I/O Behind the Scenes: Bandwidth Requirements of HPC Applications With Asynchronous I/O

Ahmad Tarraf
Department of Computer Science
Technical University of Darmstadt
Darmstadt, Germany
ahmad.tarraf@tu-darmstadt.de

Javier Fernandez Muñoz
Department of Computer Science
University Carlos III of Madrid
Leganés, Spain
jfmunoz@inf.uc3m.es

David E. Singh
Department of Computer Science
University Carlos III of Madrid
Leganés, Spain
dexposit@inf.uc3m.es

Taylan Özden
Department of Computer Science
Technical University of Darmstadt
Darmstadt, Germany
taylan.oezden@tu-darmstadt.de

Jesus Carretero
Department of Computer Science
University Carlos III of Madrid
Leganés, Spain
jcarrete@inf.uc3m.es

Felix Wolf
Department of Computer Science
Technical University of Darmstadt
Darmstadt, Germany
felix.wolf@tu-darmstadt.de

2024 IEEE International Conference on Cluster Computing (CLUSTER) | 979-8-3503-5871-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/CLUSTER59578.2024.00044

***Abstract*—I/O bandwidth is a critical resource in an HPC cluster. As with all shared resources, its availability is impacted significantly by the users and the applications they execute. Without proper restrictions, jobs consuming more prominent portions of the I/O bandwidth can severely affect other jobs by notably prolonging their runtime. In such a context, applications that perform asynchronous I/O bring unique properties that allow for the reduction of such effects. That is, by limiting the bandwidth to the required one to perform the I/O in the background of the compute phases, I/O bursts can be flattened without significantly prolonging the application time, if at all. Hence, the bandwidth consumption of such applications is limited to what they need, sparing much of the system bandwidth to other applications. At the same time, these applications achieve higher parallel efficiency due to the overlapping of different resources (e.g., compute and I/O). This paper shows these aspects and demonstrates our approach to finding the required bandwidth for applications that use asynchronous I/O. Moreover, we apply it automatically using an MPI implementation of a bandwidth limitation approach at the application level. We validate our approach with several experiments on a large production cluster and show the impact of our approach on the application behavior and its importance for the system throughput.**

***Index Terms*—Bandwidth limitation, asynchronous I/O, I/O requirements, MPI-IO, requirement engineering**

This work was funded by the European Commission and the German Federal Ministry of Education and Research (BMBF) under the EuroHPC programmes DEEP-SEA (GA no. 955606, BMBF funding no. 16HPC015) and ADMIRE (GA no. 956748, BMBF funding no. 16HPC006K), which receive support from the European Union's Horizon 2020 programme and DE, FR, ES, GR, BE, SE, GB, CH (DEEP-SEA) or DE, FR, ES, IT, PL, SE (ADMIRE). Moreover, this work was also funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project No. 449683531 (ExtraNoise). Additionally, this work received further funding from BMBF and the Hessian Ministry of Science and Research, Art and Culture (HMWK) for supporting this work as part of the NHR funding. Furthermore, this work also received funding from the Spanish Research Agency with project reference PCI2021-121966. Finally, the authors gratefully acknowledge the computing time provided to them on the high-performance computer Lichtenberg II. This is funded by BMBF and the State of Hesse.

I. Introduction

Typical high-performance computing (HPC) applications represent simulations of large scientific problems executed on massive clusters with vast resources. Since 1993, the TOP500 list [1] ranks the top clusters twice yearly according to the resources. As HPC applications usually run on several nodes, the message passing interface (MPI) has been widely adopted in the parallel programming domain. While more computing resources (typically user-exclusive) are great for boosting computational performance, other shared resources, like Input/Output (I/O) performance, are often overlooked. Recent terms, such as the storage wall [2], try to quantify the I/O performance bottleneck from the application scalability perspective. Even though recent studies have shown exascale systems will provide a far greater increase in computational speed than I/O bandwidth [3], the distribution of the latter resource often remains at the mercy of the users.

In this context, performance degradation and I/O contention are often encountered as different jobs compete for shared resources as I/O [4], [5]. Several studies have been conducted to understand and characterize the I/O behavior of HPC applications [6]–[8]. Typical HPC applications are composed of alternating I/O and computational phases. The I/O phases tend to be periodic, with dominating write I/O operations (e.g., checkpointing) occurring in bursts synchronously across several processes [2]. Due to this behavior, I/O bursts are quite common in HPC [9]–[11]. In particular, I/O bursts refer to the phenomena when an enormous amount of I/O traffic is transferred in a short period of time [12]. Recent studies also discovered *spatial I/O burst*, which can occur due to unequal distribution of I/O workload across the compute nodes or the adjacent mapping of jobs with high I/O bursts [12]. Moreover, besides its burstiness and periodicity, dominant I/O behavior tends to be *repetitive* as applications are executed several times on HPC clusters [13]. Since resources like I/O

2168-9253/24/$31.00 ©2024 IEEE
DOI 10.1109/CLUSTER59578.2024.00044

are shared between users, a single simulation that extensively consumes I/O bandwidth can severely impact the runtime of other applications that are trying to access the shared parallel file system (PFS) [14]. This *cross-application interference* can impact I/O performance significantly, leading, for example, to a more than 200 times difference between identical workloads depending on the time when they were executed [15], [16]. For performance-degraded applications, this leads to prolonged execution times due to the lack of fairness. To control shared resource distribution among users, quality of service (QoS) is needed, which has a long research history [5], [17]–[22]. While approaches like external token-bucket mechanisms for bandwidth I/O control [21] or token bucket filter implemented in the Lustre file system's network request scheduler [22] can prevent a single application from consuming the entire I/O bandwidth, there are missed optimizations potentials, especially for applications that use asynchronous I/O.

Asynchronous I/O in HPC holds huge potential for improved parallel efficiency by utilizing different components of the system at the same time. Still, systematic studies of the benefits and limitations of asynchronous I/O for applications are missing [23]. In contrast to synchronous I/O, asynchronous I/O can hide partially, if not completely, the I/O phases behind the computational or communications phases. Thus, different application phases can overlap by utilizing different system components (e.g., storage devices and CPUs), achieving higher throughput and decreasing the time the application spends on I/O solely.

In this paper, we exploit applications that use asynchronous MPI-IO to (1) spare the I/O bandwidth of the system as much as possible while (2) increasing the parallel efficiency of these kinds of applications (i.e., utilizing different system components simultaneously). To realize this, our approach combines two aspects that are often handled separately. The first part of our approach examines the asynchronous I/O behavior of an application to determine the bandwidth *required* to execute these I/O operations in the background of the compute phases. The second part of our approach limits the application's bandwidth to the found value through an extended MPICH version. This way, applications with asynchronous MPI-IO only use as much bandwidth as needed to execute the I/O operation unnoticed, ideally with unchanged runtimes, while sparing the shared I/O bandwidth of the system for other applications (e.g., synchronous I/O applications), which have less flexibility regarding their I/O consumption. Thus, this paper contributes by:

- Presenting an approach that finds for an application with asynchronous MPI-IO the *bandwidth requirements* through the open-source library TMIO[1] (Tracing MPI-IO) with a very low overhead. Furthermore, the library provides insight into the asynchronous I/O performance (i.e., hidden and visible I/O).
- Extending the MPICH MPI implementation[2] with the ability to limit the bandwidth of asynchronous MPI-IO operations. We provide means to control the consumed bandwidth at the *user-level*.
- Combining the previous points to execute I/O behind the scenes seamlessly. Both approaches are stand-alone open-source tools that are publicly available. No code modifications are required; the library is preloaded, while the other is just a modified MPI version.
- Demonstrating the applicability of our approach based on examples executed on an HPC cluster in production.

This paper is structured as follows: We provide a motivation for our work in Sec. II followed by an overview of asynchronous MPI-IO in Sec. III. While Sec. IV shows how to find the required bandwidth of an application, Sec. V describes the extensions to MPICH and how we limit the bandwidth at the user level. We provide the experimental results in Sec. VI. Finally, after looking into the state-of-the-art in Sec. VII, we provide our future work and the conclusion in Sec. VIII.

II. Motivation

To improve the user experience for interactive data exploration, the consistency of I/O performance is essential to avoid over-stressing the storage system and unexpected job termination [24]. Many applications spend 15-40% of their execution time performing I/O, with a good probability that this value even further increases in the future [5], [25]–[28]. Considering this fact, and that numerous applications have bursty I/O behaviour [9]–[11], [28], I/O contention and performance degradation [29], [30] as well as mechanism to avoid them [4], [5], [17], [25] popular research topics. While several sophisticated QoS approaches [5], [17]–[22], for example from the application side [31] or using burst buffers [32] try to solve these problems, they often miss the optimization potentials a specific kind of I/O offers, namely asynchronous I/O. When an application uses synchronous I/O, the I/O requests are done after or before the computational phases. The time needed to perform the I/O request will directly contribute to the total application time. Thus, the higher the I/O bandwidth, the faster the application will be executed, justifying the need for the highest possible bandwidth. In contrast, an asynchronous application overlaps I/O with computational phases. Ideally, the I/O bandwidth required has a maximum limit equivalent to the computational time. This allows applications that use asynchronous I/O to flatten the I/O burst to some degree, up to the required bandwidth, without affecting their runtime significantly, if at all. Furthermore, as I/O threads can compete for resources with the compute threads [33], prioritizing compute threads can yield performance gains for asynchronous I/O as we demonstrate later. These two aspects are often overlooked in HPC, as the system is usually not concerned with the type (async/sync) of I/O but instead tries to respond as fast as possible to the stream of I/O requests. On the other hand, application developers that utilize asynchronous I/O in their code often assume that the overlap with the computational phases will be *good enough*, regardless of the configuration.

[1] https://github.com/tuda-parallel/TMIO/

[2] https://github.com/jfmunoz00/MPICH-IOBandwidth-Limitation

Closer to the original idea of performing asynchronous I/O in the background of the other phases, and without wasting computational resources on waiting or checking request completion and I/O bandwidth on too fast executions, we propose a solution that does not need any code modifications. Our library automatically limits the bandwidth consumption of applications that use asynchronous I/O to what is *needed*, flattening the I/O burst and *sparing* the bandwidth for other applications whose runtime directly depends on this resource. This way, the asynchronous I/O of these applications is performed *behind the scenes* with a lower impact on the shared resource and ideally without prolonging their execution times.

Our approach does not seek to compete with QoS approaches. Generally, QoS cannot be *perfectly* achieved from an application-centric view. On the contrary, bandwidth limitation from such a perspective *can* slow down the cluster's performance since contention is more likely to happen as the affected application performs I/O for a longer duration. A global view is required to utilize the system's bandwidth completely optimally. Yet, there might be cases where limiting the bandwidth of applications with asynchronous I/O only during contention can increase the system's performance. An example of this is shown in Figure 1, which has been simulated using ElastiSim [34]. Aligning with most settings of the Lichtenberg cluster (i.e., 500 nodes, 96 cores per node, and a PFS speed of 120 GB/s), we executed eight jobs mimicking HACC-IO (explained later in Sec. VI-B) with different node configurations (16, 32, or 96 nodes). Only job 4 performs asynchronous I/O, while the others execute their I/O synchronously. The top part of Figure 2 shows the bandwidth distribution for unrestricted access (i.e., fair bandwidth distribution according to the number of nodes). In contrast, the bottom part shows the case where the bandwidth of job 4 is limited according to our methodology during contention only. As Figures 1 and 2 show, almost all jobs profited from the spared bandwidth.

This paper aims to provide an approach that identifies

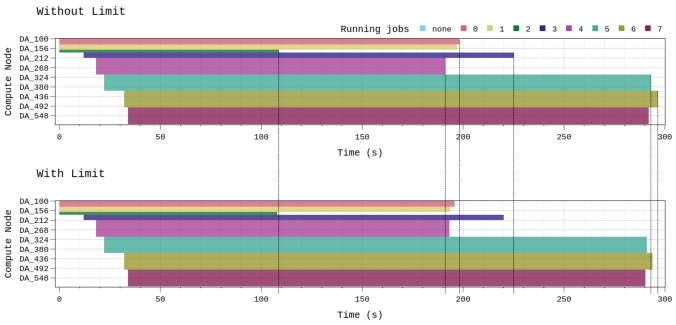

Fig. 1. Example showing that limiting the bandwidth of an application with asynchronous I/O allows other applications to utilize the spared bandwidth. The top part shows the runtime of the jobs in case there are no restrictions (fair bandwidth distribution according to the number of nodes). In contrast, the lower part shows the results in case job 4, the only job with asynchronous I/O, is slowed down to, at most, the required bandwidth during contention. Note that due to contention, the runtime of this job slightly increases.

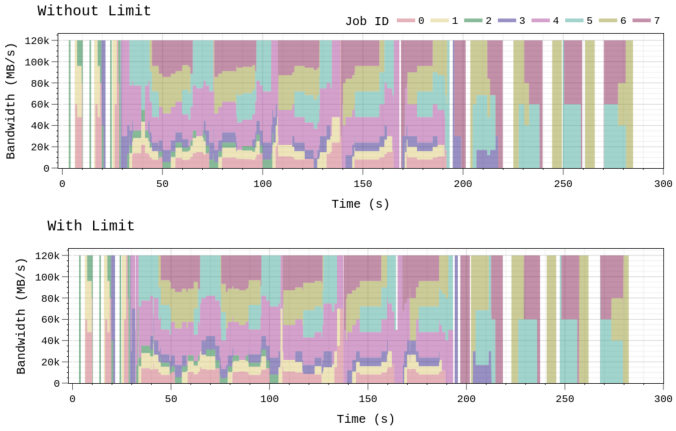

Fig. 2. Bandwidth distribution for the two cases shown in Figure 1.

the application's required bandwidth and demonstrates how it can be limited. We provide a user-level approach for broader support. This metric can be considered by the I/O scheduler to dynamically schedule I/O accesses to reduce the contention. As the example in Figure 1 showed, other decision mechanisms like a job scheduler could also profit from our approach. This allows such mechanisms to exploit the benefits that asynchronous I/O offers to the application and the system.

III. Asynchronous MPI-IO

Asynchronous I/O is gaining significant importance as it can hide some or all of the costs associated with I/O by overlapping communication and computation operations with I/O operations [35]. Several ways exist to realize asynchronous I/O (see Sec. VII). In this paper, we focus on the MPI implementation. While a single synchronous I/O call will not return until the I/O operation is complete (e.g., MPI_File_write), an asynchronous I/O call (e.g., MPI_File_iwrite) initiates an I/O operation but does not wait for it to complete. The MPI standard demands matching pairs for non-blocking operations. That is, a separate request complete call (MPI_Wait, MPI_Test, or any of their variants) is needed to complete the I/O request [36, Ch. 14.2]. Ideally, the asynchronous I/O operation would be executed in the background of the compute phase *unnoticed*. Thus, to utilize the full potential of asynchronous I/O, an asynchronous I/O operation should finish before it reaches the request-complete call, avoiding wasting resources on waiting for the I/O operation to finish. Asynchronous I/O happens entirely in the background, overlapping the computational phase if this is achieved. If the I/O operation takes longer, the advantage of asynchronous I/O is somehow diminished. This is where our developed methodology aims to contribute by providing a metric called *required bandwidth* (B), which quantifies the I/O requirements of the application to avoid the second —undesired— scenario. On the other hand, the metric that quantifies the real I/O behavior (i.e., bytes transferred per second) is referred to as *throughput* (T). Both will be explained in detail in Sec. IV-A.

According to the MPI standard [36, Ch. 14.6], for asynchronous data operations, access can occur between the call to the asynchronous data access routine and the return from the corresponding request complete routine. To gain more control of the asynchronous I/O, we implemented our runtime in a modified MPICH version that spawns a thread for I/O. This was necessary for our bandwidth-limiting approach, as explained later in Sec. V. Moreover, it brings the advantage of computing the throughput for asynchronous I/O operations more accurately. The thread can easily find this value by monitoring the transferred bytes and the elapsed time of the I/O operation. This removes the need for less accurate methods, like frequent calls to MPI_Test, though it can still be used if needed. Thus, for this paper, the throughput results are obtained based on the data and timing of the I/O thread.

IV. Asynchronous I/O requirements

I/O bandwidth is one of the most valuable resources in HPC systems. Traditionally, approaches focus on the I/O behavior of an application, like throughput change over time. In what follows, we tackle the problem differently: We assess the *bandwidth requirements* of HPC applications employing asynchronous MPI-IO.

A. I/O requirements at the rank level

As mentioned in Sec. III, we examine two aspects of asynchronous I/O: (1) the I/O requirements associated with the required bandwidth and (2) the real I/O behavior quantified by the throughput. For asynchronous I/O, we define the I/O requirement as the bandwidth required such that the application finishes an asynchronous I/O operation before reaching the matching blocking operation (e.g., MPI_Wait). This is illustrated in Figure 3. After the first computational phase (phase 0), an MPI_Wait operation is called (which returns immediately), followed by an asynchronous I/O operation, namely MPI_File_iwrite. An I/O thread is spawned that carries out the asynchronous I/O operation in the background of the next computational phase (phase 1). Assuming that the I/O operation is immediately executed once submitted (which is the case for our implementation), the time consumed to finish the I/O operation of phase 0 is $\Delta t'_{0,0}$. However, ideally the I/O operation finishes at least at the end of the second computational phase, i.e., before the matching MPI_Wait function, such that no time is wasted waiting for the I/O operation to finish. For that, the *required* (or available) time window to finish the I/O operation is $\Delta t_{0,0}$. $\Delta t_{0,0}$ can be captured by intercepting the asynchronous I/O operation (MPI_File_iwrite) and the corresponding blocking one (MPI_Wait). As for each rank $i \in [0, n)$, the number of transferred bytes $b_{i,j}$ is captured beside the required I/O time window $\Delta t_{i,j}$, the required bandwidth $B_{i,j}$ for phase $j \in [0, m)$ is simply:

$$B_{i,j} = \frac{b_{i,j}}{\Delta t_{i,j}}, \tag{1}$$

such that $\Delta t_{i,j} = te_{i,j} - ts_{i,j}$ with $te_{i,j}$ and $ts_{i,j}$ indicating the required end and start time of the asynchronous I/O operations, respectively. Thus, $B_{i,j}$ represent the bandwidth required by rank i during the j^{th} I/O phase (i.e., $(j+1)^{th}$ computational phase), such that the asynchronous I/O operation is performed completely in the background. If several requests are submitted in the same phase, for each request, the bandwidth is calculated similarly to Eq. (1). To obtain the rank-level metric $B_{i,j}$ again, either the sum or the average of the individual bandwidths can be computed. For this paper, we sum the bandwidths of the individual requests, as this results in higher values for $B_{i,j}$.

Similarly to $B_{i,j}$ from Eq. (1), by dividing $b_{i,j}$ by the *actual* time window $\Delta t'_{i,j}$ of the I/O operation, the throughput $T_{i,j}$ of rank i for phase j is obtained:

$$T_{i,j} = \frac{b_{i,j}}{\Delta t'_{i,j}} \tag{2}$$

With our modified MPICH version, we accurately determine $\Delta t'_{i,j}$. Note that, as standard MPI implementations are restricted to the MPI level and are unconcerned with how asynchronous I/O is realized, the exact value of $\Delta t'_{i,j}$ is hard to find and depends on the number of tests (e.g., MPI_Test) performed during the computational phase $j+1$. Moreover, while the required I/O time windows $\Delta t_{i,j}$ stay nearly constant (in accordance with the compute phase), the actual I/O time windows $\Delta t'_{i,j}$ can vary significantly as shown in Figure 3, due to several reasons (e.g., network congestion, slow I/O, and I/O congestion). Hence, $T_{i,j}$ is strongly affected by I/O variability. On the contrary, $B_{i,j}$ is tightly coupled to the duration of the compute phase, which can also be subjected to variations.

For the case a rank submits multiple requests during a phase, the I/O phase of the throughput $T_{i,j}$ starts once the first request is pushed into a throughput monitoring queue and ends once the last request is completed and the queue becomes empty. For the required bandwidth $B_{i,j}$, while the start time (i.e., $ts_{i,j}$) is the same (once the first request is pushed into a bandwidth monitoring queue), the I/O phase ends (i.e., $te_{i,j}$) once the first request in the queue reaches the matching wait operation. While TMIO provides options to set $te_{i,j}$ after the last request in the queue reaches the matching wait operation, we opted for the previous case, as it results in higher bandwidth requirements.

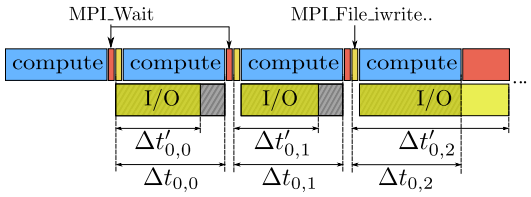

Fig. 3. Rank 0 performing asynchronous I/O during the computational phases.

B. Bandwidth limitation using I/O requirements

With the required bandwidth for each rank at hand, the throughput of each rank can be limited according to $B_{i,j}$. This can be realized through different methods, for example, aggregating $B_{i,j}$ over all involved ranks and calculating an application-level metric. We developed our methodology by

limiting the I/O throughput $T_{i,j}$ of each rank separately to the corresponding required bandwidth $B_{i,j}$ from the last phase. To be more precise, after finding $B_{i,j}$, the throughput $T_{i,j+1}$ of the *next phase* (i.e., $j+1$) is limited to this value. Note that we still refer to our approach as *bandwidth limitation* as it is widely known in the literature, though the throughput is actually limited by the values of the bandwidth.

Since $B_{i,j}$ is calculated from a previous phase and used to limit the next one, several limitations can occur. The most obvious is when the computing phase during the phase $j+1$ differs from the one during the j^{th} phase. By limiting the throughput to a too-low value, the I/O of the rank could force the application to wait until it completes. Looking at this aspect from the opposite direction, a too-high limit could result in no benefits, as the I/O operation can be too slow. However, the latter brings no disadvantages compared to a standard run, while the first could prolong the application's runtime. Hence, a methodology is needed to decide the value for limiting the throughput of the next phase based on what has been calculated so far. For that, we developed three strategies:

1) The **direct strategy** assigns the goal of the next phase to the obtained value from the last phase (i.e., $B_{i,j}$) multiplied by a tolerance factor (tol).
2) The **up-only strategy** only assigns *increasing* values of $B_{i,j} * tol$ as the limit for the next phase.
3) The **adaptive strategy** takes $B_{i,j}$ and the difference between $B_{i,j}$ and $B_{i,j-1}$ multiplied by tolerance values as the limit for the next phase. Inspired by control theory, this strategy mimics a PI controller for a softer transition.

Depending on tol, the *direct* strategy can be very restrictive, resulting in waiting phases. In contrast, while the *up-only* strategy yields less restrictive limits, it decreases the chance of spending time in waiting phases but often results in less exploitation of the compute phases by asynchronous I/O. Thus, the first strategy is the aggressive one with the highest exploitation chances, while the latter is the safer one. The third strategy lies between the previous two and is more balanced. As $B_{i,j}$ is computed at the MPI level, several aspects, like threads competing for resources [33], are ignored. The tolerance value tries to compensate for these aspects.

C. Application level I/O requirements

Our approach works on a per-rank basis. After an MPI_Wait or any of its derivatives is reached, each MPI rank sets the limit according to the used strategy. However, it's more convenient to show the results at the application level for visualization. Furthermore, providing an application-level metric can be easily interpreted by existing I/O scheduling and QoS approaches, as mentioned in Sec. II. Hence, in this section, we derive an application-level metric from the calculated rank-level metrics. Since a typical HPC application is executed with n MPI ranks, the required bandwidths $B_{i,j}$ of each rank $i \in [0,n)$ during the phases $j \in [0,m)$ are collected. We compute the required bandwidth B_r of the overlapping I/O phases in the $r \in [0,p)$ regions to summarize the individually collected data at a higher level of abstraction. That is, the regions include segments at the *application-level* in which the I/O phases of the different ranks overlap, and B_r is the sum of the required bandwidths $B_{i,j}$ inside the region across the ranks. Consequently, the maximum of B_r represents the minimal required bandwidth at the application level such that during the entire execution of the job, no time is spent waiting for a matching blocking operation (e.g., MPI_Wait) to finish.

To find the $r \in [0,p)$ regions in which the I/O phases overlap across the ranks, the start $ts_{i,j}$ and the end $te_{i,j}$ time of the collected bandwidths $B_{i,j}$ are examined. We sort these time values and detect the start time (ts_r) of each region from these sorted values. That is, by iterating over the sorted values, whenever a start time $ts_{i,j}$ or end time $te_{i,j}$ is encountered, a new region is detected, ts_r is assigned, and the bandwidth $B_{i,j}$ is added to or subtracted from the current sum to find B_r. This implies that only the start time of the regions ts_r is needed, as the end time of the regions is implicit (i.e., just before the next region starts or all data was handled). Consequently:

$$B_r = \{ \sum_{\substack{i \in [0,n) \\ j \in [0,m)}} B_{i,j} \mid ts_r \in [ts_{i,j}, te_{i,j}) \} \tag{3}$$

This calculation is done offline in the plotting script (for this paper) or optionally online if the appropriate flags are provided to TMIO. The example in Figure 4 demonstrates this approach. Three ranks performed asynchronous I/O and their required bandwidths $B_{0,0}$, $B_{1,0}$, and $B_{2,0}$ were captured. In this example, all required bandwidths belong to the same phase 0. However, this is not always the case unless collective operations are used. Five regions were identified as shown by the circled number at the top of Figure 4. The points in the lower part of the figure indicate the time (x-axis) when B_r was calculated and the value (y-axis) it attained. Once B_r is found, the value is held until the next region starts. As all data has been processed at $te_{1,0}$, no further region is added.

For simplicity, we refer to B_r as B in what follows. The application level T is similarly calculated from $T_{i,j}$. As the limiting strategies scale the values of each $B_{i,j}$, we calculate B_L in the same way as B, however, with these scaled values. Hence, B_L is the *limit* applied at the application

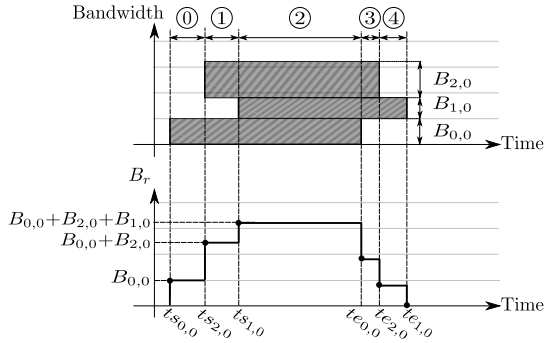

Fig. 4. Finding B_r in the $r \in [0,5)$ regions. The top part of the figure shows the required bandwidths of the first three ranks versus time, while the lower part shows how B_r is calculated.

level (depending on the strategy and tolerance), while B is the *lowest limit* needed. To sum up, the symbols stand for:

- B: Short for B_r, which is the required bandwidth of the overlapping I/O phases from different ranks in the $r \in [0, p)$ regions (see Figure 4).
- B_L: The bandwidth limit at the application level. While its calculation is similar to B_r, it uses scaled values for each $B_{i,j}$, depending on the strategy and tolerance used.
- $B_{i,j}$: The required bandwidths of each rank $i \in [0, n)$ during the phase $j \in [0, m)$ (see Eq. (1)).

The application and rank level throughputs T and $T_{i,j}$ are defined similarly to B and $B_{i,j}$ by replacing the term *required bandwidth* with *throughput*, respectively. Note that the application-level metrics B, B_L, and T are calculated for illustration purposes, as the limit is applied on the rank level.

D. Overhead of the tracing library

The tracing library TMIO has three roles: (1) trace the throughput $T_{i,j}$ and bandwidth $B_{i,j}$, (2) calculate the limits based on the strategies and pass them to the extended MPI version (see Sec. V), and (3) aggregate the collected data and write them out to a file. The first two points contribute to the overhead peri-runtime, while the last point results in overhead post-runtime. In all our experimental runs in Sec. VI, both types contributed to less than 9% of the total runtime. Figure 5 presents the runtime information about HACC-IO from Sec. VI-B. As observed, the overhead contributes only slightly to the total runtime of the application. This becomes even clearer when the runtime distribution in Figure 6 is considered. The figure also distinguishes between the overhead during (peri-) runtime and after (post-) the application run. Note that overhead post-runtime is calculated during MPI_Finalize. As observed in Figure 6, the peri-runtime overhead is negligible and lies below 0.1%. In contrast, the overhead post-runtime increases with the number of ranks due to the increased communication effort. However, this overhead is present since we want to visualize the results. Hence, this overhead can also be discarded if the collected metrics are not saved. Note that, aside from writing the data out, the library can also send the data via TCP (via ZeroMQ [37]) to avoid creating a file.

As shown in Figure 6, the application seems to become more compute-intensive as the percentage of visible I/O (synchronous I/O and asynchronous I/O during the waiting calls)

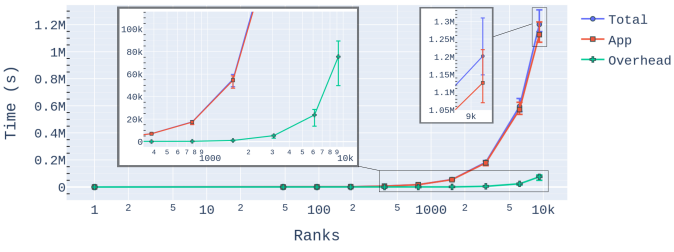

Fig. 5. Runtime variation of HACC-IO up till 9216 MPI ranks.

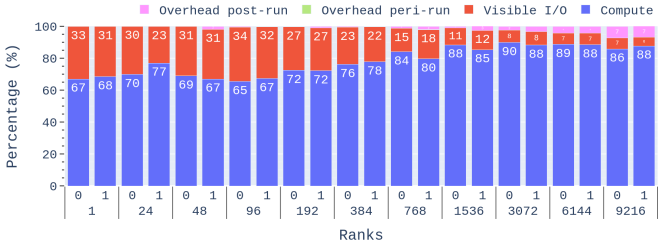

Fig. 6. Total time distribution of HACC-IO with the direct strategy (run 0) and without the bandwidth limitation (run 1) for various rank configurations.

decreases. However, this is only true for the case without limit (run 1), as the majority of I/O in case the throughput is limited (run 0) is done *behind the scenes* as shown later in Sec. VI-B.

E. Implementation

Using the LD_PRELOAD mechanism, we can link the library TMIO easily to an executable. Thus, we can extract the requirements easily without modifying the application code. Specific MPI calls are intercepted using the PMPI interface to extract metrics like the start time and bytes transferred during the MPI-IO operations. Moreover, the request complete calls like MPI_Wait and its derivatives are also intercepted to determine the length of the available time window. As we want to provide the I/O requirements of HPC applications to other bandwidth-limiting approaches, we decided not to implement it as a part of the extended MPICH version used in this paper (see Sec. V). Note that linking the library by including its headers is also possible to control the flushing of the data further. However, we avoid this opinion in this paper.

V. MPI Extension for Bandwidth Limitation

One of the contributions of this paper is to feature I/O bandwidth limitations for individual applications without modifying them. Therefore, we have limited the scope to considering only applications that rely on the MPI framework for their I/O operations. However, the main ideas behind this implementation can be easily applied to other I/O frameworks. We used MPICH, which is a widely used MPI implementation. MPICH relies on the ROMIO implementation for the MPI-IO subset. One of ROMIO's best features is to provide a flexible architecture that allows easy integration of new I/O drivers and file systems. This is achieved through a modular design that separates the I/O driver and file system layers from the rest of the code. Those inner layers are accessed through their interface called ADIO. ADIO (Advanced I/O) is a collection of library functions in ROMIO that provide an optimized I/O interface for parallel applications.

Ultimately, the ROMIO modular architecture comes in handy for our purposes. The separation between the MPI-I/O interface calls and the ADIO inner calls offers a neat opportunity to implement bandwidth limitations for synchronous and asynchronous I/O operations. This was done as follows: First, all MPI-IO calls have been modified to intercept ADIO calls

related to synchronous/asynchronous read/write operations. Second, a server/client scheme redirects all read/write ADIO calls to a new thread that performs all these operations. The thread and all the client/server-related tools (like mutex and conditions) are created using the inner facilities of the MPICH framework. This makes the implementation platform independent. Third, the I/O thread implements all read/write operations synchronously. Despite this, from the application's point of view, an asynchronous I/O operation is carried out as this realization permits overlapping the compute and I/O phases. Fourth, we created a new MPI generalized request object to notify the end of an asynchronous operation. This object is returned when the I/O call ends so the program can perform a wait operation. The I/O thread will notify the end of the I/O operation through this MPI generalized request object.

MPI generalized request is a mechanism in the MPI library that enables the implementation of custom asynchronous operations. It allows users to create non-blocking communication and I/O operations that are not supported by the standard MPI operations. To create a generalized request, the MPI_Grequest_start function is used. This call returns an MPI_Request object that tracks the progress of the non-blocking operation. The function MPI_Grequest_complete notifies the end of the operation through the MPI_Request object. As mentioned, the I/O thread limits the bandwidth of the I/O operations. It performs the following operations:

1) The request is divided into several sub-requests of predefined size. If the request is smaller than that value, then it's just executed.
2) For every sub-request, the thread calculates the time required to perform it. This is done using the value of the required bandwidth limit and the size of the sub-request being read or written. That is: $\Delta t_{i,j} = \frac{B_{i,j}}{b_{i,j}}$.
3) The thread performs each sub-request as a blocking operation. Once done, the thread compares the actual execution time with the required time calculated previously. There are two scenarios:
   - Case A: If the execution time is shorter than the required time, the thread will sleep until completing the required time.
   - Case B: If the execution time is longer than the required time, the thread will accumulate the difference and use it to reduce the sleeping time.

Two steps are required to leverage the application with these features: (1) the application has to use the modified version of the MPICH framework (once compiled and installed). (2) the application executable has to be linked to the intercepting library TMIO using the LD_PRELOAD mechanism. Thus, the application's code is not modified, but the application needs to be recompiled to use the modified MPI framework.

VI. Experimental Results

In this section, we demonstrate our methodology on an HPC cluster in production mode using two use cases: (1) the WaComM++ CFD kernel and (2) the HACC-IO benchmark, which are described below. All experiments were executed on the Lichtenberg cluster, where the typical node has 96 cores, and the access mode is user-exclusive. The shared file system (IBM Spectrum Scale) has a peak performance of 106 GB/s for writes and 120 GB/s for reads.

A. WaComM++

WaComM++ (Water Community Model) is a pollutant transport and diffusion model that operates over the model outputs. It uses a Lagrangian model to simulate marine pollutants' transport and diffusion processes. WaComM++ is a component of the *Environment Application* workflow that produces operational weather and marine forecasts and/or on-demand ad-hoc environmental simulations for scenarios and what-if analysis [38]. It is characterized by a parallelization schema based on hierarchical and heterogeneous computation and has been designed with hierarchical parallelism in mind. Nevertheless, some requirements have been strongly driven by the transport and diffusion Lagrangian model, such as the need for data exchange using standard and well-known formats. For each time interval to simulate (i.e., one hour), the total number of particles is distributed between the available processors in an MPI-distributed memory fashion. Each processor distributes its duty between the available threads leveraging OpenMP. In the original version of WaComM++, rank 0 reads particle information at the start of the application and writes the results in several files at the end of the application. In some cases, a new read operation is executed after every hour of simulation to include new particles in the model. We modified WaComM++ to write the particles asynchronously in every simulation iteration. The last write I/O operations are still synchronous, as there is no opportunity to overlap I/O with the computational phase. We selected $2 * 10^6$ particles and 50 iterations for all following experiments.

Figure 7 shows the time distribution of WaComM++ for an increasing number of MPI ranks (from 24 to 6144 ranks). We executed six runs (as indicated on the x-axis), such that each two runs had the same settings: runs 0 and 1 with the *direct* strategy and $tol = 2$, runs 2 and 3 with the *up-only* strategy and $tol = 1.1$, and runs 4 and 5 without bandwidth limitation. As illustrated, several I/O bursts occur, which pollute the filesystem with *unnecessary* short accesses. As observed, the runs with our bandwidth-limiting approach achieve higher parallel efficiency by performing asynchronous I/O (write) during the computational phases. For all runs, the time spent during the asynchronous I/O matching wait calls was negligible, except for the run with 384 ranks, where this value reached 1.9% during run 3. Moreover, the *async write exploit*, which shows the percentage from the runtime where asynchronous I/O operations were performed in the background of computational or communicational operations, is higher in the experiments with our limiting approaches. Typically, the runs with the *direct* strategy would reach higher exploitation values than those with the *up-only* strategy. However, due to different tolerance values between the strategies, the *direct* strategy (runs 0 and 1) reaches, in most cases, lower exploitation

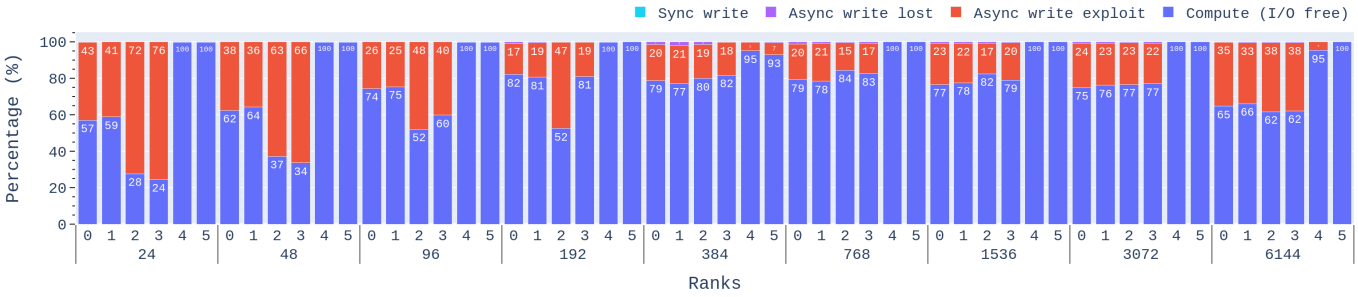

Fig. 7. Application time distribution of WaComM++ with the *direct* strategy (runs 0 and 1) with a $tol = 2$, with the *up-only* strategy (runs 2 and 3) with a $tol = 1.1$, and without bandwidth limitation (runs 4 and 5).

values than the *up-only* strategy (runs 2 and 3). Moreover, one can see that this exploitation is decreasing with an increasing number of ranks, up to 3072 ranks, where the values start to increase again. Note that we aggregated the collected rank-level metrics to calculate the values in Figure 7. For example, for the asynchronous write exploit time, we aggregated the difference between $\Delta t_{i,j}$ and $\Delta t'_{i,j}$ for positive values over all ranks i and phases j.

Figure 8 shows the experiment with 96 ranks and no bandwidth limitation, corresponding to run 1. In contrast, Figure 9 shows the result of our bandwidth limiting approach with the *up-only* strategy (run 5). Even with the most over-estimating method, the throughput is limited to a significantly lower value. Moreover, as observed, the throughput T of the next phase reaches the specified limit B_L from the previous phase. Note that, as the approach is implemented on the rank level, the notion of *phases* implies the asynchronous I/O requests handled at the matching wait operations. The vertical purple lines show when the limit is applied for the first time.

As mentioned in Sec. IV-B, the bandwidth limit is modified according to the strategy used. That is, B_L presents a usually scaled value of B, the aggregated sum of the individual required bandwidths $B_{i,j}$ (see Sec. IV-C). Hence, B can be seen as the lowest limit, so no waiting occurs. Ideally, the throughput T should have a shorter duration than B on the x-axis, such that no waiting occurs. Moreover, the height of the throughput should not significantly exceed the B, as this lowers the exploitation of the compute phases.

The results for large experiments with 9216 MPI ranks (96

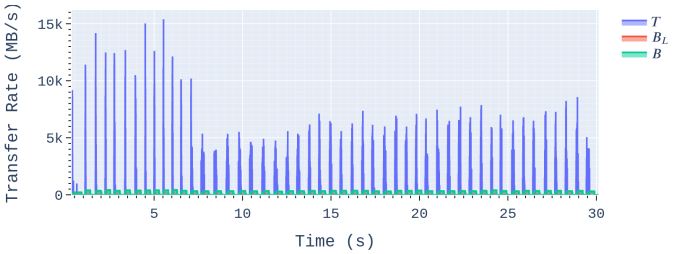

Fig. 8. WaComM++ with 96 ranks without bandwidth limit.

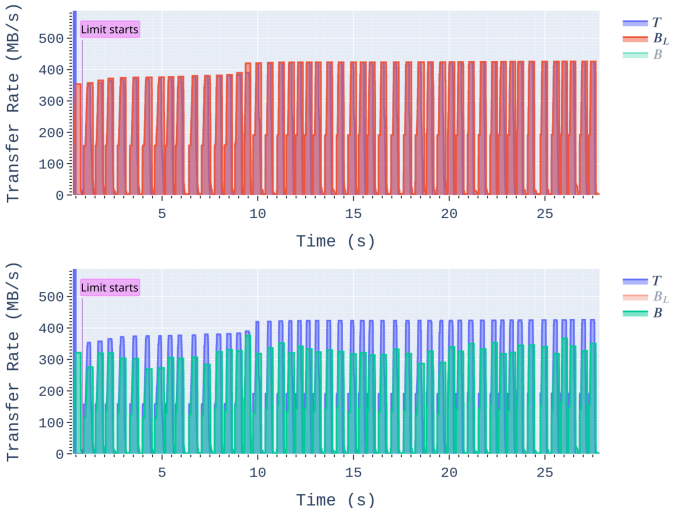

Fig. 9. WaComM++ with 96 ranks and the *up-only* strategy. As the top part shows, T follows the values of B_L from the previous phases. B_L in the top is calculated from B shown in the lower part of the figure. In every phase, T ends before B, indicating no blocking I/O.

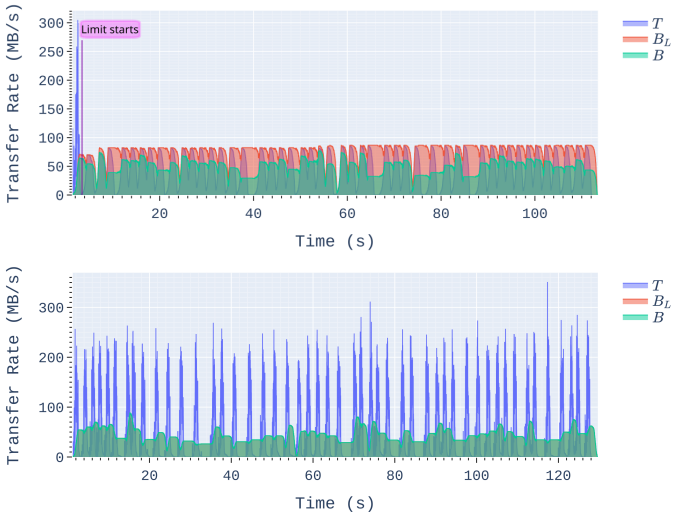

Fig. 10. WaComM++ with 9216 ranks. The result of using the *up-only* strategy (top) and no bandwidth limit (bottom) are shown.

nodes) are shown in Figure 10. The top part of the figure presents the results with the *up-only* strategy, while the bottom shows the results without the bandwidth-limiting approach. For the first case, the exploitation reaches 57% in contrast to the 3.9% obtained with the latter (i.e., without the approach). Moreover, while both cases result in nearly no waiting times (i.e., a disadvantage for the application), the first case results in a small speedup ($\approx 11.6\%$) to 113.4 s from 126.6 s. Note that the bandwidth limit is applied on the rank level based on the values captured from the previous iteration (see Sec. IV-B). Out of the 50 iterations for this example, each rank applies this limit for the first time at the start of the asynchronous I/O operations during the second iteration. The vertical purple line in the Figures 9 and 10 indicates the instance when the limit is applied for the first time by the fastest rank that reaches and executes the I/O operation during the second iteration.

B. HACC-IO

Next, we demonstrate our approach based on HACC-IO [39], which mimics an I/O phase of HACC (Hybrid/Hardware Accelerated Cosmology Code) [40]. From an abstract view, HACC-IO fills arrays of different types with the current index of a for loop, which iterates over the number of particles. Next, a header (containing metadata information such as the number of particles) and the arrays are written to a file. Finally, the file's contents are read again and compared against the values of the variables still in memory. We classified the portions of the application into four blocks in the same order as described above: compute, write, read, and verify. Moreover, we added a for loop around these blocks to execute them several times. The vanilla version of HACC-IO supports different settings for I/O. We used MPI-IO to write using an individual file pointer to distinct files, which is more challenging than collective I/O. Moreover, the header I/O operations are done synchronously.

HACC-IO uses non-collective blocking I/O routines with explicit offset (MPI_File_write_at and MPI_File_read_at), which we replaced with matching non-collective non-blocking I/O routines (MPI_File_iwrite_at and MPI_File_iread_at). Moreover, we adjusted the code such that the read/write occurs asynchronously to the compute/verify blocks as shown in

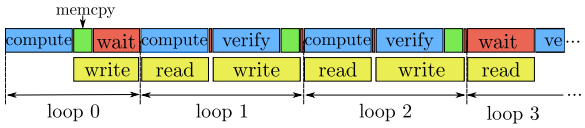

Fig. 12. The modified HACC-IO benchmark.

Figure 12. To avoid data races between the read and write blocks, we used wait blocks (MPI_Wait) at the end of the compute and verify blocks. Moreover, to make the data from one compute block available to the verify block of the same phase, we create a copy of the data using memcpy. This block is located at the end of the verify block, just before the wait block. Finally, we added global broadcast operations during the compute and verify phases for more variability.

Figure 11 shows the time distribution of HACC-IO with 10 loops and 10^6 particles per rank. Runs 0 and 1 are executed with the *direct* strategy, 2 and 3 with the *up-only* strategy, 4 and 5 with the *adaptive* strategy, and 6 and 7 without bandwidth limit. All bandwidth-limiting strategies use $tol = 1.1$. As observed, while the exploitation of the compute phases by the asynchronous write operations increases for all bandwidth-limiting approaches, it decreases when no limiting approach is used. Yet, as Figure 5 shows, the applications' runtime doesn't significantly change between the different runs. Since the required bandwidth for this application increases (from nearly 0.7 GB/s to 58 GB/s) with an increasing number of processes (from 1 to 9216 ranks), and the length of the phases increases at the same time (from 0.6 s to 10 s), a higher number of ranks is more favorable, as it provides I/O scheduling mechanisms with more flexibility regarding regulating the bandwidth for the applications with asynchronous I/O.

In Figure 13, the results of four experiments with 9216 ranks are shown. From top to bottom, the strategies *direct* (run 1), *up-only* (run 3), and *adaptive* (run 4) are used. The bottom figure doesn't use the limiting approach (run 7). As observed, nearly no waiting time at all occurs. While all bandwidth-limiting approaches achieve good exploitation of the compute phase (see Figure 11), the *up-only* strategy achieves lower values as higher limits are set compared to the

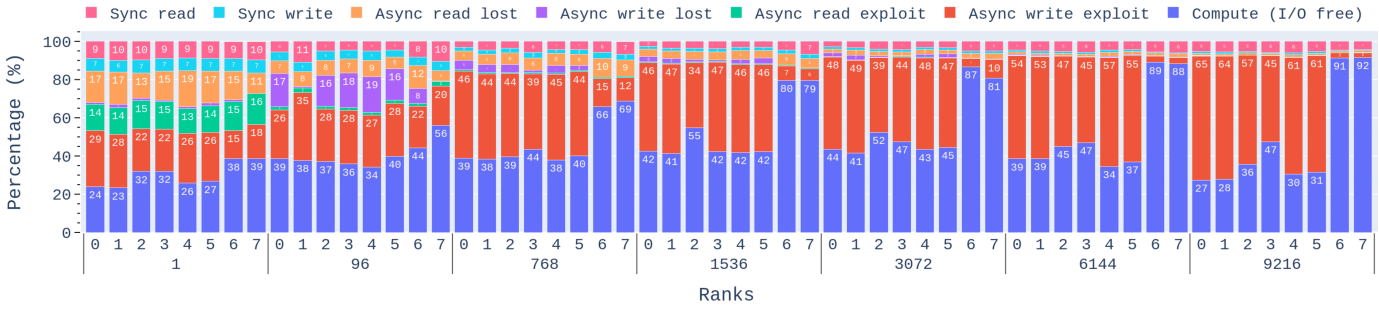

Fig. 11. Application time distribution of HACC-IO with the *direct* strategy (runs 0 and 1), *up-only* strategy (runs 2 and 3), *adaptive* strategy (runs 4 and 5), and without bandwidth limitation (runs 6 and 7). All strategies use the same tolerance value ($tol = 1.1$).

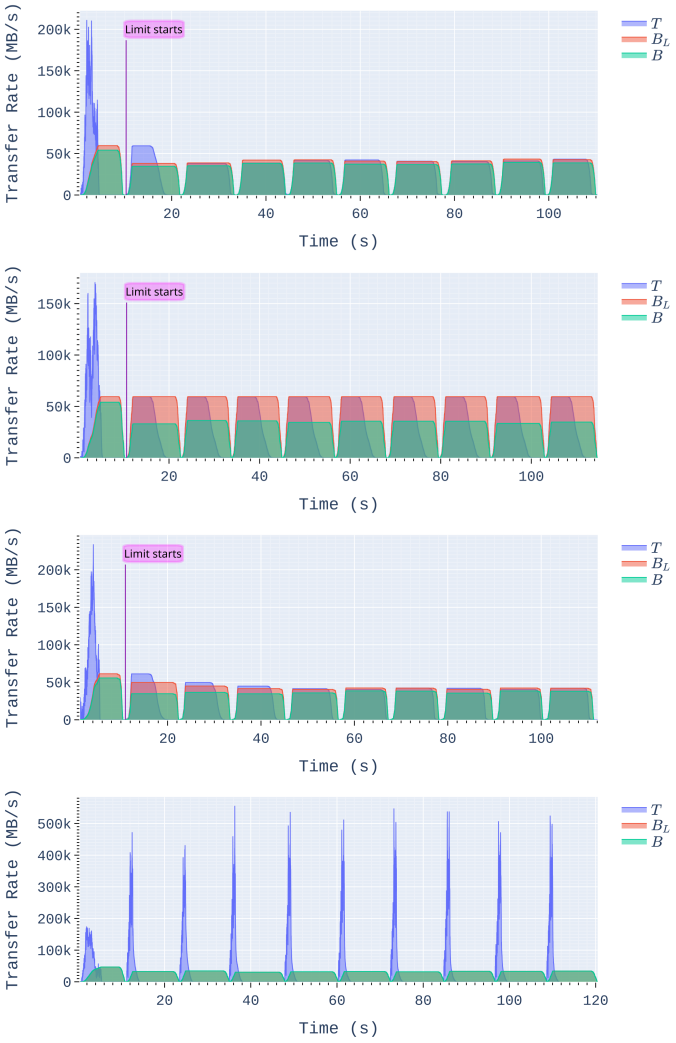

Fig. 13. HACC-IO with 9216 ranks. From top to bottom, the figures show the results for the strategies *direct*, *up-only*, and *adaptive*, respectively. The bottom one is without bandwidth limitation. The vertical purple lines indicate when the limit is applied for the first time by the fastest rank.

other two strategies. Compared to the result without bandwidth limitation, not only have we reduced the I/O burst significantly, but we also exploited the compute phases effectively. Yet, as can be seen in Figure 5 (App time) or in Figure 14 (e.g., with 1536 ranks), the limit applied with, for example, the direct strategy is not reached due to I/O variations like congestion or slow I/O resulting in short waiting times, which prolonged the runtime slightly. This is depicted by the throughput T, which is outside the green region B. To alleviate such cases, a global view and coordination of the I/O system is needed to ensure the application can either attain the required bandwidth or that all bytes in the phase are transferred in time. As our library provides the required bandwidth alongside other metrics (length of the phase per rank or transferred bytes), we plan to develop such solutions in the future. Furthermore, for asynchronous I/O, the application performance can increase

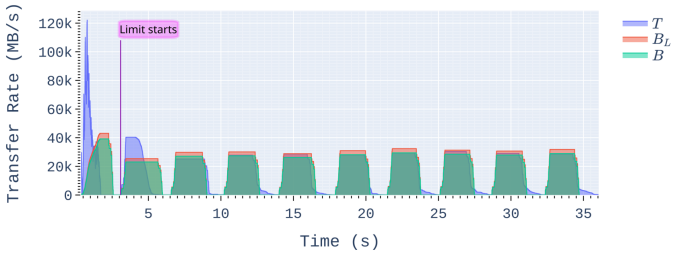

Fig. 14. HACC-IO with 1536 ranks and the *direct* strategy.

at the expense of I/O throughput, as we have observed in this work. This is possible due to less competition for resources at the beginning of the phases, as other works also observed [33]. We plan to examine this in the future. Also, we plan to improve the strategies for estimating the bandwidth requirements. Depending on the strategy (see Sec. IV-B), $B_{i,j}$ is scaled and used to limit $T_{i,j+1}$, which helps in dealing with changing behavior. Nevertheless, this can be further improved by using, for example, a most frequently used table of accesses.

VII. Related Work

There are several ways to perform asynchronous I/O. Two well-known standard implementations on Linux machines are the POSIX asynchronous I/O (POSIX-AIO) [41] and the kernel native asynchronous I/O (kernel-AIO) [42]. POSIX-AIO delegates synchronous I/O operations to a pool of threads [43]. In Linux, POSIX-AIO is provided in the user space by glibc [44]. The kernel-AIO provides an implementation at the kernel level with a set of system calls (io_setup, io_submit, etc. [44]). These system calls are independent of the set of typical synchronous system calls [43] and come with a set of features like the ability to reorder or combine the individual requests of a batched I/O to optimize the disk activity [42]. Moreover, as the kernel-AIO uses asynchronous queue-based system requests [45], it can prevent oversaturating the system (e.g., compared with POSIX-AIO with threads). Still, there are some limitations associated with the O_DIRECT flag and file systems support [43], [46]. A recently added alternative to Linux 5.1 that promises better performance is io_uring [46], which uses ring buffers shared between the user and kernel space. However, reliability and compatibility need more investigation as it almost rebuilds the traditional asynchronous I/O stack and the asynchronous I/O interfaces of applications [47]. MPI-IO implementations, such as ROMIO [48], use POSIX-AIO to realize asynchronous I/O. POSIX-AIO makes background progress typically by spawning a thread. Moreover, ROMIO uses extended generalized MPI requests [49] to query the state of pending asynchronous I/O operations without spawning a new thread. Spawning a thread once an asynchronous I/O routine is called to ensure the operations are executed asynchronously is nothing new [50]. Our approach, however, uses this thread to additional limit the bandwidth. Still, there are several challenges associated

with asynchronous I/O. For example, asynchronous I/O can cause resource contention (CPU, memory bandwidth, network bandwidth, etc.), as background I/O threads can compete with application threads for resources [33]. As our approach is implanted on the MPI level, this low-level information cannot be captured precisely. While this depends on the implementation, we aim to address this with future work.

Recent work in HPC has also focused on improving asynchronous I/O further by deploying user-level threads that support non-data operations [35], [51]. Additionally, high-level libraries, such as ADIOS [52], [53], provide asynchronous I/O support using staging nodes, and file systems like LWFS [54] offer asynchronous I/O support at the file-system level. The VOL connector [35], for instance, provides asynchronous I/O support for HDF5. Other approaches provide routines to convert I/O system calls into asynchronous calls [55]. Compared to these works, our approach does not focus on improving asynchronous I/O, but rather the parallel efficiency of the *application* through utilizing different system resources simultaneously. Moreover, we do not focus on implementing asynchronous I/O, as we rely on ROMIO, which handles this using MPI generalized requests. Our approach requires no code modification; our library is simply preloaded, and the application is compiled and launched with our modified MPICH version. As several tools often rely on MPI-IO, they can implicitly benefit from our library, aligning with our future goals, which we plan to investigate.

A typical pattern with asynchronous I/O is to submit the request as early as possible and check the request status later [35]. several studies focused on overlapping asynchronous I/O with other phases [50], [56], but only at small scale. Other tools [57] use a burst buffer file system and move the data to the PFS asynchronously or propose I/O libraries that take advantage of the modern hierarchical storage systems [58], [59]. Compared to these works, our approach does not rely on additional hardware in the I/O stack.

I/O bursts can lead to severe I/O inefficiency [9]–[12]. Several works have been devoted to reduce I/O contention through QoS approaches [5], [17]–[22], [25], [60], redesigning the I/O stack using caching approaches [29], using burst buffers [32], [61], proposing I/O bandwidth-sharing strategies [62], or introduce QoS from the application side [31]. Compared to these solutions, We do not provide a QoS approach as it is difficult to achieve from a single application perspective. Rather, our approach restricts the maximal bandwidth consumption of the application to what is needed to execute the *asynchronous* I/O unnoticed, sparing the bandwidth of the system to other applications that demand more bandwidth. Additionally, QoS approaches can still be combined with ours, as our limitation only applies to the case where the current bandwidth is higher than what is needed. Moreover, the required bandwidth could also be forwarded to these approaches, which is the intention behind providing a standalone library. Furthermore, while our limitation strategies target a single application, our methodology could be integrated into approaches that strive to balance I/O globally [62], enabling such strategies to exploit the spared bandwidth.

The quest to understand, predict, and optimize I/O performance in HPC systems has been pursued by numerous researchers [8], [63]–[66]. In this context, monitoring software offers means to profile applications to analyze and understand I/O performance and identify bottlenecks and optimization potentials. The HPC domain is rich with I/O monitoring software like TAU [67], Darshan [68], recorder [69], Score-P [70], Scalasca [71], and many more. While several of these tools are not limited to I/O only, tools like Darshan have specialized in I/O (POSIX I/O, MPI-IO, etc.). Compared to other work, our objective is not to provide a tracing or profiling tool. Rather, we capture *asynchronous* I/O data *online* to calculate the *required bandwidth*. This is done by examining individual requests using the PMPI interface of a single application. This paper described how TMIO captures several metrics associated with asynchronous I/O and applies bandwidth-limiting strategies using a modified MPICH version. Yet, the library also supports online and offline synchronous I/O tracing. Targeting synchronous I/O behavior, the tool has been recently used together with FTIO (frequency techniques for I/O) to predict online or detect offline the I/O phases of an application [72].

Several studies focused on modeling and predict I/O performance in HPC [8], [63]–[66], [73], [74]. However, none focused on asynchronous I/O in their models. While a recent approach builds performance models based on past iterations of an application [23], it focuses on high-level libraries (HDF5) and only estimates the cost associated with asynchronous I/O. Our approach not only finds the throughput and required bandwidth, but it can also limit the throughput to the required bandwidth, as demonstrated. Note that the captured data can be aggregated over the ranks to produce application-level metrics (e.g., total bandwidth) online or offline through flags.

VIII. CONCLUSION

This paper presented an approach to identify the I/O requirements of HPC applications that use asynchronous I/O. With the requirements at hand, our approach automatically limits the bandwidth at the user level, reducing I/O bursts and potential I/O congestion in the system while increasing the parallel efficiency of the application. This higher resource utilization is achieved while minimally, if at all, impacting the application runtime. We proposed different strategies that, depending on the risks they take (i.e., the tolerance value), can achieve higher or lower exploitation of the compute phases through asynchronous I/O. Future work will include integrating the setup with a system supporting malleability to even further enhance resource utilization, applying sophisticated strategies for calculating the bandwidth limits, and proposing a similar definition for synchronous I/O in the presence of burst buffers.

As mentioned in the introduction (see Sec. I), TMIO and the modified version of MPICH are publicly available on GitHub. The repository[3] describes how to reproduce the results and experiments from this paper. Furthermore, the data sets from the experiments are publicly available [75].

[3]https://github.com/tuda-parallel/TMIO/tree/main/artifacts/cluster24

REFERENCES

[1] "The TOP500 List," https://www.top500.org/, 2024.

[2] W. Hu, G.-m. Liu, Q. Li, Y.-h. Jiang, and G.-l. Cai, "Storage wall for exascale supercomputing," *Frontiers of Information Technology & Electronic Engineering*, vol. 17, no. 11, pp. 1154–1175, Nov. 2016.

[3] L. Wan, A. Huebl, J. Gu, F. Poeschel, A. Gainaru, R. Wang, J. Chen, X. Liang, D. Ganyushin, T. Munson, I. Foster, J.-L. Vay, N. Podhorszki, K. Wu, and S. Klasky, "Improving I/O performance for exascale applications through online data layout reorganization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 878–890, Apr. 2022.

[4] S. Liu, L. Huang, H. Liu, A. Ruhela, V. Trueheart, S. Lindsey, and Q. Yuan, "Practice guideline for heavy I/O workloads with lustre file systems on TACC supercomputers," in *Practice and Experience in Advanced Research Computing*, ser. PEARC '21. New York, NY, USA: Association for Computing Machinery, 2021.

[5] T. Patel, R. Garg, and D. Tiwari, "GIFT: A coupon based throttle-and-reward mechanism for fair and efficient I/O bandwidth management on parallel storage systems," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, ser. FAST'20. USA: USENIX Association, 2020, pp. 103–120.

[6] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A multiplatform study of I/O behavior on petascale supercomputers," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 33–44.

[7] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Using formal grammars to predict I/O behaviors in HPC: The Omnisc'IO approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2435–2449, 2016.

[8] B. Xie, Z. Tan, P. Carns, J. Chase, K. Harms, J. Lofstead, S. Oral, S. S. Vazhkudai, and F. Wang, "Interpreting write performance of supercomputer I/O systems with regression models," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2021, pp. 557–566.

[9] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. Miller, D. Long, and T. McLarty, "File system workload analysis for large scale scientific computing applications," United States, Jan. 2004.

[10] S. Oral, J. Simmons, J. Hill, D. Leverman, F. Wang, M. Ezell, R. Miller, D. Fuller, R. Gunasekaran, Y. Kim, S. Gupta, D. T. S. S. Vazhkudai, J. H. Rogers, D. Dillow, G. M. Shipman, and A. S. Bland, "Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 217–228.

[11] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 819–829.

[12] J. Yu, W. Yang, F. Wang, D. Dong, J. Feng, and Y. Li, "Spatially bursty I/O on supercomputers: Causes, impacts and solutions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 12, pp. 2908–2922, 2020.

[13] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Automatic identification of application I/O signatures from noisy server-side traces," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, ser. FAST'14. USA: USENIX Association, 2014, p. 213–228.

[14] B. Xie, J. S. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki, "Characterizing output bottlenecks in a supercomputer," in *Conference on High Performance Computing Networking, Storage and Analysis (SC)*. USA: IEEE, Nov. 2012.

[15] A. Miranda, A. Jackson, T. Tocci, I. Panourgias, and R. Nou, "NORNS: Extending slurm to support data-driven workflows through asynchronous data staging," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. USA: IEEE, Sep. 2019, pp. 1–12.

[16] M.-A. Vef, N. Moti, T. Süß, M. Tacke, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "GekkoFS - A temporary burst buffer file system for HPC applications," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 72–91, 2020.

[17] Y. Qian, X. Li, S. Ihara, L. Zeng, J. Kaiser, T. Süß, and A. Brinkmann, "A configurable rule based classful token bucket filter network request scheduler for the lustre file system," in *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[18] J. Carretero, E. Jeannot, G. Pallez, D. E. Singh, and N. Vidal, "Mapping and scheduling HPC applications for optimizing I/O," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. New York, NY, USA: Association for Computing Machinery, 2020.

[19] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC applications under congestion," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 1013–1022.

[20] S. Karki, B. Nguyen, and X. Zhang, "QoS support for scientific workflows using software-defined storage resource enclaves," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 95–104.

[21] Y. Hua, X. Shi, H. Jin, W. Liu, Y. Jiang, Y. Chen, and L. He, "Software-defined QoS for I/O in exascale computing," *CCF Transactions on High Performance Computing*, vol. 1, no. 1, pp. 49–59, May 2019.

[22] Y. Qian, X. Li, S. Ihara, L. Zeng, J. Kaiser, T. Süß, and A. Brinkmann, "A configurable rule based classful token bucket filter network request scheduler for the lustre file system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. New York, NY, USA: Association for Computing Machinery, Nov. 2017.

[23] J. Ravi, S. Byna, Q. Koziol, H. Tang, and M. Becchi, "Evaluating asynchronous parallel I/O on HPC systems," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 211–221.

[24] Z. Qiao, Q. Liu, N. Podhorszki, S. Klasky, and J. Chen, "Taming I/O variation on QoS-Less HPC storage: What can applications do?" in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–13.

[25] R. Macedo, M. Miranda, Y. Tanimura, J. Haga, A. Ruhela, S. L. Harrell, R. T. Evans, J. Pereira, and J. Paulo, "Taming metadata-intensive HPC jobs through dynamic, application-agnostic QoS control," in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. Bangalore, India: IEEE, May 2023, pp. 47–61.

[26] C. Daley, D. Ghoshal, G. Lockwood, S. Dosanjh, L. Ramakrishnan, and N. Wright, "Performance characterization of scientific workflows for the optimal use of burst buffers," *Future Generation Computer Systems*, vol. 110, pp. 468–480, 2020.

[27] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. Phoenix, AZ, USA: IEEE, May 2014, pp. 155–164.

[28] T. Patel, S. Byna, G. K. Lockwood, and D. Tiwari, "Revisiting I/O behavior in large-scale storage systems: The expected and the unexpected," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[29] Y. Qian, X. Li, S. Ihara, A. Dilger, C. Thomaz, S. Wang, W. Cheng, C. Li, L. Zeng, F. Wang, D. Feng, T. Süß, and A. Brinkmann, "LPCC: Hierarchical persistent client caching for lustre," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[30] J. Lüttgau, S. Snyder, P. Carns, J. M. Wozniak, J. Kunkel, and T. Ludwig, "Toward understanding I/O behavior in HPC workflows," in *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, 2018, pp. 64–75.

[31] L. Huang and S. Liu, "OOOPS: An innovative tool for IO workload management on supercomputers," in *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, 2020, pp. 486–493.

[32] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *2012 IEEE 28th Symposium on Mass Storage Systems and*

Technologies. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2012, pp. 1–11.

[33] S.-M. Tseng, B. Nicolae, F. Cappello, and A. Chandramowlishwaran, "Demystifying asynchronous I/O Interference in HPC applications," *The International Journal of High Performance Computing Applications*, vol. 35, no. 4, pp. 391–412, Jul. 2021.

[34] T. Özden, T. Beringer, A. Mazaheri, H. M. Fard, and F. Wolf, "Elastisim: A batch-system simulator for malleable workloads," in *Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3545008.3545046

[35] H. Tang, Q. Koziol, J. Ravi, and S. Byna, "Transparent asynchronous parallel I/O using background threads," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 891–902, Apr. 2022.

[36] MPI Forum, "MPI: A message-passing interface standard," Jun. 2021.

[37] ZeroMQ developers, "ZeroMQ: An open-source universal messaging library." [Online]. Available: https://zeromq.org/

[38] R. Montella, D. Di Luccio, P. Troiano, A. Riccio, A. Brizius, and I. Foster, "WaComM: A parallel water quality community model for pollutant transport and dispersion operational predictions," in *2016 12th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS)*. IEEE, 2016, pp. 717–724.

[39] LLNL, "CORAL benchmark codes - *HACC IO*," 2020. [Online]. Available: https://asc.llnl.gov/coral-benchmarks#hacc

[40] S. Habib, V. Morozov, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, T. Peterka, J. Insley, D. Daniel, P. Fasel, N. Frontiere, and Z. Lukic, "The Universe at extreme scale: Multi-petaflop sky simulation on the BG/Q," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. Salt Lake City, UT: IEEE, Nov. 2012, pp. 1–11.

[41] S. R. Walli, "The POSIX family of standards," *StandardView*, vol. 3, no. 1, pp. 11–17, Mar. 1995.

[42] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan, "Asynchronous I/O support in linux 2.5," in *Proceedings of the Linux Symposium*, 2003, pp. 371–386.

[43] A. Roca Nonell, V. Beltran Querol, and S. Mateo Bellido, "Introducing the task-aware storage I/O (TASIO) library," in *OpenMP: Conquering the Full Hardware Spectrum*, X. Fan, B. R. de Supinski, O. Sinnen, and N. Giacaman, Eds. Cham: Springer International Publishing, 2019, vol. 11718, pp. 274–288.

[44] "Aio(7) - Linux manual page," https://man7.org/linux/man-pages/man7/aio.7.html.

[45] F. Schmaus, F. Fischer, T. Hönig, and Schröder-Preikschat, Wolfgang, "Modern concurrency platforms require modern system-call techniques," Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Tech. Rep., 2021.

[46] J. Axboe, "Efficient IO with io_uring," Retrieved April 06, 2022 from https://kernel.dk/io_uring.pdf, 2019.

[47] D. Li, N. Zhang, M. Dong, H. Chen, K. Ota, and Y. Tang, "PM-AIO: An effective asynchronous I/O system for persistent memory," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2021.

[48] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems - IOPADS '99*. Atlanta, Georgia, United States: ACM Press, 1999, pp. 23–32.

[49] R. Latham, W. Gropp, R. Ross, and R. Thakur, "Extending the MPI-2 generalized request interface," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, F. Cappello, T. Herault, and J. Dongarra, Eds. Berlin, Heidelberg: Springer, 2007, pp. 223–232.

[50] C. M. Patrick, S. Son, and M. Kandemir, "Comparative evaluation of overlap strategies with study of I/O overlap in MPI-IO," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 6, pp. 43–49, Oct. 2008.

[51] H. Tang, Q. Koziol, S. Byna, J. Mainzer, and T. Li, "Enabling transparent asynchronous I/O using background threads," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, Nov. 2019, pp. 11–19.

[52] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu, "Hello ADIOS: The challenges and lessons of developing leadership class I/O frameworks: HELLO ADIOS," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, May 2014.

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

[54] R. A. Oldfield, L. Ward, R. Riesen, A. B. Maccabe, P. Widener, and T. Kordenbrock, "Lightweight I/O for scientific applications," in *2006 IEEE International Conference on Cluster Computing*, Sep. 2006, pp. 1–11.

[55] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel, "Lazy asynchronous I/O for event-driven servers," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '04. USA: USENIX Association, 2004, p. 21.

[56] S. Zhou, A. Oloso, M. Damon, and T. Clune, "Application controlled parallel asynchronous IO," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 178–es.

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

[58] T. Alturkestani, T. Tonellot, H. Ltaief, R. Abdelkhalak, V. Etienne, and D. Keyes, "MLBS: Transparent data caching in hierarchical storage for out-of-core HPC applications," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 312–322.

[59] T. Alturkestani, H. Ltaief, and D. Keyes, "Maximizing I/O bandwidth for reverse time migration on heterogeneous large-scale systems," in *Euro-Par 2020: Parallel Processing*, M. Malawski and K. Rzadca, Eds. Cham: Springer International Publishing, 2020, pp. 263–278.

[60] X. Zhang, K. Davis, and S. Jiang, "QoS support for end users of I/O-intensive applications using shared storage systems," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[61] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "GekkoFS - A temporary distributed file system for HPC applications," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 319–324.

[62] A. Benoit, T. Herault, L. Perotin, Y. Robert, and F. Vivien, "Revisiting I/O bandwidth-sharing strategies for HPC applications," INRIA, Tech. Rep. RR-9502 v3, Mar. 2023. [Online]. Available: https://inria.hal.science/hal-04038011

[63] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Omnisc'IO: A grammar-based approach to spatial and temporal I/O patterns prediction," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2014, pp. 623–634.

[64] R. McKenna, S. Herbein, A. Moody, T. Gamblin, and M. Taufer, "Machine learning predictions of runtime and IO traffic on high-end clusters," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2016, pp. 255–258.

[65] B. Xie, Z. Tan, P. Carns, J. Chase, K. Harms, J. Lofstead, S. Oral, S. S. Vazhkudai, and F. Wang, "Applying machine learning to understand write performance of large-scale parallel filesystems," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, Nov. 2019, pp. 30–39.

[66] P. J. Pavan, J. L. Bez, M. S. Serpa, F. Z. Boito, and P. O. A. Navaux, "An unsupervised learning approach for I/O behavior characterization," in *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct. 2019, pp. 33–40.

[67] S. S. Shende and A. D. Malony, "The tau parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, May 2006.

[68] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular HPC I/O characterization with darshan," in *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, Nov. 2016, pp. 9–17.

[69] C. Wang, J. Sun, M. Snir, K. Mohror, and E. Gonsiorowski, "Recorder 2.0: Efficient parallel I/O tracing and analysis," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2020, pp. 1–8.

[70] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Berlin, Heidelberg: Springer, 2012, pp. 79–91.

[71] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency and Computation: Practice & Experience*, vol. 22, no. 6, pp. 702–719, Apr. 2010.

[72] A. Tarraf, A. Bandet, F. Boito, G. Pallez, and F. Wolf, "Capturing periodic I/O using frequency techniques," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, CA, USA, May 2024, pp. 465–478.

[73] M. R. Meswani, M. A. Laurenzano, L. Carrington, and A. Snavely, "Modeling and predicting disk I/O time of HPC applications," in *2010 DoD High Performance Computing Modernization Program Users Group Conference*, 2010, pp. 478–486.

[74] B. Behzad, S. Byna, Prabhat, and M. Snir, "Optimizing I/O performance of HPC applications with autotuning," *ACM Transactions on Parallel Computing*, vol. 5, no. 4, Mar. 2019.

[75] A. Tarraf, J. F. Muñoz, D. E. Singh, T. Özden, J. Carretero, and F. Wolf, "I/O behind the scenes [data set]," Zenodo, Jul. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.12700677