Optimizing Data Aggregation by Leveraging the Deep Memory Hierarchy on Large-scale Systems

François Tessier
Argonne National Laboratory
Lemont, Illinois
ftessier@anl.gov

Paul Gressier
Argonne National Laboratory
Lemont, Illinois
pgressier@anl.gov

Venkatram Vishwanath
Argonne National Laboratory
Lemont, Illinois
venkat@anl.gov

ABSTRACT

Effective data aggregation is of paramount importance for data-centric applications in order to improve data movement for I/O or to facilitate complex workflows, such as *in-situ* analysis, as well as coupling models and data for multi-physics. A key challenge for data aggregation in current and upcoming architectures is the heterogeneity of memory and storage systems (including DRAM, MCDRAM, NVRAM or parallel file system). One has to take advantage of this hierarchy and the characteristics of each tier to achieve improved performance at scale. In this paper, we present a topology and memory-aware data movement library performing data aggregation on large-scale systems. We first detail our hardware abstraction layer to accomplish code and performance portability on various platforms. Next, we present a cost model taking into account the system interconnect and the memory properties to determine an appropriate location for aggregating data. We also describe how we have implemented a data aggregation mechanism through the read algorithm. Finally, we show how we can improve data movement on a visualization cluster and a leadership-class supercomputer up to 16K processes with a benchmark and two typical I/O kernels. Particularly, we demonstrate how our approach can decrease the I/O time of a classic workflow by 26%.

CCS CONCEPTS

• **Information systems** → *Hierarchical storage management*;

KEYWORDS

Data movement, I/O, data aggregation, deep memory hierarchy, placement

ACM Reference Format:
François Tessier, Paul Gressier, and Venkatram Vishwanath. 2018. Optimizing Data Aggregation by Leveraging the Deep Memory Hierarchy on Large-scale Systems. In *ICS '18: 2018 International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3205289.3205316

1 INTRODUCTION

Efficiently moving data on large-scale systems is decisive for improved performance. Large-scale simulations express important needs of reliability and accuracy in their results leading to a larger amount of data to manage. We typically estimate that those applications can easily spend around 10% to 20% of the wall time in I/O operations. In addition, the variety of the data structures that need to be written or read complicates the work of the application programmer to achieve good I/O performance. A particle-based application, for instance, generally needs to write multiple variables in multidimensional arrays evenly distributed among processing entities while an AMR[1] application has to manage different I/O sizes according to the input parameters. The increasing popularity of deep learning algorithms also brings new workloads requiring massive amount of input data. Last, more complex workflows such as *in-situ* visualization and analysis intensify even more this complexity. Thus, it is clear that data movement optimization is indubitably a key challenge for the coming years.

From an hardware perspective, the gap has increased for years between the amount of data to move and the memory or storage capabilities, whether it be in terms of capacity or performance. To overcome this problem, vendors have deployed intermediate tiers of memory and storage that need to be taken into account to reduce as much as possible the I/O bottleneck. However, these levels of the memory hierarchy come with their own characteristics and sometimes a dedicated software stack which make them difficult to use efficiently.

Data aggregation is a widespread technique mitigating the data movement bottleneck. It consists of aggregating data at different places of the architecture in order to optimize costly data access operations. In the context of collective I/O calls, for example, a phase of data aggregation accumulates contiguous chunks of data in memory before writing it to the storage system. Another illustration could be *in-situ* analysis workflows in which data aggregation provides more input data to the analysis process at the same location.

In this paper, we present an extended version of TAPIOCA [14, 15], an I/O library performing topology-aware data aggregation at scale. We propose to change the TAPIOCA perspective by tackling new tiers of memory and storage on current and coming large-scale systems. Thanks to an abstraction layer of the network interconnect and the deep memory hierarchy, this library can now perform data aggregation on any kind of memory or storage system and is seamlessly portable on various HPC systems. Thus, we target collective I/O operations as well as more complex workflows such as *in-situ* or *in-transit* analysis that may have needs of temporarily persistent

[1]Adaptive mesh refinement

data. A cost model we detail can determine the most appropriate location where data aggregation should take place. To validate our approach, we show how this method can outperform traditional I/O calls on a synthetic benchmark and the I/O kernels of two real applications. We focus our experiments on a visualization cluster and a leadership-class supercomputer located at Argonne National Laboratory, USA, offering the characteristics of the pending exascale architectures.

2 CONTEXT AND MOTIVATION

We first present in this section a state of the art of the current and upcoming new tiers of memory and storage one has to take into account for improved performance. We also briefly broach the subject of the necessity of an abstraction layer handling those resources. Then we dive more deeply in data aggregation algorithms and particularly though the two-phase scheme widely used in modern I/O libraries.

2.1 Complex large-scale architectures

The current and future high-performance architectures offer to address the I/O challenge by providing faster and more complex network interconnect as well as new types of memory banks and storage systems. For instance, Mira, a 10 PetaFLOPS IBM BG/Q supercomputer, features a 5D-torus interconnect (2GBps per link) while Theta, a newly deployed 11.69 PetaFLOPS Cray XC40 system hosts thousands of compute nodes connected with a Cray Aries high-speed Dragonfly network whose bisection bandwidth has been evaluated to 7.2 TBps. For its part, Sunway Taihulight in China, which has topped the Top500 list for two years, links its more than forty thousands nodes with a two-level fat tree topology (56 Gbps per link).

On the memory and storage side, the complexity is significant as well. The Intel Knight Landing processor deployed on most of the today's Cray systems embeds, in addition to DRAM, 16GB of high-bandwidth on-package memory (MCDRAM) whose I/O bandwidth can attain up to 380GBps. Some compute nodes also have a 128GB local SSD giving the user data persistency during the job lifetime. On Cori, at the National Energy Research Scientific Computing Center, burst buffers have been added as a new layer between compute nodes and the storage system. In a near future, the Intel 3D-Xpoint Optane SSD technology will provide up to 375GB of memory featuring the attributes of both memory and storage in a single tier. From a storage point of view, the architecture design of a Lustre file system or a GPFS instance shows several differences such as the stripping of the data, the number of concurrent streams or the distribution of the gateway nodes into the topology.

Coming up with an abstraction of those resources insuring code and performance portability is an open issue. A trade-off has to be found out between a specific abstraction for every system that needs to capture every phase of deployment or the software versions, and a more generalized abstraction that maps to current and expected future deep memory hierarchies and network interconnects. In addition, the feature requirements differ from one research domain to another (locality, performance, ...). We propose in this work an abstraction layer that allows to move data from/to any type of memory or storage.

Furthermore, this wide diversity and complexity of network interconnects and sorts of memory and storage has a significant performance potential on condition that they are efficiently exploited. A solution can be found in the domain of data aggregation techniques.

2.2 Data aggregation

In order to limit concurrency and non-contiguous accesses on file systems or memory, a preliminary phase of data aggregation is often necessary before moving data. In the context of I/O, the two-phase I/O algorithm is a widespread method consisting of selecting a subset of the processing entities, called aggregators, to aggregate contiguous pieces of data (aggregation phase) before writing/reading it to/from the storage system (I/O phase). Figure 1 shows an example of the two-phase I/O mechanism with four processes writing three-dimensional coordinates in a file on the storage system. In this example, processes 0 and 2 are selected to aggregate data from the other processes. The second phase aims to move contiguous chunks of aggregated data to the storage system.

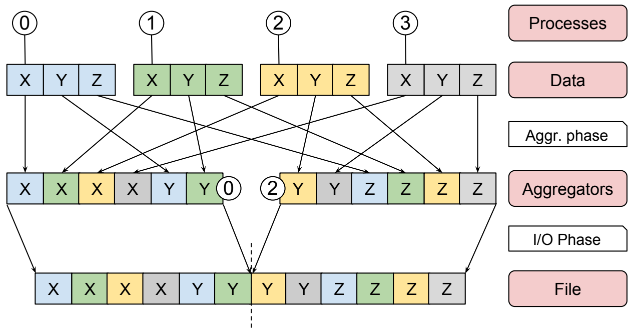

Figure 1: Example of the two-phase I/O mechanism

More generally, a number of aggregators as well as an aggregation buffer size are set and multiple aggregation and I/O rounds are performed to flush the total amount of data. That way, a fewer number of processing entities concurrently performs I/O operations on disks and those operations tend to be done with contiguous chunks of data, reducing the impact of the file system block size and the number of reads and writes. However, the standard implementations of this method suffer limitations. In MPI collective I/O operations for instance, no topology-aware aggregators placement policy has been implemented. Similarly, the aggregation phase is performed in the main memory of the aggregators, eclipsing the other available tiers of memory. The work done in complex workflows sometimes take advantage of one of the levels of memory but the proposed methods are usually designed for a specific workflow and are not easily portable on different platforms and workloads.

In this work, we extend TAPIOCA, a collective I/O library based on the two-phase scheme, to exploit the multiple tiers of memory for improved data movements. While standard approaches aggregate data in memory before moving it to/from a parallel file system, our technique takes advantage of the available layers of the deep memory hierarchy to perform the aggregation phase and can output data on any kind of persistent storage.

3 OUR APPROACH

We first introduce here the architecture abstraction on which our memory-aware placement algorithm relies on. Next, we present our cost model giving the most beneficial tier of memory or storage where data should be aggregated for an increased I/O bandwidth. Finally, we describe our data aggregation technique based on the two-phase I/O scheme through the "read" algorithm we have implemented and show a usage example. We will call our implementation "MA-TAPIOCA" for "Memory-aware TAPIOCA" for the rest of the paper.

3.1 Architecture abstraction

A key feature of our approach is to achieve code and performance portability across a broad variety of architectures and an extensibility in order to address future network interconnects and tiers of memory and storage on exascale systems and beyond. 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 2 depicts how those components fit into MA-TAPIOCA while Listings 1 and 2 show some of the API functions of those two abstractions.

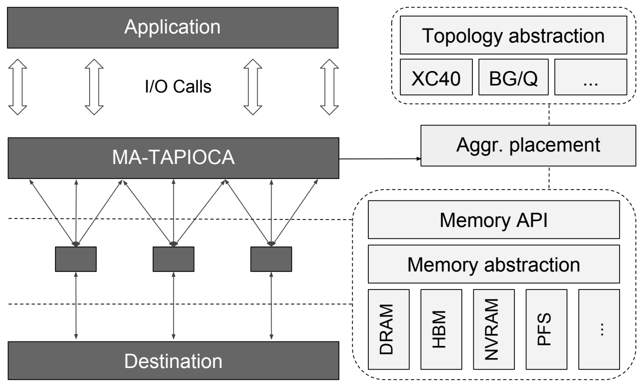

Figure 2: High-level view of MA-TAPIOCA and the two abstraction layers

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 insure 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 3.2 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, the *memkind*[2] library will be used for the (de)allocation. 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).

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);
```

The network abstraction provides the relative location of compute nodes as well as performance information. The distance between two nodes is defined according to the coordinates of the nodes in the topology. This information is either gathered through a vendor API or easy to determine on several topologies. The distance to the I/O node corresponds to the distance between the calling compute node and the gateway node through which the data packets transit to the storage system. Depending on the system, a gateway node can be one of the compute nodes directly linked to the storage system or a dedicated node whose function is to forward I/O calls to the storage system.

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);
```

3.2 Architecture-aware Aggregators Placement

The second main contribution of this work on data aggregation consists of transparently selecting the most appropriate location to place the aggregators. While some 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 included in our model concerns data persistency. A workflow including *in-situ* analysis for example may need temporary persistent local data.

More generally, our cost model is based on both the application needs (the data access pattern) and the network interconnect and memory characteristics. Therefore, for each set of data producers in which we need to elect an aggregator, given:

- V_M: The set of heterogeneous memory banks fulfilling the persistency requirements;

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

- $A \in V_M$: A memory tier able to aggregate data, chosen among the available memory banks;
- 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, $v \in V_M$;
- $d(u, v)$: The distance between memory banks u and v (hops or bus), $v \in V_M$;
- l: The latency such as $l = max\,(l_{network}, l_v)$;
- $B_{u \to v}$: The bandwidth from memory bank u to v with $v \in V_M$, such as $B_{u \to v} = min\,(B_{network}, B_u, B_v)$.
- C: The memory capacity

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.

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

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:

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

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. To do so, we define two costs $Cost_A$ and $Cost_T$. $Cost_A$ 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 slower bandwidth involved as well as the worst latency.

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

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

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

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 $\{Cost_A, Cost_T\}$ will be computed, on for each tier.

To determine the 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:

$$MemAware(A) = min\,(Cost_A + Cost_T)$$

Figure 3 illustrates this 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 inside a node are separated by one hop. Each node hosts two types of memory in addition to the main memory (DRAM): a high-bandwidth memory (HBM) and a 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. 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.

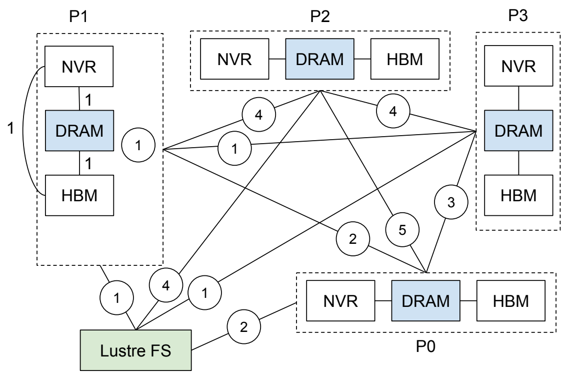

Figure 3: Toy examples of four processes collectively writing data on a Lustre file system through a data aggregation process.

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

Table 2 shows, for each process, the cost of aggregating data on its local available tiers of memory. Our model shows that the most advantageous location for aggregation is the high-bandwidth memory available on the node hosting process 1. 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 recent work [11].

3.3 Data Aggregation algorithm

The main part of our data aggregation algorithm is an optimized implementation of the well-known two-phase I/O scheme featuring multiple pipelined aggregation buffers together with an I/O

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 $Cost_T$.

scheduling wherein we describe the coming I/O transactions. In our previous work [15], we detailed those optimizations as well as the "write" algorithm we implemented. We present in Algorithm 1 the "read" algorithm. We can mainly distinguish four blocks in this algorithm. From line 14 to 20, 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). From line 23 to 30, this data is distributed from the aggregators to the other processes. The processes passing this conditional block carry out a RMA[3] operation (one-sided communication) to get data from the appropriate aggregation buffer (aggregation phase, line 33). The last block, from line 36 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 2 shows a usage example of our data movement library when reading from a file on the parallel file system three arrays representing coordinates. In this example, the aggregation layer can be either defined by the user with an environment variable (*MA-TAPIOCA_AGGRTIER*) or chosen with the aggregator placement model described in Section 3.2. When using the first method, the environment variable can be set to any memory tier implementing the memory abstraction. The memory location is then computed according to the only topology information. The second method computes the cost of aggregating data on one or the other level of the memory hierarchy and select the one with minimal cost. In this case, the level of persistency required has to be set with the *MA-TAPIOCA_PERSISTENCY* variable. Unlike MPI I/O, our approach requires to describe the upcoming I/O operations (no more than the parameters given to the I/O calls) in order to compute an internal I/O scheduling (lines 6 to 9). The read operations are similar to the MPI I/O API. The effort needed to use MA-TAPIOCA is very low for the user.

While standard implementations are unaware of the underlying architecture, we propose an architecture abstraction and a cost model such as the aggregation location is either a user-defined parameter or computed according to the data access pattern and the hardware characteristics.

Algorithm 1: MA-TAPIOCA read algorithm.

```
1  $GlobalRound \leftarrow 0$;
2  $ReadRound \leftarrow 0$;
3  $TotalRound \leftarrow$ ComputeNumberOfRounds ($datasize$);
5
6  Function MA-TAPIOCA_Read
     ($f, offset, data, size, type, status$)
7      $round \leftarrow$ GetRound();
8      $aggr \leftarrow$ GetAggregatorRank();
9      $chunkSize \leftarrow$ GetRoundSize($round$);
10     $bufferId \leftarrow globalRound$ % 2;
11     $readId \leftarrow readRound$ % 2;
13
14     if $firstRead$ then
15         if I am an aggregator then
16             Pull_Buffer ($readId$);
17             $readRound \leftarrow readRound + 1$;
18             $readId \leftarrow readRound$ % 2;
20         Fence ();
22
23     while $round \neq globalRound$ do
24         if I am an aggregator AND
             $readRound < TotalRounds$ then
25             Pull_Buffer ($readId$);
26             $readRound \leftarrow readRound + 1$;
27             $readId \leftarrow readRound$ % 2;
28         Fence ();
29         $globalRound \leftarrow globalRound + 1$;
30         $bufferId \leftarrow globalRound$ % 2;
32
33     RMA_Get ($data, chunkSize, offset, aggr, bufferId$);
35
36     if $chunkSize = size$ then
37         while $globalRound \neq TotalRounds$ do
38             if I am an aggregator AND
                 $readRound < TotalRounds$ then
39                 Pull_Buffer ($readId$);
40                 $readRound \leftarrow readRound + 1$;
41                 $readId \leftarrow readRound$ % 2;
42             Fence ();
43             $globalRound \leftarrow globalRound + 1$;
44             $bufferId \leftarrow globalRound$ % 2;
45     else
46         MA-TAPIOCA_Read ($f, offset + chunkSize, data +$
             $roundSize, size - chunkSize, type, status$);
```

[3]Remote Memory Access

Algorithm 2: Collective reads using MA-TAPIOCA.

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

for $i \leftarrow 0, i < 3, i \leftarrow i + 1$ do
    $count[i] \leftarrow n$;
    $type[i] \leftarrow$ sizeof $(type)$;
    $ofst[i] \leftarrow offset + i \times n$;

MA-TAPIOCA_Init $(count, type, ofst, 3)$;

MA-TAPIOCA_Read $(f, offset, x, n, type, status)$;
$offset \leftarrow offset + n$ ;
MA-TAPIOCA_Read $(f, offset, y, n, type, status)$;
$offset \leftarrow offset + n$;
MA-TAPIOCA_Read $(f, offset, z, n, type, status)$;

MA-TAPIOCA_Finalize ();
```

4 EXPERIMENTS

4.1 Experimental setup

To evaluate our data aggregation algorithm, we concentrated our effort on two systems located at Argonne National Laboratory. First, we targeted Theta, a recently deployed 11.69 PetaFLOPS Cray XC40 system that features some of the memory tiers we should expect in exascale systems. This Cray system counts more than four thousands Intel Knight Landing (KNL) nodes interconnected through an Aries dragon-fly network. Each node embeds 64 hyper-threaded cores and 192 GB of DRAM. 16 GB of MCDRAM per node are also available. This high-bandwidth memory can be used either as cache or as an allocatable memory. Each KNL node also hosts a 128 GB SSD which is cleared between two consecutive allocations. A Lustre file-system manages the 9.2 PB of storage dedicated to this platform. For our experiments, this file system were configured to use 48 OSTs and a 8MB stripe size. The second system, called Cooley, is an 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 storage are managed with a GPFS file system. The interconnect is a 56Gbps FDR Infiniband CLOS network.

We selected three different MPI applications to show the benefit of our approach: a synthetic benchmark and two I/O kernels respectively extracted from a computational fluid dynamics (CFD) application and a cosmological simulation. The purpose of the synthetic benchmark is to calibrate the subfiling technique and give insights about initial data distribution. The I/O kernel of the CFD application, called S3D-IO, takes advantage of the memory-aware aggregator placement described in Section 3.2. Finally, we use HACC-IO, the I/O kernel of a cosmological application, to study various aggregation techniques performed with MA-TAPIOCA. For all of our experiments, we set to 16 the number of MPI processes per node performing I/O on the Cray system and 12 processes per node on the Haswell-based cluster. We compared our results with the MPI-IO implementation available on the target platforms. The results presented in the rest of the paper are averages of at least 10 runs. Table 3 summarize our experimental setup. To fit in this paper, we will limit the experiments on the visualization cluster to HACC-IO.

Table 3: Experimental Setup

HPC Systems	Cray XC40, Haswell-based cluster
Comparison	MA-TAPIOCA, MPI-IO
Workloads	Synthetic benchmark 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 HDD: Hard disk drive (Lustre and GPFS) RAN: Network-attached memory bank

4.2 1D-Array benchmark

This 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 allocate 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 MA-TAPIOCA.

Figure 4 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 MA-TAPIOCA. We carried out three use-cases: the first one with an array of 25K integers per process (100 KB), the second one with a random distribution of the data among the processes (a value between 0 and 100 KB) and the last one with a normal distribution among the processes. Our approach outperforms MPI-IO on the three types of distributions. However, the performance gap is particularly significant with a random and a normal distribution seeing as the write bandwidth is respectively approximately 6 and 29 times higher while we read data 3 times faster.

Performing I/O operations on a single shared file is known to provide poor performance. Subfiling is usually preferred. Figure 5 presents the results we obtained on the same platform while performing subfiling, from one file per node to one file per 8 nodes. In such a use-case, one aggregator is selected per group of nodes writing or reading the same file. Data aggregation is performed on DRAM while the destination of the data is the Lustre file system. In addition, Unlike MPI-IO, MA-TAPIOCA allows to set the local SSD as a shared destination tier. We also ran experiments showing this feature. It has to be noted that the file created on each local SSD is temporary (allocation lifetime). We can conclude from these results

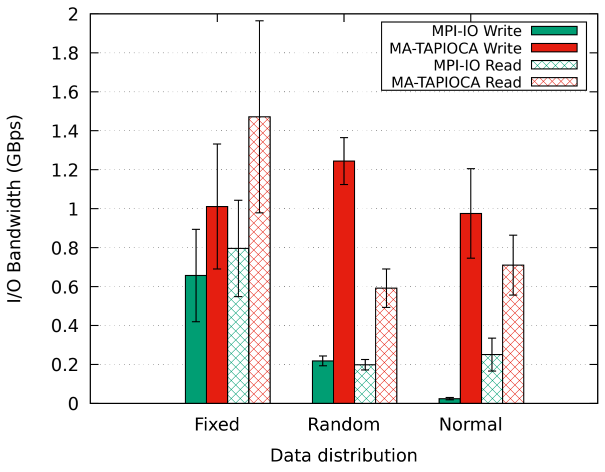

Figure 4: 1D-array on Cray XC40, single shared file on Lustre

that one file per node is the configuration offering the best I/O bandwidth for MPI-IO and our library. This "1:1" case was also evaluated with a random data distribution as shown in Table 4. Again, the best I/O performance is achieved with MA-TAPIOCA except on the read case from the Lustre file system. We are still investigating the poor read bandwidth obtained in most of our experiments.

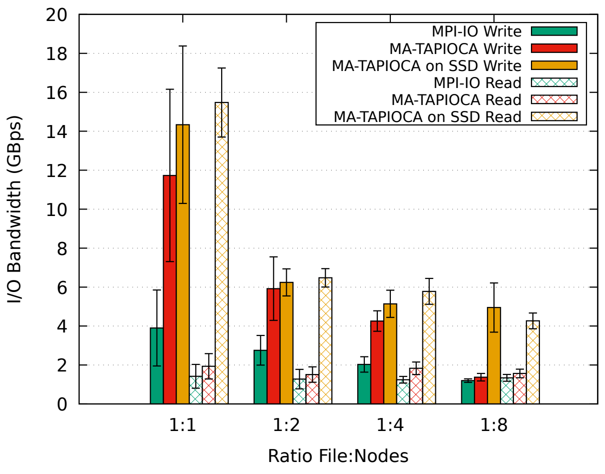

Figure 5: 1D-array on Cray XC40, ratio nodes per file

Table 4: MPI-IO vs MA-TAPIOCA, one file per node on Lustre and SSD (MA-TAPIOCA only) with random data distribution

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

Last, Table 5 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 observe that the difference in performance is not significant between aggregation on DRAM and HBM. This experiment corroborates the cost model evaluation presented in Section 3.2. We can also notice the overhead due to the file mapping in memory (*mmap*) when aggregating data on the local SSD.

Table 5: MA-TAPIOCA, one file per node on Lustre while aggregating on the three tiers of memory and storage available on nodes. 1 MB 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

4.3 S3D-IO

S3D [7] is a state-of-the-art direct numerical simulation (DNS) Fortran and MPI code in the field of computational fluid dynamics (CFD), focusing 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 MA-TAPIOCA. For those experiments, we let our architecture-aware algorithm described in Section 3.2 automatically decide the most appropriate tiers of memory for data aggregation among the compute nodes.

We first present in Table 6 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 MA-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, MA-TAPIOCA significantly outperforms MPI-IO. When running on 1024 nodes, the I/O bandwidth is multiplied by 3.

Table 6: Maximum write bandwidth (GBps). Aggregation performed on HBM with MA-TAPIOCA

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

Table 7: Maximum write bandwidth (GBps) while artificially reducing the memory capacity. For each run, the grey box corresponds to the memory tier selected for aggregation.

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

The second example, running on 256 nodes with 134 millions grid points, artificially emphasizes the adaptability of our approach in the event of the faster tier of memory available does not have enough memory for aggregated data. Table 7 shows the results obtained with S3D-IO on 256 nodes while the capacity of the high-bandwidth memory (run 2) then the DRAM (run 3) has been artificially decreased to 32 MB. At the same time, we set the number of aggregation buffers to 3 and their size to 16 MB. The capacity requirement described in Section 3.2 not being fulfilled, the second then the third faster memory tier are selected.

4.4 HACC-IO

HACC is a large-scale cosmological application simulating the mass evolution of the universe with particle-mesh techniques. HACC-IO is the I/O kernel of HACC, in other words the application without computation. In HACC, every process manages a number of particles. A particle is defined by 9 variables describing the coordinates, the movement vectors and some state values. Data is stored in file in an array of structure data layout. We previously showed with the 1D-Array experiments that the best I/O bandwidth is achieved with one file per node when doing subfiling. We kept this setting for our experiments with HACC-IO. As HACC-IO is designed to use a single shared file, we implemented another version using one file per node. Except when explicitly mentioned, the experiments on the Cray System where carried out on 1024 nodes (16K processes) and on 64 nodes on the visualization cluster. For the coming experiments, we explicitly set the aggregation layer through the *MA-TAPIOCA_AGGRTIER* environment variable.

Figure 6 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 MA-TAPIOCA can achieve on a standard workflow, from the application to a parallel file system. Data aggregation is performed on the DRAM in this set of experiments. On both read and write, MA-TAPIOCA outperforms MPI-IO respectively by a factor of 5.4 and 13.8 with a 1 MB data size per process.

As demonstrated in 4.2, 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 7, writing one file per node on the parallel file system improves the performance up to 40 times with a large amount of data per process. On this case, MPI-IO and MA-TAPIOCA offer I/O performance in the same confidence interval. As mentioned previously, whatever the subfiling granularity chosen, MA-TAPIOCA is able to use the local SSD as a file destination. Therefore, we included the results when writing and reading data

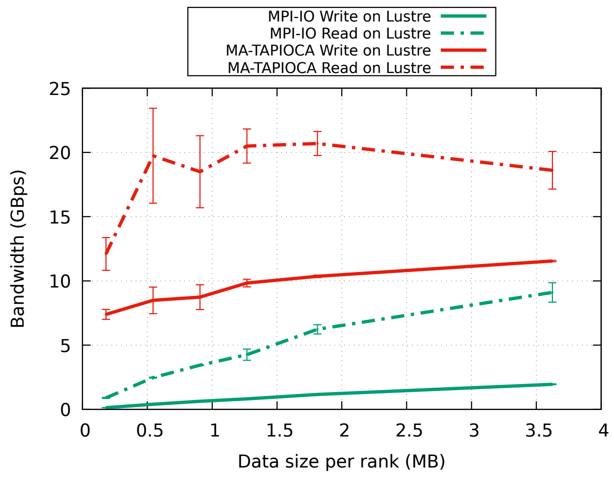

Figure 6: HACC-IO on 1024 Cray XC40 nodes, single shared file

to/from this storage layer. In this case, the I/O bandwidth is 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.

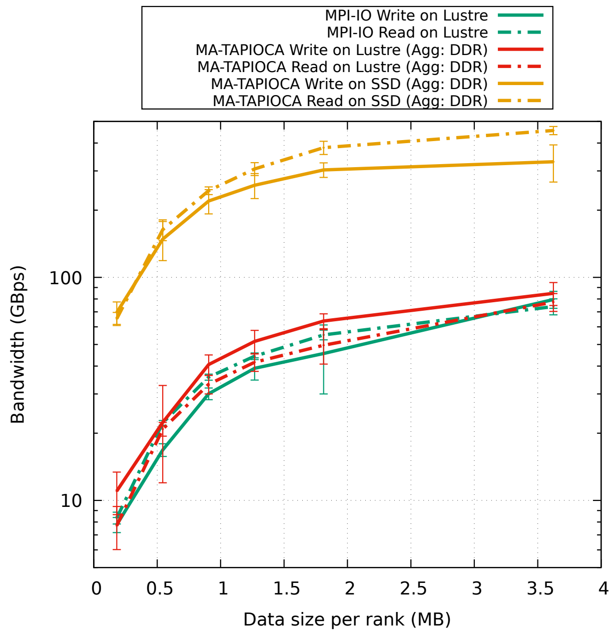

Figure 7: HACC-IO on 1024 Cray XC40 nodes, one file per node on Lustre and local SSD. Log-scale on y-axis.

Figure 8 shows a weak scaling study of the previous experiment. A process manages 1 MB of data. The aggregation is performed on the DRAM of each aggregator and the target for output data is set to the parallel file system and the on-node SSD. This last method reveals a very good scalability as the I/O performance attained increases by more or less 50% every time we double the number of compute nodes.

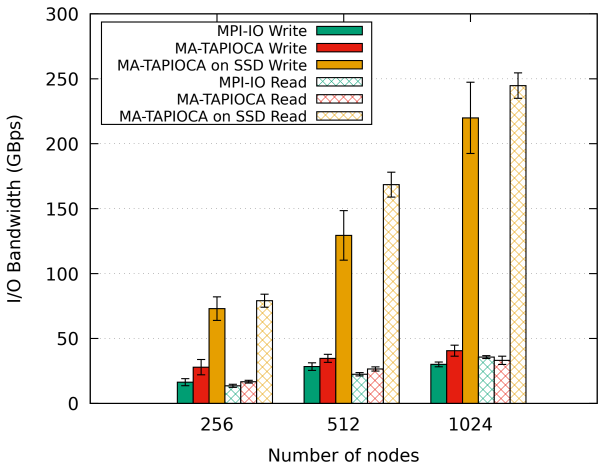

Figure 8: HACC-IO, one file per Cray XC40 node on Lustre and local SSD. 1MB per process, varying the number of nodes.

Our library, thanks to the memory abstraction we have proposed, is able to aggregate data on the high-bandwidth memory available on the compute nodes. Figure 9 compares an execution with aggregation on DRAM and on HBM while writing and reading on local SSD. In case of I/O operations on the Lustre file system, our model shows that performance is limited by the network. The I/O performance achieved is comparable with aggregation on DRAM and on HBM. This result was expected given the performance gap between the HBM and the SSD. In the future, the coming generation of NVDIMM could completely disrupt this result.

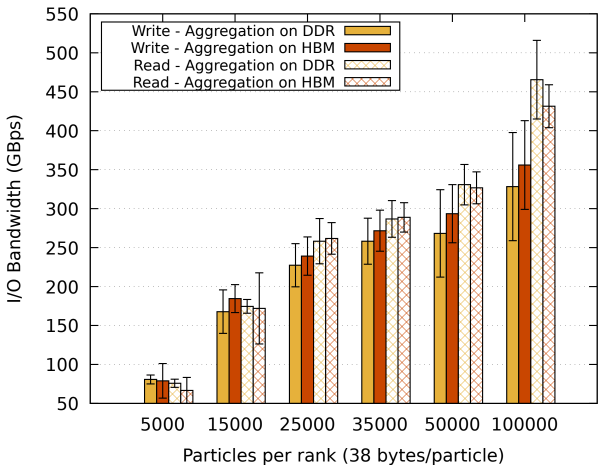

Figure 9: HACC-IO on 1024 Cray XC40 nodes, one file per node on local SSD. Comparison of the aggregation on DDR and on HBM.

Finally, Figure 10 depicts a typical workflow that can be seamlessly implemented with MA-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 3, MA-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. An MPI window then exhibits this buffer to local and remote nodes.

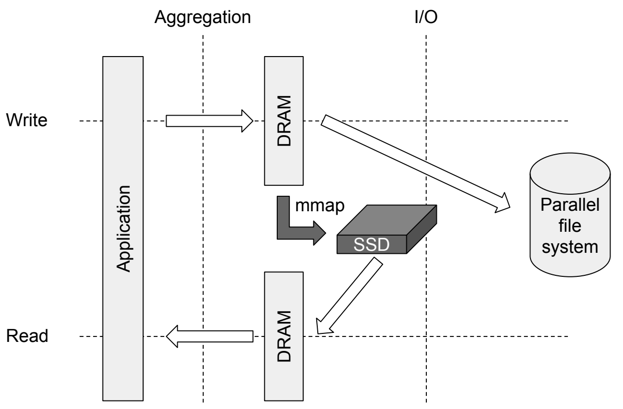

Figure 10: Write/Read workflow using MA-TAPIOCA and SSDs as both an aggregation buffer and a target.

Table 8 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 MA-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 8: 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
MA-TAPIOCA	DDR	47.50	38.92	693.88 ms
MPI-IO	DDR	32.95	37.74	843.73 ms
MA-TAPIOCA	SSD	26.88	227.22	617.46 ms
Variation		-36.10%	+446.94%	-26.82%

4.5 Portability

To assess portability of our architecture-aware data aggregation algorithm, we ran experiments with HACC-IO on 64 nodes of an 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 test-bed 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 as presented is Figure 10. Beyond the I/O performance, these experiments are more a proof of concept.

We show in Table 9 the results obtained with the workflow described in Figure 10. 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 visualization cluster.

Table 9: 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
MA-TAPIOCA	DDR	6.60	38.80	123.41 ms
MPI-IO	DDR	6.02	17.46	155.40 ms
MA-TAPIOCA	HDD	5.97	35.86	135.86 ms
Variation		-0.83%	+105.38%	-12.57%

5 RELATED WORKS

Data aggregation is a widespread technique, particularly developed in the context of intensive parallel I/O [12]. In MPI I/O libraries such as ROMIO [16] based on the MPI-2 [6] standard, collective I/O operations take advantage of data aggregation thanks to the two-phase I/O scheme [2]. Some work has been done to optimize this algorithm, whether it be within the MPI standard [19], as part of an MPI-IO implementation [17, 18, 20] or while proposing a new MPI-based collective I/O library [13, 15]. However, those approaches are unaware of the available tiers of memory and storage and don't allow to easily benefit from these new resources.

Data movement optimizations based on data aggregation phases are also studied in research areas focusing on workflows such as *in-situ* or *in-transit* data processing. In [9], the authors have studied how to take advantage of available SSDs to overcome the shortage of DRAM for a specific workflow. Another approach consists in finely describing the workflow and the movements of data involved [4, 5]. Nonetheless, those techniques require the user to have a strong knowledge of its applications.

On the contrary, some work has been done from the runtime perspective. In [3], the authors have proposed to address the I/O bottleneck challenge by transparently moving data from the application to the storage system through an intermediate fast storage layer. Another interesting work [22] focused on using fast storage layers (here, burst buffers) as a distributed file system while another one [10] came up with a driver for MPI-IO able to take advantage of a network-attached memory tier. However, those research targets very specific architectures and memory tiers, restraining the portability.

To insure code portability and take into account the coming exascale machines embedding new tiers of memory and intermediate storage, an architecture abstraction is necessary. Hwloc [1] is probably the most common hardware abstraction. However, this library only provides qualitative information and does not consider the interconnect network. At an higher-level, SharP [21] provides an abstraction layer to allocate memory on any available tier. Yet, this approach depends on the data model to handle (multi-dimensional arrays, meshes, ...) and does not target workflows involving data aggregation. Older work [8] focused on SMP architectures and developed a runtime interface able to compose with multiple runtimes to leverage the memory hierarchy.

Our approach differs from all the above solutions by adopting a method based on data aggregation taking into account the underlying architecture through a memory and network interconnect abstraction. Thus, MA-TAPIOCA can perform aggregation on any available memory and storage tier and proposes a model minimizing the cost of data movement. Finally, our algorithm does not depend on the application data model.

6 CONCLUSION

We have presented in this paper a data aggregation algorithm able to take advantage of the underlying architecture. Particularly, we showed how an architecture abstraction as well as a cost model can mitigate the I/O bottleneck on current and future large-scale systems. Our library has significantly improved performance on traditional I/O workloads as well as more complex workflows expressing different I/O efficiency and data persistency requirements. The way data is aggregated and moved from/to the storage system can be either specified by the user or computed by our cost model. This model will be at the center of our future research as we would like to extend it such as it will be possible to describe a complete workflow (multiple applications sharing data for instance) instead of just the I/O behavior of one application.

In terms of performance improvements, we have demonstrated on a benchmark and two real applications reduced to the only I/O phases that we can outperform MPI-IO and offer much more flexibility. We ran a large set of experiments up to 16K processes on two systems at Argonne National Laboratory: Theta, a 11.69 PetaFLOPS Cray XC40 system and Cooley a visualization cluster, both of them prefiguring the coming exascale systems. Particularly, we showed on a workflow involving two workloads sharing data that we can reduce the execution time by more than 26%.

As a future work, we would like to strengthen this approach, particularly by first studying the impact of input parameters in the cost of moving data. The data access pattern for instance will be of interest to characterize the applications and offer better improvements. Another research track we would like to develop is multi-level data aggregation. MA-TAPIOCA allows to determine or explicitly point out one aggregation layer. A multi-level approach could surely be of benefit to various workloads such as checkpointing.

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 and DE-AC02-05CH11231. This work was supported in part by the U. Department of Energy, Office of Science, ASCR, under award numbers 57L32, 57L11, 57K50, and 5080500.

REFERENCES

[1] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. 2010. 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. http://hal.inria.fr/inria-00429889

[2] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. 1993. Improved Parallel I/O via a Two-phase Run-time Access Strategy. *SIGARCH Comput. Archit. News* 21, 5 (Dec. 1993), 31–38. https://doi.org/10.1145/165660.165667

[3] B. Dong, S. Byna, K. Wu, Prabhat, H. Johansen, J. N. Johnson, and N. Keen. 2016. Data Elevator: Low-Contention Data Movement in Hierarchical Storage System. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. 152–161. https://doi.org/10.1109/HiPC.2016.026

[4] M. Dreher and T. Peterka. 2017. *Decaf: Decoupled Dataflows for In Situ High-Performance Workflows*. https://doi.org/10.2172/1372113

[5] M. Dreher, K. Sasikumar, S. Sankaranarayanan, and T. Peterka. 2017. Manala: A Flexible Flow Control Library for Asynchronous Task Communication. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 509–519. https://doi.org/10.1109/CLUSTER.2017.31

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

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

[8] Michael C. Houston. 2008. *A Portable Runtime Interface for Multi-level Memory Hierarchies*. Ph.D. Dissertation. Stanford, CA, USA. Advisor(s) Hanrahan, Patrick M. AAI3302833.

[9] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Romanus, N. Podhorszki, S. Klasky, H. Kolla, J. Chen, R. Hager, C. S. Chang, and M. Parashar. 2015. Exploring Data Staging Across Deep Memory Hierarchies for Coupled Data Intensive Simulation Workflows. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 1033–1042. https://doi.org/10.1109/IPDPS.2015.50

[10] Julian M. Kunkel and Eugen Betke. 2017. An MPI-IO In-Memory Driver for Non-volatile Pooled Memory of the Kove XPD. In *High Performance Computing - ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers (Lecture Notes in Computer Science)*, Julian M. Kunkel, Rio Yokota, Michela Taufer, and John Shalf (Eds.), Vol. 10524. Springer, 679–690. https://doi.org/10.1007/978-3-319-67630-2_48

[11] Jialin Liu, Quincey Koziol, Houjun Tang, Francois Tessier, Wahid Bhimji, Brandon Cook, Brian Austin, Suren Byna, Bhupender Thakur, Glenn Lockwood, et al. 2017. Understanding the IO Performance Gap Between Cori KNL and Haswell. In *Cray User Group Meeting*.

[12] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. 2015. 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, 33–44. https://doi.org/10.1145/2749246.2749269

[13] Preeti Malakar and Venkatram Vishwanath. 2017. Hierarchical Read–Write Optimizations for Scientific Applications with Multi-variable Structured Datasets. *International Journal of Parallel Programming* 45, 1 (01 Feb 2017), 94–108. https://doi.org/10.1007/s10766-015-0388-z

[14] François Tessier, Preeti Malakar, Venkatram Vishwanath, Emmanuel Jeannot, and Florin Isaila. 2016. Topology-aware Data Aggregation for Intensive I/O on Large-scale Supercomputers. In *Proceedings of the First Workshop on Optimization of Communication in HPC (COM-HPC '16)*. IEEE Press, Piscataway, NJ, USA, 73–81. https://doi.org/10.1109/COM-HPC.2016.13

[15] F. Tessier, V. Vishwanath, and E. Jeannot. 2017. TAPIOCA: An I/O Library for Optimized Topology-Aware Data Aggregation on Large-Scale Supercomputers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 70–80. https://doi.org/10.1109/CLUSTER.2017.80

[16] Rajeev Thakur, William Gropp, and Ewing Lusk. 1998. A Case for Using MPIâĂŹs Derived Datatypes to Improve I/O Performance. In *Proceedings of SC98: High Performance Networking and Computing*. ACM Press. http://www.mcs.anl.gov/~thakur/dtype/

[17] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. 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, 182–. http://dl.acm.org/citation.cfm?id=795668.796733

[18] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. 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, 23–32. https://doi.org/10.1145/301816.301826

[19] Rajeev Thakur, William Gropp, and Ewing Lusk. 2002. Optimizing Noncontiguous Accesses in MPI I/O. *Parallel Comput.* 28, 1 (Jan. 2002), 83–105. https://doi.org/10.1016/S0167-8191(01)00129-6

[20] Yuichi Tsujita, Hidetaka Muguruma, Kazumi Yoshinaga, Atsushi Hori, Mitaro Namiki, and Yutaka Ishikawa. 2012. 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, Article 7, 8 pages. http://dl.acm.org/citation.cfm?id=2338816.2338823

[21] M. G. Venkata, F. Aderholdt, and Z. Parchman. 2017. SharP: Towards Programming Extreme-Scale Systems with Hierarchical Heterogeneous Memory. In *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*. 145–154. https://doi.org/10.1109/ICPPW.2017.32

[22] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu. 2016. An Ephemeral Burst-Buffer File System for Scientific Applications. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. 807–818. https://doi.org/10.1109/SC.2016.68