses by aggregating data in a manner that aligns with the layout of the parallel file-system. Figure 2 provides an illustrative example of this technique, featuring four processes with two selected as aggregators. By minimizing network contention around the storage system and maximizing the I/O bandwidth through the writing of large contiguous data chunks, substantial performance improvements are achieved. However, the current implementation of this approach exhibits several limitations. Firstly, despite offering improved I/O performance compared to direct access, it often falls short of achieving the peak I/O bandwidth. Secondly, there is an observed inefficiency in the placement policy for aggregators, despite the potential impact on performance through smart mapping. Lastly, existing implementations fail to leverage the data model, data layout, and memory and system hierarchy effectively.

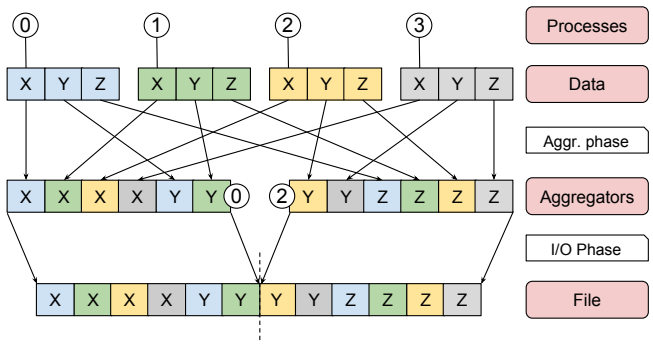

Figure 2: Example of the two-phase I/O mechanism

This work focuses on addressing these limitations within the context of the two-phase I/O scheme. Specifically, we present TAPIOCA, an I/O library built on top of MPI I/O, designed to optimize the two-phase I/O scheme for large-scale supercomputers with a keen awareness of system topology. TAPIOCA encompasses three primary directions: an efficient implementation of the two-phase I/O algorithm, an enhanced aggregator placement strategy that accounts for system characteristics, and a versatile interface to query system topology information.

3. Related work

Parallel I/O [7] is an active research topic, primarily developed in the context of intensive parallel I/O. While I/O tuning is a crucial step to increase I/O bandwidth, improvements at various layers of the I/O software stack are necessary. Parallel file systems like GPFS [8] and Lustre are widely used [9], and parallel I/O libraries such as MPI I/O and its ROMIO [6] implementation, part of the MPI-2 [5] standard, are common for performing reads and writes. Collective I/O techniques, like Chaarawi et al.'s evaluation of various write algorithms [10], have been deployed to enhance performance. The two-phase I/O algorithm [11], which aggregates data on a subset of processes before writing it to the storage system, is a de facto collective I/O approach. Various efforts have been made to optimize this algorithm [12, 13, 14, 15, 16, 17], but they usually lack awareness of the available tiers of memory and storage, limiting their ability to leverage these resources effectively. Other research has investigated multithreading to overlap aggregation and I/O phases using double buffering [15, 18]. However, the optimal number of aggregators and buffer size in collective I/O remains an open question. In general, other I/O libraries offer aggregation techniques, but these are generally not very advanced or scalable [19].

Data movement optimizations based on data aggregation has also been explored beyond low-level I/O libraries. At an ephemeral file-system level for instance, UnifyFS has implemented aggregation as a unique namespace on node-local storage resources [20]. Other work, for example in the field of checkpointing [21, 22], has developed aggregation techniques to accelerate write phases, notably via asynchronous operations. However, these approaches are limited to a specific tool or framework and are architecture-agnostic, whereas TAPIOCA proposes to take advantage of MPI, which is widely used in the community, while also leveraging the machine's topology. From a workflow point of view, some authors have investigated using available SSDs to overcome DRAM shortages in specific workflows [23], while others have focused on finely describing workflow data movements [24, 25]. However, these techniques often require users to possess in-depth knowledge of their applications.

In contrast, some research has been conducted from a runtime perspective. For example, authors have proposed transparently moving data from applications to storage systems through an intermediate fast storage layer [26]. Another approach explored the use of fast storage layers (e.g., burst buffers) as a distributed file system [4], and a driver for MPI-IO was developed to take advantage of network-attached memory tiers [27]. However, these approaches are tailored to specific architectures and memory tiers, limiting portability.

To ensure code portability and accommodate the emerging exascale machines with new memory and storage tiers, an architecture abstraction is essential. Hwloc [28] offers a common hardware abstraction, although it provides qualitative information and does not account for the interconnect network. At a higher level, SharP [29] provides an abstraction layer for allocating memory on any available tier, but it is dependent on the data model to handle. Our approach stands out by adopting a data aggregation method that considers the underlying architecture through memory and network interconnect abstractions. TAPIOCA can perform aggregation on any available memory and storage tier, offering a model that minimizes the cost of data movement, while being independent of the application's data model.

Our current approach differs from existing solutions by com-

bining an optimized buffering system with an architecture-aware quantitative aggregator mapping strategy. It targets various systems, such as IBM BG/Q and Cray XC40, along with both GPFS and Lustre. Furthermore, it is extensible to accommodate new storage tiers and takes into account the application's I/O pattern.

4. Our Approach

In this paper, we introduce TAPIOCA (standing for Topology-Aware Parallel I/O: Collective Algorithm), a MPI-based library for collective I/O operations using an optimized architecture-aware two-phase I/O algorithm. By relying on an architecture abstraction layer, TAPIOCA enables the placement of aggregators taking into account the network topology and the available memory and storage spaces. Our library also optimizes data aggregation by taking the data layout into account through a description of the I/O phases in the application's code. Finally, we focused on an efficient implementation leveraging one-sided communication (Remote Memory Access) and multi-buffering.

In the rest of this section, we present these different aspects of TAPIOCA. We begin by detailing our hardware abstraction layer, then we introduce our architecture-aware cost model for data aggregation. We conclude this section by presenting our aggregation algorithms for both read and write collective operations. For the remainder of this paper, we will use the term *buffer* to refer to the memory space dedicated to aggregation on the "aggregator" processes, and *target* to designate the destination of the data (typically, a parallel file system).

4.1. Architecture Abstraction

A key feature of our approach is to achieve code and performance portability across a broad variety of architectures, including emerging and future network interconnects and tiers of memory and storage. To do so, we have developed two abstraction layers with which our library interacts for both efficient aggregators placement and management of reads and writes on different levels of memory and storage. Figure 3 depicts how those components fit into TAPIOCA while Listings 1 and 2 show some of the API functions of those two abstractions.

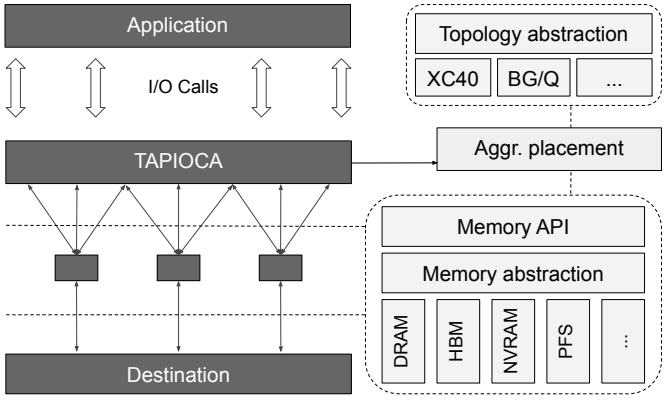

Figure 3: High-level view of TAPIOCA and the two abstraction layers

Listing 1: Function prototypes for memory/storage data movements

```
buff_t* memAlloc       (mem_t mem, int buffSize,
                        bool masterRank, char* fileName,
                        MPI_Comm comm);
void    memFree        (buff_t *buff);
int     memWrite       (buff_t *buff, void* srcBuffer,
                        int srcSize, int offset,
                        int destRank);
int     memRead        (buff_t *buff, void* srcBuffer,
                        int srcSize, int offset,
                        int srcRank);
void    memFlush       (buff_t *buff);
int     memLatency     (mem_t mem);
int     memBandwidth   (mem_t mem);
int     memCapacity    (mem_t mem);
int     memPersistency (mem_t mem);
```

Our memory abstraction (Listing 1) allows to allocate and free buffers on any kind of memory or storage. *memRead()* and *memWrite()* functions are in charge of data movements from/to an allocated buffer. As some operations are either asynchronous or need a process involved to be completed, a *memFlush()* function has been implemented to ensure that all the initiated operations on the buffer are finished. Functions giving vendors or experimental performance values for the memory tiers are also available. The *memPersistency()* function returns the persistency capability of a memory tier. The cost model we describe in 4.2 queries those values.

Technically, this memory abstraction internally calls the appropriate functions according to the type of memory managed. For instance, if data is aggregated on the high-bandwidth memory (HBM), the *memkind*[2] library will be used for allocation and deallocation. Depending on the scope of the memory bank, the memory management technique may vary. An on-node SSD, for example, is locally accessible with regular I/O calls (POSIX, MPI, ...) but has to be exposed to remote nodes in case of aggregation from multiple compute nodes. In this case, we implemented this feature by mapping a file on SSD into the main memory through a *mmap* system call then by exposing this buffer to remote nodes with a MPI Window (RMA).

The network abstraction provides the relative location of compute nodes as well as performance information. In order to tackle various topologies making our approach work on a diverse set of supercomputers, we developed a generic C++ interface to implement our data aggregation method for use on any system. Listing 2 presents the main function prototypes to implement to take advantage of a topology-aware aggregator placement. 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 a hardware-specific MPI extension (MPIX library [30]) offers a set of functions providing information such as the distance to the I/O node (`MPIX_IO_distance`) while on a Cray XC40 machine 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

[2] `http://memkind.github.io/memkind/`

Listing 2: Function prototypes for network interconnect

```
int networkBandwidth                (int level);
int networkLatency                  ();
int networkDistanceToIONode         (int rank, int IONode);
int networkDistanceBetweenRanks     (int srcRank, int destRank);
```

application.

4.2. Architecture-Aware Aggregators Placement

The second main contribution of this work on data aggregation concerns the aggregators placement policy. The various implementations of the MPI standard propose a set of aggregators mapping strategies for the two-phase I/O scheme. For example, in MPICH [31] 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. We propose in TAPIOCA a topology-aware approach for aggregators placement. Also, while existing methods select a subset of nodes to gather chunks of data, we consider a set of available memory banks on nodes and chose among those tiers the ones fulfilling the persistency and performance requirements. For instance, if the number of aggregators is equal to the number of nodes (i.e. one aggregator per node), we can locally aggregate data on the fastest available memory tier. In case we have more than one node sending data to an aggregator, the I/O bandwidth and the latency will be probably bounded by the performance of the network interconnect. Thus, a memory tier with enough capacity is sufficient. Another criteria we include in our model concerns data persistency. A workflow including *in-situ* analysis, for example, may need temporary persistent local data.

Therefore, our strategy involves considering the topology of the target system and the memory/storage requirements in an objective function in order to determine a near-optimal aggregator placement minimizing 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_M: The set of heterogeneous memory banks present in the partition and fulfilling the persistency requirements;
- $A \in V_M$: A memory tier able to aggregate data, chosen among the available memory banks;
- Cap_A: The capacity of a memory tier A
- T: The target memory, usually a file system;
- N_{buff}: The number of aggregation buffers;
- S_{buff}: The aggregation buffer size;
- $\omega(u, v)$: The amount of data to move from one memory bank u to another v with $u, v \in V_M$;
- $d(u, v)$: The distance between memory banks u and v (hops or bus) with $u, v \in V_M$;
- l: The latency such as $l = max(l_{network}, l_v)$ with $v \in V_M$;
- $B_{u \to v}$: The bandwidth from memory bank u to v with $u, v \in V_M$, such as $B_{u \to v} = min(B_{network}, B_u, B_v)$.

4.2.1. Memory Requirements

First, the selected aggregator has to fulfill a memory capacity condition. The memory bank chosen to aggregate data has to have a capacity greater or equal than the size needed for the aggregation buffers. We consider two cases: with and without a need of persistency. If the aggregated data needs to be persistent in memory, the memory capacity has to be at least the sum of the data produced for an aggregator.

$$Cap_A \geq \sum_{u \in V_M, u \neq A} \omega(u, A)$$

However, if persistency is not necessary, the memory capacity must be able to contain the number of buffers required for aggregation. More formally, the memory capacity has to be such as:

$$Cap_A \geq N_{buff} \times S_{buff}$$

Once this prerequisite has been met, we obtain a subset $V_m \subseteq V_M$ containing the aggregators candidates from the set of the memory banks. The next step consists of selecting the most appropriate memory tier providing the best I/O bandwidth among the candidates.

4.2.2. Objective function

To do so, we define two costs C_1 and C_2 as depicted in Figure 4. C_1 corresponds to the cost of aggregating data onto the aggregator. To compute this cost, we sum up the cost of each data producer i of sending an amount of data $\omega(i, A)$ to a memory bank A used for aggregation. This cost takes into account the slowest bandwidth involved as well as the worst latency.

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

C_2 is the cost of sending the aggregated data to the destination (typically, the storage system).

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

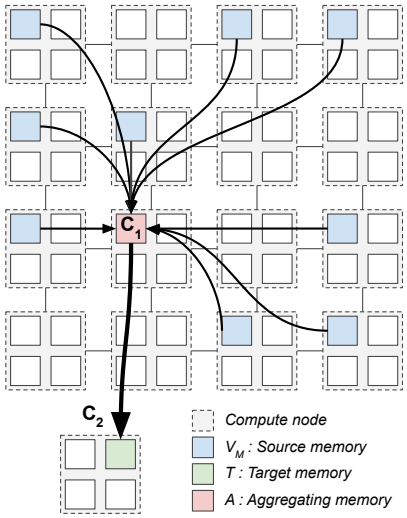

Figure 4: Objective function minimizing the communication costs to and from an aggregator.

Every node is in charge of computing the cost, for each of its local memory bank, of being an aggregator. Let's take as an example a node hosting three different types of memory complying with the persistency and capacity requirements mentioned previously. Three pairs of $\{C_1, C_2\}$ will be computed, one for each tier.

To determine the near-optimal location for data aggregation, we find out the minimal value of the sum of these two costs among the elements of V_m. More formally, our objective function is:

$$ArchAware(A) = min\,(C_1 + C_2)$$

A call to `MPI_Allreduce` across a partition with the `MPI_MINLOC` parameter enables our algorithm to choose as an aggregator the process with the minimal cost. Hence, for each partition an aggregator is elected.

4.2.3. Toy Example

Figure 5 illustrates our model with four processes that need to collectively write data on a parallel file system (PFS). We consider that each process is located on a different node. Two memory banks within a node are separated by one hop while the distance between nodes is noted on the links (white circles). Each node hosts two types of memory in addition to the main memory (DRAM): a high-bandwidth memory (HBM) and a HDD-based non-volatile memory (NVR). The source of the data is the DRAM (blue boxes) while the destination is a Lustre file system (green box). There is no need for intermediate persistency. Processes P0, P1, P2 and P3 respectively produce 10MB, 50MB, 20MB and 5MB. Based on vendors values, we set in Table 1 the latency, bandwidth, capacity and level of persistency of the available tiers of memory and the interconnect network for this toy example.

Table 2 shows, for each process, the cost of aggregating data on its local available tiers of memory. Our model shows

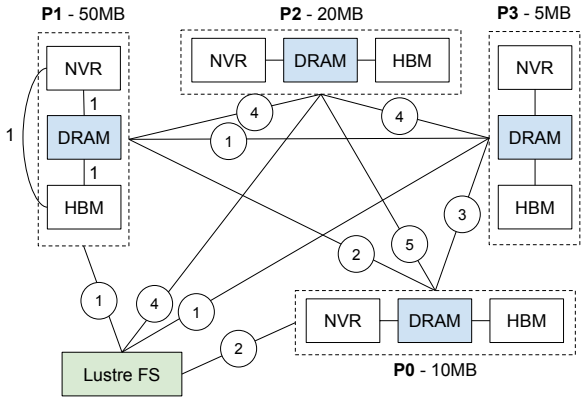

Figure 5: Toy examples of four processes collectively writing data on a Lustre file system through a data aggregation process.

Value#	HBM	DRAM	NVR	Network
Latency (ms)	10	20	100	30
Bandwidth (GBps)	180	90	0.15	12.5
Capacity (GB)	16	192	128	N/A
Persistency	No	No	job lifetime	N/A

Table 1: Memory and network capabilities based on vendors information

that the most advantageous location for aggregation is the high-bandwidth memory available on the node hosting process $P1$. We can notice that the difference between aggregation on HBM and DRAM is negligible. We observed this result with real experiments on a supercomputer equipped with those types of memory. Likewise, this behavior has also been observed in a related work [32]. Finally, Figure 6 depicts the decision taken by TAPIOCA for the aggregator selection.

P#	$\omega(i, A)$	HBM	DRAM	NVR
0	10	0.593	0.603	2.350
1	50	**0.470**	0.480	2.020
2	20	0.742	0.752	2.710
3	5	0.503	0.513	2.120

Table 2: For each process, according to the amount of data produced (ω) and the network and memory information, sum of the aggregation cost $Cost_A$ and the I/O cost $Cost_T$.

It has to be noted that the aggregation memory can be also defined by the user through an environment variable (*TAPIOCA_AGGRTIER*). When using this method, the environment variable can be set to any memory tier implemented in our memory abstraction. The aggregators location is then computed according to the only topology information.

4.3. Advanced Data Aggregation Algorithms

In this section we show how to use TAPIOCA in applications to perform collective I/O operations. Then we detail our architecture-aware implementation of read and write calls taking advantage of advanced techniques such as one-sided communication and multi-buffering.

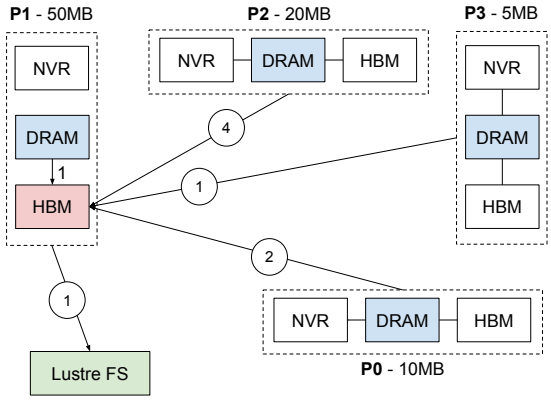

Figure 6: Decision taken by TAPIOCA for selecting the most appropriate aggregator given the initial state described in Figure 5.

4.3.1. Data Pattern Awareness

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 is the key to better scheduling I/O and to reduce the idle time for all the MPI tasks. As an example, Algorithm 1 describes the collective MPI I/O calls needed for a set of 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. Each call to `MPI_File_write_at_all` is a collective operation independent of the next calls.

Algorithm 1: Collective MPI I/O writes.

```
1  n ← 5;
2  x[n], y[n], z[n];
3  offset ← rank × 3 × n;
5
6  MPI_File_write_at_all (f, offset, x, n, type, status);
7  offset ← offset + n ;
9
10 MPI_File_write_at_all (f, offset, y, n, type, status);
11 offset ← offset + n;
13
14 MPI_File_write_at_all (f, offset, z, n, type, status);
```

With TAPIOCA, application developers have to describe the upcoming writes. This description contains nothing more than what is already known and requires less than a dozen lines of code. Algorithm 2 is the TAPIOCA version of Algorithm 1. 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, TAPIOCA is initialized with this information. This phase enabled our library to schedule the aggregation phase in order to completely fill an aggregator buffer before flushing it to the target. Figure 7 gives another perspective of what happens when performing this write phase with MPI I/O and TAPIOCA. In our example, MPI I/O has to flush three almost empty buffers in file while TAPIOCA can aggregate all the data. Moreover, TAPIOCA also takes advantage of buffers pipelining to further optimize the aggregation and I/O phases.

Algorithm 2: Collective TAPIOCA writes.

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

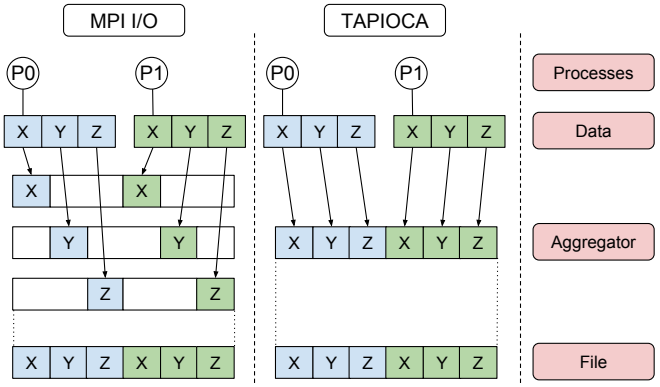

Figure 7: Calling three collective writes for an array of structure data layout with MPI I/O and TAPIOCA.

4.3.2. Buffers Pipelining

In order to optimize both the aggregation phase and the I/O phase, each aggregator manages at least two buffers. Therefore, while data is aggregated into a buffer, another one can be flushed into the target. In our implementation, 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 target (usually a storage system) making themselves available for other operations. In this way the aggregators are able to flush a full buffer while receiving data into another one. This loop is performed as many times as necessary

to process the data. The buffers used by the aggregators to stage data are allocated as a multiple of the target file-system block size to avoid lock penalties during the I/O phase. As depicted in Figure 8, a series of experiments in which we ran a simple benchmark from 2048 BG/Q nodes on a GPFS file-system (each process writes the same chunk size to different offsets of a single shared file) helped motivate this choice, although this behavior is known.

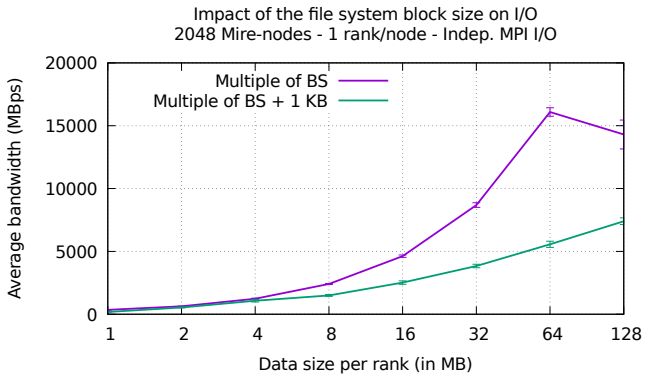

Figure 8: Benchmark measuring the impact of the file-system block size for write operations.

Each instance of a buffer filling and flushing itself is called a *round*. A *global round* is equivalent to a round performed by the same buffer on all the aggregators.

4.3.3. TAPIOCA's Collective Write and Read Algorithms

Using the contributions of the previous sections, we present here our write and read algorithms implemented in TAPIOCA and executed by the aggregation processes selected thanks to our cost model presented in Section 4.2.

Algorithm 3 details the write method implemented in our library. For each call to `TAPIOCA_Write`, we retrieve information computed during the initialization phase such as the number of aggergation buffers, the round number, the target aggregator, the amount of data to write during this round and the aggregator buffer to put data in (lines 6 to 10). Then, the *while* loop starting from line 13 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 (I/O phase). Line 21 just puts the data into the target buffer by way of a one-sided operation (Aggregation phase). If the process has written all its data, it enters a portion of code similar to the one starting from line 13. Else, we recursively call this `TAPIOCA_Write` function again while updating the function parameters.

We present in Algorithm 4 our read procedure. We can mainly distinguish four blocks in this algorithm. From line 15 to 21, we perform a first synchronization of the processes involved in the read operation. During this phase, the processes chosen to act as aggregators read from the input file a chunk of data whose size is the size of an aggregation buffer (I/O phase).

Algorithm 3: TAPIOCA Write Algorithm

```
1  GlobalRound ← 0;
2  TotalRound ← ComputeNumberOfRounds (datasize);
4
5  Function TAPIOCA_Write
     (f, offset, data, size, type, status)
6      round ← GetRound();
7      aggr ← GetAggregatorRank();
8      chunkSize ← GetRoundSize(round);
9      buffCount ← GetBuffCount();
10     buffId ← globalRound % buffCount;
12
13     while round ≠ globalRound do
14         Fence ();
15         if I am an aggregator then
16             iFlush_Buffer (buffId);
17         globalRound ← globalRound + 1;
18         buffId ← globalRound % buffCount;
20
21     RMA_Put (data, chunkSize, offset, aggr,
         buffId);
23
24     if chunkSize = size then
25         while globalRound ≠ TotalRounds do
26             Fence ();
27             if I am an aggregator then
28                 iFlush_Buffer (buffId);
29             globalRound ← globalRound + 1;
30             buffId ← globalRound % buffCount;
31     else
32         TAPIOCA_Write
             (f, offset + chunkSize, data +
             roundSize, size − chunkSize, type, status);
```

From line 24 to 31, this data is distributed from the aggregators to the other processes. The processes passing this conditional block carry out a RMA operation to get data from the appropriate aggregation buffer (aggregation phase, line 34). The last block, from line 37 to the end, is quite similar to the second block. The processes whose data has been fully retrieved, get stuck in a waiting loop, while the others recursively call the read function.

Algorithm 4: TAPIOCA Read Algorithm

```
1  GlobalRound ← 0;
2  ReadRound ← 0;
3  TotalRound ← ComputeNumberOfRounds (datasize);
5
6  Function TAPIOCA_Read
     (f, offset, data, size, type, status)
7      round ← GetRound();
8      aggr ← GetAggregatorRank();
9      chunkSize ← GetRoundSize(round);
10     buffCount ← GetBuffCount();
11     buffId ← globalRound % buffCount;
12     readId ← readRound % buffCount;
14
15     if firstRead then
16         if I am an aggregator then
17             Pull_Buffer (readId);
18             readRound ← readRound + 1;
19             readId ← readRound % buffCount;
21         Fence ();
23
24     while round ≠ globalRound do
25         if I am an aggregator AND
             readRound < TotalRounds then
26             Pull_Buffer (readId);
27             readRound ← readRound + 1;
28             readId ← readRound % buffCount;
29         Fence ();
30         globalRound ← globalRound + 1;
31         buffId ← globalRound % buffCount;
33
34     RMA_Get (data, chunkSize, offset, aggr,
         buffId);
36
37     if chunkSize = size then
38         while globalRound ≠ TotalRounds do
39             if I am an aggregator AND
                 readRound < TotalRounds then
40                 Pull_Buffer (readId);
41                 readRound ← readRound + 1;
42                 readId ← readRound % buffCount;
43             Fence ();
44             globalRound ← globalRound + 1;
45             buffId ← globalRound % buffCount;
46     else
47         TAPIOCA_Read (f, offset + chunkSize, data +
             roundSize, size − chunkSize, type, status);
```

5. Evaluation

To validate our approach, we ran a large series of experiments on Mira and Theta, two leadership-class supercomputers at Argonne National Laboratorywhich have been decommissioned respectively in 2019 and 2023. We also used Cooley, a mid-scale visualisation cluster, to highlight the portability of our method. TAPIOCA was assessed on I/O benchmarks and on two I/O kernels of large-scale applications: a cosmological simulation and a computational fluid dynamics (CFD) code.

In this section, we first describe the three testbeds we carried out our experiments on. Then, we demonstrate in 5.2 the impact of user-defined parameters on collective I/O operations. This step calibrates TAPIOCA and MPI I/O for a fair comparison. Finally, starting from Section 5.3 we present a comparative study of TAPIOCA and MPI I/O on diverse use-cases.

Table 3 summarizes the experimental setup used to evaluate our architecture-aware data aggregation technique.

5.1. Testbeds

5.1.1. Mira

Mira is a 10 PetaFLOPS IBM BG/Q supercomputer ranked in the top ten of the Top500 ranking for years, until June 2017 (see Figure 9). 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 [8] 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).

5.1.2. Theta

Theta is a 11.7 PetaFLOPS Cray XC40 supercomputer. This architecture (see Figure 10) 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 (HBM), can be used as an additional cache or as a high-speed allocatable

Table 3: Experimental Setup

HPC Systems	Cray XC40, IBM BG/Q, Haswell-based cluster
Comparison	TAPIOCA, MPI-IO
Workloads	IOR Synthetic I/O benchmarks IO kernel of a cosmological application (HACC) IO kernel of a direct numerical simulation (S3D)
Operations	Write and read with various subfiling techniques
Memory, Storage	DDR: Main Memory HBM: High-bandwidth memory NVR: NVRAM, either a on-node SSD or HDD PFS: Parallel file-system (Lustre or GPFS)

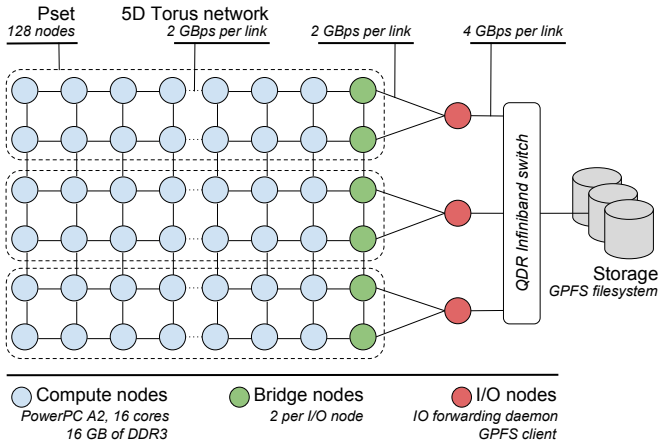

Figure 9: IBM BG/Q architecture.

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 [33, 9]. Figure 11 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 provide a way to know how the data is distributed on LNET nodes. It explains why aggregators placement on this platform do not take the I/O phase into account.

5.1.3. Cooley

Cooley, is a Haswell-based analysis and visualization cluster featuring 126 Intel Haswell E5-2620 nodes, each with 12 cores, 384GB of memory and a local hard-disk drive (HDD). The 27 PB of shared storage are managed with a GPFS filesystem. The interconnect is a 56Gbps FDR Infiniband CLOS network. We limited our experiments on this cluster to evaluating the portability of our library and abstraction layer.

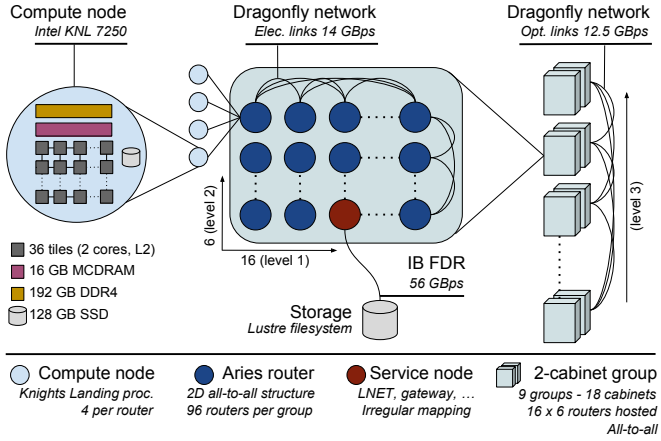

Figure 10: Cray XC40 architecture.

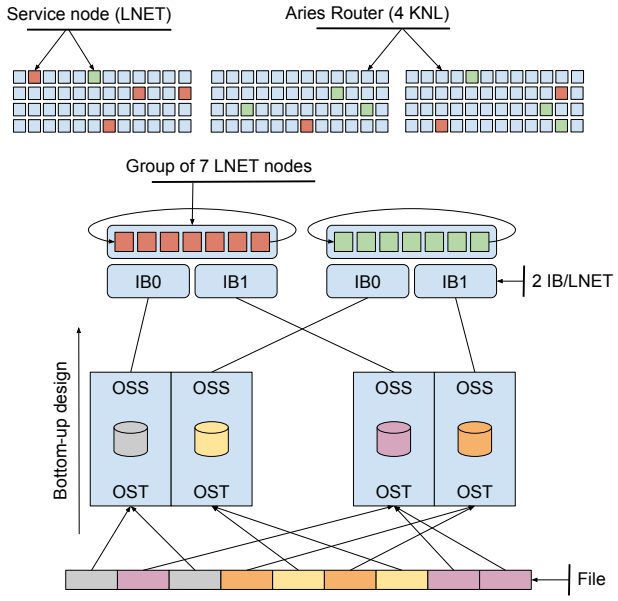

Figure 11: Storage system on Theta managed by Lustre.

5.2. Calibration of Collective I/O Operations 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 Mira and Theta a baseline I/O bandwidth with the default parameters and an optimized run with I/O tuning. This first study allows to present a fair comparison between TAPIOCA and MPI I/O in the rest of this paper.

To evaluate I/O performance, we ran IOR, a popular I/O benchmark [34]. 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 while MPI processes managed a single shared file on Theta.

On Mira (Figure 12), 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.

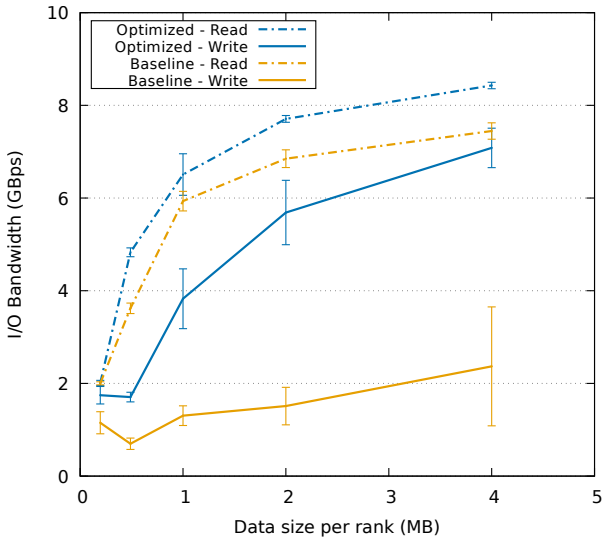

Figure 12: I/O bandwidth achieved with IOR benchmark on 512 Mira nodes, 16 ranks per node, with and without user-defined optimizations.

Figure 13 depicts the same experiment on Theta. On this platform, IOR with the default parameters revealed a read bandwidth of approximately 800 MBps while up to 36 GBps were reached with optimized parameters. The write bandwidth was increased from nearly 200 MBps to 10 GBps in the best case. The gap was substantial between these two scenarios. Indeed, by default on Theta's Lustre file-system, 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 increased the I/O bandwidth. As on Mira, two locking modes are available. Lock sharing set for collective operations reduced the lock contention and took 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 provided better I/O bandwidth.

This preliminary study also allowed to highlight a decisive correlation between aggregator buffer size set in TAPIOCA and stripe size of the Lustre file system. Table 4 shows the average I/O bandwidth achieved on 512 nodes and 16 ranks per node

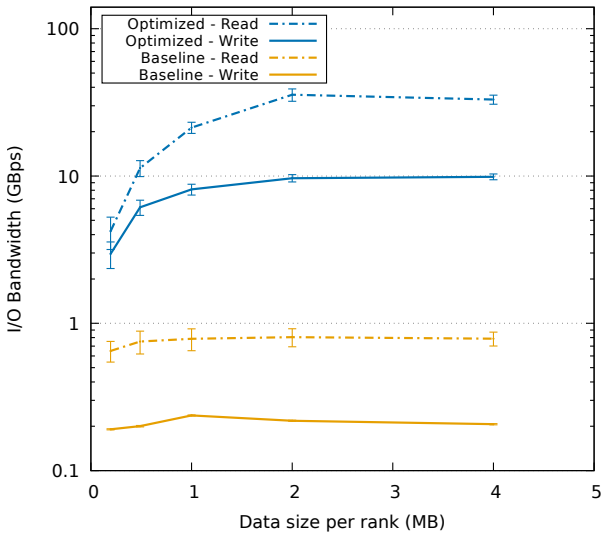

Figure 13: I/O bandwidth achieved with IOR benchmark on 512 nodes on Theta, 16 ranks per node, with and without user-defined optimizations. Log scale on y-axis.

with various aggregator buffer sizes and stripe sizes on a simple use-case: each MPI process writes 1MB of data into a single shared file. 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 could maintain a certain ratio. We observed that a 1:1 ratio—that is, an aggregator buffer size equal to the stripe size—gives the best performance.

Table 4: 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

For the rest of our experiments, and to ensure a fair comparison between TAPIOCA and MPI-IO, we configured each environment with the optimal parameters determined in this section.

5.3. 1D-Array

We first ran a series of experiments with a micro-benchmark called 1D-Array. In this code, every MPI process writes a contiguous piece of data in one or multiple shared files (subfiling) during a collective call. We used this benchmark to provide an initial assessment of TAPIOCA's full range of capabilities, namely architecture-aware aggregator placement, I/O scheduling, and the means to use any type of memory and storage level for aggregation. As Theta is our most recent architecture featuring multiple memory and storage tiers, we focused on this platform for this first analysis. We compared TAPIOCA with the MPI I/O implementation installed on Theta while varying the data distribution among the processes, the number of nodes involved in files read and written, and the aggregation memory tiers.

This micro-benchmark allocates one buffer per process filled with random values and collectively write/read it to/from the storage system. We tried out three different configurations for

the buffer size: every process allocates the same buffer or a random buffer size is chosen or the buffer sizes follow a normal distribution. To have a fair comparison, the data distributions were preserved between experiments with MPI-IO and TAPIOCA.

Figure 14 shows experiments on 128 Cray XC40 nodes while writing and reading data to a single shared file on the Lustre file system. We selected 48 aggregators (DRAM) for both MPI-IO and TAPIOCA. We carried out three use-cases: the first one with an array of 25K integers per process (100KB), the second one with a random distribution of the data among the processes (a value between 0KB and 100KB) and the last one with a normal distribution among the processes. Our approach outperformed MPI-IO on the three types of distributions. However, the performance gap was particularly significant with a random and a normal distribution seeing as the write bandwidth was respectively approximately 6 and 29 times higher while we read data 3 times faster.

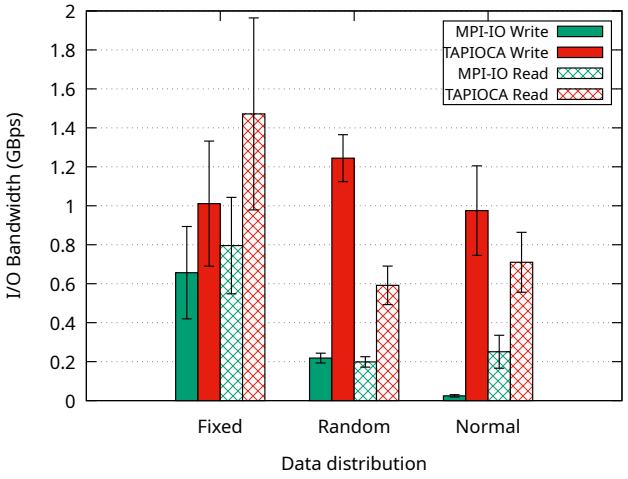

Figure 14: I/O bandwidth achieved with 1D-array from 128 Cray XC40 nodes while varying the data distribution. Data read/written into a single shared file on Lustre.

Performing I/O operations on a single shared file is known to often provide poor performance. Subfiling is usually preferred. Figure 15 presents the results we obtained on the same platform while performing subfiling, from one file per node (1:1) to one file per 8 nodes (1:8). In such a use-case, one aggregator was selected per group of nodes writing or reading the same file. Data aggregation was performed on DRAM while the destination of the data was the Lustre file system. Unlike MPI-IO, TAPIOCA allows to set the local SSD as a shared destination tier. The storage space is mapped into a memory space exposed to one-sided communication. We also ran experiments showing this feature. It has to be noted that the file created on each local SSD was temporary (allocation lifetime). We can conclude from these results that one file per node is the configuration offering the best I/O bandwidth for MPI-IO and our library. We also observe that setting the SSD as a destination provides better performance. However, this must be moderated by the fact that the volume of data read and written is small and that a cache effect undoubtedly comes into play. The "1:1" case was also evaluated with a random data distribution as shown in Table 5. Again, the best I/O performance was achieved with TAPIOCA except on the read case from the Lustre file system. We are still investigating the poor read bandwidth obtained in some of our experiments.

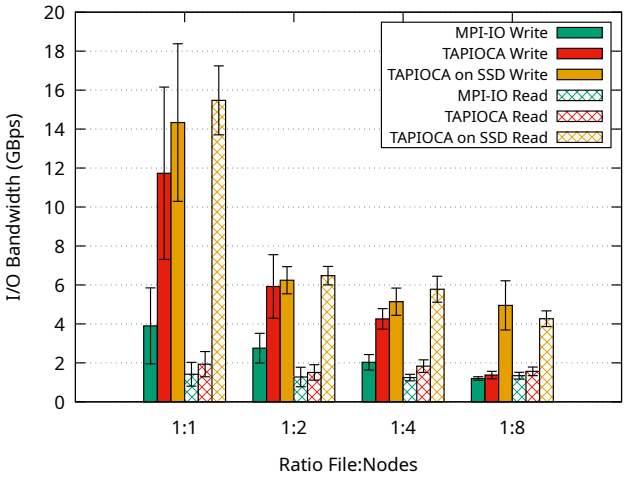

Figure 15: I/O bandwidth achieved with 1D-array from 128 Cray XC40 nodes while varying the number of nodes per file. The node-local SSD was also considered as a target.

Table 5: MPI-IO vs TAPIOCA, one file per node on Lustre and SSD (TAPIOCA only) with random data distribution

I/O Operation	**MPI-IO**	**TAPIOCA**	
		Lustre	**SSD**
Read Bw (GBps)	0.99	0.80	4.47
Write Bw (GBps)	2.46	5.89	4.32

Last, Table 6 gives the read and write I/O bandwidth achieved on the Lustre file system when performing data aggregation on the three tiers of memory available on the Cray system. In order to highlight the differences, we increased the data size per process to 1 MB. We first observed that the difference in performance was not significant between aggregation on DRAM and HBM. This experiment corroborates the cost model evaluation presented in Section 4.2. We can also notice the overhead due to the file mapping in memory (*mmap*) when aggregating data on the local SSD.

Table 6: Reading and writing one file per node on Lustre with TAPIOCA while aggregating on the three tiers of memory and storage available on nodes. 1MB read/written per process.

I/O Operation	**DRAM**	**HBM**	**SSD**
Read Bw (GBps)	8.96	8.24	7.80
Write Bw (GBps)	19.15	19.36	10.70

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 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) while only writing data, the reading phase providing similar results for both methods. Then we show the results on Theta with 1,024 and 2,048 nodes and 16 ranks per node (resp. 16K and 32K processes) for both read and write.

5.4.1. Mira

Figure 16 shows the results on 1,024 Mira nodes, with 16 ranks per node and one file per *Pset* as output. We compared our approach to MPI I/O on this platform with two data layouts: array of structures (AoS) and structure of arrays (SoA). For these experiments, we varied the data size per rank from 5K to 100K particles. We first note from the results 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 was achieved by our topology-aware strategy. We also note that we outperformed the default implementation even on large messages.

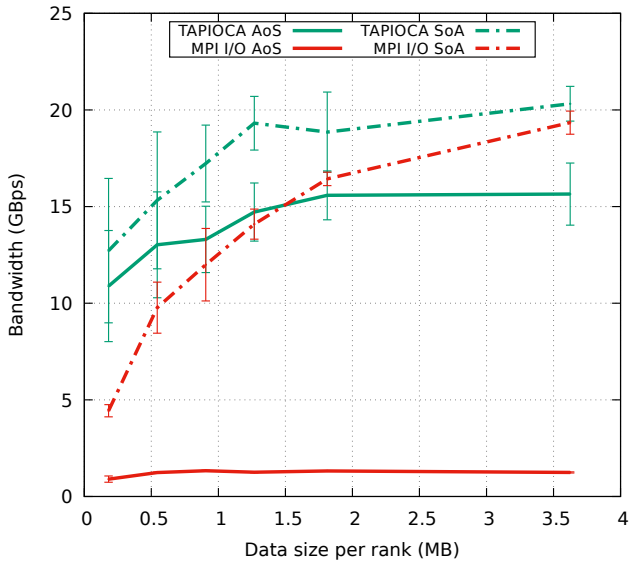

Figure 16: Write bandwidth achieved with HACC-IO on Mira by writing one file per *Pset* from 1,024 nodes (16 ranks/node). TAPIOCA: 16 aggregators per *Pset*, 16 MB for the aggregator buffer size.

Figure 17 presents experiments with the same configuration as the previous one except that we ran it on 4,096 Mira nodes. The behavior was similar, with the peak write bandwidth almost reached with TAPIOCA (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 decreased as the data size increased. In any case, the I/O performance was substantially improved for both AoS and SoA layouts.

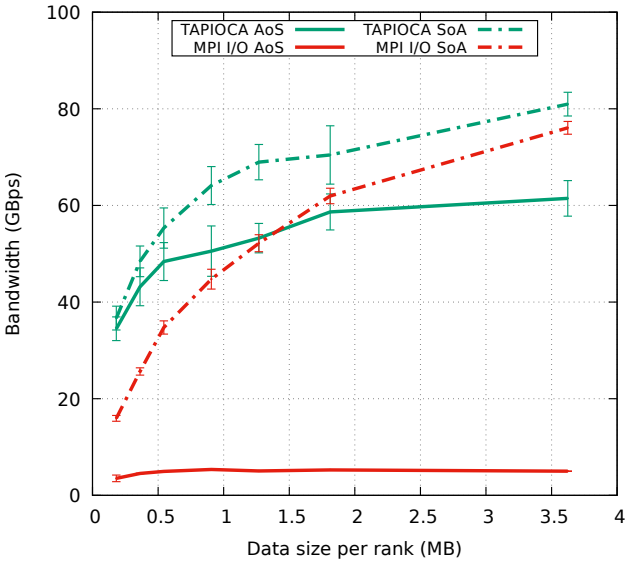

Figure 17: Write bandwidth achieved with HACC-IO on Mira by writing one file per *Pset* from 4,096 nodes (16 ranks/node). TAPIOCA: 16 aggregators per *Pset*, 16 MB aggregator buffer size.

5.4.2. Theta

Our experiments on Theta showed a good I/O performance gain as well. Figure 18 depicts the read and write bandwidth achieved on 1024 nodes on the Cray XC40 supercomputer while sharing a single file as output and varying the data size per process. This result highlights the performance improvement TAPIOCA can achieve on a standard workflow, from the application to a parallel file system. Data aggregation was performed on the DRAM in this set of experiments. On both read and write, TAPIOCA outperformed MPI-IO respectively by a factor of 5.4 and 13.8 with a 1 MB data size per process.

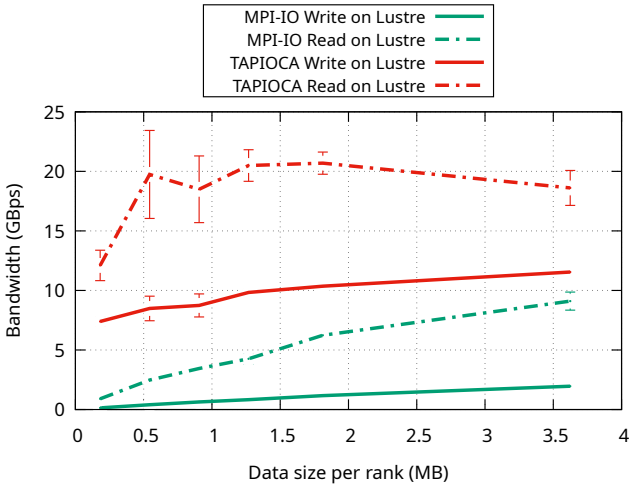

Figure 18: Read and write bandwidth achieved with HACC-IO from 1024 Cray XC40 nodes while writing into a single shared file on the Lustre file-system

As demonstrated in 5.3 and 5.4.1, subfiling is a key method to improve I/O bandwidth and reduce the proportion of the wall time spent in I/O. As shown in Figure 19, writing one file per

node on the parallel file system improved the performance up to 40 times with a large amount of data per process. On this case, MPI-IO and TAPIOCA offered I/O performance in the same confidence interval. As mentioned previously, whatever the subfiling granularity chosen, TAPIOCA is able to use the local SSD as a file destination (as well as an aggregation layer). Therefore, we included the results when writing and reading data to/from this storage layer. In this case, the I/O bandwidth was boosted in the range of 4 and 9 times when writing data and in the range of 6 and 8 when reading compared to the parallel file system.

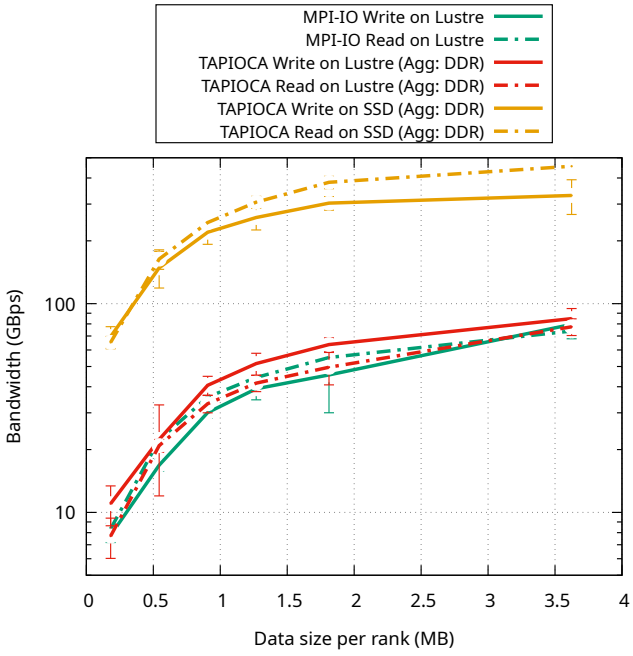

Figure 19: Read and write bandwidth achieved with HACC-IO from 1024 Cray XC40 nodes while writing one file per node on the Lustre file-system and on the local SSD (TAPIOCA only). Log-scale on y-axis.

To extend the analysis of this use-case, we ran a weak scaling study of the previous experiment as depicted in Figure 20. Here, every process managed 1MB of data. The aggregation was performed on the DRAM of each aggregator and the target for output data was set to the Lustre parallel file system and the on-node SSD. This last method revealed a very strong scalability as the I/O performance attained increased by more or less 50% every time we doubled the number of compute nodes.

Eventually, thanks to the memory abstraction we have proposed (see Section 4.1), we carried out experiments with data aggregation on the high-bandwidth memory available on the compute nodes. For the purposes of this experiment, we chose to take the case with the best performance so far, i.e. one file written to the local SSD per node. This choice was motivated by the fact that in other configurations (writing to the Lustre file system, for example), our model shows that performance is limited by the network, regardless of the aggregation layer used. Thereby, Figure 21 compares an execution with aggregation on DRAM and on HBM while writing and reading one file per node on the node-local SSD. Even so, the performance gap between the two memory banks and the SSDs remains wide. The two-phase I/O operations is still bounded by the SSD's bandwidth as expected, showing no difference between data agregation on the DRAM or on the HBM.

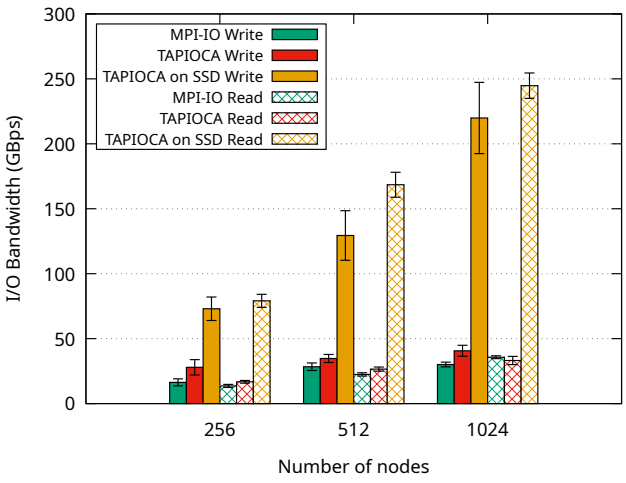

Figure 20: HACC-IO, one file per Cray XC40 node on Lustre and local SSD. 1MB per process, varying the number of nodes.

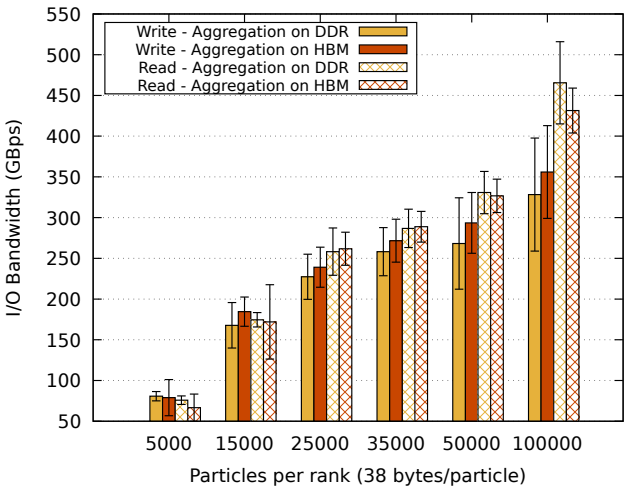

Figure 21: HACC-IO on 1024 Cray XC40 nodes, one file per node on local SSD. Comparison of the aggregation on DDR and on HBM.

Workflow. HACC is a very large-scale simulation code generating data that can be analyzed or visualized, in real time if possible. We propose here a situation of this application in a workflow as described in Figure 22 that can be seamlessly implemented with TAPIOCA. The workflow might be either a single application performing write (simulation) and read (analysis) operations consecutively like an in-situ analysis with co-located processes or two different applications running during the same allocation as the data is persistent on SSD for the allocation lifetime. As described in section 4, TAPIOCA allows to use on-node local SSD as an aggregation layer. This task is done by mapping a file created for the occasion on the SSD to

the DRAM of the node. A MPI window then exhibits this buffer to local and remote nodes.

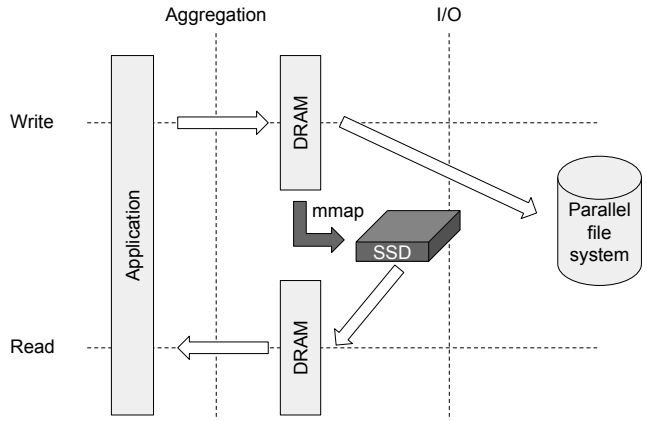

Figure 22: Write/Read workflow using TAPIOCA and SSDs as both an aggregation buffer (write) and a target (read).

Table 7 shows the best I/O bandwidth achieved for write and read as well as the best time to solution for the whole workflow. The performance variation is based on the MPI-IO case and the TAPIOCA case using SSD. The first result row is for information purpose. We can see that the overhead due to the *mmap* system call is widely counterbalanced by the performance attained with the read operation. The total time to solution is reduced by 26.82%.

Table 7: Max. Write and Read bandwidth (GBps) and total I/O time achieved with and without aggregation on SSD

	Agg. Tier	Write	Read	I/O time
TAPIOCA	DDR	47.50	38.92	693.88 ms
MPI-IO	DDR	32.95	37.74	843.73 ms
TAPIOCA	SSD	26.88	227.22	617.46 ms
Variation		-36.10%	+446.94%	-26.82%

5.4.3. Cooley

To assess the portability of our architecture-aware data aggregation algorithm, we ran experiments with HACC-IO on Cooley, a 64-node Haswell-based visualization cluster. To take advantage of the features we proposed in our data aggregation library on another platform, there is no need to modify the application. Only the compilation process and an implementation of the memory and network abstraction are necessary.

The testbed we targeted is not designed for intensive I/O. In addition, the on-node disks are hard disk drives with poor performance. However, this machine is suitable for workflows combining simulation and visualization as presented is Figure 22. Beyond the I/O performance, these experiments are more a proof of concept.

We show in Table 8 the results obtained with the workflow described in Figure 22. To control the impact of GPFS caching, we interleaved random I/O with HACC-IO write and read runs. We can notice that the overhead caused by local aggregation on HDD is very low. Again, the read bandwidth is significantly increased while the overall I/O time is reduced by more than 12% on this cluster.

Table 8: Max. Write and Read bandwidth (GBps) and total I/O time achieved with and without aggregation on local HDD

	Agg. Tier	Write	Read	I/O Time
TAPIOCA	DDR	6.60	38.80	123.41 ms
MPI-IO	DDR	6.02	17.46	155.40 ms
TAPIOCA	HDD	5.97	35.86	135.86 ms
Variation		-0.83%	+105.38%	-12.57%

5.5. S3D-IO

S3D [35] is a state-of-the-art direct numerical simulation (DNS) code written in Fortan and MPI, in the field of computational fluid dynamics (CFD). S3D focuses on turbulence-chemistry interactions in combustion. The DNS approach aims to address small domain problems to calibrate physical models for macro-scale CFD simulations. S3D is based on a 3D domain decomposition distributed across the MPI processes. In terms of I/O, a new single shared file is collectively written every n timesteps. The state of each element of the studied domain is stored following an array of structure data layout. The file as output is used both as a checkpoint in case of failure and for data analysis. S3D-IO is a version of the S3D production code whose physics modules have been removed. The memory arrangement as well as the I/O routines have been kept though.

We implemented a module in S3D-IO using TAPIOCA for managing I/O operations. For these experiments, we let our architecture-aware algorithm described in Section 4.2 automatically decide the most appropriate tiers of memory for data aggregation among the compute nodes.

We first present in Table 9 a typical use-case of S3D with 134 and 537 millions grid points respectively distributed on 256 and 1024 nodes on the Cray XC40 system (16 ranks per node). We set the number of aggregators to 96 on 256 nodes and 384 on 1024 nodes for both MPI-IO and TAPIOCA. For this use-case, our aggregator placement algorithm selected the HBM as an aggregation layer for all the 96 aggregating nodes. We can see that on the two problem sizes, TAPIOCA significantly outperforms MPI-IO. When running on 1024 nodes, the I/O bandwidth is multiplied by 3.

Table 9: Maximum write bandwidth (GBps) achieved with aggregation performed on HBM using the TAPIOCA library.

	Points	Size	256 nodes	1024 nodes
MPI-IO	134M	160 GB	3.02 GBps	4.42 GBps
TAPIOCA	537M	640 GB	4.86 GBps	13.75 GBps
Variation	N/A	N/A	+60.93%	+210.91%

In order to emphasize the adaptability of our approach, we ran another series of experiments on 256 nodes with 134 millions grid points while artificially decreasing the capacity of the high-bandwidth memory then the DRAM to 32MB. At the same time, we set the number of aggregation buffers to 3 and their size to 16 MB (so 48MB total, above the memory capacity(. The goal was to show the behavior of TAPIOCA in case of the

fastest tier of memory available does not have enough space for aggregated data. Table 10 presents the results. The capacity requirement described in Section 4.2 not being fulfilled, the second then the third fastest memory tier are selected. In the third scenario, data is aggregated on the node-local SSD, offering poor I/O bandwidth compared to HBM or DRAM. However, the application can still be carried out.

Table 10: Maximum write bandwidth (GBps) while artificially reducing the memory capacity of the HBM then the DRAM. For each run, the grey box corresponds to the memory tier selected for aggregation by TAPIOCA.

Run	HBM	DDR	NVR	Bandwidth	Std dev.
1	16 GB	192 GB	128 GB	4.86 GBps	0.39 GBps
2	↓ 32 MB	192 GB	128 GB	4.90 GBps	0.43 GBps
3	↓ 32 MB	↓ 32 MB	128 GB	2.98 GBps	0.15 GBps

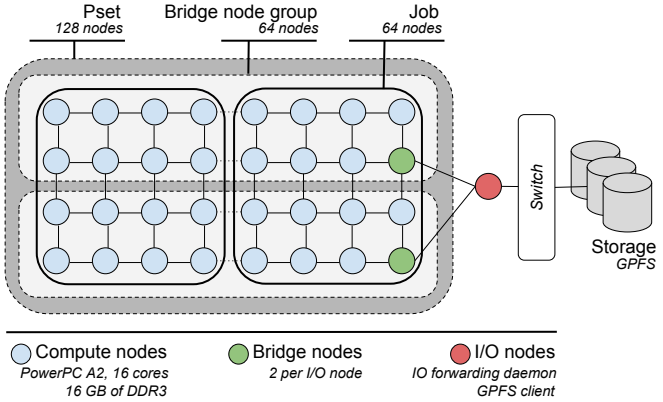

Figure 23: Job partitioning on a *Pset* on Mira to demonstrate the impact of I/O interference on performance.

6. Discussion

In this section, we discuss several challenges, including those faced while pursuing this research. These highlight the need for better co-design between hardware and software stacks, as well as the need for domain-driven research for I/O data management.

6.1. Impact of network interference

While carrying out experiments with our I/O library, we observed a certain variability in the I/O bandwidth measurements. This instability was due to I/O interference from other conncurently running jobs. On Mira, a set of I/O nodes is isolated only as part of a 512-nodes allocation.

To emphasize this behavior, we ran controlled benchmark tests using one *Pset* (128 nodes compute nodes, two bridge nodes and one I/O node). Our tests were run to highlight the impact of I/O interference. In one case, we ran a single I/O intensive HACC-IO job on 64 of the 128 nodes, while leaving the other 64 nodes idle. This case eliminated interference on the bridge and I/O nodes. In the other case, we ran the same I/O intense job on 64 of the 128 nodes, while simultaneously running jobs of varying I/O intensity on the other 64 nodes. Node allocation was distributed such that each 64 node job used 32 nodes per bridge node. This configuration corresponded to the default distribution on BG/Q. Figure 23 depicts a 5D Torus flattened on 2 dimensions and the aforementioned jobs partitioning.

Table 11 shows the mean I/O bandwidth achieved with HACC-IO with and without interference. A single I/O intensive HACC-IO job running on 64 nodes sharing two bridge nodes can reach more than 60% of the peak I/O bandwidth. However, the performance is decreased by 13% when a concurrent job is running on the same *Pset*. We can also notice a rise in variability (standard deviation) of 37.5%. This result demonstrates the need for a good understanding of the underlying topology and better ways to leverage this knowledge by conducting more research in the domain of topology-aware resource allocation or I/O contention management (I/O scheduling or I/O priority). On BG/Q for instance, we have learnt that the minimal unit to consider for a node allocation is a block of four *Psets* (512 nodes) to reduce as much as possible the impact of I/O interference and ensure a good reproducibility.

Table 11: Mean I/O bandwidth achieved with HACC-IO (2 MB per rank) through our I/O library with and without interference. Concurrent jobs have variable I/O intensity (0.2 MB to 4 MB per rank).

	HACC-IO		Other	
	Average	Std_dev	Average	Std_dev
no-interference	2.20 GBps	0.10 GBps	N/A	N/A
interference	1.92 GBps	0.16 GBps	1.15 GBps	0.35 GBps

6.2. Architecture-Aware Limitations

This work has highlighted some of the limitations of the "architecture-aware" approach. For example, on Theta, the lack of information on the placement of LNET nodes (the nodes through which I/O transits to the Lustre file system) makes it challenging to take advantage of the topology for the I/O phase. Similarly, the packet routing algorithms on the network may not be known or may be "adaptive" as is the case on the Cray machine in comparison to systems with static routing such as on BG/Q. To fully making these approaches, we need to deal with the uncertainties in the routing algorithms and require capabilities for system introspection.

Given the increasing adoption of heterogenous memory and storage on HPC systems, at a node-level, rack-level and system-level, architecture-aware methods such as the one implemented in TAPIOCA will be needed to fully realize the performance potentials. Similarly, the emergence of disaggregation technologies such as CXL (Compute Express Link) [36] should make architecture-aware placement even more important.

More generally, to address such issues, we advocate for better co-design between the hardware and software stack to provide feedback from the underlying architecture with accurate tools and libraries.

6.3. Potential Applications

The two-phase I/O algorithm is just one use-case among others to illustrate our contribution of coupling data between

computation and storage. Our approaches are widely applicable to data coupling between various stages of a scientific workflow and will enable the efficient movement of data between the stages. This is a critical component for science workflows combining simulations, analysis, AI, among others. As we are witnessing these workflows being executed on heterogeneous systems with diverse memory, storage, compute and networking characteristics, approaches such as TAPIOCA, can provide a holistic data-movement acceleration.

7. Conclusion

In this paper, we have introduced TAPIOCA, a data aggregation algorithm designed to leverage supercomputer's architecture effectively. Specifically, we have demonstrated how an architectural abstraction, coupled with an aggregator's placement model, can alleviate the I/O bottleneck across present and future large-scale systems. Our library has exhibited significant performance improvements across typical I/O workloads and more intricate workflows that express diverse I/O requirements. Our assessment on benchmarks and on two real applications narrowed down to the I/O phases, have demonstrated our ability to outperform MPI-IO while offering extended flexibility. We conducted experiments with up to 16K processes on three systems at Argonne National Laboratory including Theta, a 11.69 PetaFLOPS Cray XC40 system and Mira, a 10 PetaFLOPS IBM BG/Q supercomputer. In particular, on an example of a typical "simulation + analysis" workflow configuration, we have demonstrated an execution time saving of the order of 26%, while transparently taking advantage of local storage resources.

In future endeavors, we aim to strengthen this approach, beginning with an in-depth exploration of the influence of input parameters on the cost of data movement. Analyzing the data access pattern, for instance, could allow a better characterization of applications and deliver performance improvements. Another future direction concerns the number of aggregators. As mentioned in Section 5, the way we determine the appropriate number of aggregators is empirical or based on the system's default value. Therefore, we plan to implement a contention model determining the number of aggregators given the bandwidth degradation due to concurrent accesses on the aggregators, and the number of streams required to achieve high I/O performance on the parallel file-system. In the longer term, we will extend this result from the "n to 1" paradigm, i.e. a fixed set of processes sending data to a single aggregator, to "n to m" models where processes can supply data to several aggregators. A contention model in this case will be all the more decisive. Finally, from an evaluation perspective, we will study how TAPIOCA performs on AI workloads. The read-intensive access pattern of these applications is a relevant use case for our library.

More than just optimizing TAPIOCA, we also plan to delve into multi-level data aggregation. While our library currently allows the identification or explicit specification of a single aggregation layer, adopting a multi-level approach could undoubtedly benefit various workloads, particularly in scenarios such as checkpointing. In that context, we also plan to extend our model beyond the two-phase I/O algorithm. In hybrid HPC/-Cloud systems, for example, data aggregation as a preamble to data movements between geo-distributed infrastructures is a key element for which our placement model can provide a solution.

Acknowledgment

This research has been funded in part by the NCSA-Inria-ANL-BSC-JSC-Riken-UTK Joint-Laboratory on Extreme Scale Computing (JLESC).

This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

References

[1] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, J. 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, ACM, New York, NY, USA, 2011, pp. 26:1–26:10. doi:10.1145/2063384.2063419.
URL http://doi.acm.org/10.1145/2063384.2063419

[2] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia, S. Byna, S. Farrell, D. Gursoy, C. Daley, V. Beckner, B. Van Straalen, D. Trebotich, C. Tull, G. H. Weber, N. J. Wright, K. Antypas, Prabhat, Accelerating science with the NERSC burst buffer early user program, in: CUG2016 Proceedings, 2016, best paper award, in press.

[3] M.-A. Vef, N. Moti, T. Süß, M. Tacke, T. Tocci, R. Nou, A. Miranda, T. Cortes, A. Brinkmann, Gekkofs—a temporary burst buffer file system for hpc applications, Journal of Computer Science and Technology 35 (1) (2020) 72–91. doi:10.1007/s11390-020-9797-6.
URL https://jcst.ict.ac.cn/en/article/doi/10.1007/s11390-020-9797-6

[4] T. Wang, K. Mohror, A. Moody, K. Sato, W. Yu, An ephemeral burst-buffer file system for scientific applications, in: SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 807–818. doi:10.1109/SC.2016.68.

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

[6] R. Thakur, W. Gropp, E. Lusk, A case for using MPI's derived datatypes to improve I/O performance, in: Proceedings of SC98: High Performance Networking and Computing, ACM Press, 1998.
URL http://www.mcs.anl.gov/~thakur/dtype/

[7] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, 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, HPDC '15, ACM, New York, NY, USA, 2015, pp. 33–44. doi:10.1145/2749246.2749269.
URL http://doi.acm.org/10.1145/2749246.2749269

[8] F. Schmuck, R. 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, USENIX Association, Berkeley, CA, USA, 2002.
URL http://dl.acm.org/citation.cfm?id=1083323.1083349

[9] Lustre filesystem website, http://lustre.org/.

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

[11] J. M. del Rosario, R. Bordawekar, A. Choudhary, Improved parallel I/O via a two-phase run-time access strategy, SIGARCH Comput. Archit. News 21 (5) (1993) 31–38. doi:10.1145/165660.165667.
URL http://doi.acm.org/10.1145/165660.165667

[12] R. Thakur, W. Gropp, E. Lusk, Optimizing noncontiguous accesses in MPI I/O, Parallel Comput. 28 (1) (2002) 83–105. doi:10.1016/S0167-8191(01)00129-6.
URL http://dx.doi.org/10.1016/S0167-8191(01)00129-6

[13] R. Thakur, W. Gropp, E. 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, IEEE Computer Society, Washington, DC, USA, 1999, pp. 182–.
URL http://dl.acm.org/citation.cfm?id=795668.796733

[14] R. Thakur, W. Gropp, E. 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, ACM, New York, NY, USA, 1999, pp. 23–32. doi:10.1145/301816.301826.
URL http://doi.acm.org/10.1145/301816.301826

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

[16] F. Tessier, V. Vishwanath, E. Jeannot, Tapioca: An i/o library for optimized topology-aware data aggregation on large-scale supercomputers, in: 2017 IEEE International Conference on Cluster Computing (CLUSTER), 2017, pp. 70–80. doi:10.1109/CLUSTER.2017.80.

[17] P. Malakar, V. Vishwanath, Hierarchical read–write optimizations for scientific applications with multi-variable structured datasets, International Journal of Parallel Programming 45 (1) (2017) 94–108. doi:10.1007/s10766-015-0388-z.
URL https://doi.org/10.1007/s10766-015-0388-z

[18] Y. Tsujita, K. Yoshinaga, A. Hori, M. Sato, M. Namiki, 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, 2014, pp. 232–235. doi:10.1109/PDP.2014.46.

[19] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, C. Jin, Flexible io and integration for scientific codes through the adaptable io system (adios), in: Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments, CLADE '08, Association for Computing Machinery, New York, NY, USA, 2008, p. 15–24. doi:10.1145/1383529.1383533.
URL https://doi.org/10.1145/1383529.1383533

[20] M. J. Brim, A. T. Moody, S.-H. Lim, R. Miller, S. Boehm, C. Stanavige, K. M. Mohror, S. Oral, Unifyfs: A user-level shared file system for unified access to distributed local storage, in: 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2023, pp. 290–300. doi:10.1109/IPDPS54959.2023.00037.

[21] M. Gossman, B. Nicolae, J. Calhoun, Modeling Multi-Threaded Aggregated I/O for Asynchronous Checkpointing on HPC Systems, in: ISPDC 2023: The 22nd International Symposium on Parallel and Distributed Computing, IEEE, Bucharest, Romania, 2023, pp. 101–105. doi:10.1109/ISPDC59212.2023.00021.
URL https://hal.science/hal-04343661

[22] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, F. Cappello, Veloc: Towards high performance adaptive asynchronous checkpointing at large scale, in: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2019, pp. 911–920. doi:10.1109/IPDPS.2019.00099.

[23] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Romanus, N. Podhorszki, S. Klasky, H. Kolla, J. Chen, R. Hager, C. S. Chang, M. Parashar, Exploring data staging across deep memory hierarchies for coupled data intensive simulation workflows, in: 2015 IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 1033–1042. doi:10.1109/IPDPS.2015.50.

[24] M. Dreher, T. Peterka, Decaf: Decoupled Dataflows for In Situ High-Performance Workflows, 2017. doi:10.2172/1372113.
URL http://www.osti.gov/scitech/servlets/purl/1372113

[25] M. Dreher, K. Sasikumar, S. Sankaranarayanan, T. Peterka, Manala: A flexible flow control library for asynchronous task communication, in: 2017 IEEE International Conference on Cluster Computing (CLUSTER), 2017, pp. 509–519. doi:10.1109/CLUSTER.2017.31.

[26] B. Dong, S. Byna, K. Wu, Prabhat, H. Johansen, J. N. Johnson, N. Keen, Data elevator: Low-contention data movement in hierarchical storage system, in: 2016 IEEE 23rd International Conference on High Performance Computing (HiPC), 2016, pp. 152–161. doi:10.1109/HiPC.2016.026.

[27] J. Kunkel, E. Betke, An mpi-io in-memory driver for non-volatile pooled memory of the kove xpd, in: J. M. Kunkel, R. Yokota, M. Taufer, J. Shalf (Eds.), High Performance Computing, Springer International Publishing, Cham, 2017, pp. 679–690.

[28] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, Hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications, in: Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010), IEEE Computer Society Press, Pisa, Italia, 2010.
URL http://hal.inria.fr/inria-00429889

[29] M. G. Venkata, F. Aderholdt, Z. Parchman, Sharp: Towards programming extreme-scale systems with hierarchical heterogeneous memory, in: 2017 46th International Conference on Parallel Processing Workshops (ICPPW), 2017, pp. 145–154. doi:10.1109/ICPPW.2017.32.

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

[31] 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, Springer-Verlag, London, UK, UK, 2002, pp. 7–.
URL http://dl.acm.org/citation.cfm?id=648139.749473

[32] J. Liu, Q. Koziol, H. Tang, F. Tessier, W. Bhimji, B. . Cook, B. Austin, S. Byna, B. Thakur, G. Lockwood, et al., Understanding the io performance gap between cori knl and haswell, in: Cray User Group Meeting, 2017.

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

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

[35] E. R. Hawkes, R. Sankaran, J. C. Sutherland, J. H. Chen, Direct numerical simulation of turbulent combustion: fundamental insights towards predictive models, Journal of Physics: Conference Series 16 (1) (2005) 65.
URL http://stacks.iop.org/1742-6596/16/i=1/a=009

[36] D. D. Sharma, Compute express link®: An open industry-standard interconnect enabling heterogeneous data-centric computing, in: 2022 IEEE Symposium on High-Performance Interconnects (HOTI), 2022, pp. 5–12. doi:10.1109/HOTI55740.2022.00017.