Orthrus: A Framework for Implementing Efficient Collective I/O in Multi-core Clusters

Xuechen Zhang[1] Jianqiang Ou[2] Kei Davis[3] Song Jiang[2]

[1] Georgia Institute of Technology,
[2] Wayne State University,
[3] Los Alamos National Laboratory

Abstract. Optimization of access patterns using collective I/O imposes the overhead of exchanging data between processes. In a multi-core-based cluster the costs of inter-node and intra-node data communication are vastly different, and heterogeneity in the efficiency of data exchange poses both a challenge and an opportunity for implementing efficient collective I/O. The opportunity is to effectively exploit fast intra-node communication. We propose to improve communication locality for greater data exchange efficiency. However, such an effort is at odds with improving access locality for I/O efficiency, which can also be critical to collective-I/O performance. To address this issue we propose a framework, *Orthrus*, that can accommodate multiple collective-I/O implementations, each optimized for some performance aspects, and dynamically select the best performing one accordingly to current workload and system patterns. We have implemented Orthrus in the ROMIO library. Our experimental results with representative MPI-IO benchmarks on both a small dedicated cluster and a large production HPC system show that Orthrus can significantly improve collective I/O performance under various workloads and system scenarios.

1 Introduction

Petascale HPC systems are current, and issues critical to the delivery of exascale systems in the next decade are being actively identified and studied [1]. A fundamental approach to reaching very high compute capacity is to employ increasingly large numbers of compute nodes, each with increasingly more CPU cores. A major challenge on such a technical path is application scalability. For parallel programs or system facilities that need global communication and coordination, the benefit of increased system scale cannot be fully realized if locality in the operations is not carefully exploited. In this paper we take collective I/O, a widely-used and performance-critical facility in the MPI I/O library, as a representative case for studying how to introduce communication locality into its implementation, and to demonstrate a strategy to effectively exploit locality.

1.1 Potential Challenges in the Performance of Collective I/O

Collective I/O is a technique commonly employed in MPI programs to coordinate and reorganize requests from the multiple processes of a program before sending them to the data nodes. The idea is simple: if each process needs to access one or a number of segments of data in a file domain, and collectively these processes access all or a major part of the domain, it is more efficient to have each of (a subset of) the processes access a large and mostly contiguous segment of data in the domain. In this way there can be fewer requests to the data nodes for higher data access efficiency.

There are two factors that may compromise performance. One is the potentially high cost of the data exchange that is central to collective I/O. In collective I/O a process, the aggregator, is responsible for access to non-overlapping sections of a file, which is called the aggregator's file realm. An aggregator's file realm may include data requested by other processes, so a data exchange phase is required in addition to the I/O phase. For reads the data exchange happens after the aggregators have retrieved the data into their respective buffers. Data exchange for writes, wherein the aggregator collects data for writing, precedes the I/O phase. The overhead of data exchange can be significant if a large amount of data needs to be exchanged, especially when these aggregators and the processes communicating with them are on different compute nodes. If there are multiple cores on each node—the common case in today's HPC systems—the ideal scenario in terms of minimizing the cost of data exchange is to have each aggregator only be responsible for accessing data for processes on the same node and so have all data exchange occur within individual nodes. However, this assignment of file realms to aggregators may compromise the efficiency of I/O operations on the data nodes—the second factor that affects collective-I/O performance.

When data is stored on hard disks—currently the dominant storage devices in HPC systems—disk efficiency determines I/O efficiency, especially for requests that are not very large. For a disk to efficiently serve a set of temporally proximal requests, the order in which they reach the disk, which largely determines the order in which they are served, is a major factor in the disk's efficiency. Because a hard disk relies on disk-head seek and disk-platter rotation to reach requested data, arrival of requests in ascending disk order minimizes its mechanical operations and maximizes its throughput. However, if the requests to a disk are issued from different aggregators, there is no way to ensure ascending order unless the involved aggregators synchronize their issuance, which can be excessively expensive, especially in a large-scale system. The ideal scenario in terms of maximizing disk efficiency is to have all requests to a disk (or data node) be issued by one aggregator that can send them in the ascending order according to requested data's offsets in the file [35].

1.2 Data Exchange Efficiency vs. Disk Service Efficiency

As described, a strategy for assigning file realms to aggregators to achieve high efficiency for data exchange is at odds with one for achieving high disk service

efficiency. For example, data on a disk can be requested by processes on different compute nodes, and if an aggregator's file realm is only the data requested by processes on the same node to avoid inter-node data exchange, the hard disk would receive requests from different aggregators and so risk loss of efficiency.

It is not clear how to define a file realm assignment that is optimized for both disk access and data exchange, and an implementation of one strategy or the other is not expected to perform well for collective I/O with diverse patterns. However, we can *dynamically* determine how biasing the trade-off will affect collective-I/O service time and accordingly apply the appropriate strategy to improve efficiency.

We propose a general framework that allows multiple collective-I/O implementations, each optimized for one or more performance aspects, e.g. I/O pattern or relative hardware performance characteristics, to co-exist in a library. Based on a prediction of which would perform best for the current I/O pattern and system load, and applying it, the framework essentially provides MPI programmers a collective-I/O library that adapts to access pattern changes and other dynamic system characteristics. This framework, named *Orthrus*, extensibly accommodates multiple collective-I/O implementations.

1.3 The Challenge and our Contributions

A major challenge for the framework to achieve effectiveness is in the dynamic prediction of the performance of the various candidate implementations in a given scenario. Both off-line modeling and on-line simulation are unlikely to provide predictions accurate enough to distinguish the candidates. One reason is that both modeling and simulation need not only the workload information (such as number of aggregators, request count, volume and distribution of requested data across data nodes) but also the system information, especially that about data nodes, such as number of data nodes, type of storage devices, and actual data layout on the devices. Furthermore, the information may change from one run to another. In general there will be system dynamics that are impossible to predict such as communication traffic generated by other programs on the compute-node side, or streams of I/O requests from unrelated programs sharing the same data nodes. Some information, such as data layout, is simply not available to user-level programs, including the MPI libraries. Much of the information is hard to accurately capture in an on-line manner, let alone for the modeling or simulation facilities to use for accurate and on-the-fly prediction.

We propose a simple, efficient, accurate, and portable method for selecting the best performer for a collective-I/O operation. To this end it does not take any workload or system information as input and does not involve any complex modeling or simulation for performance prediction. The key technique is to use performance examination, rather than performance modeling, in the prediction. Each candidate collective-I/O implementation is given opportunities to demonstrate its performance, which is recorded for comparison and selection. In this way all of the hard-to-capture information is distilled in the actual performance of a candidate's examination run, which provides an accurate prediction of the

performance that would be exhibited should the candidate be selected for executing the collective I/O in the near future. In summary we make the following contributions.

- We propose a framework for accommodating multiple collective I/O implementations, each optimized for some class of I/O patterns, that dynamically selects from them for best performance, and is thus adaptive to changing workloads and system dynamics.
- We develop an efficient mechanism to evaluate the performance of collective-I/O implementations so that the best performer can be identified and deployed on the fly. With this mechanism parallel I/O developers need not commit to compromising trade-offs in a collective-I/O implementation. Instead, they can implement schemes optimized for specific workloads or system characteristics and simply plug them into the framework, where their performance advantages would be realized whenever their targeted characteristics appear.
- We experimentally evaluate our implementation of the *Orthrus* framework in the ROMIO library, and provide a detailed analysis of the results. Our experimental results with representative MPI-IO benchmarks on both a small dedicated cluster and a large production HPC system show that *Orthrus* can significantly improve the I/O throughput of storage systems.

2 The Design of Orthrus

The objective of *Orthrus* is to provide a framework for implementing a high-performance collective I/O library that can adaptively achieve high I/O efficiency with various workload patterns and system setups. It represents a deviation from the traditional practice of attempting to build a single, highly-versatile implementation to efficiently handle different kinds of workloads. Realizing that a monolithic design tends to lack flexibility through extensibility, we subdivide the effort. A key element is the design of a framework ready for multiple implementations to plug into. The other elements are the provision of collective-I/O implementations. One advantage of this approach is making the second effort open to other practitioners in the HPC community. For workloads that are of particular importance to them but not handled well by existing implementations, they can develop new implementations specific to their workloads (and machine architectures) and plug them into the framework without the concerns associated with the traditional approach, such as compromising the existing implementation or complicating the design. In this section, we describe our efforts on these two fronts.

2.1 The Orthrus Framework

The crux of the *Orthrus* strategy is knowing how each candidate implementation would likely perform if it were used to execute a collective-I/O function call.

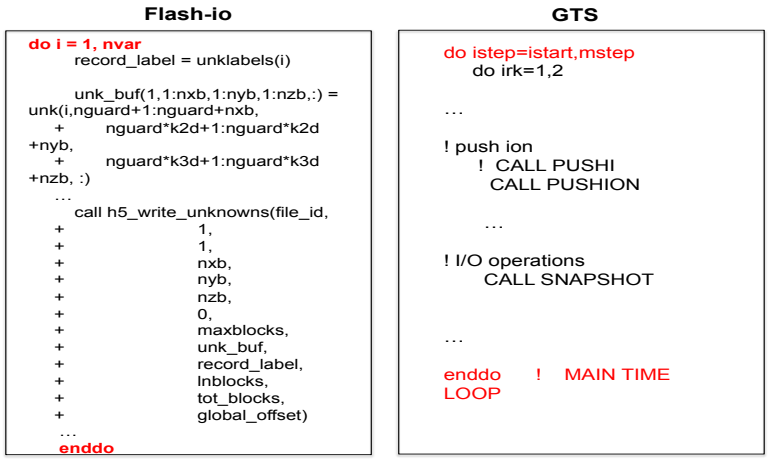

Fig. 1. ***Flash-io*** periodically writes out checkpointing data using the function ***h5_write_unknowns()***. In the ***GTS*** code, both checkpoint and visualization data are written back to the disks by the ***SNAPSHOT*** function. Thousands of iterations are typically taken in their execution in production HPC runs. Different I/O transports can be specified, e.g. MPI collective I/O, in the code when the ADIOS [25] library is used.

As mentioned, *Orthrus* examines the performance of each candidate. Certainly in the library one application-level call cannot be executed more than once: the overhead would be excessive and the caching effect would invalidate the performance results of all but the first call. However, with different candidates executing different calls we would need to ensure that their performance results were comparable by invoking these candidates within the same workload, i.e., have the data requested by the calls have the same pattern. In addition, the performance examination period must constitute only a small fraction of the total collective-I/O time because many candidates might not provide efficient I/O service.

Our solution is based on the assumption that a collective-I/O function call is usually in a loop of many iterations. For our purpose this is a reasonable assumption, otherwise the call would be executed only one or a few times, either generating requests for a small aggregate amount of data, whose I/O performance would likely be insignificant for the entire program's performance, or generating a few large requests, each for a large amount of data, whose I/O performance is generally not at issue. Through experimental observations, we also find that many scientific applications, such as *Flash-io* [15] and *GTS* [28], use loop statements to carry out I/O operations for checkpointing or dumping visualization data during their entire execution periods. Their I/O patterns are outlined in Figure 1.

We also assume a consistent access pattern across the loop iterations in the execution of a collective I/O function call. The iteration's data access can be

characterized by a number of factors, including name of accessed file, number of processes issuing I/O requests, number of requests, request sizes, and their offsets in the file. While offsets of requests issued in different iterations cannot be compared directly, for characterization we define a so-called relative offset and use this as a signature to identify similar spatial access patterns quickly. Suppose that in the execution of iteration k of a loop, a collective I/O call produces n I/O requests that sorted according to the offsets of their requested data are $[R_1, R_2, \ldots, R_n]$. If the size and file offset of request R_i are $Size_i$ and $Offset_i$, respectively, the relative offset of these requests is $rel\_offset = \frac{\sum_{i=2}^{n}[Offset_i - (Offset_{i-1} + Size_{i-1})]}{n-1}$. We refer to these values collectively as the call's *signature*. Two collective I/O calls are deemed to have the same pattern and comparable performance results only when their signatures are the same. I/O signature-based pattern identification has been effectively used in previous works, for example in data prefetching [4].

The assumption of consistency provides us with two capabilities. One is that we can use the observed performance of one execution of a collective I/O call as an estimate of its performance in subsequent executions. Second, it allows us to identify calls from the same program statement—we do not assume the availability of the program's source code or the ability to instrument it to explicitly notify the ROMIO library in which *Orthrus* is implemented.

Under these assumptions *Orthrus* carries out its operations as follows. When a collective-I/O call is made, first its signature is compared to the signature of the previous call. If they are not the same, a new candidate examination period is started. If they are the same, then either the system is in a candidate examination period, in which case *Orthrus* keeps testing a candidate implementation, or the system uses the currently selected candidate implementation to execute the call. When a new candidate examination period is started, calls are serviced by the candidates in rotation, each for a fixed number of times (three by default), as long as the signature does not change. The throughput of each candidate is computed as an average to minimize the effects of transient changes in the execution environment. When the examination period ends without a change in signature, the candidate with the highest throughput is selected to execute the calls until the signature changes.

During a regular execution period, even if the collective-I/O calls do not change their access pattern, the dynamic system environment, such as communication and I/O request traffic initiated by other programs, could change, and accordingly the best performing candidate might change. To detect such system variations we monitor throughput of the selected candidate. If its deviation exceeds a certain threshold (15% by default) of the average recorded in the examination period the system returns to the examination mode.

2.2 Candidate Collective-I/O Implementations

To test our idea we implemented two simple, contrasting strategies. One, *core-first*, constrains data exchange to be intra-node. In the current implementation of *core-first* there is one aggregator per compute node to represent all the processes

running on that node. We choose as the aggregator the process in that node requesting the largest amount of data. The processes' I/O requests are collected by the aggregator, which then sorts them in the ascending order according to the offsets of the requested data. When the number of cores in each node is relatively small (e.g. less than 100) using one aggregator is sufficient to handle the data management. With more cores multiple aggregators might be needed and a more sophisticated algorithm could be developed to distribute the load.

The second strategy, *disk-first*, is designed to maximize disk efficiency. It sets up the same number of aggregators as the number of data nodes, each collecting and sending requests to one data node, sorting the requests as *core-first* does. However, *disk-first* ensures that each data node receives requests in well-ordered sequences, while in *core-first* each data node receives requests from multiple (possibly all) aggregators, and the order of requests from different aggregators is essentially random. While maintaining an equal number of aggregators as data nodes allows the data nodes to receive fully sorted sequences of requests, it may limit the I/O bandwidth if the number of data nodes is small. One solution is having multiple aggregators coordinated to access a data node [35].

Though the existing collective-I/O implementation in the ROMIO library does not attempt to reduce communication or maximize disk efficiency, it does reduce the number of requests. One method used is to designate a contiguous file domain to each aggregator. As an I/O request to operating system must be for a contiguous segment of data, this helps reduce the number of requests. However, in ROMIO such a contiguous file domain may contain holes, or gaps in the domain that are not requested. Such holes break the contiguity, and also breaks a potentially large request into multiple smaller ones. To remedy this ROMIO uses data sieving [22] to remove the holes to yield one large request. The benefits of having a smaller number of large requests include reduced request processing overhead and increased disk efficiency. As the *core-first* and the *disk-first* schemes do not use *list I/O* [10] to pack requests, the number of requests to the kernels of the data nodes is not reduced. When the requests are very small (even if they are well ordered), ROMIO's increased disk efficiency through data sieving can be still substantial in comparison. Therefore, as a third strategy we plug the ROMIO implementation into the *Orthrus* framework. For fair comparison we set up only one aggregator per compute node in this ROMIO instantiation.

We emphasize that we are *not* attempting to provide a comprehensive or near-optimal set of strategies in this paper. Rather, the purpose of our choices is to provide a simple set of contrasting strategies to demonstrate the effective adaptivity of the *Orthrus* framework. We include the ROMIO implementation as a candidate strategy as a touchstone of realism—if the performance of these three strategies can exceed that of ROMIO alone, our case is much stronger than it might be if only synthetic strategies were used.

3 Performance Evaluation

The *Orthrus* framework was implemented as an extension of ROMIO in mpich2-1.4.1, a widely used MPI-IO implementation. It currently hosts three collective-

I/O implementations: *core-first*, *disk-first*, and ROMIO's implementation (*ROMIO* hereafter). In this section we evaluate the framework with these three self-contained and independent collective-I/O implementations. We specifically answer three questions: (1) When does *Orthrus* outperform the existing ROMIO implementation? (2) How effective is *Orthrus* in identifying the best performing candidate implementation? (3) Does *Orthrus* work as expected when the execution environment changes dynamically?

3.1 Experimental Setup

To evaluate the performance of *Orthrus* we used a dedicated cluster allowing full control of the running environment. This allowed collection of low-level performance statistics and controlled injection of interference. The cluster consisted of 11 compute nodes and six data nodes. Each compute node was equipped with an 8-core L5410 2.33 GHz Intel Xeon CPU and 64 GB DRAM. Each data node had a 2.13GHz Intel Core 2 CPU, 2 GB DRAM, and one 500 GB SATA hard disk (WDC WD5000AAKS). We used PVFS2 version pvfs-2.8.2 for parallel storage management with its default 64KB striping configuration [19]. All of the nodes were connected through a 1 Gbps Ethernet network. We used MPICH2-1.4.1 [27] compiled with ROMIO to generate MPI executables. The processes were evenly distributed across the nodes and their respective cores. Each node ran a CentOS Linux distribution with kernel-3.3.6. The CFQ I/O scheduler [5] was used for the disks as is standard practice.

We used four benchmarks in the evaluation: *Matrix*, *Noncontig*, *Hpio*, and *Flash-io*. The first one simulates a common pattern of matrix access in scientific applications. The next two have access patterns that may pressure both the disks and the communication, having non-contiguous I/O patterns and the need to access a substantial amount of data that may be involved in the data exchange (usually more than 10 MB in one collective operation). *Flash-io* periodically writes checkpoint and visualization data using HDF5 [17] with parallel I/O. Following we describe experiments with each benchmark.

3.2 Matrix

In the *Matrix* benchmark data are viewed as elements of a two-dimensional matrix with columns evenly distributed among processes. The matrix is serialized by row-major order. Each process accesses a set of contiguous columns, and the data in one row of these columns constitutes a block as illustrated in Figure 2. In each collective-I/O operation all processes access the same number of contiguous blocks, the *access depth*, in their respective columns. If ROMIO is used, the total amount of data accessed in a single collective-IO operation is the product of the block size, access depth, and number of processes. Because any two blocks accessed in an operation by a process are not adjacent, they are in two separate requests if *core-first* or *disk-first* is applied.

We first run the benchmark with 64 processes and 4-block access depth and vary the block size between 4 KB to 4096 KB. Figures 3(a) and 3(b) show the I/O

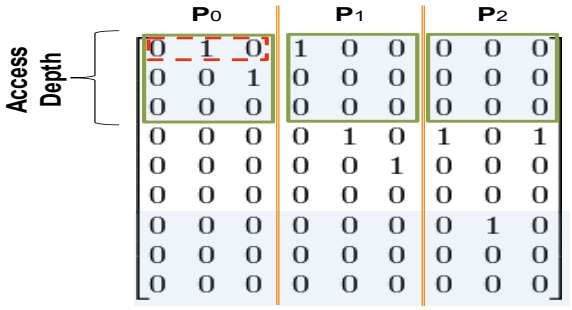

Fig. 2. The access pattern of the *Matrix* benchmark. Data in a dotted rectangle comprises a block, and the set of blocks in a solid rectangle are accessed by a process (P_0, P_1, or P_2) in each collective operation when the access depth is 3. The three contiguous rows in a shaded area are the total amount of data accessed by one collective I/O operation.

throughputs of the benchmark for read and write access, respectively. *Orthrus* increases the throughputs on average by 21% and 36% for reads and writes respectively, compared to the stock ROMIO library. This increase is greater up to a block size of 64 KB because the file striping unit size is 64 KB and blocks (requests) no greater than 64 KB give *disk-first* more opportunity to improve disk efficiency. Once blocks are larger than 64 KB the request size usually stays at 64 KB and improvement does not increase. Interestingly, the throughputs of both ROMIO and *Orthrus* are reduced for block sizes of 256 KB and 1024 KB. This is caused by unsynchronized service of a large request that is spread over multiple data nodes. Figure 3 also shows the throughputs with *disk-first* and *core-first*. When the block size is small *disk-first* performs better than *core-first* because it improves disk access locality. However, with increasing block (request) size this advantage is less significant because the inter-node communication cost increases with the increase of the total amount of accessed data. Thus the performance gap between *disk-first* and *core-first* is reduced with increase of block size either before or after it reaches 64 KB. Finally, with 4096 KB blocks *core-first* performs better than *disk-first*, and *Orthrus* chooses *disk-first* over *core-first*.

In the second experiment we vary the access depth from 4 blocks to 128 blocks with a fixed block size of 16 KB and process count of 64. Figure 4 shows the results of *Orthrus* selecting different schemes for executing collective I/O. When the access depth is four blocks the amount of data accessed in one collective I/O operation is only 4 MB. In such a scenario the order of requests issued to the data nodes determines the I/O efficiency, making *disk-first* outperform ROMIO by 20%. However, when the access depth is 128 blocks, the data communication volume is increased to 128 MB and the cost of data exchange increases accordingly. For example, for one ROMIO collective I/O with an access depth of 4 blocks the data exchange time accounts for 28% of its service time. When data exchange accounts for 41% with an access depth of 128 blocks, *Orthrus* selects

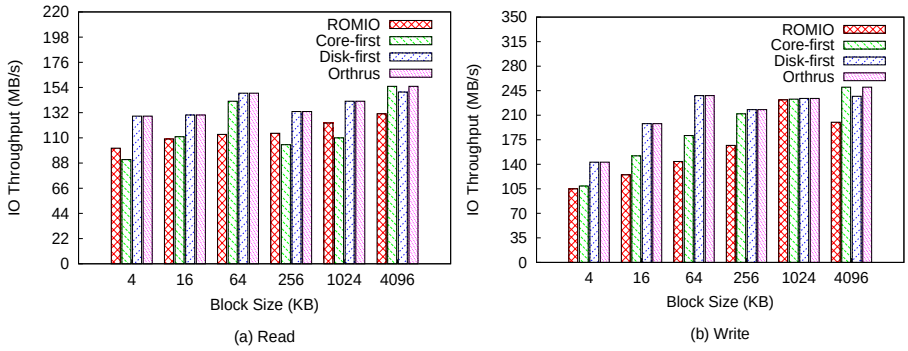

Fig. 3. Throughput of *Matrix* as a function of block size, which is also request size for *disk-first* and *core-first*, increasing from 16 KB to 4096 KB.

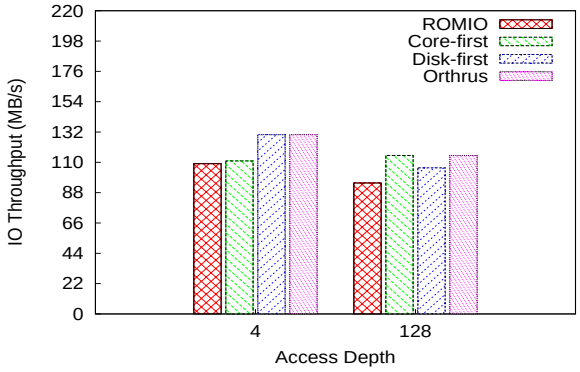

Fig. 4. Throughputs of the *Matrix* benchmark with different access depths.

core-first and reduces this to 17%. Using *core-first* compromises I/O efficiency, which is why its throughput is not significantly greater than that of *disk-first*.

3.3 Noncontig

Noncontig was developed at Argonne National Laboratory [29, 23]. In the experiments we configure the benchmark to simulate noncontiguous data access using the *vector* MPI derived data type. *Noncontig* uses *elmtcount* to describe size of a contiguously accessed data chunk. Each element is an MPI_INT, which is four bytes on our system, so the request size without using ROMIO collective I/O is 4**elmtcount*. We adjusted *veclen*, a parameter which describes the number of data chunks in a vector, so that the total accessed data size is less than 20 GB. We ran *Noncontig* with 64 processes and *elmtcount* ranging from 16 to 16384.

	2 KB	4 KB	8 KB
Random Read (MB/s)	1.0	4.1	10.2
Random Write (MB/s)	0.7	0.8	2.0

Table 1. Hard disk throughputs with random read and write requests of various sizes.

Figures 5(a) and 5(b) show read and write throughputs, respectively, with increasing *elmtcount* for each of the schemes when the data are accessed from the hard disks. *Orthrus* consistently tracks the highest achievable throughputs produced by the three candidate implementations. When request sizes are 16, 64, and 256 times *elmtcount*, i.e. the effective request sizes are 64, 256 and 1024 bytes, which are smaller than the memory page size (4 KB), issuing such small random requests to disks significantly compromises disk efficiency. This explains the low throughput with *core-first* which does not form large requests or ensure that data nodes receive sorted request streams. Interestingly, when the request size is small, i.e. *elmtcount* is 16/64/256 bytes for reads and 16/64/256/1024 bytes for writes, the throughput of ROMIO is greater than that of *disk-first* which is specifically optimized to produce fully sorted requests at each data server. ROMIO achieves the performance advantage by transforming a large number of small requests into one large one, which can be more effective when the request count is very large and requests are very small. As *elmtcount* increases to 1024 for read (4096 for write), the amount of accessed data in one collective-I/O operation is very large and the data exchange cost becomes significant. For read at 1024**elmtcount*, 67% of the operation's service time is spent on data exchange. By using *core-first* this is reduced to 6% and we observe that the throughput of *core-first* starts to exceed that of ROMIO. When *elmtcount* reaches 16,384 the data exchange time is 90% of the service time while for *core-first* it is only 2%. *Core-first* has a clear performance advantage over both ROMIO and *disk-first*, and *Orthrus* is accordingly represented by *core-first*. Figure 5 also shows that in general read has higher throughput than write. This is because hard-disk write bandwidth is lower than read bandwidth when the request size is not large (Table 1), and the fact that the benchmark does I/O synchronization after every I/O operation, which effectively disables the optimization of system write-back.

3.4 Flash-io

We used macro-benchmark *Flash-io* to further evaluate *Orthrus*. *Flash-io* is designed to provide a controlled environment for tuning the I/O performance of the multi-scale multi-physics simulation code FLASH [20], so the benchmark has an identical data access pattern to the original code and the performance improvements made to the benchmark can be realized by the FLASH application. In the benchmark three data files are generated using HDF5 and MPI-IO libraries—a checkpointing file (*chk*), a plot file with centered data (*plt_cnt*), and a plot file with corner data (*plt_crn*). Table 2 gives the times for the collective I/O operations for each of the files. The I/O characteristics of the benchmark

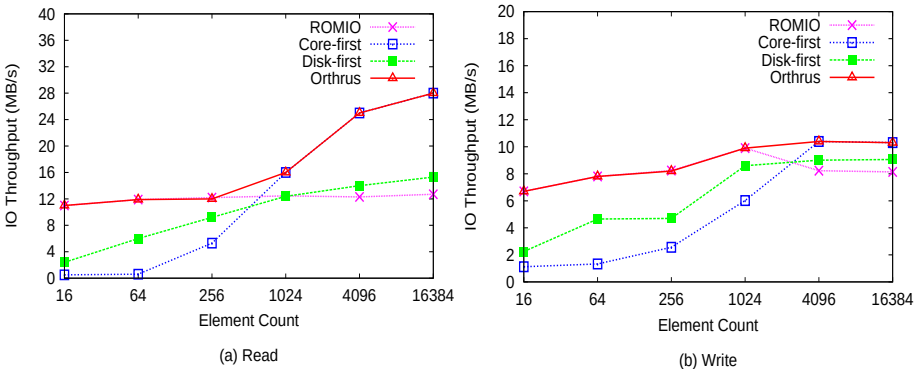

Fig. 5. Throughput of the *Noncontig* benchmark with increasing *elmtcount* when data is accessed from the hard disks.

can be changed by setting the maximum number *MAXBLOCK* of blocks per processor, the number *IONMAX* of fluids to track, and the size *nxb/nyb/nzb* of the grid blocks. Two configurations are used in experiments. Configuration A has MAXBLOCK=1500, IONMAX=100, and nxb/nyb/nzb=1; Configuration B has MAXBLOCK=600, IONMAX=100, and nxb/nyb/nzb=8. We ran the benchmark with 64 processes. As shown in Table 2, the I/O times of *chk* are reduced in both Configurations A and B by 47.2% and 26.4%, respectively. Although the performance improvements are significant in both scenarios, the explanations are different. For Configuration A the average request size is 10 KB because of the small size of grid blocks. As a result, requests issued by each of the collective I/O aggregators are not large enough, and I/O performance hinges on the service order of the requests on the disks. By choosing *disk-first Orthrus* allows each data node to receive the requests in ascending order, ameliorating the performance bottleneck of disk efficiency. Unlike Configuration A, processes in Configuration B issue much larger requests (2 MB), making the communication cost a dominant performance issue. By choosing *core-first*, *Orthrus* can effectively reduce the amount of inter-node communication. Another observation is that *Orthrus* incurs about 5.2% and 9.8% overhead for accessing *plt_cnt* and *plt_crn*, respectively, in Configuration A. We determined that this is because there are only four loop iterations when writing the plot files, more than nullifying the performance advantage of *Orthrus* with the overhead of examining collective I/O candidates.

3.5 Orthrus in a Dynamic Execution Environment

To analyze how *Orthrus* responds to changes in the run-time environment during a program's execution we ran *Matrix* with 32 processes and 64 KB block size to read a 10 GB file and injected a high volume of inter-node communication during its execution. Specifically, from the 60th second of *Matrix*'s execution we ran the FT program (discrete 3D fast Fourier Transform) from the NAS

Exp	Policy	chk(s)	plt_cnt(s)	plt_crn(s)
Configuration A	ROMIO	22.8	0.38	0.51
	Orthrus	12.03	0.4	0.56
Configuration B	ROMIO	174	4.2	5.9
	Orthrus	128	3.3	3.7

Table 2. Collective I/O times for two different configurations of *Flash-io*. Configuration A: MAXBLOCK=1500, IONMAX=100, and nxb=1; Configuration B: MAXBLOCK=600, IONMAX=100, nxb=8. *chk* represents I/O times for checkpointing, *plt_cnt* and *plt_crn* are times for writing centered and corner plot file, respectively.

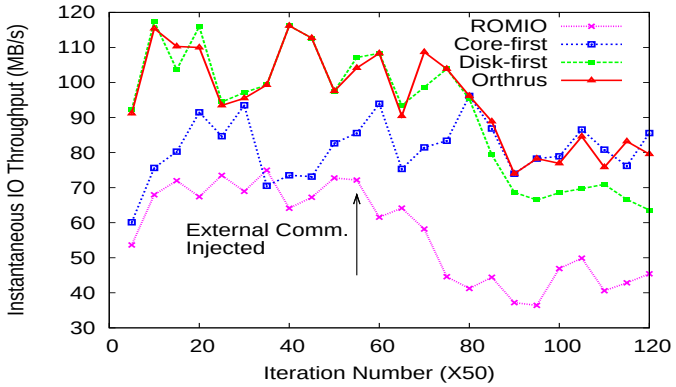

Fig. 6. Instantaneous throughput measured for the schemes after execution of every 250 iterations. The FT program starts executing at the 60th second.

Parallel Benchmarks (NPB) [30, 16] with 32 processes to generate inter-node all-to-all communication. Figure 6 shows the instantaneous I/O throughput of *Matrix*, at each multiple of 250 iterations, using each of the four strategies. Initially *Orthrus* selects *disk-first* as it did in the experiment in Section 3.2. With the injection of the external communication traffic the inter-node bandwidth available to *Matrix* is reduced and its data exchange becomes more expensive. Accordingly the throughputs of ROMIO and *disk-first* are reduced by up to 46% and 60%, respectively, and *core-first* shows its advantage. As shown in Table 3, *core-first* generates a much smaller number of IP segments as reported by /proc/net/snmp. When *Orthrus* detects the throughput degradation with *disk-first* it re-evaluates the candidates and responsively switches to *core-first*.

4 Related Work

Orthrus considers both the efficiency of storage devices and the heterogeneity of process communication, whereas most of other extant works consider only one of them. We classify these works into two categories, one seeking to optimize I/O

Schemes	ROMIO	*core-first*	*disk-first*
# of In Segs.	701,600	337,257	701,056
# of Out Segs.	1,385,008	322,884	1,412,066

Table 3. Average number of incoming and outgoing IP segments transmitted at each compute node during the execution of *Matrix* with different collective-I/O schemes. Statistics are collected by reading networking stats from /proc/net/snmp.

streams for greater spatial locality, the other to reduce inter-node communication but preserve intra-node communication in multi-core clusters.

Optimizing Request Pattern to Improve Spatial Locality. Storage systems, especially those using hard disks for direct data access, usually cannot sustain high I/O throughput with small requests for random data from a large number of parallel processes. Many techniques have been proposed to improve requests' spatial locality by transforming them into large sequential requests, such as *data sieving* [11] in ROMIO. There are variants of collective I/O recently proposed to take account of on-disk data layout, including Resonant I/O [35] and reducing locking overhead [24] during request transformation. As *Orthrus* is a framework for dynamically selecting the best performing implementation, these variants are ready to be ported into *Orthrus* which would then inherit their performance attributes. In fact, *disk-first* is a simplified version of Resonant I/O.

To provision a framework with adaptivity to accommodate workload and system dynamics to maximize performance an effort has been made using data sieving [8]. Instead of conducting actual test runs to select the best performer among multiple candidates they a model is used to predict the performance of candidates in various settings. The model takes system/workload specifics such as disk seek time, system call time, request size, and network bandwidth as inputs. In this endeavor not only is constructing an accurate model itself a significant challenge, but also the required inputs can change and may not be (immediately) available.

Parallel I/O Optimization for Multi-core CPUs. Adoption of multi-core processors in clusters introduces non-uniform communication costs [2, 13]. In such a cluster the cost of inter-node communication can be an order of magnitude higher than for intra-node communication. Furthermore, communication between cores on the same chip can be much faster than communication between chips on the same socket. Zhang et al. proposed to use the ratio of inter-node and intra-node communication costs to represent the performance effects of process affinity, which determines how processes are mapped to cores on the same or different nodes [32]. Because the implementation of ROMIO collective I/O incurs expensive all-to-all communication in the data exchange phase, Cha et al. studied the effect of aggregator assignment in the collective-I/O implementation and proposed to optimize the placement of aggregators to reduce both inter-node and intra-node communication costs [12]. Chaarawi et al. proposed a scheme for selecting the number of aggregators based on consideration of process topology, file view, and the actual amount of data requested [14]. A study

by Zou et al. found that parallel I/O performance can be compromised by lack of affinity between the core receiving an I/O interrupt and the process serving the interrupts [34]. Their proposed interrupt-scheduling scheme recovers the loss of data locality on private caches of multi-core CPUs. In the design of *Orthrus* we also consider affinity by having the option of aggregators only serving data requested by processes on the same nodes.

Self-adapting Techniques. Orthrus essentially adopts a self-adapting technique for improving I/O performance. Similar techniques have been used in other domains. For example, the load balancer of CHARM++ takes both computation and communication patterns into account at run-time to inform object migration for improving application scalability [6, 33]. To match DRAM page size and CPU cache size, the CMSSL library contains routines for automatic selection of optimal parameters, such as loop order and operator alignments, for matrix multiplication in both local and global scopes [18]. Benkert et al. uses an empirical approach for MPI communication auto-tuning with ADCL library [3]. In comparison, Orthrus introduces a self-adapting technique to improve collective I/O performance and demonstrates its feasibility and effectiveness, which may inspire more innovative applications of the technique to address I/O issues in large-scale HPC systems.

5 Conclusion

We have presented the design and implementation of *Orthrus*, a framework for hosting multiple collective-I/O implementations and adaptively selecting the one that provides the highest I/O throughput according to current workload and system dynamics. Instead of attempting to optimize an existing collective-I/O implementations, or develop a new one from scratch, we demonstrate an open framework allowing multiple implementations to compete with, and complement, each other. This represents a unique approach that we have shown to be effective in the context of a multi-core cluster that has both I/O and communication efficiencies to optimize. More abstractly, it suggests a general approach for dynamic selection of multiple performance-optimization strategies where accurate modeling or simulation would be infeasible or non-portable. Experiments with the prototyped *Orthrus* in the ROMIO library show that it can improve the throughput of collective I/O under various workloads and system scenarios by up to several times.

6 Acknowledgments

This work was supported by US National Science Foundation under CAREER CCF 0845711, CNS 1117772, and CNS 1217948. This work was also funded in part by the Accelerated Strategic Computing program of the US Department of Energy. Los Alamos National Laboratory is operated by Los Alamos National Security LLC for the US Department of Energy under contract DE-AC52-06NA25396.

References

1. The Opportunities and Challenges of Exascale Computing, In http://science.energy.gov/media/ascr/ascac/pdf/reports/ exascale_subcommittee_report.pdf/.
2. S. Alam, R. Barrett, J. Kuehn, P. Roth, and J. Vetter, "Characterization of Scientific Workloads on Systems with Multi-core processors", In *IEEE International Symposium on Workload Characterization*, 2006.
3. K. Benkert and E. Gabriel, "Empirical Optimization of Collective Communications with ADCL" In *High Performance Computing on Vector Systems 2010* , 2010.
4. S. Byna, Y. Chen, X. Sun, R. Thakur, W. Gropp, "Parallel I/O Prefetching Using MPI File Caching and I/O Signatures" In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2008.
5. A. Carroll, "Linux Block I/O Scheduling", In http://www.cse.unsw.edu.au/aaronc/ iosched/doc/sched.pdf, 2007.
6. "Parallel Languages/Paradigms: Charm ++ - Parallel Objects", http://charm.cs.uiuc.edu/research/charm/.
7. A. Ching, A. Choudhary, W. Liao, L. Ward, and N. Pundit, "Evaluating I/O Characteristics and Methods for Storing Sturctured Scientific Data", In *IEEE Internatinal Parallel & Distributed Processing Symposium*, 1996.
8. Y. Lu, Y. Chen, P. Amritkar, R. Thakur, and Y. Zhuang, "A New Data Sieving Approach for High Performance I/O", In *the 7th International Conference on Future Information Technology*, 2012.
9. A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp, "Efficient Structured Data Access in Parallel File Systems", In *IEEE Internatinal Conference on Cluster Computing*, 2003.
10. A. Ching, A. Choudhary, K. Coloma, and W. Liao, "Noncontiguous I/O Access Through MPI-IO", In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2003.
11. K. Coloma, A. Ching, A. Choudhary, W. Liao, R. Ross, R. Thakur, and L. Ward "A New Flexible MPI Collective I/O Implementation", In *IEEE International Conference on Cluster Computing*, 2006.
12. K. Cha and S. Maeng, "An Efficient I/O Aggregator Assignment Scheme for Collective I/O Considering Processor Affinity", In *7th International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems*, 2011.
13. L. Chai, Q. Gao, and D. Panda, "Understanding the Impact of Multi-core Architecture in Cluster Computing", In *the 7th IEEE International Symposium on Cluster Computing and the Grid*, 2007.
14. M. Chaarawi, and E. Gabriel, "Automatically Selecting the Number of Aggregators for Collective I/O Operations", In *IEEE International Conference on Cluster Computing*, 2011.
15. FLASH IO Benchmark Routine-Parallel HDF 5, http://www.ucolick.org/ zingale/flash_benchmark_io/.
16. "FT: Discrete 3D Fast Fourier Transform", *http://www.nas.nasa.gov/publications/npb.html*
17. HDF5 documents, http://www.hdfgroup.org/HDF5/whatishdf5.html.
18. S. L. Johnsson, "CMSSL: a Scalable Scientific Software Library", *In Proceedings of Scalable Parallel Libraries Conference*, Mississippi State, MS, 1993.
19. PVFS2, Parallel Virtual File System (Version 2), http://www.pvfs.org/.

20. K. Riley, "Introduction to Flash", http://flash.uchicago.edu/site/flashcode/ user_support/tutorial_talks/home.py?submit=May2004.txt.
21. "Tuning I/O Performance", In http://doc.opensuse.org/products/draft/SLES/SLES-tuning_sd_draft/cha.tuning.io.html, 2012.
22. R. Thakur, W. Gropp, and E. Lusk, "Data Sieving and Collective I/O in ROMIO", *In Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, Annapolis, MD, 1999.
23. R. Latham, and R. Ross, "PVFS, ROMIO, and the noncontig Benchmark", 2005.
24. W. Liao and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols" In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2008.
25. J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Proc. CLADE'08*, 2008.
26. "Lustre File System", In https://www.xyratex.com/products/lustre, 2012.
27. MPICH2, A High Performance Message Passing Interface, http://www.mcs.anl.gov/ research/projects/mpich2/.
28. K. Madduri, K. Ibrahim, S. Williams, E. Im, S. Ethier, J. Shalf, and L. Oliker, "Gyrokinetic Toroidal Simulations on Leading Multi- and Manycore HPC Systems" In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2011.
29. Noncontig I/O Benchmark, http://www-unix.mcs.anl.gov/thakur/pio-benchmarks.html.
30. "NAS parallel benchmarks, NASA Ames Research Center, 2009", http://www.nas.nasa.gov/Software/NPB.
31. Spider - the Center-Wide Lustre File System, http://www.olcf.ornl.gov/kb_articles/ spider-the-center-wide-lustre-file-system/.
32. C. Zhang, X. Yuan, and A. Srinivasan, "Processor Affinity and MPI Performance on SMP-CMP Clusters", In *11th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing*, 2010.
33. G. Zheng, "Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing", In *PhD Thesis*, 2005.
34. H. Zou, X. Sun, S. Ma, and X. Duan, "A Source-Aware Interrupt Scheduling for Modern Parallel I/O Systems", In *26th IEEE International Parallel & Distributed Processing Symposium*, 2012.
35. X. Zhang, S. Jiang, and K. Davis, "Making Resonance a Common Case: A High-Performance Implementation of Collective I/O on Parallel File Systems", In *23th IEEE International Parallel & Distributed Processing Symposium*, 2009.
36. X. Zhang, Y. Xu, and S. Jiang, "YouChoose: A Performance Interface Enabling Convenient and Efficient QoS Support for Consolidated Storage Systems", In *27th IEEE Symposium on Massive Storage Systems and Technologies*, 2011.