Cooperative Collective I/O: Speeding up Parallel Producer/Consumer Patterns

Florin Isaila, Jesus Carretero

University Carlos III

Robert Bindar, Georgiana Dolocan

Politechnic University of Bucharest

Tom Peterka

Argonne National Laboratory

Abstract

Scientific applications running on current high-performance computing (HPC) platforms are increasingly based on parallel producer-consumer patterns consisting of simulation and data analysis components. This paper presents the design and implementation of cooperative collective I/O (CCIO), a novel I/O optimization technique for accelerating data-parallel producer-consumer patterns using the MPI-IO interface. CCIO relies on Bufferflow, a novel modular node-local buffering approach, offering data-parallel access operations, collective buffering, elasticity of the memory pool, secondary storage backup, and flexibility of memory allocation and replacement policies. The experimental results run on up to 49,152 cores demonstrate a significant performance improvement over the current MPI-IO implementation: up to 57 times for collective writes, up to 5 times for collective reads, and up to one order of magnitude for data-parallel producer/consumer patterns using collective writes and collective reads.

Keywords: HPC, parallel I/O, collective I/O, MPI-IO, producer/consumer patterns

1. Introduction

The increasing computational and data requirements of scientific applications are main driving factors for the continuous increase of the scale of HPC platforms towards Exascale and beyond within the next decade. With the advent of big data and the foreseen growing imbalance between computational and I/O capabilities there is a critical need to fundamentally redesign the storage I/O system of large-scale HPC systems for meeting the Exascale requirements [1, 2].

The current software I/O stack of large-scale HPC systems consists of several layers as shown in Figure 1 [3]: scientific libraries (e.g., HDF5[1]), middleware (e.g., ROMIO [4] implementing the MPI-IO standard [2]), I/O forwarding (e.g., IOFSL [5]), and file systems (e.g., GPFS [6], Lustre [7]).

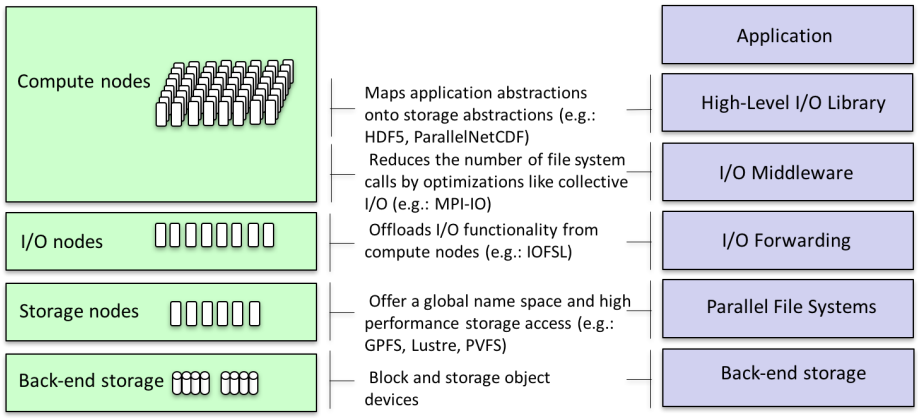

Figure 1: Mapping of the I/O software stack (right-hand side) on the architecture of current large-scale high-performance computing systems (left-hand side)

One of the main challenges to achieve the required scaling of this software I/O stack by several orders of magnitude is the fact that the I/O bandwidth (between compute nodes and I/O nodes) is expected to increase significantly less than the computation and memory capabilities of the compute nodes[3].

[1] HDF5 is available at `http://www.hdfgroup.org/HDF5/`.

[2] Available at `http://www.mpi-forum.org/`.

[3] https://science.energy.gov/ascr/facilities/

Additionally, current shared parallel file systems are reaching their scalability limits [8].

Parallel producers and consumers are an increasingly common pattern for sharing data between scientific simulations and data analysis on large-scale HPC infrastructures. For addressing the storage I/O challenges described above, the data access requirements of these patterns have been addressed by two classes of techniques targetting to reduce the required peak I/O bandwidth and/or the file system utilization: in-situ computation and buffering. In-situ computation [9, 10] leverages the locality of data produced by parallel simulations for analyzing/visualizing data without moving them from compute nodes. While this approach works fine for low-overhead analysis/visualization, it can heavily penalize the execution time for complex analyzes, given that in most cases it requires the simulation to stop. Alternatively, buffering is aguably the most common technique for optimizing the usage of storage systems in large-scale HPC systems. Buffering approaches include: reducing the number of storage I/O operations by collective I/O [4], hiding the storage I/O latency through burst-buffers[11] or in-memory multiple-stage buffering [12], reducing the storage I/O volume by performing in-transit computation [13], and speed up the storage I/O of workflow patterns [14, 15, 16].

The main contribution of this paper is the design and implementation of cooperative collective I/O (CCIO), a novel I/O optimization technique for accelerating data-parallel producer-consumer patterns. To the best of our knowledge CCIO is the first portable MPI-IO implementation of *coordinated collective write and read operations*. CCIO relies on Bufferflow, a novel modular node-local buffering approach. Bufferflow goes beyond the traditional put/get store by offering *data-parallel* primitives in both scalar and vector versions (i.e., analogue to POSIX readv/writev operations). These primitives can be used to efficiently address key properties of parallel producer/consumer patterns including spatial locality, temporal locality, and access non-contiguity. Additionally, Bufferflow provides elasticity through an *adaptive buffering* mechanism that reacts to either high or low volume of data access operations. Its adaptability enables

the library to expand or shrink its internal memory pools, when specific watermarks are reached, asynchronously from the consumer and producer execution contexts. Finally, Bufferflow opens up the buffer memory and storage management to user-specific allocation and replacement policies (both these policies are currently embedded in various layers of the I/O stack).

The solution described in this paper has been implemented in two software packages that are freely available for download and evaluation: Bufferflow[4] and CLARISSE [5].

The remainder of the paper is organized as follows. Sections 2 presents an overview of the design and implementation of CCIO. The details of the Bufferflow implementation are discussed in Section 3. Section 4 describes the algorithmic details of CCIO. Section 5 discusses the experimental results. We contrast our work with related approaches in Section 6. Section 7 summarizes our work and looks at remaining challenges and next steps.

2. Overview of Cooperative Collective I/O

In this section we present an overview of Cooperative Collective I/O (CCIO) technique. Figure 2 shows the place of existing collective I/O optimization in the current software I/O stack of large-scale HPC platforms. The left-hand side shows the most common configuration of this stack. The most popular MPI-IO implementation is ROMIO [4]. ROMIO is implemented on top of several file systems, which are typically offering a POSIX-like interface. The file system calls are forwarded from compute nodes to remote I/O nodes through RPCs by a I/O forwarding layer. The file systems are mounted on I/O nodes.

Collective I/O is used for improving the I/O performance by coalescing small- and medium-sized I/O requests, thus reducing the pressure on the back-end storage. ROMIO includes a collective I/O algorithm called two-phase I/O

[4]The Bufferflow code is available for download at https://github.com/robertbindar/CLARISSE-AdaptiveBuffering.

[5]The CLARISSE code is available for download at https://bitbucket.org/fisaila/clarisse.

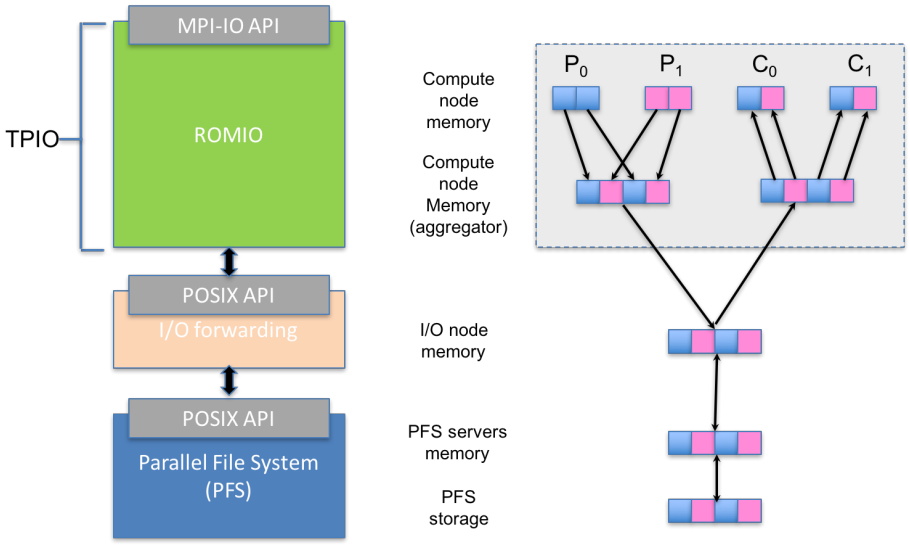

Figure 2: Two-phase I/O (TPIO) and the storage I/O stack.

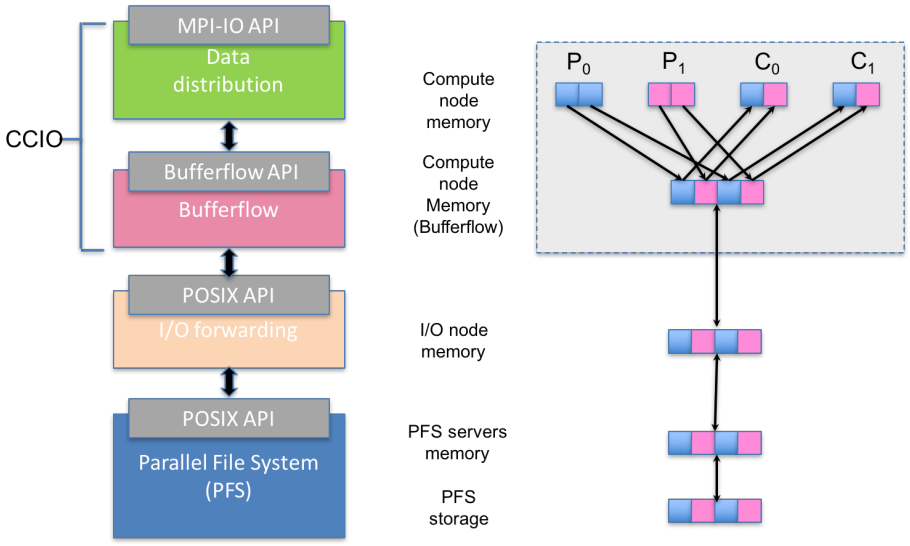

Figure 3: Cooperative Collective I/O (CCIO) and the storage I/O stack.

(TPIO). The right-hand side of Figure 2 illustrates TPIO for a parallel producer with $n_p = 2$ processes and a parallel consumer with $n_c = 2$ proceses. For n_p processes issuing collectively write I/O requests, TPIO consists of two-phases:

1) the data are merged on buffers on a subset of $n_s \leq n_p$ processes called aggregators and 2) the buffers are flushed to the file system. The reverse-symmetrical operations are executed for reads.

TPIO can cause several performance limitations when used by parallel producer-consumer patterns. First, the aggregators are a subset of application processes, causing the I/O latency of the slowest aggregator to be perceived by all processes. Second, each aggregator uses exactly one buffer, which is statically allocated. Thus, when memory availability varies, it is not possible to dynamically grow/shrink the buffer space for accounting for the speed difference between producers and consumers. Third, the TPIO buffers can not be shared among producers and consumers, because the memory spaces of producers and consumers are distinct. This causes the data to be buffered twice in read and write buffers. Additionally, in the best case the data is moved between these two buffers through the file system server caches.

The left-hand side of Figure 3 shows our solution: the MPI-IO implementation is organized in two modules: a data distribution module offering an MPI-IO interface and implementing the data movement between application processes and buffering processes and a buffering module called Bufferflow consisting of several aggregators running as independent processes.

CCIO addresses all TPIO limitations described above. First, for allowing relatively fast producers to progress, CCIO decouples aggregator processes from application processes. Each CCIO aggregator is an independent process running a node-local buffering server offering the Bufferflow API. Second, each Bufferflow instance employs dynamic management of buffer memory: buffer space can grow/shrink dynamically within availability limits for adapting to the speed mismatch of producers and consumers. Moreover, buffer allocation and replacement are modularly implemented and can be customized by the user for meeting specific needs. Third, producers and consumers share aggregator buffers, while exploiting spatial locality through the implicit coordination offered by the Bufferflow API. The remainder of this section presents an overview of CCIO.

2.1. Overview of CCIO data distribution

The CCIO data distribution has been developed as a part of the data plane of the CLARISSE middleware [19]. The data distribution abstraction is an MPI-IO file distributed round-robin over n_s servers. The implementation is portable and supports all MPI-IO file operations including open, close, setview, write, and read calls.

The right-hand side of Figure 3 shows the data flow in the CCIO. The figure shows n_p producers and n_c consumers performing data-parallel accesses on a shared data set abstracted as an MPI-IO file. For simplicity we used $n_s = 1$ in the figure. For instance, the data produced by P_O and P_1 are non-contiguously distributed to a remote buffer at addresses represented by a list of offset-length tuples. The mapping of the data to the buffer and the transfer to the Bufferflow server is performed by CLARISSE, while the access to the buffer by a Bufferflow put call. For the read operation, the data are retrieved locally by a Bufferflow get call and distributed to the consumers by CLARISSE. The details of the algorithms used for implementing the data distribution in CLARISSE are presented in Section 4.

2.2. Bufferflow architecture

Bufferflow is composed of three layers, shown in Figure 4: *coordination, scheduling*, and *allocation* layers. Each of the layers exposes a clean interface and an implementation agnostic of the other layers' internal details, thus allowing developers to easily plug different implementations of entire layers into Bufferflow. This is an important design decision, because it opens up the buffering service to custom implementations depending on the application needs. The lack of scalability of the current software I/O stack is due in part to the inflexible embedding of buffering at various layers, which prevents the design and insertion of novel policies for key aspects such as access coordination, scheduling, and allocation.

The coordination layer shown in the upper part of the figure is responsible for coordinating concurrent data-parallel accesses to the buffering systems. As one

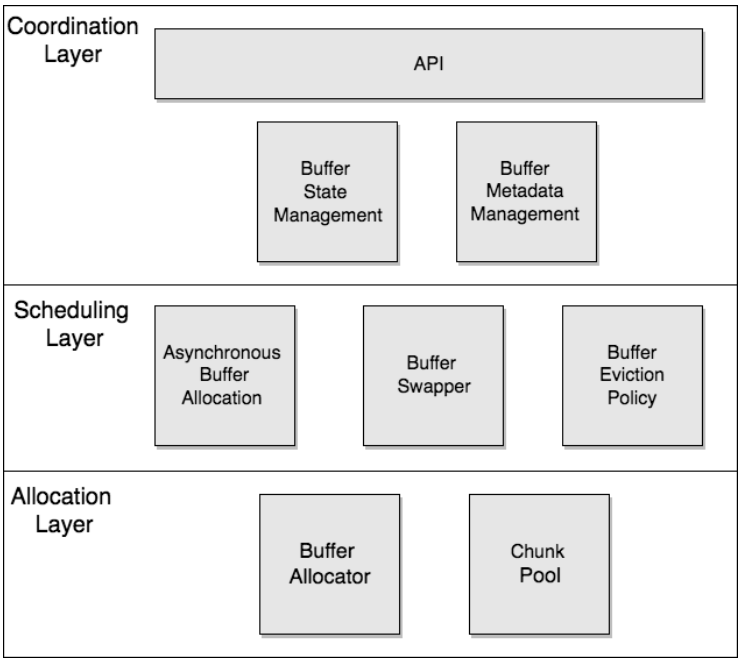

Figure 4: Bufferflow high-level architecture.

of the key contributions, Bufferflow offers mechanisms for coordinating both data-parallel accesses within one application and across applications, that is, between data-parallel producers and consumers. To achieve this coordination, the coordination layer manages the life cycle of the high-level abstraction of a buffer, including the transitions of a buffer through different states. Moreover, the coordination layer provides synchronization mechanisms for the concurrent data-parallel participants, thus enabling Bufferflow to offer powerful API semantics, described in the following section.

The scheduling layer shown in the middle area of the figure mediates the interaction between the coordination and allocation layers. As the most complex part of the architecture, the scheduling layer makes Bufferflow adaptive to various conditions that may occur in the system. These conditions include inefficient memory management due to differences of speed between producer and consumer processes, exceeding buffer memory footprint thresholds,

and slow memory allocations/deallocations due to certain data access patterns. The scheduling layer offers asynchronous mechanisms for both buffer allocation (asynchronous buffer allocation) and swapping (buffer swapper). Additionally, the buffer selection for swapping is driven by a customizable buffer eviction policy.

The allocation layer shown in the lower part of Figure 4 specializes in managing chunks of raw data. Bufferflow provides a modular allocation layer structure consisting of a buffer allocator and a chunk pool. In Section 3.3 we describe an implementation of the buffer allocator that is efficient for bulk allocations/deallocations required by the asynchronous allocation policy implemented in the scheduling layer and discussed in Section 3.2. However, the modular design allows for a straightforward implementation of any other allocation policy in the buffer allocator module.

3. Bufferflow Implementation

This section describes an implementation of the three architectural layers shown in Figure 4.

3.1. Coordination layer

The coordination layer exposes the Bufferflow API that allows data-parallel concurrent producers and consumers to be expressed with simple and powerful semantics, as presented in Section 3.1.1. The coordination layer manages the buffer metadata as presented in the Section 3.1.2. For coordinating the data-parallel accesses of producers and consumers, the coordination layer also manages the life cycle of buffers by directly controlling the buffer state and the transitions between states as discussed in Section 3.1.3.

3.1.1. API

Bufferflow exposes a simple and flexible interface. Each buffer is uniquely identified by a buffer handle *buf_handle_t*. In order to facilitate the association

of a buffer with an external object such as the block of a file, the handle is described as a structure consisting of a unique global descriptor and an offset.

Bufferflow offers data-parallel put/get operations on buffers with both scalar and vector access patterns, as shown in Listing 1. Each of these operations is executed locally on a node.

```
error_code get(buffering_t *bufservice, const
    buf_handle_t bh, const size_t offset, byte_t *
    data, const size_t count)
error_code get_all(buffering_t *bufservice, const
    buf_handle_t bh, const size_t offset, byte_t *
    data, const size_t count, const uint32_t
    nr_participants);
error_code get_vector(buffering_t *bufservice, const
     buf_handle_t bh, const size_t *offsetv, const
    size_t *countv, const size_t vector_size, byte_t
    *data);
error_code get_vector_all(buffering_t *bufservice,
    const buf_handle_t bh, const size_t *offsetv,
    const size_t *countv, const size_t vector_size,
    byte_t *data, const uint32_t nr_participants);
```

Listing 1: API of get operation. The matching put operations have the same syntax except that the type of the `data` variable is `const byte_t`.

The `put/get` calls atomically modify/retrieve the chunk of data residing at `offset` bytes from the buffer identified by `bh` in the buffering service `bufservice`.

The `put_all/get_all` calls are data-parallel versions of `put/get` calls. They can be called by a subset of processes of a parallel application for collectively accessing a shared buffer. These calls allow the parallel accesses to be coordinated for making the best use of temporal and spatial locality of accesses typical of parallel workflows.

The `put_all` call has the following semantics: `nr_participants` concurrent processes are writing to a shared buffer `bh`. Processes of scientific applications commonly write nonoverlapping buffer regions [17]. This scenario is optimally supported by Bufferflow. If the buffer regions overlap, the outcome can be any possible combination of interleaving patterns of individual accesses, unless higher-level middleware imposes a special order on the access enforcement. The shared buffer `bh` is not eligible for replacement until all `nr_participants` have finished. This approach enables the optimal exploitation of spatial locality, a common property of scientific applications [17].

The `get_all` call allows `nr_participants` consumers to concurrently read data from a shared buffer. The buffer is automatically released after all the consumers have finished. A buffer accessed by `get_all` is swappable (Section 3.1.3 discusses in more detail the swapping process in the context of the buffer life cycle). Temporal locality could be exploited by a `get_all` arriving after a `put_all`, but this is not guaranteed. For enforcing the exploitation of temporal locality Bufferflow offers non-swappable versions of the put/get functions shown above. For these versions the buffers written by the concurrent producers are kept in memory until concurrent consumers read them. The usage of these calls can significantly improve performance when the consumers are as fast as or faster than the producers, because the data are guaranteed not to reach the disk.

The vector versions of `put/get/put_all/get_all` allow accessing several segments of a buffer with a single call, similar to how `readv` and `writev` calls access memory in the POSIX standard.

3.1.2. Buffer metadata management

Bufferflow keeps track of the existing buffers in the system and their state in various internal data structures.

Internally, each buffer is associated with a small data structure containing information such as the buffer handle, buffer state, two integers representing the number of arrived producers and consumers, and the data pointer. The

buffer handle, discussed in the beginning of Section 3.1.1, is used for mapping the external buffer representation on the internal data structure. The buffer state indicates the current state of each buffer and, together with the number of expected and arrived numbers of producers/consumers is useful for controlling the transitioning of the buffers through various states, as discussed in more detail in Section 3.1.3. The arrived number of producers and consumers is kept in the internal data structure, while the expected numbers of producers and consumers involved in the put/get operations are provided through the put/get calls. The data pointer points to a region of an allocated chunk, which is managed by the allocation layer.

The global metadata includes a buffer count and a hash table containing the information about all the buffers in the system. The scheduling and the allocation layers contain their own metadata used for implementing the specific policies described in Sections 3.2 and 3.3.

3.1.3. Buffer state management

A buffer transitions through various states during its lifetime in Bufferflow. Figure 5 illustrates the possible states of a buffer and the operation types that might trigger a transition. In the diagram, we assume that the events triggering a state change arrive either from the user or from internal system modules. For the user events, we illustrate a collective put_all/get_all access, in which a number of producers collectively write a buffer through a `put_all` call and a number of consumers collectively read a buffer through a `get_all` call (for noncollective calls the number of expected producers and consumers will be 1). The system modules implementing various policies including scheduling and allocations can cause buffer changes depending on internal events and thresholds, as described below.

Each buffer enters into the system in the *ALLOCATED* state, when it is created. A buffer remains in the *ALLOCATED* state until all producers finish updating the buffer. For facilitating spatial locality, a buffer is not allowed to be swapped out from the *ALLOCATED* state. This approach gives buffer

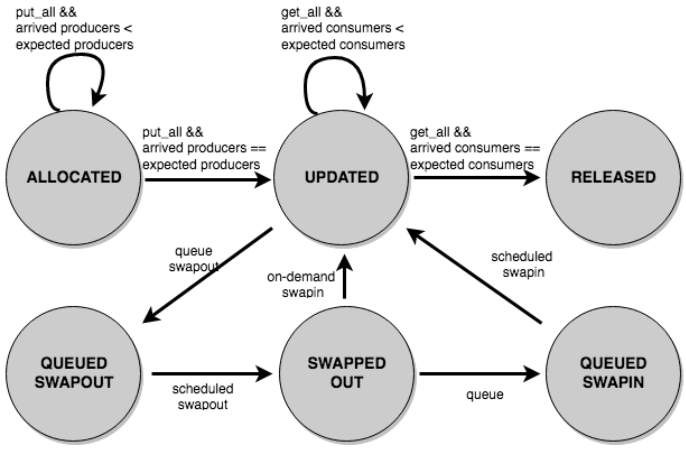

Figure 5: Buffer states diagram.

allocation priority to data-parallel producers of a started collective operation instead of later arrived operations. Assuming application progress, the buffer state will eventually transit to *UPDATED* when all the producers have arrived and finished their `put_all` operations.

Once in the *UPDATED* state a buffer can be either used for serving waiting consumers or swapped out to secondary storage. The swapping is triggered by an eviction policy implemented in the scheduling layer. If the buffer is elected for eviction, it is enqueued waiting for being swapped-out (*QUEUED SWAPPOUT* state). Once the buffer is swapped-out its state transitions to *SWAPPED OUT*. Subsequently, the buffer can be swapped-in either on-demand upon a consumer request or proactively by placing the swap-in request into a queue (*QUEUED SWAPIN* state).

When all expected consumers have arrived, the buffer is released and returned to the allocation layer.

3.2. Scheduling layer

The *buffer scheduler* component is in charge of hiding the latency of costly buffer management activities by scheduling buffer allocation and buffer swapping events. It consists of three modules: asynchronous buffer allocation, buffer

swapper, and buffer eviction. Each module can implement different policies. In the remainder of the section, we describe each module and the policies that have been implemented so far.

3.2.1. Asynchronous Buffer Allocation

Two classical approaches exist for performing synchronous allocation in buffering systems: on-demand allocation and preallocation. In this section we present the implementation of a hybrid asynchronous approach for performing buffer allocation in Bufferflow.

The asynchronous buffer allocation module provides implementations of buffer allocation scheduling policies that allow costly memory allocations to be asynchronously performed concurrent with other system activities. The time and space of any of these policies are bounded by the two classical approaches named above.

At one extreme, an on-demand buffer allocation scheduler makes optimal use of space and worse use of time (the whole overhead is paid at the time of synchronous allocation). At the other extreme, a static buffer allocation scheduler preallocates in a best-effort approach a large amount of memory before the application starts. If the preallocated memory is less than the total memory required by the application, this scheduling policy offers zero dynamic allocation time overhead (everything is statically allocated) and worst space overhead (the space is kept allocated even during periods when it is not needed).

Our new adaptive asynchronous allocation policy can be controlled to offer space and time overheads within the bounds of these two extremes. This policy can be customized for finding a suitable trade-off between the time and space overheads within the optimal and worst-case limits of the on-demand and static scheduling policies. The implementation of this policy provides adaptiveness to the producer/consumer access rates, asynchronous memory allocation, and asynchronous buffer eviction support according to a custom replacement policy (e.g., most recently used).

The adaptive scheduling policy is based on high and low watermarks. Its

purpose is to keep the memory footprint of the library under control by reacting to the difference of speed between producers and consumers.

The low watermark value is used by the adaptive scheduling policy as a threshold for the number of available allocated buffers (both in use and free). The low watermark is reached when the producers are faster than consumers and are continuously requesting buffers from the scheduling layer without the consumers having the chance to free some of them. When it is reached, the buffer scheduler component dispatches an asynchronous task for increasing the number of allocated buffers.

The high watermark is used to limit the number of available buffers the scheduling policy keeps in memory. Imagine a situation where consumers are becoming faster than producers and they start to release buffers. A large number of free buffers may reside in the main memory. Usually, there is no justification to keep such a large amount of memory allocated, and therefore an asynchronous task is dispatched that shrinks the internal memory pools.

The low-high watermark technique is a good heuristic for producer/consumer problems. By introducing this technique in the adaptive scheduling policy we want to leave a window of buffers that is large enough so that the asynchronous allocation mechanism can expand or shrink the available buffer space with minimal performance impact on the producers or consumers.

3.2.2. Buffer Swapper

The *buffer swapper* component assumes the existence of a local or distributed storage system. This component processes the requests received from the buffer scheduler to write or read buffers to and from persistent storage.

Bufferflow currently includes two swapping policies: *on-demand* and *asynchronous* swapping. On-demand swapping occurs when one or more consumers try to read a swapped buffer, causing the coordination layer to call into the scheduling layer to swap in the buffer. This policy corresponds to the transition from *SWAPPED OUT* to *UPDATED* state in Figure 5. Asynchronous swapping is activated when the memory footprint goes low. Instead of releasing the

free unused memory, the scheduling layer may decide to bring the buffer from storage into memory so that the consumers trying to read the buffer at some later point do not incur the latency involved in storage access. In this case, the buffers are placed in a queue in the order in which they have been swapped-out. Subsequently, the queue is processed for swapping in buffers until the memory capacity is full or the queue is empty. This policy corresponds to the state transitions from *SWAPPED OUT* to *QUEUED SWAPIN* to *UPDATED* in Figure 5.

On-demand swapping or a slow asynchronous swapping may cause back pressure on the producers, causing them to slow down. When the producers are faster than the consumers, the buffer space increases until reaching the maximum available memory. At this moment producers waiting for new available buffer space will have to slow down at the rate of the file system performance. However, this behavior is normal on current systems being caused by the limited amount of memory on the compute nodes.

3.2.3. Buffer eviction policy

The choice of buffers to be swapped out is driven by an eviction policy. The most recently used policy seems like the straightforward choice for producer/-consumer patterns: a buffer that was produced recently has a higher probability of being consumed later than the other buffers produced before. While this policy might fit some applications, however, others might have a special behavior that will not benefit from it. For these cases, Bufferflow provides an open interface that can be used for implementing of any new eviction policy.

3.3. Allocation layer

The allocation layer in Bufferflow contains optimizations that can offer significant performance improvement for applications with a particular allocation pattern. We designed the allocation layer as a flexible, highly optimized component that the scheduling layer can leverage to obtain a significant performance boost when coupled with the asynchronous buffer allocation mechanism. It is

composed of two layers: the *chunk pool* layer, which manages a memory area composed of fixed-size memory blocks, and the *buffer allocator* layer, which uses the chunk pool as a building block to support memory allocations that exceed the capacity of a *chunk pool*.

The *buffer allocator* component of Bufferflow is a heap memory allocator optimized for fixed-size chunks of memory that are relatively small. The default C heap allocator is slow mostly because it is designed as a general-purpose memory allocator for variable-size memory blocks having medium to large dimensions (hundreds of kilobytes). Moreover, programs do not have any control over the allocation mechanisms or the internal data structures of the allocator [18].

When designing a memory allocator, three main concerns arise: the allocation complexity, the deallocation complexity, and the expected allocation pattern. No silver bullet exists, since every allocator has its own weakness when it comes to a specific allocation pattern. Alexandrescu [18] describes four main allocation patterns that must be considered when designing an allocator: (1) bulk allocation (multiple objects allocated at the same time, such as arrays of objects); (2) reverse deallocation (objects are deallocated in the reverse order of their allocation); (3) same order for allocation and deallocation (objects are allocated and deallocated in the same order); and (4) butterfly pattern (allocations and deallocations do not follow a specific pattern).

While Bufferflow can support any of these patterns, for this work we chose to implement the third pattern, because we target scientific pipelines consisting of parallel producers/consumers. For these applications, a buffer allocated before other buffers has a higher probability of being deallocated first because it waited longer for its consumers to arrive.

4. Cooperative collective I/O implementation

This section presents the algorithmic details of CCIO, which have been included in the CLARISSE and Bufferflow implementations.

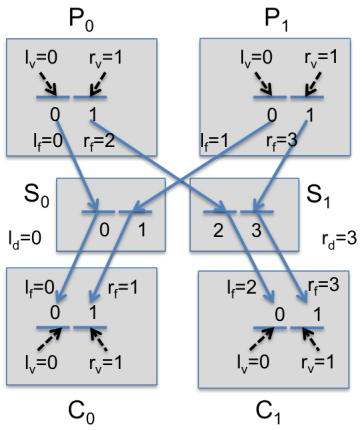

Figure 6: Numerical example for $n_p = 2$, $n_c = 2$, $n_s = 2$, $d = 4$, $b = 2$.

Algorithms 1-5 illustrate the integration between CLARISSE and Bufferflow. Our approach assumes that n_p concurrent and cooperative producers and n_c concurrent and cooperative consumers exchange information through a data set (file) of size d consisting of several fixed-sized (b) blocks distributed over n_s Bufferflow servers. In our first implementation, the blocks of the data set are distributed round-robin over all servers, but other distributions can be added with little effort in CLARISSE without any change in Bufferflow. The producers and consumers can declare views (e.g., through MPI-IO) on the data that they want to write or read, respectively. This is not a limiting assumption; our approach works well without declared views, for example when the view is the whole data set.

Figure 6 shows a simple numeric example to explain the algorithms. In this example we use $n_p = 2$, $n_c = 2$, $n_s = 2$, $d = 4$, and $b = 2$. The producer P_0 has declared a view on bytes 0 and 2 (sees them as if they were contiguously stored), and the producer P_1 has declared a view on bytes 1 and 3. The consumer C_0 has declared a view on bytes 0 and 1, and the consumer C_1 has declared a view on bytes 2 and 3. The block consisting of bytes 0 and 1 is mapped to server S_0, and the block consisting of bytes 2 and 3 is mapped to server S_1.

In the algorithms we use the following notation. The variable l subscripted

by v, f, or d is the leftmost byte of the data region of a view (v), a subset of the whole data set on which the view maps (f), or the whole data set (d). Analogously, r is the rightmost byte of a data region. For example, for P_1 $l_v = 0$, $r_v = 1$, $l_f = 0$, and $r_f = 2$. The whole data set is within $[l_d = 0, r_d = 3]$. We also denote fh as a handle representing the whole data set and bh as a handle representing a buffer of the data set.

Algorithm 1 Consumer collective read

```
1:  procedure MPI_FILE_READ_ALL(fh, offset, buf, count, type)
2:      l_v ← offset
3:      r_v ← offset + size(type) × count
4:      type_v ← view_type(fh)
5:      l_f ← map_offset(l_v, type_v)
6:      r_f ← map_offset(r_v, type_v)
7:      l_d ← min_{∀p∈P} l_f
8:      r_d ← max_{∀p∈P} r_f
9:      for all s ∈ S do
10:         bitmap, offs, lens, datamap ← map_data(l_d,
11:                 r_d, l_v, r_v, type_v, s)
12:         rpc_bitmap_op(s, fh, l_d, r_d, bitmap, READ_ALL)
13:         rpc_read_op(s, fh, offs, lens, datamap)
14:     end for
15: end procedure
```

For Algorithms 1 and 2, the variable *type* represents the MPI data type used for representing data in the memory of consumer or producer processes, and `size(type)` returns the number of bytes in the type (e.g., using `MPI_Type_size`). The function `view_type` returns the type used for constructing the view of a process, which is stored internally by CLARISSE when the view is declared (not shown here for simplicity). Additionally, the original CLARISSE code offers functions for mapping views to shared data. The function `map_offset` maps a view offset to a offset within the shared data set. For example, for P_0

r_f = `map_offset`(r_v) = `map_offset`(1) = 2. The function `map_data` maps a whole-view region to a particular server by computing a *bitmap* of the active blocks on the server, a list of segments represented as offsets and lengths within each block, and a *data_map* representing the addresses in the local memory of the data to be sent to/received from servers. In our example, the bitmap is 1 for the mappings P_0-S_0, P_0-S_1, P_1-S_0, P_1-S_1, C_0-S_0, and C_1-S_1 and 0 for C_0-S_1 and C_1-S_0. For P_0-S_0 the offsets for the first block are 0 and 2 with lengths 1 and 1. The data map has offset 0 starting from buf and length 2.

Algorithm 2 Producer collective write

```
1:  procedure MPI_File_write_at_all(fh, offset, buf, count, type)
2:      l_v ← offset
3:      r_v ← offset + size(type) × count
4:      type_v ← view_type(fh)
5:      l_f ← map_offset(l_v, type_v)
6:      r_f ← map_offset(r_v, type_v)
7:      l_d ← min_{∀p∈P} l_f
8:      r_d ← max_{∀p∈P} r_f
9:      for all s ∈ S do
10:         bitmap, offs, lens, datamap ← map_data(l_d,
11:                 r_d, l_v, r_v, type_v, s)
12:         rpc_bitmap_op(s, fh, l_d, r_d, bitmap, WRITE_ALL)
13:         rpc_write_op(s, fh, offs, lens, datamap)
14:     end for
15: end procedure
```

Algorithm 1 presents the pseudo-code of the implementation of the collective read operation executed by the concurrent consumers. For our numerical example, *count* is 2, and *type* is `MPI_BYTE` (having the size 1). Lines 1 and 2 compute l_v and r_v (they have value 0 and 1 for both processes). Line 3 retrieves the view set in a previous step (in our example the view of both consumer processes is contiguous mapping at global offsets 0 and 2, respectively). Lines 4 and 5

Algorithm 3 Server bitmap operation

```
1: procedure BITMAP_OP(client, fh, l_d, r_d, bitmap, op)
2:     B ← data_blocks(fh, l_d, r_d)
3:     n ← 0
4:     for all bh ∈ B do
5:         if GET_NTH_BIT(bitmap, n) == 1 then
6:             if op == WRITE_ALL then
7:                 bh.cnt_producers++
8:             else
9:                 bh.cnt_consumers++
10:            end if
11:        end if
12:        n++
13:    end for
14: end procedure
```

compute the lower and upper bound of the mapping of consumer processes on the data set (C_1: l_f=0, r_f=1 and C_2: l_f=2, r_f=3). Lines 6 and 7 compute the lower and upper bound of the data set (for both C_1 and C2: l_d=0, r_d=3). This can be implemented as an "all reduce" operation in MPI (`MPI_Allreduce`). Subsequently, for all servers, we first call remotely the bitmap operation and then remotely call the read operation. The separation of the two operations is motivated by the fact that for large operations, the data is transferred in several rounds, while the bitmap can be sent only once given the small space overhead involved with each operation. All server operations are remotely called by producers and consumers through remote procedure calls (RPCs). The RPCs are implemented based on MPI point-to-point communication.

Algorithm 2 presents the pseudo-code of the implementation of the collective write operation executed by the concurrent consumers. For our numerical example *count* is 2, and *type* is `MPI_BYTE` (having the size 1). The pseudo-code is similar to the one for read except that the write operation is called remotely

at all servers. For our numerical example the computed values are as follows: P_0 and P_1: l_v=0,r_v=1, P_0:l_f=0 and r_f=2, P_1:l_f=1 and r_f=3, and P_0 and P_1: l_d=0, r_d=3.

Algorithm 3 presents the pseudo-code of the implementation of the bitmap operation at the server. This operation is called by both producers and consumers and is necessary for detecting termination based on counting the number of producer and consumer processes involved in read/write. Line 1 identifies the buffers managed by the server and involved in the operation in the domain comprised between l_d and r_d. Subsequently, for each block, if the associated bit is 1, the number of producers or consumers is incremented according to the type of operations. The numbers of the active producers and consumers are stored in a structure associated with the buffer handle *bh*.

Algorithm 4 presents the pseudo-code of the implementation of the read operation at the server. The code first increments the number of arrived consumers. Subsequently, it either calls the Bufferflow operation `get_vector_all` and returns the data if all the producers have already arrived or enqueues the operation for further processing otherwise.

Algorithm 4 Server read operation

```
1:  procedure READ_OP(consumer_rank, bh, offs, lens, size)
2:      bh.arrived_consumers++;
3:      if bh.arrived_producers == bh.cnt_producers then
4:          get_vector_all(bufservice, bh,
5:              c.offs, c.lens, c.size, dataout,
6:              bh.cnt_consumers)
7:          send(c.rank, dataout)
8:      else
9:          c = {consumer_rank, offs, lens, size}
10:         enqueue(bh.waiting_consumers, c)
11:     end if
12: end procedure
```

The write operation at the server is also straightforward. It first calls the `put_vector_all` operation of Bufferflow for writing the data into the buffers, and then it increments the number of arrived producers. If all active producers have arrived, it serves all waiting consumers by calling the `get_vector_all` operation of Bufferflow.

Algorithm 5 Server write operation

```
1: procedure WRITE_OP(bh, offs, lens, size, data)
2:     put_vector_all(bufservice, bh, offs,
3:         lens, size, data, bh.cnt\_producers)
4:     bh.arrived_producers++;
5:     if bh.arrived_producers == bh.cnt_producers then
6:         for c ∈ bh.waiting_consumers do
7:             get_vector_all(bufservice, bh,
8:                 c.offs, c.lens, c.size, dataout,
9:                 bh.cnt_consumers)
10:            send(c.rank, dataout)
11:        end for
12:    end if
13: end procedure
```

5. Experimental results

This section presents the experimental results. After describing the experimental setup, the section focuses on three main aspects targeting to quantify the benefits of our solution:

- Section 5.2 compares CCIO and TPIO at large scale in terms of performance and resource cost.
- Section 5.3 evaluates the performance of producer-consumer patterns for decoupling buffering with Bufferflow

- Section 5.4 describes experiments for evaluating the elasticity of the buffer memory pool and the impact on the memory usage for various values of low and high watermarks.

In the evaluation, we used synthetic benchmarks and kernels of real scientific applications.

5.1. Experimental setup

All results presented in the following sections were obtained by deploying Bufferflow on the Archer supercomputer, located at EPCC and part of the UK National Supercomputing Service. Archer is based on the Cray XC30 MPP [6] with the addition of external login nodes, postprocessing nodes, and storage system. The Cray XC30 consists of 4,290 compute nodes each with two 12-core Intel Ivy bridge processors and 64 GB of memory. Each 12-core processor in a compute node is an independent NUMA region with 32GB of local memory.

Archer is equipped with the Cray proprietary Aries interconnect [7], which links all the compute nodes in a dragonfly topology. The topology consists of Aries routers that connect four compute nodes, cabinets that are composed of 188 compute nodes, and groups that consist of two cabinets grouped together. Aries connects groups to each other by all-to-all optical links and the nodes within a group by 2D all-to-all electrical links. All these characteristics enable Archer to provide low-latency high-throughput data transfer between the compute nodes. The data infrastructure in Archer relies on multiple high-performance parallel Lustre [7] file systems that manage a total amount of 4.4 PB of storage.

In our experiments, we used two application kernels (VPICIO and VORPALIO) and a synthetic benchmark that allows evaluating Bufferflow in a controlled environment.

[6] A description of Cray XC30 is available at `http://www.cray.com/products/computing/xc-series`.

[7] A desciption of the Archer hardware is available at `http://www.archer.ac.uk/about-archer/hardware/`.

VPICIO and VORPALIO are two I/O kernels extracted from real scalable applications at LBNL [8]. VPICIO is an I/O kernel of VPIC, a scalable 3D electromagnetic relativistic kinetic plasma simulation [22]. VPICIO receives as parameters the number of particles and a file name, generates a 1D array of particles, and writes them to a file. We extended VPICIO to write the array over a number of time steps. VORPALIO is an I/O kernel of VORPAL, a parallel code simulating the dynamics of electromagnetic systems and plasmas [23]. The relevant parameters of VORPALIO are 3D block dimensions (x, y, and z), a 3D decomposition over p processes (p_x, p_y, and p_z, where $p_x \times p_y \times p_z = p$), and the number of time steps. In each step, VORPALIO creates a 3D partition of blocks and writes it to a file.

Both I/O kernels perform storage I/O through the H5Part library, which can store and access time-varying, multivariate data sets through the HDF5 library. For collective I/O, the HDF5 library employs MPI-IO. In the evaluation for this paper, we used two MPI-IO implementations: the MPI-IO implementation on top of CLARISSE and the MPI-IO implementation of MPICH [9] The original benchmarks perform only file system write. We integrated the original benchmarks into a miniworkflow by implementing the symmetrical read functionality. By this approach, a data analysis code can be plugged directly into the original benchmark for extracting the output, thus avoiding costly storage accesses. In this way, Bufferflow acts as an intermediary elastic parallel I/O storage layer.

The synthetic benchmark does a parallel file copy based on the MPI framework. It assumes the existence of a global namespace for buffers (e.g., handled by an underlying distributed file system).

The benchmark architecture, illustrated in Figure 7, decouples the parallel processes of an application into three categories: producers, consumers, and servers. Instead of tightly coupling parallel producer and consumer processes in order to transfer the content of a file, we chose to have the producers delegate

[8] VPICIO and VORPALIO are available at https://sdm.lbl.gov/exahdf5/software.html

[9] MPICH distribution is available at `http://www.mpich.org/`.

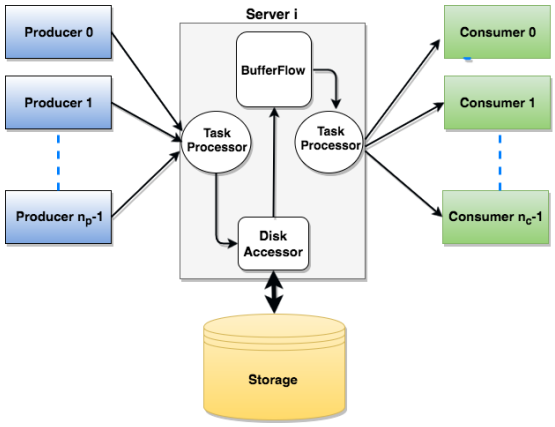

Figure 7: Benchmark architecture.

the data transfer to the servers so that they can continue to execute the rest of the application logic without facing the high time penalty of data transfers.

We assigned equal pieces of the file to each producer process in a MapReduce fashion. Instead of starting to send the data to the consumer processes, the producers send metadata requests to the servers that are responsible for later transferring the data to the consumers based on the information listed in the metadata received from the producers. The metadata consists of the operation type (put for producers and get for consumers), the Bufferflow buffer handler, and the access size. The benchmark executes a configurable number of server processes. Each producer request goes through a shuffling phase that redirects it to a particular server, thus ensuring a balanced load on each server.

Inside each server process there are four components linked in a pipeline: a task processor responsible of forwarding the requests from the producers to the disk accessor component. The disk accessor fetches the data from the system storage based on the metadata received from the task processor and sends it to Bufferflow. Later the data are read by the task processor responsible for transmitting the data to the consumer processes.

5.2. Comparison between CCIO and TPIO

In this section we report the results of the experimental evaluation of CCIO and TPIO for the VPICIO and VORPAL kernels. For both kernels, we evaluated a producer-consumer pattern consisting of the original benchmark, which is a parallel producer of n_p processes generating data, and a parallel consumer running on n_c processes. Bufferflow was employed for the communication among parallel producers and consumers through n_s servers. We evaluated 9 configuration cases: n_p= 88, 176, 352, 704, 1,408, 2,816, 5,632, 11,264, and 22,528 processes; n_c= 88, 176, 352, 704, 1,408, 2,816, 5,632, 11,264, and 22,528 processes, and n_s= 16, 32, 64, 128, 256, 512, 1,024, 2,048, and 4,096 processes. For each of these cases we compared the two scenarios from Figures 3 and 2: (1) the MPI-IO implementation of CCIO on top of Bufferflow and (2) the MPI-IO implementation of TPIO in ROMIO on top of a Lustre file system. For both cases, we used the same buffer size (16 MB), which was the default value for TPIO on Archer and the same number of aggregators for TPIO as buffering servers for CCIO. CCIO used 9 % more processes than TPIO, given that the buffering servers are independent processes. Due to the 1 to 1 mapping of processes to physical cores imposed by Archer's job scheduler, CCIO required 9% more resources than TPIO.

VPICIO was run for 1,048,256 particles per process and 1 time step, which for n_p processes generated total data-set sizes of $n_p \times 4$ MB (i.e., 88 GB for 22,528 processes). For VORPALIO, we used block dimensions of sizes $x = 100$, $y = 100$, and $z = 60$; decompositions of sizes $p_x = 1$, $p_y = 1$, and $p_z = n_p$; and 1 time step, which for n_p processes generated total data-set sizes of $n_p \times 13.7$ MB (i.e., 301.6 GB for 22,528 processes). Given these parameters, the data pressure per server is about 22 MB/server for VPICIO and 75 MB/server for VORPALIO in all six configured cases.

The experiments for n_p= 88, 176, 352 and n_c= 88, 176, 352 were repeated 5 times; the experiments for n_p= 704, 1,408, 2,816 and n_c= 704, 1,408, 2,816 were repeated 3 times; and the experiments for n_p= 5,632, 11,264, 22,528 and n_c= 5,632, 11,264, 22,528 were executed only once. The reason for using fewer

repetitions for larger process counts was that we had a limited core-hour allocation on the Archer machine. The standard deviation for the write and read results is plotted in the graphs.

Figures 8 and 9 show the results for VPICIO and VORPALIO. For each kernel, the upper graph shows the speedup of CCIO over TPIO for aggregated throughput of write and read operations. The middle graph represents the aggregate write throughput of CCIO and TPIO in logarithmic scale. The lower graph plots the aggregate read throughput of CCIO and TPIO in logarithmic scale.

We note that for aggregated write and read operations, the speedup of CCIO over TPIO is higher than 1 in all cases. In the best case it is larger than 11.7 for VORPALIO $n_p = n_c = 5632$. For VPICIO, the speedup decreases when the access size and number of accessing producers/consumers increase. In this case the data pressure per server is low (22 MB/server), which most probably allows the read operations to fully benefit from client-side caching of Lustre. For larger data pressure per server, this phenomenon does not occur anymore. For VORPALIO the speedup is 5.3 for $n_p = n_c = 88$ and 5.7 for $n_p = n_c = 176$. Then it decreases to 3.6 for $n_p = n_c = 352$ and 2 for $n_p = n_c = 704$, but it strongly increases up to 11.7 for $n_p = n_c = 5632$.

To better understand these results, we analyzed the aggregate write and read performance. For write operations, the fact that Bufferflow is decoupled from a back-end file system such as Lustre enables better utilization of the local memory of the compute nodes without the need to write back the data remotely. This results in significant speedups for write operations, reaching up to 57 times improvement for VORPALIO when $n_p = n_c = 176$ and 50 times for VPICIO when $n_p = n_c = 352$. For read operations, CCIO outperforms TPIO in most cases, with speedups of up to 5.1 times for VORPALIO when $n_p = n_c = 88$ and 4 times for VPICIO when $n_p = n_c = 88$. TPIO reads slightly outperform CCIO reads for VPICIO when $n_p = n_c = 2{,}816$ and 22,528. These results show that the speedup improvement for the aggregated operations stems mostly from the write operations that are not efficiently supported by the Lustre

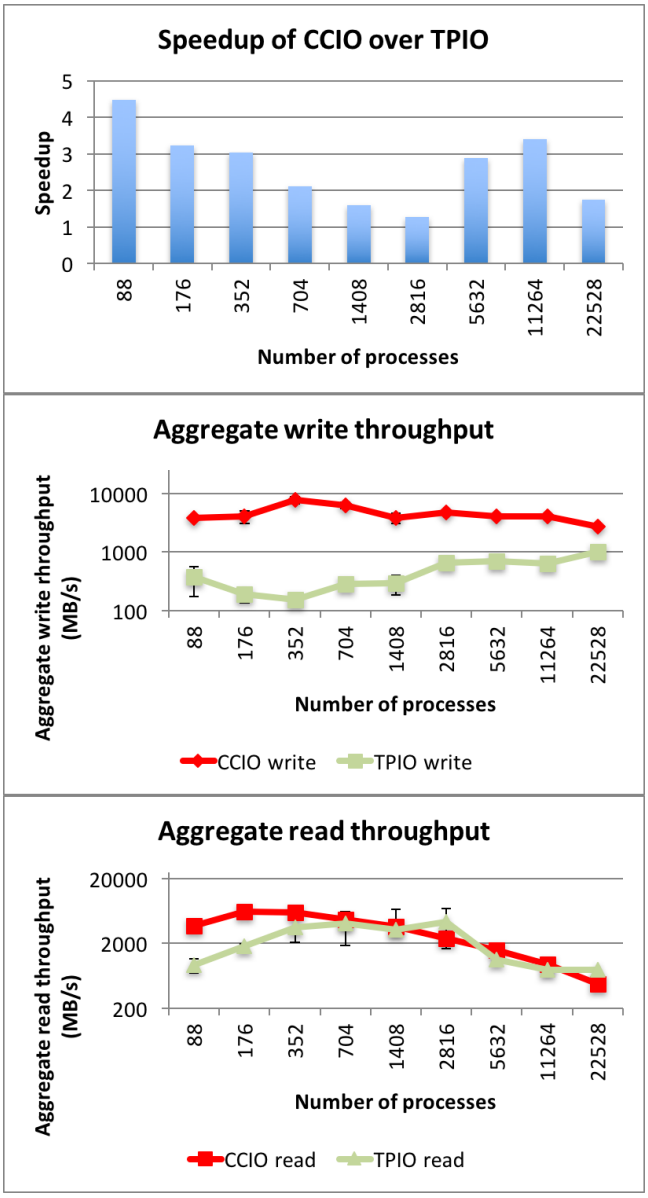

Figure 8: VPICIO results for the nine configuration cases: the speedup of CCIO over TPIO for aggregated write and read operations, the aggregate write throughput for CCIO and TPIO (in log-scale), and the aggregate read throughput for CCIO and TPIO (in log-scale).

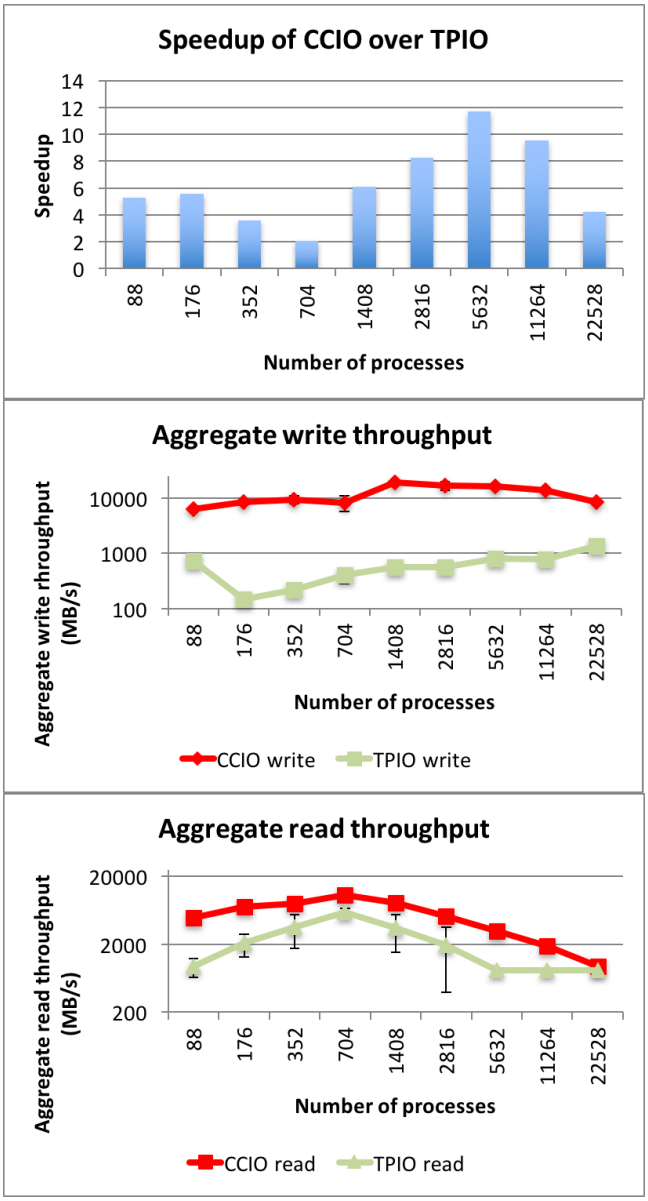

Figure 9: VORPALIO results for the nine configuration cases: the speedup of CCIO over TPIO for aggregated write and read operations, the aggregate write throughput for CCIO and TPIO (in log-scale), and the aggregate read throughput for CCIO and TPIO (in log-scale).

file system. Thus, decoupling the producers and consumers from the file system provides significant performance improvement, in some cases even of one order of magnitude.

Based on the write and read results, we note that CCIO shows better performance stability than does TPIO. The relative standard deviation, measured as standard deviation divided by mean, is much higher for TPIO operations than for CCIO operations in all cases but two in which they are almost the same. The relative standard deviation for TPIO writes can get as high as 52% from the mean for VPICIO when $n_p = n_c = 88$ and 103% for reads for VPICIO $n_p = n_c = 1408$. In comparison, the maximum relative standard deviation for CCIO writes can get as high as 31% from the mean for VORPALIO when $n_p = n_c = 704$ and 17% for reads for VORPALIO $n_p = n_c = 88$. The relative standard deviation indicates that the TPIO solution is more prone to performance variation than is CCIO. The reason is that the data transfer can cause contention due to both network traffic and the file system in TPIO. In comparison, the performance of the CCIO solution is affected only by network traffic.

5.3. Decoupling of parallel producers-consumers

The objective of this section is to evaluate the performance benefits of decoupling of parallel producers and consumers through Bufferflow. For this evaluation we use the synthetic benchmark described in Section 5.1, which we will refer to as Bufferflow-Copy, and a tightly coupled producer-consumer implementation of the same functionality based on MPI, which we will refer to as MPI-Copy. Both benchmarks copy a file and are implemented using MPI point-to-point communication.

For Bufferflow-Copy, the producers divide the files into chunks and delegate the copying process to Bufferflow servers by evenly shuffling the assigned chunks. The Bufferflow-Copy consumers receive the chunks and write them to the file. For MPI-Copy, the producers divide the file into chunks, shuffle the chunks among themselves, read the assign chunks, and send them to consumers using MPI. The MPI-Copy consumers simply receive chunks and write them to the

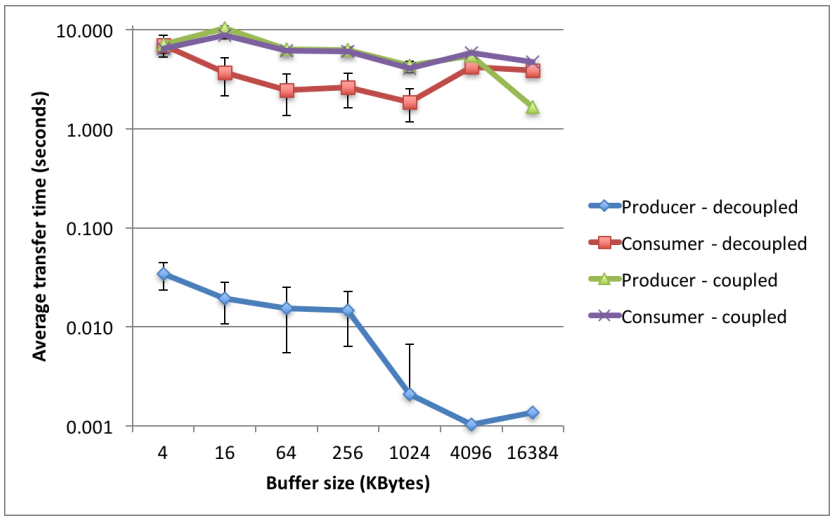

Figure 10: Decoupling MPI data transfers through Bufferflow servers

file.

Our hypothesis is that tightly coupling producers and consumers causes higher data transfer times due to the synchronization of involved processes.

To evaluate Bufferflow-Copy, we deployed 64 parallel producers, 64 parallel consumers and 64 servers for copying a file of 1 GB. We varied the Bufferflow buffer size between 4 KB to 1 MB. The low watermark is 10% and the high watermark is 60%. In this experiment we intentionally avoided reaching the low watermark by setting the aggregate size of buffering of all 64 servers to 16 GB (i.e., 256 MB/server).

Figure 10 shows the results. As expected, the Bufferflow producers delegating the file copy to servers finish fast. For 4 KB buffers the producers finish in 34 milliseconds on average. As the buffer size increases, the number of transfers decreases, causing the finish time of producers to drop to 2 milliseconds for buffers of 1 MB. In comparison, the MPI-Copy producers require on average between 4.3 seconds and 10.4 seconds, because they have to synchronously carry out the data transfer to the consumers.

The significant speedup of the producers is justified by delegating the asyn-

chronous transfers to the servers. However, the interesting part of this experiment is the consumer results. The average transfer time of the consumers is up to 2.5 times faster for buffers of 64 KB. The slow transfers of MPI-Copy can be explained by the tight coupling between producers and consumers, causing an MPI-Copy consumer process to wait for a matching producer request to complete.

This experiment proves that the decoupling technique provided by Bufferflow can significantly benefit data-parallel producer-consumer applications at the cost of additional processes and memory used by the Bufferflow servers. The additional processes are not expected to consume critical resources, given that the computational and networking capacity of HPC machines is increasing significantly more than the storage I/O bandwidth. However, memory is expected to become a scarcer resource. Thus, using memory efficiently will become increasingly important.

5.4. Evaluation of the allocation scheduling policies

The target of this section is to evaluate the elasticity of the buffer memory pool and the impact on the memory usage for various values of low and high watermarks. We analyze the three allocation scheduling policies presented in Section 3.2.1: adaptive, static, and on-demand. In this evaluation we use the benchmarking framework from Figure 7, with 64 producers and 64 consumers concurrently generating 16,384 requests to 1 server. The size of each request was 64 KB. For the adaptive policy the low watermark was 10% and the high watermark was 50%. Figure 11 illustrates the buffer memory footprint as a time series of the number of allocated buffers after serving each request.

The static allocation scheduling policy has a constant buffer memory footprint during the execution of the benchmark, in other words, the maximum number of buffers Bufferflow was configured with in the initialization phase.

The on-demand scheduling generates a constant increase in the allocated buffers, with the ascending part matching the duration of the producers' execution continuously requesting buffers from the scheduler. When the producers

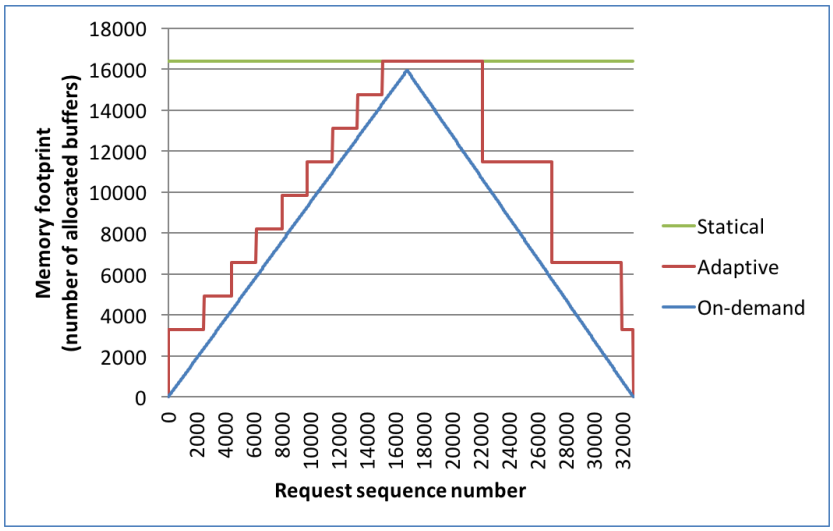

Figure 11: Memory footprint for adaptive, on-demand, and static allocation scheduling

finish their jobs, the consumers release buffers until the benchmark finishes.

The memory footprint of the adaptive policy evolves as a step function, because the policy allocates and deallocates buffers in chunks when it reaches the low and high watermarks.

Figure 12 shows that the adaptive scheduling policy has a higher average buffer memory footprint than the on-demand policy, but lower than the static allocation policy. For this experiment, the average memory footprint of the adaptive allocation policy is 10303, which is 37% less than the static allocation policy and 29% worse than the on-demand allocation policy.

The following figures illustrate the memory footprint generated by the adaptive scheduling policy for the Bufferflow-Copy benchmark. For each figure we ran Bufferflow against specific values of the low and high watermarks represented as percentages of the max memory limit configured in the initialization phase.

Figure 13 illustrates a deployment strategy for the scheduling layer in Bufferflow where extreme values are used for the low and high watermarks, that is 1% and 100% of the maximum memory configured in the initialization phase. The

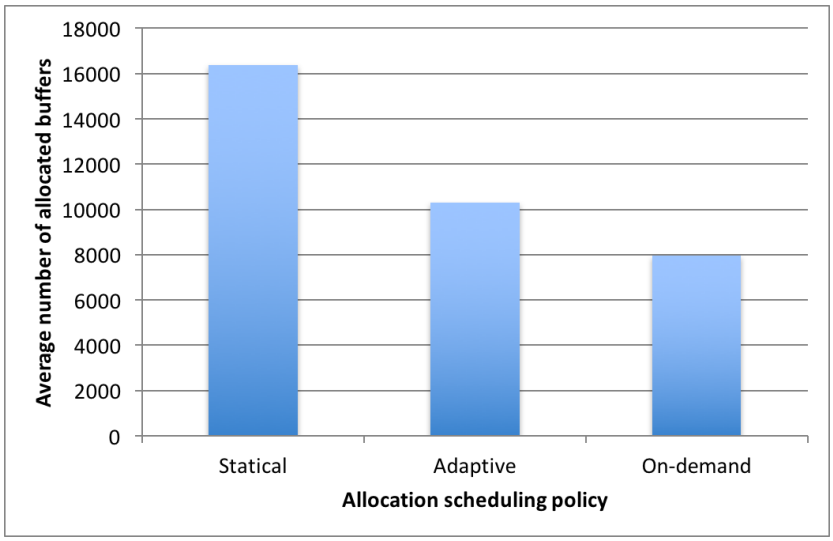

Figure 12: Average number of buffers allocated by adaptive, on-demand, and static allocation scheduling

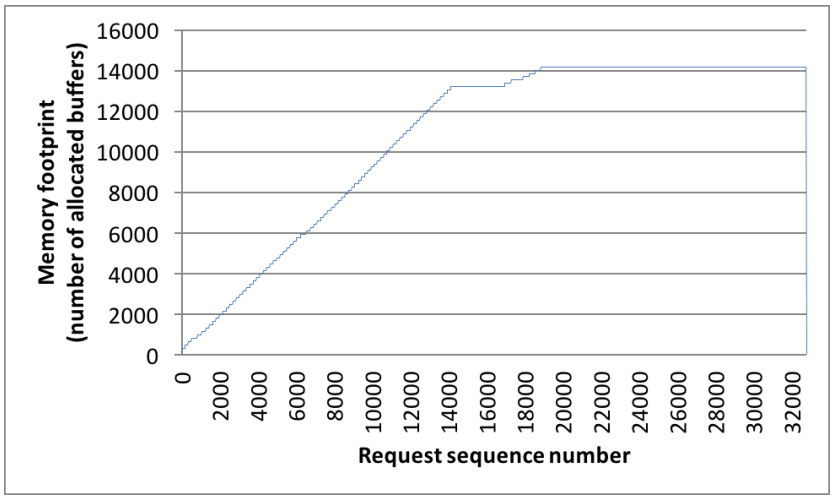

Figure 13: Memory footprint for adaptive allocation scheduling with 1% low watermark and 100% high watermark

results are expected. The low watermark is hit repeatedly, while the high watermark is not. As we can see in the plot, the buffer memory footprint increases until the producers finish their execution; then it reaches a standstill because

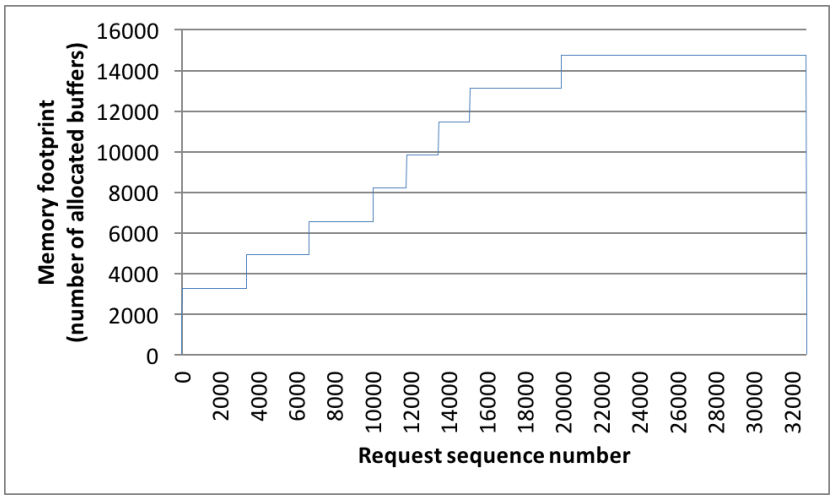

Figure 14: Memory footprint for adaptive allocation scheduling with 10% low watermark and 90% high watermark

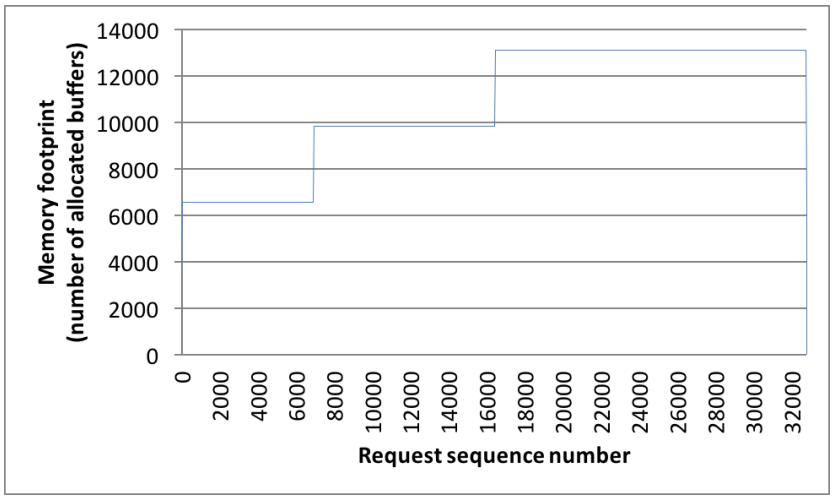

Figure 15: Memory footprint for adaptive allocation scheduling with 20% low watermark and 80% high watermark

neither of the consumers triggers a shrink event, since the high watermark is too high. Such an approach is undesirable: it hurts the producers' access time because of the low watermark being hit too frequently; it may overload the internal event queues in Bufferflow; and it is memory inefficient, because the high

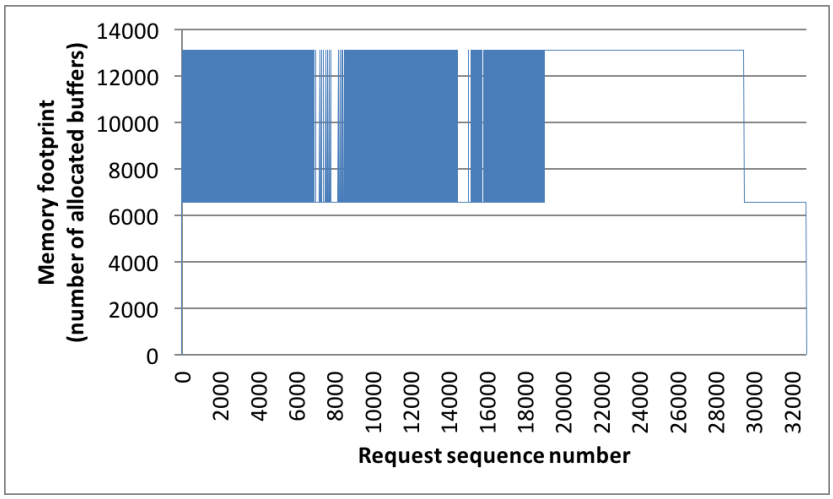

Figure 16: Memory footprint for adaptive allocation scheduling with 40% low watermark and 60% high watermark

watermark does not get hit at all.

A 10%-90% choice for the low-high watermarks is better than the previous choice, as shown in Figure 14. It both makes producers hit the low watermark less frequently and allows a small amount of flexibility for consumers to trigger memory shrinks.

Figure 15 illustrates the buffer memory footprint for a 20%-80% low-high watermark choice. As we can observe from the plot, the producers trigger fewer expand events. However, more free buffers in the window waiting to be requested, means a higher probability for the producers to trigger expand events less frequently. The interesting fact in this figure is that given a lower high-watermark than in the previous figure, one would expect a shorter plateau max value of the buffer memory footprint, and thus more shrink events on the descending side of the plot. But in fact, the standstill buffer memory footprint is longer. This is due to an overlapping of the producer and consumer execution such that when the buffer memory footprint reached to top value, the producers achieved the same access rate as the consumers.

We also present a case that should be avoided at all costs: choosing the

low and high watermarks too close to each other. As we see in Figure 16 for a 40%-60% low-high watermark choice, the left side of the plot is dominated by an aggressive burst of expand and shrink events. This kind of unstable behavior results in increased latency penalties for both consumers and producers.

The evaluation shows that good low and high watermark values should not be far apart or too close to each other. In the first case the adaptive policy constantly increases the memory footprint with a too low memory deallocation, thus wasting memory. In the second case, the adaptive policy incurs a high penalty of memory allocation and deallocation operations due to the lack of stability of the buffer pool. While the fine-tuning of the values of low and high water marks can be specifically done depending on each application, we expect that a 10%-50% low-high watermark choice will work fine in majority of the cases.

6. Related work

This section presents related work in three areas: buffering in the HPC storage I/O software stack, storage I/O buffering for clouds, and buffering for HPC workflows.

6.1. Buffering in the HPC storage I/O software stack

In most approaches for the HPC software I/O stack the buffer management is embedded in the individual stack layers. Its policies are hard-wired and expose a few configuration parameters (e.g., for GPFS [6] buffer space size, write-back activation threshold). Moreover, buffering is decoupled from the coordination mechanisms that can be used for efficiently exploiting concurrent access patterns showing spatial and temporal locality, which are common properties of scientific workflows [24].

Parallel file systems [7, 6] running on large-scale HPC infrastructures use client-side caching, server-side caching, or both for buffering. However, these approaches have currently reached a scalability limit [8] because the storage

I/O bandwidth increases substantially more slow than does the computational power [25], and the still widely used POSIX file access requirements are poorly suited for highly concurrent I/O-intensive scientific applications and emerging workflow patterns.

For decoupling the I/O performance requirements of scientific applications from the modest performance of parallel file systems, the new generation of supercomputers contain burst buffers [11], nonvolatile memory, or storage technologies meant to absorb high peaks of data transfers. The architectural role of burst buffers and the design of suitable system software is still an open problem. Depending on different avenues and technologies, the Bufferflow approach could be used either as an intermediate memory layer with burst buffer persistent support or even as a buffer management for the nonvolatile memories.

In software for parallel programming, such as MPI, buffering is used in collective I/O methods [4, 26] for aggregating small pieces of data in order to reduce the contention that would result if these requests were presented directly to disks/file systems. These approaches typically use a fixed-size buffer, whose access is internally controlled by the implementation. Additionally, temporal locality cannot be exploited, for example by external consumers that try to access these data. For exploiting intraapplication temporal locality, previous work has integrated collective buffering and distributed caching [27, 28]. However, their solutions do not efficiently address the requirements of scientific workflows consisting of data-parallel consumers and data-parallel producers. Other solutions use write-back caches for absorbing the collective I/O writes, flushing the data in back ground and serving reads from caches [29, 12]. However, these solutions do not target a flexible open buffering architecture that allows the design and deployment of novel policies for elastic buffering, allocation, and replacement. The work presented in this paper addresses all these issues and opens up the software stack to a large number of combinations of novel techniques that can be used for improving the performance and scalability.

6.2. Buffering systems for clouds

Redis [10] is a distributed in-memory store that can be used as a database, cache, or message broker. Redis accepts various representations of data (e.g., strings, hash maps, lists). It provides data replication, multiple cache eviction policies, and disk persistence. Memcached[30] is a distributed key-value store optimized for storing small variable-sized chunks of data. Memcached does not have any understanding of the data it stores. It is designed to act as an in-memory cache across a set of servers in a cluster that is meant to speed data-intensive Web applications. While Redis and Memcached are widely used in the industry, they are less applicable to the scientific community programming environment. In particular, they do not efficiently support highly concurrent read/write access or data-parallel collective operations for efficiently exploiting the noncontiguity and spatial locality access properties of scientific applications [17].

Storage systems such as RAMCloud [31] and cluster computing frameworks such as Spark [32] offer best-effort approaches to keep the highest possible amount of application data in RAM, transforming the memory of distributed nodes into a large buffer space. This approach cannot be readily applied to currnet HPC environments, however, because scientific workflows consist of independent monolithic simulations and analysis programs that cannot be easily restructured in terms of other computational paradigms. Currently, there is an open discussion about the convergence of big data and HPC worlds [25], but that convergence is still in its infancy.

6.3. Buffering systems for HPC workflows

Scientific workflow frameworks such as Pegasus [33] and Swift [34] have traditionally relied on shared parallel file systems for transfering data between dependent tasks. However, the increasing scale of workflows and the modest file system scalability (as discussed above) have motivated the need to develop

[10] Available at `http://redis.io/`.

novel techniques for reducing the storage I/O dependency. Existing approaches include data staging [15, 35], in situ and in-transit data analysis and visualization [9, 36, 37], distributed scalable key-value stores as intermediary storage [16], publish-subscribe paradigms for coupling large-scale analytics [38], and flexible analytics placement tools [39].

Our approach to a large extent complements most of these efforts. Bufferflow can be used on nodes running these solutions for providing elastic buffer management and data parallel collective access to buffers. Additionally, our approach simplifies the implementation of coordinated access for concurrent parallel producers and consumers. The open architecture provides a good ground for tailoring buffer allocation and replacement to the individual needs of each application.

7. Conclusions

In this paper we have presented the design, implementation, and evaluation of cooperative collective I/O (CCIO), a new collective I/O technique for speeding up parallel producers and consumers, an increasingly common pattern for sharing data between scientific simulations and data analysis on large-scale HPC infrastructures. CCIO can be used through the portable MPI-IO interface and its design consists of two main modules: a data distribution module implemented in the data plane of the CLARISSE middleware [19] and a buffering module based on a novel buffering framework called Bufferflow. The data distribution module is in charge of distributing data between parallel producers or consumers through a data space of an MPI-IO file mapped on a set of distributed buffers. The buffering module consists of a list of distributed Bufferflow servers offering a novel buffering interface used for efficiently orchestrating parallel producers and consumers sharing data.

The experimental results, run up to 49,152 cores, illustrate the performance and scalability of CCIO. When compared to two-phased I/O, the parallel producers are shown to be sped up to 57 times and the parallel consumers up

to 5 times, leading to an aggregate improvement of producers and consumers of up to more than one order of magnitude. These performance improvement was obtained with about 9% more computing resources (cores). Additionally, we demonstrate that Bufferflow servers can be used to significantly speed up the shared data access by decoupling tightly interacting parallel producers and consumers. Finally, we show that Bufferflow can be efficiently used for elastic memory allocation policies that make a better usage of available memory through a mechanism based on low and high water marks.

The work presented in this paper represents just one step towards reforming the system software I/O stack of large-scale high-performance platforms. Currently, there is a strong need to dynamically balance the compute and storage I/O throughput by adapting to variable bursty I/O requirements of competing applications [1, 2]. In the future we plan to use the decoupling of control and data flow from the CLARISSE middleware [19] for designing novel control approaches that holistically address this problem for competing applications including producer-consumer patterns supported by the work described in this paper. Thus, we aim at globally compensating for the relatively slow increase of storage I/O bandwidth on future systems and the current limited scalability of parallel file systems.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 328582. We acknowledge that the results of this research have been achieved using the DECI resource ARCHER based in the UK at EPCC with support from the PRACE aisbl. This work has been partially supported by the Spanish Ministry of Economy and Competitiveness thought the project grant TIN2016-79637-P Towards Unification of HPC and Big Data paradigms. This work is

partially supported by the EU under the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS)

References

[1] Exascale Requirements Review, Tech. rep. (2016).

[2] ETP4HPC, ETP4HPC Strategic Research Agenda 3, Tech. rep., ETP4HPC (2017).

[3] F. Isaila, J. Garcia, J. Carretero, R. Ross, D. Kimpe, Making the Case for Reforming the I/O Software Stack of Extreme-Scale Systems, Elsevier's Journal Advances in Engineering Software.

[4] R. Thakur, W. Gropp, E. Lusk, Data sieving and collective I/O in ROMIO, in: Frontiers of Massively Parallel Computation, 1999. Frontiers '99. The Seventh Symposium on the, 1999, pp. 182–189. doi:10.1109/FMPC.1999.750599.

[5] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, P. Sadayappan, Scalable I/O Forwarding Framework for high-performance computing systems, in: Proceedings of IEEE Custer, 2009.

[6] F. Schmuck, R. Haskin, GPFS: A shared-disk file system for large computing clusters, in: Proceedings of the 1st USENIX FAST '02, USENIX Association, Berkeley, CA, 2002.

[7] Architecting a high performance storage system, Tech. rep., Intel HPC Data Division. (2014).

[8] N. Hemsoth, The slow death of the parallel file system, Next Platform.Available at http://www.nextplatform.com/2016/01/12/the-slow-death-of-the-parallel-file-system/.

[9] M. Dreher, B. Raffin, A flexible framework for asynchronous in situ and in transit analytics for scientific simulations, in: 14th IEEE/ACM CCGrid, Chicago, IL, USA, May 26-29, 2014, pp. 277–286.

[10] M. Dreher, B. Raffin, A Flexible Framework for Asynchronous In Situ and In Transit Analytics for Scientific Simulations, in: 14th IEEE/ACM CCGrid, Chicago, United States, 2014.

[11] N. Liu, J. Cope, P. H. Carns, C. D. Carothers, R. B. Ross, G. Grider, A. Crume, C. Maltzahn, On the role of burst buffers in leadership-class storage systems, in: Proceedings of MSST/SNAPI 2012, Pacific Grove, CA, 2012.

[12] F. Isaila, J. Garcia Blas, J. Carretero, R. Latham, R. Ross, Design and evaluation of multiple-level data staging for blue gene systems, IEEE Trans. Parallel Distrib. Syst. 22 (6) (2011) 946–959. doi:10.1109/TPDS.2010.127.
URL http://dx.doi.org/10.1109/TPDS.2010.127

[13] C. Chen, M. Lang, L. Ionkov, Y. Chen, Active burst-buffer: In-transit processing integrated into hierarchical storage, in: Proceedings of the 11th IEEE International Conference on Networking, Architecture, and Storage, 2016.

[14] T. Peterka, F. Cappello, Decaf: High-performance decoupling of tightly coupled flows. http://www.mcs.anl.gov/project/decaf-high-performance-decoupling-tightly-coupled-flows.

[15] C. Docan, M. Parashar, S. Klasky, Dataspaces: an interaction and coordination framework for coupled simulation workflows, Cluster Computing 15 (2) (2012) 163–181. doi:10.1007/s10586-011-0162-y.

[16] F. R. Duro, J. G. Blas, F. Isaila, J. M. Wozniak, J. Carretero, R. Ross, Flexible Data-Aware Scheduling for Workflows over an In-memory Object Store, in: 2016 16th IEEE/ACM CCGrid, 2016, pp. 321–324. doi:10.1109/CCGrid.2016.40.

[17] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, M. L. Best, File-

Access Characteristics of Parallel Scientific Workloads, IEEE Trans. Parallel Distrib. Syst. 7 (10) (1996) 1075–1089. doi:10.1109/71.539739.

[18] A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[19] F. Isaila, J. Carretero, R. Ross, Clarisse: A middleware for data-staging coordination and control on large-scale hpc platforms, 2016 16th IEEE/ACM CCGrid (2016) 346–355.

[20] The HDF group., Available at http://www.hdfgroup.org/HDF5/ (2017).

[21] MPI Forum., Available at http://www.mpi-forum.org/ (2017).

[22] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, T. J. T. Kwan, Ultrahigh Performance Three-Dimensional Electromagnetic Relativistic Kinetic Plasma Simulationa), Physics of Plasmas 15 (5) (2008) 055703.

[23] C. Nieter, J. R. Cary, VORPAL: A Versatile Plasma Simulation Code, J. Comput. Phys. 196 (2) (2004) 448–473.

[24] T. Shibata, S. Choi, K. Taura, File-access patterns of data-intensive workflow applications and their implications to distributed filesystems, in: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10, ACM, New York, NY, USA, 2010, pp. 746–755. doi:10.1145/1851476.1851585.

[25] D. A. Reed, J. Dongarra, Exascale Computing and Big Data, Commun. ACM 58 (7) (2015) 56–68. doi:10.1145/2699414.

[26] M. Chaarawi, S. Chandok, E. Gabriel, Performance Evaluation of Collective Write Algorithms in MPI I/O, in: Proceedings of the 9th International Conference on Computational Science: Part I, ICCS '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 185–194. doi:10.1007/978-3-642-01970-8_19.

[27] F. Isaila, G. Malpohl, V. Olaru, G. Szeder, W. Tichy, Integrating Collective I/O and Cooperative Caching into the Clusterfile Parallel File System, in: Proceedings of the 18th ICS, New York, NY, USA, 2004, pp. 58–67. doi:10.1145/1006209.1006219.

[28] W.-k. Liao, A. Ching, K. Coloma, A. Nisar, A. Choudhary, J. Chen, R. Sankaran, S. Klasky, Using MPI File Caching to Improve Parallel Write Performance for Large-scale Scientific Applications, in: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC '07, ACM, New York, NY, USA, 2007, pp. 8:1–8:11. doi:10.1145/1362622.1362634.

[29] W.-k. Liao, K. Coloma, A. Choudhary, L. Ward, Cooperative write-behind data buffering for mpi i/o, in: Proceedings of the 12th European PVM/MPI Users' Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface, PVM/MPI'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 102–109. doi:10.1007/11557265_17.
URL http://dx.doi.org/10.1007/11557265_17

[30] B. Fitzpatrick, Distributed caching with Memcached, Linux J. 2004 (124) (2004) 5–.

[31] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, S. Yang, The ramcloud storage system, ACM Trans. Comput. Syst. 33 (3) (2015) 7:1–7:55. doi:10.1145/2806887.

[32] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets, in: Proceedings of the 2Nd USENIX HotCloud, Berkeley, CA, USA, 2010, pp. 10–10.

[33] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, D. S. Katz, Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Sci. Program. 13 (3) (2005) 219–237.

[34] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, I. Foster, Swift: A language for distributed parallel scripting, Parallel Comput. 37 (9) (2011) 633–652. doi:10.1016/j.parco.2011.05.005.

[35] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan, S. Klasky, Just in Time: Adding Value to the IO Pipelines of High Performance Applications with JITStaging, in: Proceedings of the 20th International Symposium on High Performance Distributed Computing, HPDC '11, ACM, New York, NY, USA, 2011, pp. 27–36. doi:10.1145/1996130.1996137.

[36] V. Vishwanath, M. Hereld, M. E. Papka, Toward Simulation-Time Data Analysis and I/O Acceleration on Leadership-Class Systems., in: LDAV, IEEE, 2011, pp. 9–14.

[37] M. Dorier, G. Antoniu, F. Cappello, M. Snir, L. Orf, Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O, in: CLUSTER - IEEE International Conference on Cluster Computing, IEEE, Beijing, China, 2012.

[38] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, N. Podhorszki, Flexpath: Type-Based Publish/Subscribe System for Large-Scale Science Analytics, in: 14th IEEE/ACM CCGrid, 2014, pp. 246–255.

[39] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, H. Yu, FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics, in: 27th IEEE IPDPS 2013, 2013, pp. 320–331.