Bufferflow: Collective Data-Parallel Buffering For Scientific Workflows

Florin Isailă, Jesus Carretero - *University Carlos III of Madrid (Spain)*
Robert Bindar, Georgiana Dolocan - *Politechnic University of Bucharest (Romania)*
Tom Peterka - *Argonne National Laboratory (USA)*

Abstract—Scientific applications running on current high-performance computing (HPC) platforms are increasingly based on complex workflows that require continuous interactions and coupling between multiple parallel components in order to avoid the current storage systems bottleneck. In this paper we present Bufferflow, a novel buffering framework that compensates for the lack of scalability of parallel file systems with customizable buffering policies that can be used for end-to-end performance optimization of data flows. Bufferflow goes beyond the traditional put/get store model by offering data-parallel collective primitives that can be used for efficiently composing parallel applications (e.g., simulation, analysis, visualizations) into complex workflows. Additionally, Bufferflow provides various high-performance implementations of asynchronous policies: concurrent data-parallel read/write access, elastic buffer allocation, and swapping. The experimental results demonstrate the flexibility, performance, and scalability of the Bufferflow framework.

Index Terms—HPC, parallel I/O, workflows

1 Introduction

The scale of contemporary HPC platforms increased significantly in the past few years. This explosive growth of computational resources engendered new and innovative scientific practices capable of generating a vast amount of data. The composition of multiple computations into workflows is an active area of research in the HPC community as it progresses toward the exascale era [1].

An important topic of discussion is the need to decompose monolithic systems such as large-scale parallel file systems into independent services that can exceed the current uses on contemporary HPC infrastructures. For instance, today there is poor support for scientific workflow I/O or for popular data analytics patterns, such as MapReduce, streaming, and iterative computations [2]. Also, monolithic systems cannot cope with the increasing depth and variety of storage hierarchy [3]. Additionally, computing and storage I/O need to be decoupled in order to allow each system to scale independently.

Scientific workflows are constructed from multiple data parallel producer-consumer dataflows, which are the primitives of data transfer between components. These dataflows require coordination between the endpoints to avoid overflows, but synchronous control is often undesirable because it forces the producer and consumer processes to advance at the same pace. Decoupling a tightly coupled producer-consumer pipeline into a buffered dataflow leads to key improvements such as better overall performance and fault tolerance. In order to achieve this decoupling, a buffering scheme must be introduced between the participants in the data exchange so that the producers can function at the required speed while buffered data is asynchronously sent to the consumers.

The current software I/O stack of large-scale HPC systems does not offer proper mechanisms to address this problem. This stack consists of several layers: scientific libraries (e.g., HDF5 [4]), middleware (e.g., MPI-IO [5]), I/O forwarding (e.g., IOFSL [6]), and file systems (e.g., GPFS [7], Lustre [8]). Improving the scaling of this distributed stack by several orders of magnitude is complex, because the lack of data flow coordination across layers. In particular, data buffering is hard-wired in each stack layer; is managed mostly statically; and typically lacks support for either on-demand space management, efficient vectorial operations, or collective I/O. For instance, the MPI-IO implementations [9] offer support for collective I/O, but the buffer management is not elastic; the data shuffling is hard-wired; the small granular noncontiguous access is inefficient [10]; and control and data management are intermingled and not customizable [11].

Recently, our work on CLARISSE (Cross-Layer Abstractions and Run-time for I/O Software Stack of Extreme-scale systems) [12] started to address the poor extensibility and the lack of global data flow coordination of the existing I/O stack. CLARISSE decomposes the software I/O stack into three layers: data, control, and policy [13]. The control backplane coordinates the data transfer reacting to different system events coming from a publish/subscribe infrastructure. Policies such as load balancing, fault tolerance, and data staging can easily be implemented by using the control plane API. The data plane in CLARISSE is composed of multiple producer-consumer pipelines through which the data is transferred. This paper introduces Bufferflow, a novel buffering scheme that efficiently supports parallel dataflows bewteen producers and consumers in the CLARISSE data plane. Bufferflow has been integrated in CLARISSE, but it can be used as an independent service by any third-party workflow middleware.

The work presented in this paper is motivated by the following steps that are needed to improve the scalability of the I/O stack: (1) decoupling the tightly coupled producer-consumer pipelines; (2) identifying novel primitives that can be used for composing highly efficient services for the future-generation software I/O stack; (3) opening up the memory and storage management of each data plane layer to user-specific allocation and replacement policies (both these policies are currently embedded in various layers of the I/O stack); and (4) compensating for the lack of scalability of parallel file systems with customizable buffering policies that tailor end-to-end performance optimization of data flows.

In this work we take an important step toward decomposing the system software stack into independent services. We present a novel buffering service that can be used as a building block for scalable storage systems. Our buffering system productively and efficiently supports various requirements such as adaptiveness to unpredicted data volume variations and low-overhead data transfers of both individual HPC simulations and composite workflows of scientific applications.

The main contributions of this paper are the following. First, Bufferflow goes beyond the traditional put/get store by offering *data-parallel* primitives in both scalar and vector versions. These primitives can be used to efficiently compose parallel applications (e.g., simulation, analysis, visualizations) into complex workflows. Second, Bufferflow is the first system to offer *collective access semantics* for efficiently sharing buffers among decoupled parallel producers and consumers. Third, Bufferflow provides 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. Fourth, we present a novel portable MPI-IO-based implementation of *coordinated collective write and read operations* leveraging the CLARISSE middleware and Bufferflow framework. The experimental results demonstrate the flexibility, performance, and scalability of the Bufferflow framework.

The ideas described in this paper have been implemented in two software packages that are freely available for download and evaluation: Bufferflow[1] and CLARISSE [2].

The remainder of the paper is organized as follows. Sections 2 and 3 describe the design and implementation of Bufferflow. Section 4 presents the integration of Bufferflow in the software I/O stack of large-scale parallel supercomputers. 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 Architectural overview

Bufferflow is an adaptive buffering system targeting concurrent data-parallel producer-consumer coordination and high-rate transfers of chunks of raw data.

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

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

2.1 Integration of CLARISSE and Bufferflow

Figure 1 displays a high-level view of the integration between CLARISSE and Bufferflow. The figure shows n_p producers and n_c consumers performing data-parallel accesses on n_s servers. Each CLARISSE server leverages Bufferflow for local buffer management, while CLARISSE orchestrates the global data flows. Data and control planes are decoupled in the design of CLARISSE, so that novel policies can be enforced on top of the control layer. This paper focuses on novel techniques for integrating Bufferflow into the data plane. However, this work also opens up the possibility of leveraging the control plane for the adaptation of the buffering substrate. This particular issue will be the subject of future work.

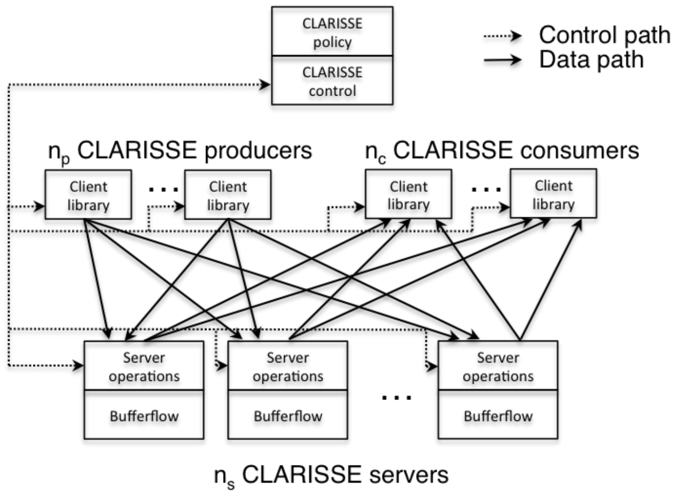

Figure 1. High-level architecture of the integration of CLARISSE and Bufferflow.

The remainder of this section focuses on an overview of Bufferflow as an independent library that can be used by CLARISSE or by any other workflow engine. Sections 3 and 4 present the implementation details of the CLARISSE and Bufferflow integration, respectively.

2.2 Bufferflow architecture

Bufferflow is composed of three layers, shown in Figure 2: the *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 of the key contributions, Bufferflow offers mechanisms for coordinating both data-parallel accesses within one application

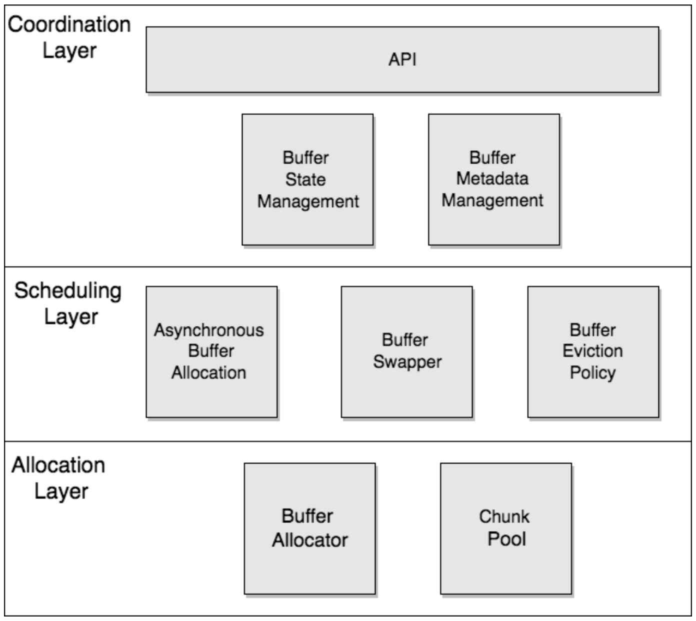

Figure 2. Bufferflow high-level architecture.

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 2 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 IMPLEMENTATION

This section describes an implementation of the three architectural layers shown in Figure 2.

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.

Middleware such as CLARISSE uses Bufferflow by creating handles for data blocks and leveraging the put/get calls described below for reading and writing data from/into internal buffers.

Bufferflow offers independent and 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. The mapping of a data set on a set of distributed Bufferflow buffers and the required access orchestration is delegated to the middleware by using a set of Bufferflow nodes as shown in the lower part of Figure 1. Section 4 discusses how this is achieved in the integration of Bufferflow into the CLARISSE middleware.

```
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 [14]. 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 [14].

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 nonswappable 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.

The API calls discussed above are blocking. However, the Bufferflow API does not necessarily enforce synchronization. A blocking get can be always called from an I/O thread, while the main computation thread is executed asynchronously.

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.

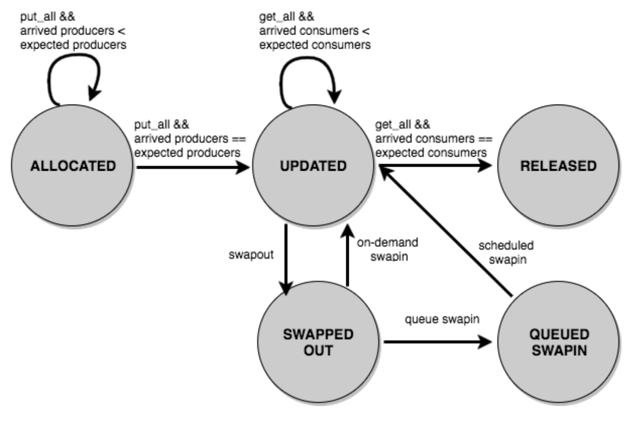

Figure 3. Buffer states diagram.

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 3 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 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 transitions into the *SWAPPED OUT state*. 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), whenever the system load demand decreases.

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 such 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 3. 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 3.

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 [15].

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 [15] 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 CLARISSE AND BUFFERFLOW INTEGRATION

This section presents the integration of Bufferflow into the CLARISSE data plane.

The CLARISSE data plane includes mechanisms for concurrently transferring data from compute nodes to storage nodes either collectively or independently [12]. Our previous work offered no efficient mechanism for decoupling and coordinating the transfers between concurrent producers and consumers. Bufferflow addresses this gap in a transparent manner in the storage I/O stack. Applications built on high-level scientific libraries such as HDF5 [4] or middleware such as MPI-IO [5] can run unmodified on top of CLARISSE/Bufferflow.

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 CLARISSE 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 limitating assumption; our approach works well without declared views, for example when the view is the whole data set.

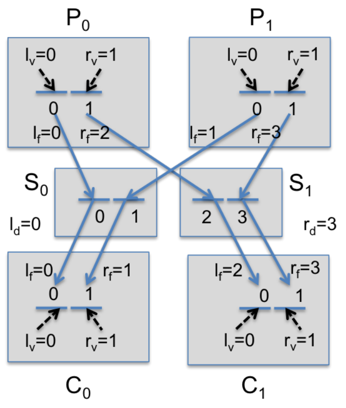

Figure 4. Numerical example for $n_p = 2$, $n_c = 2$, $n_s = 2$, $d = 4$, $b = 2$.

Figure 4 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.

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 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 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 1 Consumer collective read

```
1:  procedure READ_ALL(fh, buf, count, type, offset)
2:      $l_v \leftarrow offset$
3:      $r_v \leftarrow offset + \texttt{size}(type) \times count$
4:      $type_v \leftarrow \texttt{view\_type}(fh)$
5:      $l_f \leftarrow \texttt{map\_offset}(l_v, type_v)$
6:      $r_f \leftarrow \texttt{map\_offset}(r_v, type_v)$
7:      $l_d \leftarrow \min_{\forall p \in P} l_f$
8:      $r_d \leftarrow \max_{\forall p \in P} r_f$
9:      for all $s \in S$ do
10:         $bitmap, offs, lens, datamap \leftarrow \texttt{map\_data}(l_d,$
11:                 $r_d, l_v, r_v, type_v, s)$
12:         $\texttt{rpc\_bitmap\_op}(s, fh, l_d, r_d, bitmap, \texttt{READ\_ALL})$
13:         $\texttt{rpc\_read\_op}(s, fh, offs, lens, datamap)$
14:     end for
15: end procedure
```

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 2 Producer collective write

```
1:  procedure WRITE_ALL(fh, buf, count, type, offset)
2:      $l_v \leftarrow offset$
3:      $r_v \leftarrow offset + \texttt{size}(type) \times count$
4:      $type_v \leftarrow \texttt{view\_type}(fh)$
5:      $l_f \leftarrow \texttt{map\_offset}(l_v, type_v)$
6:      $r_f \leftarrow \texttt{map\_offset}(r_v, type_v)$
7:      $l_d \leftarrow \min_{\forall p \in P} l_f$
8:      $r_d \leftarrow \max_{\forall p \in P} r_f$
9:      for all $s \in S$ do
10:         $bitmap, offs, lens, datamap \leftarrow \texttt{map\_data}(l_d,$
11:                 $r_d, l_v, r_v, type_v, s)$
12:         $\texttt{rpc\_bitmap\_op}(s, fh, l_d, r_d, bitmap, \texttt{WRITE\_ALL})$
13:         $\texttt{rpc\_write\_op}(s, fh, offs, lens, datamap)$
14:     end for
15: end procedure
```

Algorithm 3 Server bitmap operation

```
1:  procedure BITMAP_OP(client, fh, $l_d$, $r_d$, bitmap, op)
2:      $B \leftarrow \texttt{data\_blocks}(fh, l_d, r_d)$
3:      $n \leftarrow 0$
4:      for all $bh \in B$ 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
```

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 $c \in bh$.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
```

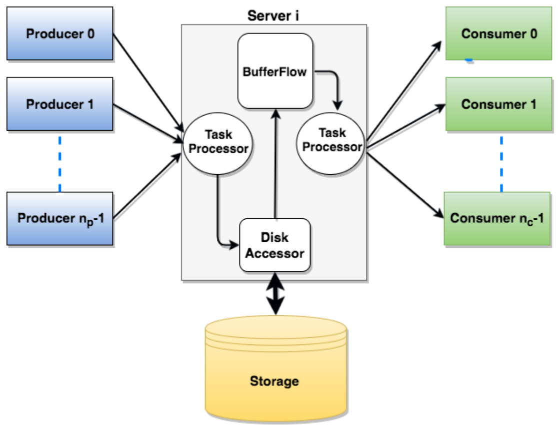

Figure 5. Benchmark architecture.

5 Experimental results

We start by describing the experimental setup. We then present results from running Bufferflow both in a controlled environment and integrated into the CLARISSE middleware. 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 [16] 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 [17], 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 [8] filesystems that manage a total amount of 4.4 PB of storage.

In our experiments, we used a synthetic benchmark that allows evaluating Bufferflow in a controlled environment and two application kernels (VPICIO and VORPALIO) that are used for evaluating the integration between Bufferflow and CLARISSE at large scale.

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 5, 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 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.

VPICIO and VORPALIO are two I/O kernels extracted from real scalable applications at LBNL [3]. VPICIO is an I/O kernel of VPIC, a scalable 3D electromagnetic relativistic kinetic plasma simulation [18]. 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 [19]. 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.

3. Available at https://sdm.lbl.gov/exahdf5/software.html

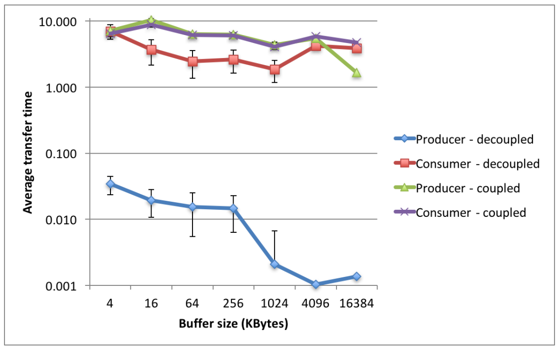

Figure 6. Decoupling MPI data transfers through Bufferflow servers

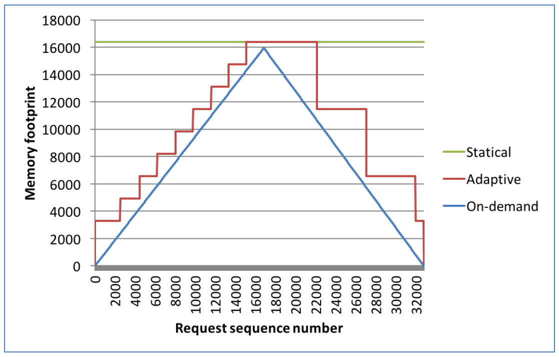

Figure 7. Memory footprint for adaptive, on-demand, and static allocation scheduling

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 [20]. 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.

5.2 Dataflow decoupling

In this section we compare the benchmark described above, 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. 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 point-to-point communications. The MPI-Copy consumers simply receive chunks and write them to the 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), in other words 10% of 16 GB is larger than 1 GB.

Figure 6 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 asynchronous 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.

5.3 Evaluation of the allocation scheduling policies

In this section we analyze the three allocation scheduling policies presented in Section 3.2.1: adaptive, static, and on-demand. In this evaluation we used the benchmarking framework from Figure 5, with 64 producers and 64 consumers concurrently generating 16,384 requests to 1 server. The size of each request was 64 KB. Thus, in this case the benchmark was copying a file of 1 GB. For the adaptive policy the low watermark was 10% and the high watermark was 50%. Figure 7 illustrates the buffer memory footprint for this evaluation.

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 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. The adaptive scheduling policy has a higher 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 does 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 8 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 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 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 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 9. It both makes producers hit the low watermark less frequently and allows a small amount of flexibility for consumers to trigger memory shrinks.

Figure 10 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 11, 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.

No perfect solution exists for picking the low and high watermark values. Each application has a different data access pattern, different rates for producers and consumers and runs on platforms with specific characteristics. Thus, the election of the low and high watermarks should be done empirically, e.g. based on benchmarks such as the one presented in this paper.

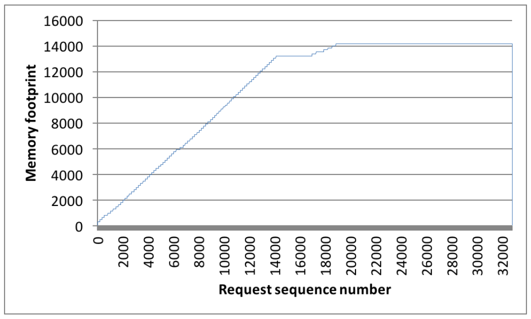

Figure 8. 1% low watermark, 100% high watermark

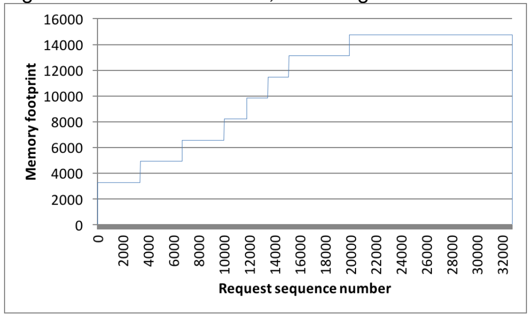

Figure 9. 10% low watermark, 90% high watermark

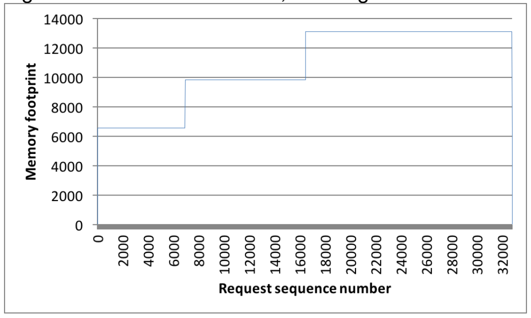

Figure 10. 20% low watermark, 80% high watermark

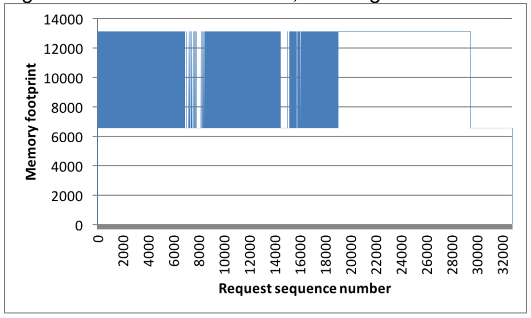

Figure 11. 40% low watermark, 60% high watermark

5.4 CLARISSE/Bufferflow evaluation

In this section we report the results of the experimental evaluation of CLARISSE based on Bufferflow for the VPICIO and VORPAL kernels in a weak scaling scenario. For both kernels, we evaluated a miniworkflow 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 6 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 two scenarios: (1) the MPI-IO implementation of collective I/O in CLARISSE on top of Bufferflow and (2) the MPI-IO implementation of collective I/O 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 ROMIO on Archer and the same number of aggregators for ROMIO as servers for CLARISSE. In the ROMIO implementation of collective I/O, aggregators play the same role as servers in CLARISSE: they aggregate data from parallel producers before writing to the file system for collective write and the reverse process for read [5]. CLARISSE servers are independent processes however, whereas ROMIO aggregators are a subset of the processes requesting the read or write operations.

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., 11 GB for 2,816 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., 37.7 GB for 2,816 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 result is plotted in the graphs.

Figures 12 and 13 show the results for VPICIO and VORPALIO. For each kernel, the upper graph shows the speedup of CLARISSE over ROMIO for aggregated throughput of write and read operations. The middle graph represents the aggregate write throughput of CLARISSE and ROMIO in logarithmic scale. The lower graph plots the aggregate read throughput of CLARISSE and ROMIO in logarithmic scale.

We note that for aggregated write and read operations, the speedup of CLARISSE/Bufferflow over ROMIO/Lustre 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, CLARISSE outperforms ROMIO 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. ROMIO reads slightly outperform CLARISSE 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 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 CLARISSE/Bufferflow shows better performance stability than does ROMIO/Lustre. The relative standard deviation, measured as standard deviation divided by mean, is much higher for ROMIO operations than for CLARISSE operations in all cases but two in which they are almost the same. The relative standard deviation for ROMIO 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 CLARISSE 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 ROMIO solution is more prone to performance variation than is CLARISSE. The reason is that the data transfer can have contention due to both network traffic and the file system in ROMIO. In comparison, the performance of the CLARISSE solution is affected only by network traffic.

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 [7] 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 [21].

Parallel file systems [7], [8] 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 [3] because the storage I/O bandwidth increases substantially more slow than does the

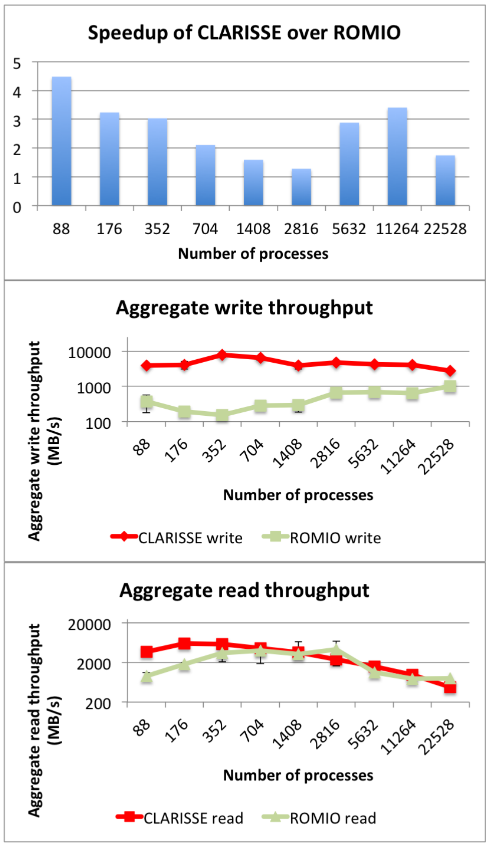

Figure 12. VPICIO results for the six configuration cases: the speedup of CLARISSE over ROMIO for aggregated write and read operations, the aggregate write throughput for CLARISSE and ROMIO (in log-scale), and the aggregate read throughput for CLARISSE and ROMIO (in log-scale).

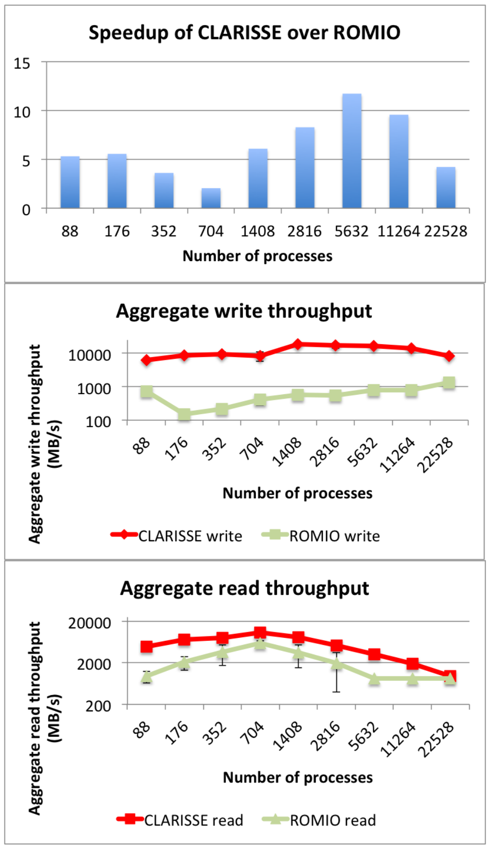

Figure 13. VORPALIO results for the six configuration cases: the speedup of CLARISSE over ROMIO for aggregated write and read operations, the aggregate write throughput for CLARISSE and ROMIO (in log-scale), and the aggregate read throughput for CLARISSE and ROMIO (in log-scale).

computational power [2], 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 [22], 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 [9], [23] 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 [24], [25]. However, their solutions do not efficiently address the requirements of scientific workflows consisting of data-parallel consumers and data-parallel producers. Additionally, to the best of our knowledge, the HPC software I/O stack lacks an 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 [26] 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 [27] 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 [14].

Storage systems such as RAMCloud [28] and cluster computing frameworks such as Spark [29] 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 [2], but that convergence is still in its infancy.

6.3 Buffering systems for HPC workflows

Scientific workflow frameworks such as Pegasus [30] and Swift [31] 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 novel techniques for reducing the storage I/O dependency. Existing approaches include data staging [32], [33], in situ and in-transit data analysis and visualization [34], [35], [36], distributed scalable key-value stores as intermediary storage [37], 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 Bufferflow, a new buffering framework to improve the performance and scalability of the I/O performed by large-scale parallel scientific workflows. We described Bufferflow's novel data-parallel collective put/get primitives that enable a simple portable implementation of complex workflow patterns in the CLARISSE middleware. Additionally, Bufferflow has an open architecture that allows for user-specific allocation and replacement policies. In this paper, we presented the design and implementation of a set of policies at various architectural layers including: asynchronous data access, buffer life-cycle management, asynchronous buffer allocation, asynchronous swapping, eviction, and memory allocation.

The experimental results illustrate the performance and scalability benefits of Bufferflow. First, we demonstrated that decoupling workflows, which are tightly coupled based on MPI communication, can offer performance improvements of several orders of magnitude for producers and significant speedups for consumers. Second, we showed how our asynchronous allocation policy can be used for trading off time and space in order to adapt to different requirements of scientific workflows. Third, we demonstrated that a solution integrating Bufferflow and CLARISSE significantly outperforms the existing software I/O stack of current large-scale high-performance platforms and offers higher performance stability.

The work presented in this paper represents just one step towards decomposing the system software I/O stack of large-scale high-performance platforms into independent services. Further research is needed to better understand how independent services such as Bufferflow can compensate for the relatively slow increase of storage I/O bandwidth on future systems and the current lack of 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] E. D. et al., "PANORAMA: An approach to performance modeling and diagnosis of extreme scale workflows," *International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 4–18, 2017.

[2] D. A. Reed and J. Dongarra, "Exascale Computing and Big Data," *Commun. ACM*, vol. 58, no. 7, pp. 56–68, Jun. 2015.

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

[4] "The HDF group." 2017, Available at http://www.hdfgroup.org/HDF5/.

[5] "MPI Forum." 2017, Available at http://www.mpi-forum.org/.

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

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

[8] "Lustre parallel file system." Available at http://lustre.org/.

[9] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *Frontiers of Massively Parallel Computation, 1999. Frontiers '99. The Seventh Symposium on the*, Feb 1999, pp. 182–189.

[10] F. Tessier, P. Malakar, V. Vishwanath, E. Jeannot, and F. Isaila, "Topology-aware Data Aggregation for Intensive I/O on Large-scale Supercomputers," in *Proc. of the 1st Workshop on Optimization of Communication in HPC*, 2016, pp. 73–81.

[11] F. Isaila, J. Garcia, J. Carretero, R. Ross, and D. Kimpe, "Making the Case for Reforming the I/O Software Stack of Extreme-Scale Systems," *Elsevier's Journal Advances in Engineering Software*, 2016.

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

[13] F. Isaila and J. Carretero, “Making the case for data staging coordination and control for parallel applications,” 2015, Available at http://www.epigram-project.eu/wp-content/uploads/2015/07/Isaila-ExaMPI15.pdf.
[14] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis, and M. L. Best, “File-Access Characteristics of Parallel Scientific Workloads,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 10, pp. 1075–1089, Oct. 1996.
[15] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.
[16] “Cray xc30,” Available at http://www.cray.com/products/computing/xc-series.
[17] “Archer hardware.” Available at http://www.archer.ac.uk/about-archer/hardware/.
[18] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, “Ultrahigh Performance Three-Dimensional Electromagnetic Relativistic Kinetic Plasma Simulationa),” *Physics of Plasmas*, vol. 15, no. 5, p. 055703, May 2008.
[19] C. Nieter and J. R. Cary, “VORPAL: A Versatile Plasma Simulation Code,” *J. Comput. Phys.*, vol. 196, no. 2, pp. 448–473, May 2004.
[20] “MPICH,” http://www.mpich.org/.
[21] T. Shibata, S. Choi, and 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*, ser. HPDC ’10. New York, NY, USA: ACM, 2010, pp. 746–755.
[22] N. Liu, J. Cope, P. H. Carns, C. D. Carothers, R. B. Ross, G. Grider, A. Crume, and C. Maltzahn, “On the role of burst buffers in leadership-class storage systems,” in *Proceedings of MSST/SNAPI 2012*, Pacific Grove, CA, 04/2012 2012.
[23] M. Chaarawi, S. Chandok, and E. Gabriel, “Performance Evaluation of Collective Write Algorithms in MPI I/O,” in *Proceedings of the 9th International Conference on Computational Science: Part I*, ser. ICCS ’09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 185–194.
[24] F. Isaila, G. Malpohl, V. Olaru, G. Szeder, and 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.
[25] W.-k. Liao, A. Ching, K. Coloma, A. Nisar, A. Choudhary, J. Chen, R. Sankaran, and 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*, ser. SC ’07. New York, NY, USA: ACM, 2007, pp. 8:1–8:11.
[26] “Redis,” Available at http://redis.io/.
[27] B. Fitzpatrick, “Distributed caching with Memcached,” *Linux J.*, vol. 2004, no. 124, pp. 5–, aug 2004.
[28] 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, and S. Yang, “The ramcloud storage system,” *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, Aug. 2015.
[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *Proceedings of the 2Nd USENIX HotCloud*, Berkeley, CA, USA, 2010, pp. 10–10.
[30] 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, and D. S. Katz, “Pegasus: A framework for mapping complex scientific workflows onto distributed systems,” *Sci. Program.*, vol. 13, no. 3, pp. 219–237, Jul. 2005.
[31] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, “Swift: A language for distributed parallel scripting,” *Parallel Comput.*, vol. 37, no. 9, pp. 633–652, Sep. 2011.
[32] C. Docan, M. Parashar, and S. Klasky, “Dataspaces: an interaction and coordination framework for coupled simulation workflows,” *Cluster Computing*, vol. 15, no. 2, pp. 163–181, 2012.
[33] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan, and 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*, ser. HPDC ’11. New York, NY, USA: ACM, 2011, pp. 27–36.
[34] M. Dreher and 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.
[35] V. Vishwanath, M. Hereld, and M. E. Papka, “Toward Simulation-Time Data Analysis and I/O Acceleration on Leadership-Class Systems.” in *LDAV*. IEEE, 2011, pp. 9–14.
[36] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, “Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O,” in *CLUSTER - IEEE International Conference on Cluster Computing*. Beijing, China: IEEE, Sep. 2012.
[37] F. R. Duro, J. G. Blas, F. Isaila, J. M. Wozniak, J. Carretero, and R. Ross, “Flexible Data-Aware Scheduling for Workflows over an In-memory Object Store,” in *2016 16th IEEE/ACM CCGrid*, May 2016, pp. 321–324.
[38] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and 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, and H. Yu, “FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics,” in *27th IEEE IPDPS 2013*, 2013, pp. 320–331.

Florin Isaila is an Associate Professor of the University Carlos III of Madrid (Spain). As a Marie Curie fellow, he has been a visiting scholar at Argonne National Laboratory (USA) in the MCS Division (2013-2015). He received a PhD in Computer Science from University of Karlsruhe (Germany) in 2004 and a MS from Rutgers The State University of New Jersey in 2000. His primary research interests include high-performance computing, parallel I/O, storage systems, and distributed systems.

Jesus Carretero is a Full Professor of Computer Architecture and Technology at Universidad Carlos III de Madrid (Spain). His research activity is centered on high-performance computing systems, large-scale distributed systems, data-intensive computing, IoT and real-time systems. He is Action Chair of the IC1305 COST Action Network for Sustainable Ultrascale Computing Systems (NESUS). He has published more than 250 papers in journals and international conferences.

Robert Bindar is a graduate student at University Politehnica of Bucharest (UPB) in Romania. He received a BS degree in 2016 from UPB and during his last semester he was an exchange student at University Carlos III of Madrid. His current research interests include high-performance computing, distributed computing and machine learning.

Georgiana Dolocan is a graduate student at University Politehnica of Bucharest (Romania) and graduated from the same university with a BS degree in 2016. She interned as a Software Engineer during her university years in companies such as Intel and Bloomberg LP and in the last year of BS she was an ERASMUS student at University Carlos III of Madrid. Her research interests include high-performance computing, machine learning and big data computing.

Tom Peterka is a computer scientist at Argonne National Laboratory, fellow at the Computation Institute of the University of Chicago, adjunct assistant professor at the University of Illinois at Chicago, and fellow at the Northwestern Argonne Institute for Science and Engineering. His research interests are in large-scale parallelism for in situ analysis of scientific data. His work has led to three best paper awards and publications in ACM SIGGRAPH, IEEE VR, IEEE TVCG, and ACM/IEEE SC, among others. Peterka received his Ph.D. in computer science from the University of Illinois at Chicago, and he currently works actively in several DOE- and NSF-funded projects.