Making Resonance a Common Case: A High-Performance Implementation of Collective I/O on Parallel File Systems

Xuechen Zhang[1], Song Jiang[1], and Kei Davis[2]
[1]ECE Department
Wayne State University
Detroit, MI 48202, USA
{xczhang,sjiang}@eng.wayne.edu

[2] Computer and Computational Sciences
Los Alamos National Laboratory
Los Alamos, NM 87545, USA
kei.davis@lanl.gov

Abstract

Collective I/O is a widely used technique to improve I/O performance in parallel computing. It can be implemented as a client-based or server-based scheme. The client-based implementation is more widely adopted in MPI-IO software such as ROMIO because of its independence from the storage system configuration and its greater portability. However, existing implementations of client-side collective I/O do not take into account the actual pattern of file striping over multiple I/O nodes in the storage system. This can cause a significant number of requests for non-sequential data at I/O nodes, substantially degrading I/O performance.

Investigating the surprisingly high I/O throughput achieved when there is an accidental match between a particular request pattern and the data striping pattern on the I/O nodes, we reveal the resonance *phenomenon as the cause. Exploiting readily available information on data striping from the metadata server in popular file systems such as PVFS2 and Lustre, we design a new collective I/O implementation technique,* resonant I/O*, that makes resonance a common case. Resonant I/O rearranges requests from multiple MPI processes according to the presumed data layout on the disks of I/O nodes so that non-sequential access of disk data can be turned into sequential access, significantly improving I/O performance without compromising the independence of a client-based implementation. We have implemented our design in ROMIO. Our experimental results on a small- and medium-scale cluster show that the scheme can increase I/O throughput for some commonly used parallel I/O benchmarks such as* mpi-io-test *and* ior-mpi-io *over the existing implementation of ROMIO by up to 157%, with no scenario demonstrating significantly decreased performance.*

1. Introduction

As large-scale scientific applications running on clusters become increasingly I/O intensive, it is important to have effective system support for efficient I/O between the processes on the compute nodes issuing I/O requests, and the disks on the I/O nodes servicing the requests [4, 5, 18]. A problematic situation in I/O performance is the issuance of requests for many small non-contiguous I/O accesses, because un-optimized servicing of these requests results in low disk efficiency and high request processing cost. Many techniques have been proposed to address this problem, including data sieving [26], list I/O [6], datatype I/O [7], and collective I/O [26]. Of these, collective I/O is one of the more commonly used techniques and usually yields the greatest improvement in I/O performance. This is because collective I/O rearranges requests from multiple processes (global optimization), rather than optimizing requests from each individual process (local optimization).

1.1 Transforming Non-contiguous Access into Contiguous Access

A common technique used in the aforementioned schemes for optimizing I/O performance is to transform small requests of non-contiguous access into large requests of contiguous access. Let us first see how the read operation can benefit from collective I/O. As depicted in Figure 1, four processes, P_0, P_1, P_2, and P_3, each requests four segments that are not adjacent in the logical file space. Because an I/O request must be issued for logically contiguous data, each process issues four requests. Without collective I/O there would be 16 small requests from the compute nodes to the I/O nodes, with each I/O node receiving and servicing four requests in random order.

With collective I/O, all the requested data is divided into

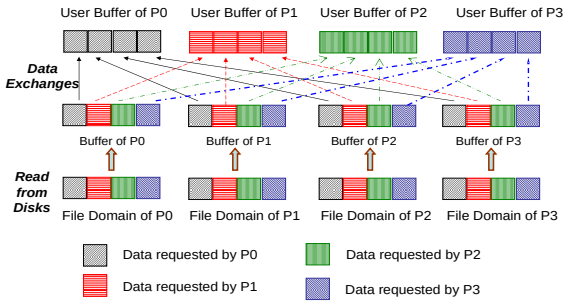

Figure 1. Illustration of the ROMIO implementation of two-phase collective I/O. Data is read by each process (the aggregator), P_0, P_1, P_2, or P_3, which is assigned a contiguous file domain in the logical file space, first into its temporary buffer in phase I and then to the user buffer of the process that actually requested them in phase II.

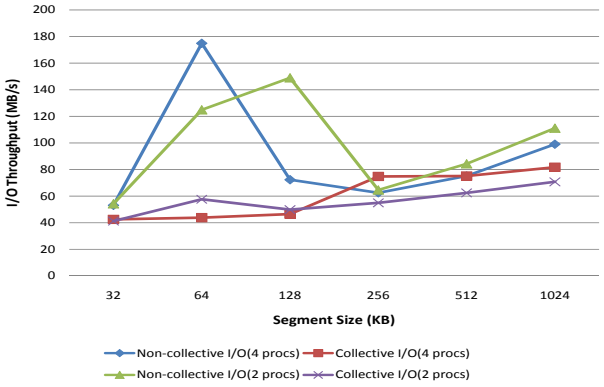

Figure 2. Throughput of I/O nodes when running a demonstration MPI program with two and four processes, varying the segment size from 32KB to 1024KB, with and without collective I/O. Throughput peaks at 64KB with *non-collective-4*, and at 128KB with *non-collective-2*, exhibiting *resonance* between the data request pattern and the striping pattern.

four file domains, each consisting of four contiguous segments, and each process issues a single request to read data belonging to a single file domain into its buffer. After the reads complete, each process retrieves its respective data from the others' buffers via inter-node message passing. As an example of a widely used collective-I/O implementation, ROMIO [26] adopts a two-phase strategy. In the first phase, each process serves as an *aggregator*, with process P_k ($k \geq 0$) responsible for reading the kth file domain into its buffer. In the second phase, data is exchanged among the processes to satisfy their actual requests. The rationale for this implementation of collective I/O is two-fold. First, both the number of requests issued to the I/O nodes, and the request processing overhead, are reduced. Second, contiguous access is expected to be more efficiently serviced on the disk-based I/O servers than non-contiguous access because contiguous access requires fewer disk head movements, which can account for more than an order of magnitude disparity in disk throughput. Clearly, for collective I/O to improve rather than degrade performance, the gains must outweigh the communication overhead incurred in this second phase that does not exist in the traditional non-collective I/O scheme,

1.2 The Resonance Phenomenon

To analyze how collective I/O performs in a typical cluster computing environment, we set up an experimental platform consisting of eight nodes, four configured as compute nodes, and the other four as I/O nodes, managed by a PVFS2 parallel file system [1]. File data was striped over the I/O nodes. We used the default PVFS2 striping unit size of 64KB. (More details of the experimental platforms are given in Section 4.)

In our experiment we ran N-process MPI programs, where N was 2 or 4, one process per compute node, that read data from a 10GB file striped over the four I/O nodes. The access pattern was generally the same as that illustrated in Figure 1. Specifically, the processes repeatedly call collective I/O to read the entire file from beginning to end. In each call, process i, $i \in \{0, 1, \ldots, N-1\}$, reads segments $k * N + i$, $k \in \{0, 1, 2, 3\}$, in the file range specified by the call. The size of the segment was varied from 32KB to 1024KB (powers of two times 32KB) over different runs of the program. Figure 2 shows the I/O throughput of the system using collective I/O with N processes and the various segment sizes, denoted as *collective-I/O-N*, where N is 2 or 4. The graph also shows the throughput with N processes when each process makes four distinct I/O calls for each of its four segments of contiguous data, denoted as *non-collective-I/O-N*. As we expect, with *collective-I/O-N*, increasing segment size (amount of requested data) gives increasing throughput. This is consistent with the fact that the disk is more efficient with large contiguous data access because of better amortized disk seek time. Surprisingly, however, we see that the throughput for *non-collective-I/O-4* reaches a peak value of 175MB/s at 64KB segment size, which is much higher than the 42MB/s throughput for *collective-I/O-4* at the same segment size. Similarly, for *non-collective-I/O-2* there is a peak of 149MB/s at 128KB, versus 72MB/s for *collective-I/O-2*. This appears to be inconsistent with the assumption that requests for larger contiguous data would be more efficiently serviced.

The reason for these throughput peaks lies in the order in which the requests arrive at each I/O node. Figure 3 illustrates the different orders when collective I/O is used (Figure 3(a)) and is not used (Figure 3(b)) in the case of four processes. When collective I/O is used, four contiguous segments are assigned to a process as a file domain.

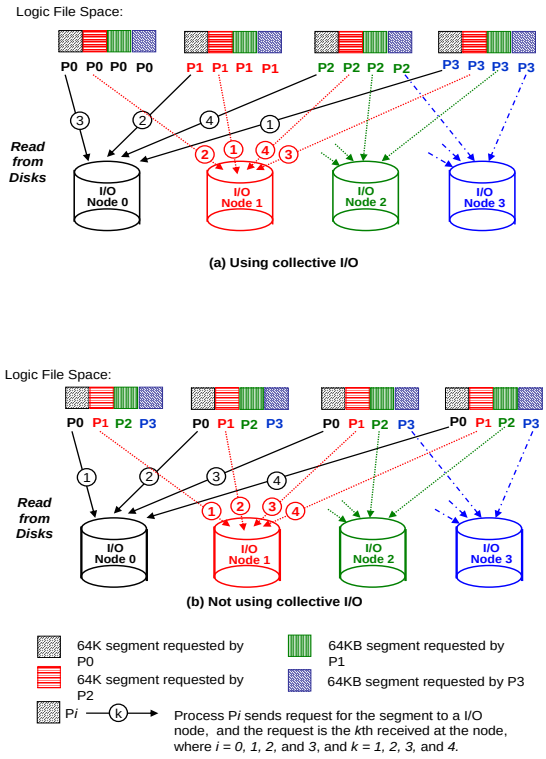

Figure 3. Illustration of how a *resonance* is developed: when collective I/O is used (a), each process reads four contiguous segments, but each I/O node receives requests from four processes in an unpredictable order. When collective I/O is not used (b), a process sends four requests for four non-contiguous segments to one I/O node, making the service order of the requests at the node consistent with the offsets of the requested data in the file domain.

Because both segment size and striping unit size are 64KB, the four requests to a particular I/O node, each for 64KB data, come from four concurrent processes and arrive in an order determined by the relative progress of the processes, which is unpredictable. In an operation manual for the Lustre cluster file system this issue is raised as a disadvantage of striping a file into multiple objects (the portion of file data on one I/O server). Consider a cluster with 100 clients and 100 I/O servers. Each client has its own file to access. The manual [24] states: *"If each file has 100 objects, then the clients all compete with one another for the attention of the servers, and the disks on each node seek in 100 different directions. In this case, there is needless contention."* This exactly describes the situation with collective I/O when multiple processes access the same file on multiple I/O nodes simultaneously. We note that while the I/O scheduler at the I/O node can re-order the requests for a sequential dispatching order, this re-ordering operation will rarely occur unless the I/O system is saturated and many requests are pending. Therefore, an I/O node usually serves requests in the order that they are received—in random order from the viewpoint of I/O node—which degrades disk performance. In contrast, when collective I/O is not used, all four requests to an I/O node are from the same process, which sends them one by one in the order of their offsets in the logical file space. Because the file system generally allocates data on the disk in an order consistent with their offsets in the file domain, the consequent sequential service order at an I/O node leads to an effective prefetching at the I/O node [15]. We name this scenario, in which an accidental match between data request pattern and data striping pattern produces sequential disk access and peak disk performance, *resonance* in the distributed I/O service, a term borrowed from the physics field. A similar resonance exists with *non-collective-I/O-2* with 128KB segment size, in which two I/O nodes are dedicated to service requests from one process (one segment is striped on two nodes), and no I/O node receives requests from multiple processes that cause random disk accesses. We also observe that *non-collective-I/O-2* with 64KB segment size generates a resonance, though with a throughput (125MB/s) lower than the one (149MB/s) at 128KB segment size. The lower throughput is a result of under-utilized I/O nodes, because at any time only two of the four I/O nodes are servicing requests from the two processes.

Analyzing the conditions for resonance to occur, we see that the key factor for high I/O throughput is not simply accessing a contiguous file domain, rather, it is ensuring sequential access of data on an I/O node. When data is striped over multiple I/O nodes, collective I/O, which designates one contiguous file domain to a process, allows requests for data on an I/O node to be from multiple processes, which introduces the indeterminacy that leads to non-sequential access. If we can rearrange requests involved in collective I/O such that all the requests for data on an I/O node come from one process, then resonance would be a common case when each process requests its data in ascending order of file offset. This is one the of techniques used in our proposed implementation of collective I/O, called *resonant I/O*.

The rest of this paper is organized as follows. Section 2 gives a brief description of related work. In Section 3 we describe the design and implementation of resonant I/O in detail. Performance evaluation results are presented in Section 4, followed by conclusions in Section 5.

2 Related Work

There is a large body of work on improving I/O performance for high-performance computing. Among the work directly related to our work presented here, there are two important issues that have been studied.

The first issue is how to efficiently access a large number of small, non-contiguous pieces of data. This access pattern, typically produced by directly using UNIX-style *read* and *write*, can incur a large overhead in processing requests. It also prevents systems from inferring the 'big picture' of access patterns to enable optimization in a larger

scope, such as rearrangement of requests for sequential disk access. Data sieving [26] is one of the techniques proposed to address this issue by aggregating small requests into large ones. Instead of accessing each small piece of data separately, data sieving accesses a large contiguous scope of data that includes the small pieces of data. If the additional unrequested data, called holes, is not excessive, the benefit can be significant. However, data sieving cannot ensure that its aggregated large requests from multiple clients are serviced at each I/O node in an order that minimizes disk seeks, which is the objective of resonant I/O.

Datatype I/O [7] and list I/O [6] are the two other techniques that allow users to specify multiple non-contiguous accesses with a single I/O function call. Datatype I/O is used for accesses with certain regularity, while list I/O handles a more general case. Compared to these techniques collective I/O is more aggressive in aggregating small requests by re-arranging requests collectively issued by a group of processes. However, while collective I/O does increase the size of a request for data contiguous in the logical file space, it may adversely cause requests to arrive at the I/O nodes in random order, as we have shown. (As will be described later, list I/O will play a role in our design.) In other work [11], an optimization is made to improve the ROMIO collective-I/O efficiency in a cluster where data is striped on the disks local to each compute node. The efficiency is achieved by making each agent process access only data on its local disks. In contrast, resonant I/O addresses I/O efficiency in a cluster with dedicated I/O nodes that may each service requests from multiple compute nodes.

The second issue is portability. As most high-performance cluster computing platforms are customized configurations, there are many variations in software and hardware architectures. To be widely adopted, a technique must be minimally dependent on specific system structures and configurations. As an example, ROMIO [26] is a high-performance and portable implementation of MPI-IO [27] in which the aforementioned optimization techniques, including collective I/O, are included. ROMIO uses ADIO (Abstract Device Interface for MPI-I/O) [25], an internal layer to accommodate the machine-dependent aspects of the implementation of MPI-IO, so that MPI-IO can be implemented portably on top of ADIO. Because the configuration of the storage subsystem of a cluster may be modified independently of the computing subsystem, it is desirable to implement I/O optimization techniques on the client side to keep them independent of configuration of storage subsystem. Collective I/O, as well as other commonly used techniques, are usually implemented on the client side. In contrast, server-side implementations such as server-directed collective I/O [23] are less adopted. Server-directed collective I/O was developed as a component of Panda, an I/O library for accessing multi-dimensional arrays, on the IBM SP2 supercomputer [23]. In this system I/O nodes are heavily involved in the re-arrangement of I/O requests by collecting request information from compute nodes and then directing them for sending/receiving data. Disk-directed I/O [12] is a strategy similar to server-directed collective I/O, with the addition of explicit scheduling of requests according to the data layout on disk. While these two techniques can provide performance benefits similar to resonant I/O, both of them compromise the independence of middleware on compute-side I/O, such as MPI-IO, from configuration changes on the I/O-node side.

3 The Design of Resonant I/O

The design objective of resonant I/O is to ensure that requests arrive at each I/O node in ascending order of file offsets for requested data from the same file. While data layout on disk usually matches offsets in the logical file space, the design allows the disk to service the requests in its preferred order, i.e., from small disk addresses to high addresses (possibly sequential), to achieve high disk throughput.

3.1 Making Collective I/O Aware of Data Layout

To induce resonance the compute node must know on which I/O node requested data is stored. Because an important design goal for the compute-node-side middleware is keeping the middleware independent of the I/O node side's configuration to ensure portability and system flexibility, explicitly requesting this information from the I/O nodes is undesirable.

Fortunately, the configuration information that is needed in resonant I/O is readily available on the compute node side in many commonly used parallel file systems, including PVFS2 [1, 13], Lustre [8, 22], and GPFS [21]. In these systems meta-data service is separate from data service to avoid bottlenecks in data transfer. As such, a compute node needs to first communicate with the meta-data server to acquire the locations of its requested data on the I/O nodes before it can access data on I/O nodes. In fact, we only need to know the striping unit size and number of I/O nodes, from which we can determine which requested data is on the same I/O node. We are aware that these two parameters may be set by users when the file is created in some file systems such as Lustre. However, to keep the design general and the interfaces of collective I/O unchanged, we do not assume that users would provide these parameters when they call collective I/O functions.

3.2 Process Access Set and I/O Node Access Set

Because resonant I/O is an implementation of collective I/O, it does not make any changes to the function interfaces seen by programmers. As usual, each participant in a resonant-I/O operation needs to call the same collective-I/O function to specify one file segment or multiple non-adjacent file segments in a request. To execute the function call the processes are first synchronized to exchange information on the requested file segments so that every process knows all the file data requested in the collective I/O. After that, a collective-I/O implementation strategy needs to decide, for each process, which data the process is responsible for accessing. We call the set of data that is assigned to a process its *access set*. Once a process knows its access set it generates one or multiple requests to the I/O nodes to access the data specified by the access set. In ROMIO collective I/O all file data to be requested are evenly partitioned into contiguous file domains. Each file domain is the access set of a process, which then uses only one request to access the data. Because this method of forming access sets based on contiguity in the logical file seeks to reduce the number of requests as well as their processing overhead, the resulting pattern of requests does not necessarily help improve disk efficiency, as described in Section 1.

To achieve disk efficiency in the implementation of collective I/O, we define an I/O node's access set as the set of data that is requested in a collective I/O *and* is stored on the I/O node. One of the objectives of resonant I/O is to ensure that an I/O node's access set is accessed by requests arriving in the ascending order of the offsets of the data in the logical file domain. Note that it is the LBNs (logical block numbers)[1] of the data that represent the on-disk locations of the data and directly determine the disk efficiency, and there is a mapping from the logical file offsets to on-disk LBNs by file systems. Therefore, in theory, ascending file offsets do not necessarily correspond to ascending LBNs, but in practice the correspondence generally holds, especially for file systems managing large files. Furthermore, our objective is that client-side optimization, such as resonant I/O, not require detailed configuration information on the server side. Using file offsets for this purpose fulfills this objective. Because the striping unit size and the number of I/O nodes are available, processes on the compute nodes can easily calculate the access set of each I/O node.

The reasons that an I/O node's access set might be requested in a random order are that (1) data in the I/O node's access set belongs to multiple processes' access sets; *and*, (2) these processes send their requests in random order because of their unpredictable relative progress. To produce an ascending access order at an I/O node, resonant I/O can take either of two actions: (1) make *one* process' access set be an I/O node's access set; or, (2) make multiple processes send their requests in a pre-defined order. In the following we describe how resonant I/O takes the first action as its basic approach to produce an ascending access order, and takes the second action to make an optimization for a particular request pattern.

[1] If the I/O node is attached to a disk array the LBN refers to the address in the array.

3.3 Designating Agent Processes According to the I/O Node's Access Set

If a process' access set is the same as an I/O node's access set, and the process sends its requests to the I/O node in ascending order of offset, then the I/O node will receive all of its requests in the preferred order. We call such a process the I/O node's *agent process*. Assuming each I/O node needs one agent process, for a given I/O node we select the process that requests the largest amount of data from the I/O node and has not been selected as another I/O node's agent process. If more than one such process exists, we arbitrarily choose the one with lowest rank in the MPI process group. As some data requested by an agent process may belong to other processes and need to be transferred between the agent process and their owner processes, this strategy minimizes the data to be transferred. The data transfer takes place before access to the I/O nodes in the write operation, and after access to the I/O nodes in the read operation. This data transfer is similar to the inter-process communication phase in ROMIO collective I/O. However, we make a special optimization for the read operation in this phase to minimize the transfer cost, as follows.

Synchronization is usually required after each agent has read data from I/O nodes into its buffer and before the inter-process data transfer starts. This synchronization can degrade I/O performance by forcing all processes to wait for the slowest process to read its data; moving the synchronization ahead of the read operation would obviate this. To this end, we let all agent processes send their requests for their access sets in a non-blocking fashion in the first phase of the read operation, assuming non-blocking I/O is supported, and synchronize their progress immediately after sending requests instead of after the data has been read. Then each process proceeds to read *directly* from the I/O nodes the data that it needs but has not requested in the first phase. If the process is not an agent, the data is actually all that it needs to access. This step replaces inter-process data transfer to eliminate synchronization immediately before the second phase. In this arrangement, we actually make many requests issued in the first phase serve as prefetching hints for the requests issued in the second phase. By performing the synchronization we ensure that requests in the second phase arrive after the I/O nodes receive re-

quests from the agents in the first phase. Thus the request service order at an I/O node is determined by the arrival order of requests in the first phase. When data is read from the disk, the requests of the second phase would be satisfied in the buffer cache of the I/O node. Usually the buffer cache is large enough to hold the data when the requests in the second phase arrive. By using the prefetching-like method, the two phases in resonant I/O can be overlapped to achieve higher efficiency.

Because an agent process may send many requests to an I/O node in resonant I/O, compared with one request in the ROMIO collective I/O, the request processing cost can be substantially higher. To reduce this cost resonant I/O uses list I/O to pack small requests for non-contiguous data segments into one or a few large requests to minimize request processing overhead. For the ROMIO implementation in MPICH2, one list I/O can accommodate up to MAX_ARRAY_SIZE (64) non-contiguous data segments, which can significantly reduce the cost.

3.4 Timing Requests from Different Processes

Because the second phase in the conventional implementation of collective I/O is the additional cost that does not exist in the non-collective I/O scheme, we seek to eliminate it subject to the condition that the access pattern satisfies a *non-overlapping condition*. This condition requires that in a collective I/O call the file offsets of the data requested by process i are smaller than those of data requested by process $i+1$ ($i = 0, 1, ..., N-2$; N is the number of processes). If a collective I/O call satisfies the condition for all the requests in the call to a given I/O node, those from process i will be for data with offsets smaller than those from process j ($i < j$). If we place the processes into sets according to the I/O nodes to which they send their requests such that processes in different sets do not share I/O nodes, and ensure that for all processes in a set, a process with lower rank always sends its request earlier than a process of higher rank, then the I/O nodes will receive the requests in the preferred order. For this particular request pattern, by timing the sending of requests in different processes, we can produce the same effect on request arrival order as by using process agents. Then we can eliminate the second phase in which data is transferred to their owner processes, because each process requests its own data.

When the non-overlapping condition is satisfied, in each process set the process with lowest rank sends its request(s) first, and after a short delay it sends a synchronous message to the process with the next higher rank in the set, which then repeats the procedure. The delay is introduced to ensure that requests arrive at I/O nodes in the preferred order. Our study has shown that because disk access time is usually much higher than message passing time, this delay can be chosen from a relatively large range, such as from 0.1ms to 1ms, with little effect on I/O performance, especially in a system supporting non-blocking I/O where a process can send its message without waiting to receive its requested data. (We note that the choice of delay does not affect the *correctness* of the protocol, only performance.) If non-blocking I/O is not supported no delay would be imposed and I/O access among processes would be fully serialized.

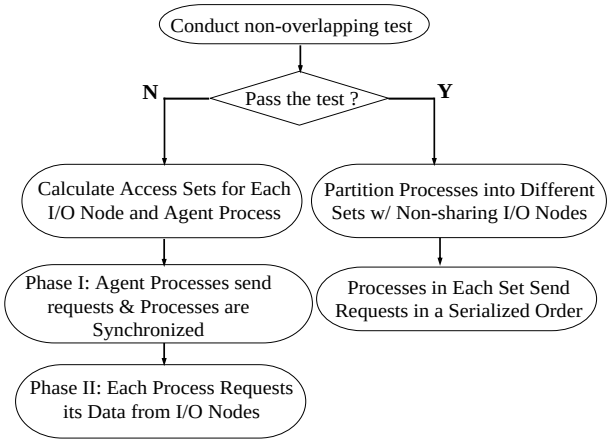

Figure 4. Algorithmic Description of Resonant I/O

Because we coordinate request sending among processes, the benefits of improved disk efficiency will outweigh the penalty of reduced concurrency of I/O operations if the number of processes is comparable to the number of I/O nodes. Otherwise, the serialization could become a performance bottleneck. To maintain balance, we set up n process groups in each process set sharing a common set of I/O nodes, where n is the number of the I/O nodes. We place the ith process in a set, sorted by rank, into group k, where $k = i/n$. Then processes in the same group send their requests without coordination, and the timing (or serialization) is carried out between process groups.

This timing technique can also be applied to make the approach using agent processes more scalable. When the number of processes in a collective I/O is much larger than the number of I/O nodes, and the amount of data to be requested is very large, resonant I/O can designate more than one process agent for each I/O node for higher network bandwidth. This is made possible by timing the request sending in these multiple agent processes.

3.5 Putting it All Together

Figure 4 summarizes the design of resonant I/O. The objective in the design is to make requests served at each I/O node arrive in the preferred order. This is achieved by ei-

ther allowing requests to one I/O node to be from the same agent process or by coordinating the issuance of requests from multiple processes. In achieving this objective, several optimizations were applied, including minimization of the cost of synchronization and elimination of the second phase of a conventional implementation of collective I/O.

4 Performance Evaluation and Analysis

To evaluate the performance of resonant I/O and compare it with the widely used collective I/O implementation in ROMIO, we used two different experimental platforms. The first is our own dedicated system, an eight-node cluster. All nodes are of identical configuration, each with dual 1.6GHz Pentium processors, 1GB memory, and an 80GB SATA hard disk. The cluster uses the PVFS2 parallel virtual file system (version 2.6.3), in which four nodes were configured as compute nodes and the other four as I/O nodes. Each node runs Linux 2.6.21 and uses GNU libc 2.6. One of the I/O nodes is also configured as the meta-data server of the file system. We used MPICH2-1.0.6 with ROMIO for our MPI programs. All nodes are connected through a switched Gigabit Ethernet network. The default striping unit size, 64KB, is used to stripe file data over the I/O nodes. The second platform, used to evaluate how the performance of resonant I/O scales in a shared production environment, is described in the section on scaling.

Our resonant I/O is implemented in ADIO on top of PVFS2. The current version of ADIO does not provide genuine support for non-blocking I/O functions [14]. Because of this limitation our implementation of resonant I/O makes some compromises: (1) for the read operation, the second phase is not initiated until the data requested in the first phase has been received by the agent processes, which nullifies much of the benefit of using prefetching-like data access in the second phase; and, (2) the I/O operations among process groups are serialized. The consequence of these compromises is that experimental results for resonant I/O presented here are conservative, and potential performance advantages may not be fully revealed.

In addition to the demonstration program we used in Section 1 to exhibit the resonance scenario, we used five well-known benchmark programs for the evaluation: *coll_perf* from the MPICH2 software package, *mpi-io-test* from the PVFS2 software package, *ior-mpi-io* from the ASCI Purple benchmark suite developed at Lawrence Livermore National Laboratory [10], *noncontig* from the Parallel I/O Benchmarking Consortium [20] at Argonne National Laboratory to test I/O characteristics with noncontiguous file access [19], and *hpio*, designed by Northwestern University and Sandia National Laboratories, to systematically evaluate performance with a diverse set of I/O access patterns [9, 2].

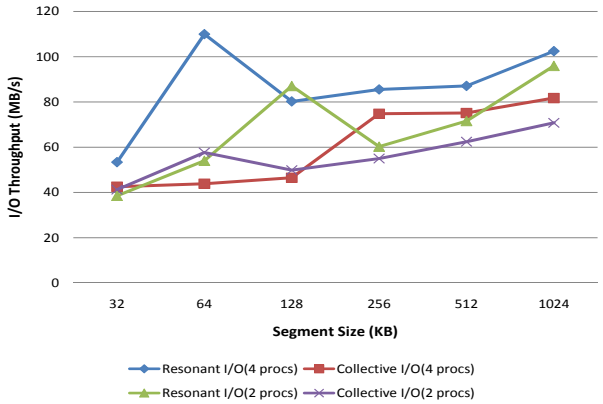

Figure 5. I/O throughput of the demonstration program with varying segment sizes and number of processes.

All presented measurements represent arithmetic means of three runs. The variation coefficients—the ratio of the standard deviation to the mean—are less than 5% for the experiments on the dedicated cluster and less than 20% on the production system. To ensure that all data was accessed from the disk, we flushed the system buffer caches of the compute nodes and I/O nodes before each test run.

4.1 Revisiting the Demonstration Program

We first revisit the demonstration program presented in Section 1. Figure 5 shows the I/O throughput observed when running the program with ROMIO collective I/O and resonant I/O with two and four MPI processes. The figure shows that resonant I/O can significantly improve I/O performance. It produces its peak throughput for segment size of 64KB with four processes and for segment size of 128KB with two processes, the two scenarios where resonance take place when I/O requests are not collectively issued (c.f. Figure 2). In these two scenarios, resonant I/O increases I/O throughput by 151% and 75% over their counterparts in ROMIO collective I/O, respectively. However, the throughput of resonant I/O in these two scenarios is less than those of non-collective I/O shown in Figure 2. This is because resonant I/O needs synchronization in each call, which slows the faster processes. In fact a collective call is not necessary when an I/O node is dedicated to a process. For a segment size of 32KB and with two processes, ROMIO collective I/O coincidentally requests data in the same pattern as resonant I/O, so it has almost the same throughput as that of resonant I/O.

4.2 Results on the Dedicated Cluster

We ran benchmarks *coll_perf*, *mpi-io-test*, *ior-mpi-io*, *noncontig*, and *HPIO* on the dedicated cluster to measure

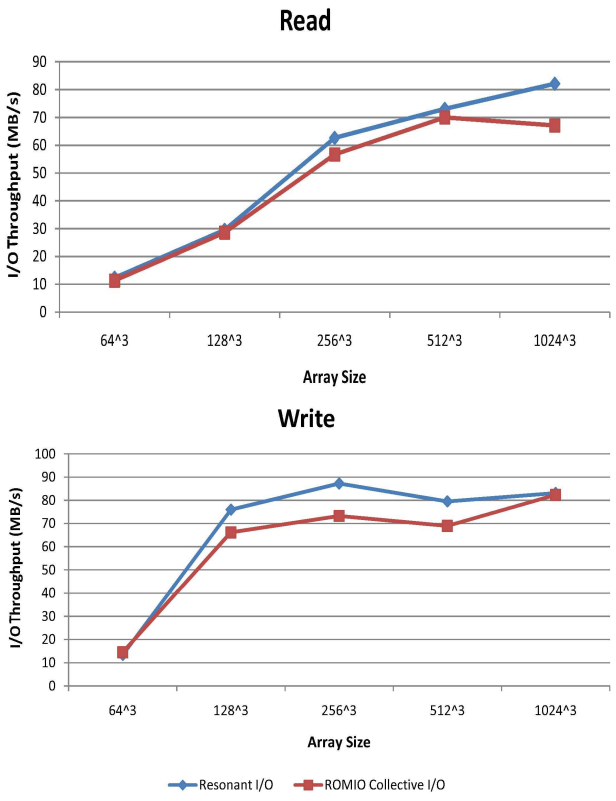

Figure 6. I/O throughput of benchmark *coll_perf* with varying scale of arrays.

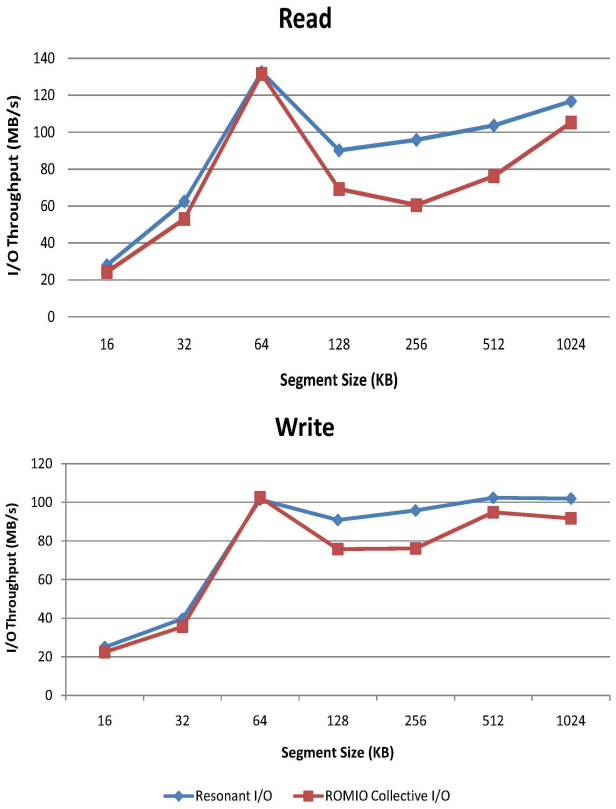

Figure 7. I/O throughput of benchmark *mpi-io-test* with varying segment sizes.

their achieved aggregate I/O throughput when resonant I/O, and ROMIO collective I/O, were used. Because the I/O operation in all but *coll_perf* can be either file *read* or file *write*, for all but *coll_perf* we measured the read and write throughputs separately.

4.2.1 Benchmark *coll_perf*

The benchmark *coll_perf* comes from the MPI source package. Using collective I/O, this benchmark first writes a 3D block-distributed array to a file which resides on the parallel file system corresponding to the global array in row-major order and then reads it back, and checks if the data is consistent with the written data [3]. We scaled the array size between 64^3 and 1024^3 to test the effect of storage throughput. We isolated read and write phases with memory flushing instead of read-after-write used in the original implementation. Figure 6 shows the read and write throughput for both resonant I/O and ROMIO collective I/O. Because the I/O request size is proportional to the array size, as the array size increases the disk becomes very efficient in servicing individual requests, and the benchmark quickly achieves its peak throughput in the system (around 80MB/s). Therefore, while resonant I/O produces higher throughput, the improvements over ROMIO collective I/O are modest.

4.2.2 Benchmark *mpi-io-test*

In the *mpi-io-test* benchmark we used four MPI processes, one on each compute node, to read a 10GB file. Each process reads one segment of contiguous data at a time. In each collective call, four processes read four segments in a row, respectively. In the next call, the next four segments are read. Figure 7 shows the throughput of the benchmark when resonant I/O and ROMIO collective I/O are used. As expected for this benchmark we see an I/O resonance (a spike in I/O throughput) at segment size 64KB. This resonance occurs with resonant I/O for both the read and write versions of the benchmark. Interestingly, we found that the ROMIO collective I/O also exhibits these resonances. Because there is no overlapping of processes' access ranges, ROMIO collective I/O does not re-arrange requests, and executes its I/O as non-collective I/O does. However, for other segment sizes, ROMIO collective I/O allows each I/O node to receive requests from multiple processes, and resonant I/O is able to order request arrivals and substantially increases the throughput by up to 61%. The figures also show that the write bandwidth is higher than read bandwidth when the

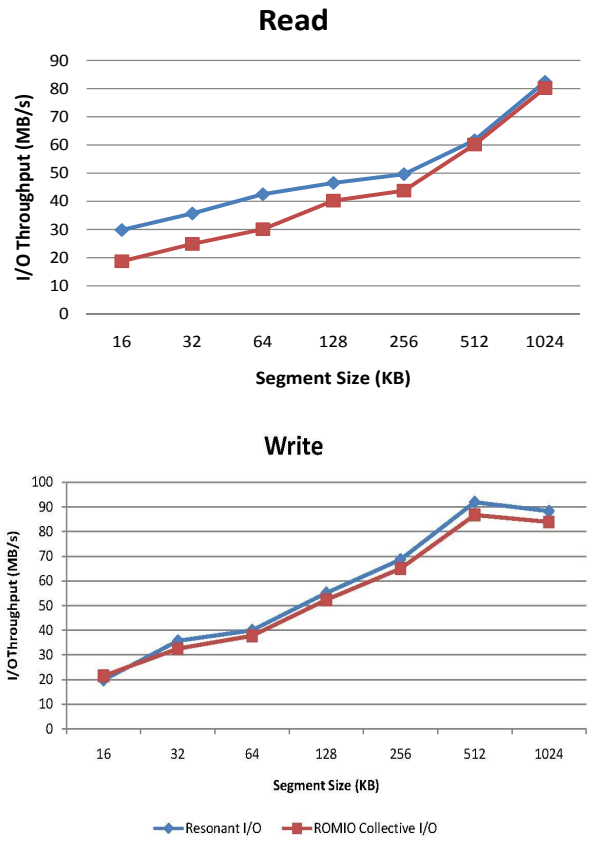

Figure 8. I/O throughput of benchmark *ior-mpi-io* with varying segment sizes.

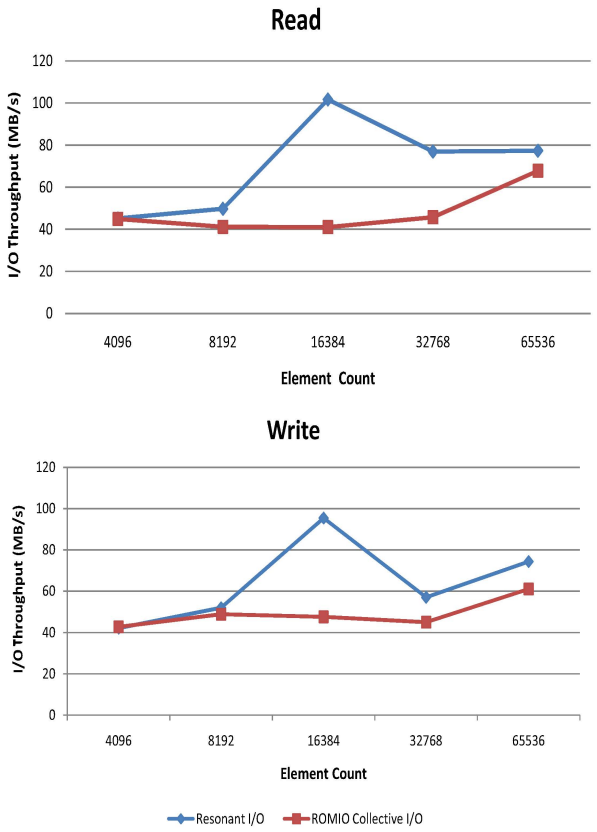

Figure 9. I/O throughput of benchmark *noncontig* with varying segment sizes.

segment size is larger than 64KB; this is mainly due to delayed write-back.

4.2.3 Benchmark *ior-mpi-io*

In benchmark *ior-mpi-io* each of the four MPI processes reads one quarter of a 1GB file: process 0 reads the first quarter, process 1 reads the second quarter, and so on. The reads are executed as a sequence of collective calls. In a call, each of the four processes reads a segment with the same relative offset in their respective access scope, starting at offset 0. Figure 8 shows the throughput with different segment sizes. When the segment size is less than 64KB only one I/O node is involved in servicing requests in each call, so the throughput is very low. The difference is that requests are from one agent process in resonant I/O and from four processes in ROMIO collective I/O, which explains their performance difference in the read version of the benchmark. The performance advantage of resonant I/O diminishes with increasing segment size because increasingly amortized disk seek time reduces the penalty of non-sequential disk access in collective I/O.

4.2.4 Benchmark *noncontig*

Benchmark *noncontig* uses four MPI processes to read a 10GB file using the *vector* derived MPI datatype. If the file is considered to be a two-dimensional array, there are four columns in the array. Each process reads a column of the array, starting at row 0 of its designated column. In each row of a column there are *elmtcount* elements of the *MPI_INT* type, so the width of a column is *elmtcount*sizeof(MPI_INT)*. In each collective call, the total amount of data read by the processes is fixed, determined by the buffer size, which is 16MB in our experiment. Thus the larger *elmtcount* the more small pieces of non-contiguous data are accessed by each process.

When *elmtcount* is small, such as 4096, resonant I/O would need to send requests for a large number of non-contiguous data segments. Because each list I/O can contain at most 64 non-contiguous segments using the default list I/O parameter, multiple list-I/O requests must be made by each agent process. This creates extra overhead for resonant I/O as ROMIO collective I/O uses only four requests. Figure 9 shows that the I/O throughput of resonant I/O is a little lower than that of ROMIO collective I/O when *elmtcount* is

4096. However, when *elmtcount* is increased, resonant I/O yields higher throughput. Both read and write throughput peaks at *elmtcount* of 16K when the segment size equals the striping unit size and all the data requested by an agent process is for itself. For read the peak throughput is 101MB/s, an improvement of 157% over that of ROMIO collective I/O, and for write the peak throughput is 96MB/s, an improvement of 97% over that of ROMIO collective I/O.

4.2.5 Benchmark *HPIO*

The benchmark *HPIO* can generate various data access patterns by changing three parameters: *region_count*, *region_spacing*, and *region_size* [2]. In our experiment, we set *region_count* to 4096, *region_spacing* to 0, and vary *region_size* from 2KB to 64KB. Using four MPI processes, the access pattern is similar to the one described for benchmark *noncontig*. Here the length of a column is fixed as *region_count* (4096) and the width of a column varies from 2KB to 64KB (powers of two times 2KB). Each process reads its designated column with a collective call. Only one collective call is made in the benchmark.

Compared with the 16MB data requested in one collective call in *noncontig*, *HPIO* accesses much more data in one collective call, from 32MB to 1GB. This helps the benchmark to achieve a higher throughput and the high throughput is consistent across different region sizes, as we compare Figures 9 and 10. Resonant I/O provides even higher throughput by rearranging requests to an I/O node, and produces a resonance peak at a region size of 64KB.

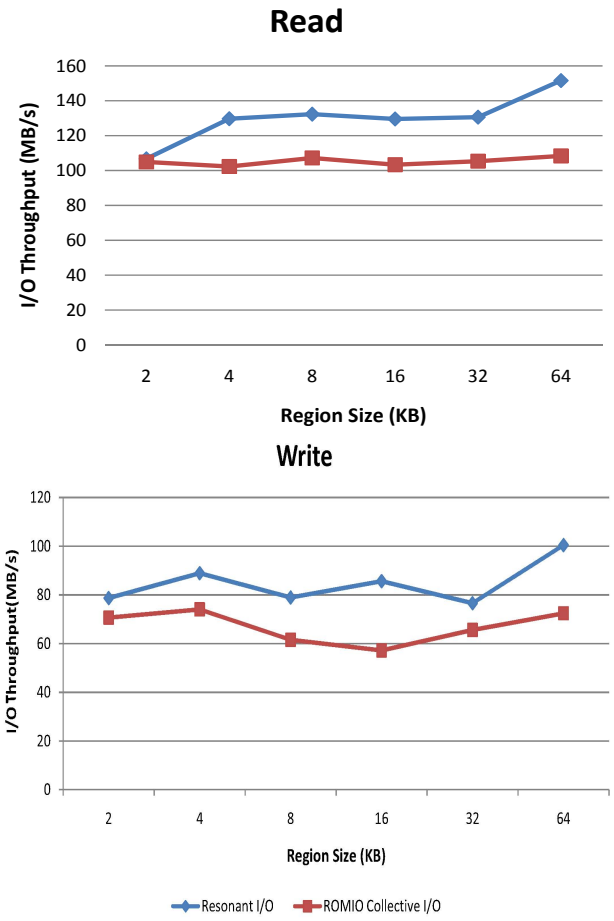

Figure 10. I/O throughput of benchmark *HPIO* with varying segment sizes.

4.3 Resonant I/O Under Interference

In this section we study the impact of interference due to external competing I/O requests on the performance of resonant I/O. For comparison we also show the impact of interference on ROMIO collective I/O. We run two programs, the demonstration program, denoted by *demo*, and *mpi-io-test*, which concurrently access their respective files that are striped over the same set of I/O nodes. We use four parallel processes for each program with 64KB segment size. In this experiment we consider *mpi-io-test* to be the source of interference with *demo*. To control intensity of inteference we insert a period of compute time between two consecutive I/O requests in *mpi-io-test*. Thus the interference intensity is quantitatively represented by the magnitude of the compute time—the smaller the computer time the higher the interference. We also define a metric called *relative throughput* as the ratio of the throughput of the program under interference and the throughput of the program with exclusive access to the same storage system. We measure both absolute throughput and relative throughput of *demo* and *mpi-io-test* with inter-call compute time ranging from 1 second to 0 seconds using resonant I/O and ROMIO collective I/O, respectively (see Figure 11).

For the *demo* program, the relative throughput of resonant I/O drops from 0.90 to 0.43 as the compute time decreases from 1 second to 0. In contrast, the relative throughput of ROMIO collective I/O drops from 0.98 to 0.47. The relative throughput of resonant I/O drops at a greater relative rate, which demonstrates that resonant I/O is more sensitive to interference because sequential request-service order is more difficult to retain with increasingly high interference from concurrently I/O requests. However, even when there is no compute time between two consecutive I/O calls (and so the highest interference intensity), in *mpi-io-test*, resonant I/O still achieves an *absolute* throughput of 48MB/s for *demo*, which is more than twice the throughput of ROMIO collective I/O (22MB/s). Meanwhile, when the interference intensity is the highest, *mpi-io-test* could potentially reduce the throughput of *demo* by at least half. From this perspective, the relative throughput of resonant I/O for *demo*, which is 0.43, can be deemed quite acceptable. This

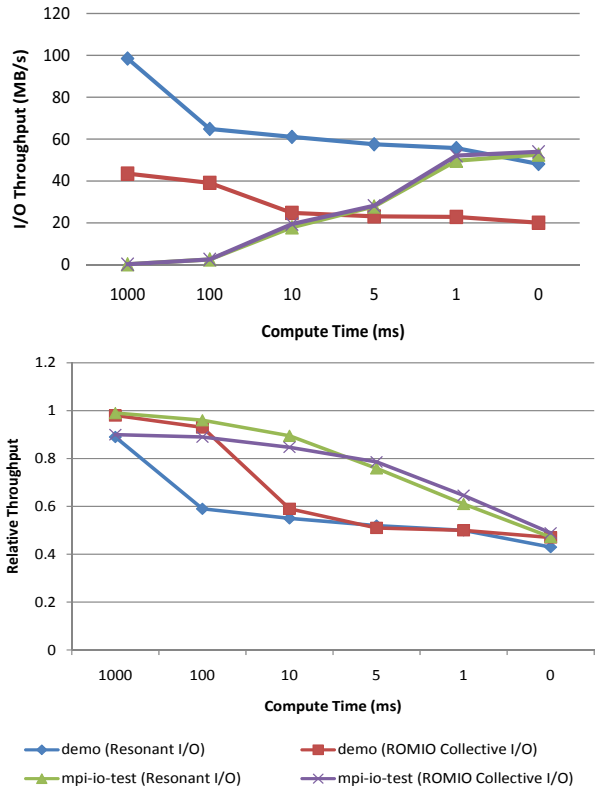

Figure 11. Absolute and relative throughput of resonant I/O and ROMIO collective I/O under different interference intensity, represented by length of the compute time between two consecutive I/O calls in *mpi-io-test*.

result shows that the effort at the application/runtime level to maintain preferred request arrival order still help to improve disk scheduling efficiency even when the competing load on the disk system is high and there are many pending requests for the disk scheduler to reorder.

For *mpi-io-test*, the relative throughput also drops but at a relatively moderate rate with the increase in interference intensity. Higher interference intensity means more I/O time in the program's run time, and the I/O time could be at least doubled when *mpi-io-test* runs concurrently with *demo* in comparison to when it has exclusive use of the I/O subsystem. Here the relative throughput of resonant I/O is slightly higher than that of ROMIO collective I/O. The rising curves representing absolute throughput of *mpi-io-test* are due to its increasing I/O demand as its compute time is reduced.

4.4 Scalability of Resonant I/O

In this section we study the scalability of resonant I/O in a production system environment, the Itanium 2 Cluster at Ohio Supercomputer Center, which has 110 compute nodes and 16 storage nodes, each with 4 GB of memory, running the PVFS2 file system. We ran benchmark *mpi-io-test* with

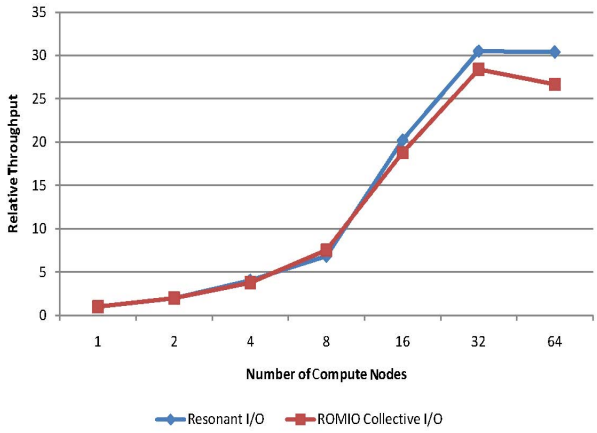

Figure 12. I/O throughput as a function of the number of compute nodes, relative to a single node, for benchmark *mpi-io-test*.

10GB file size and 1MB segment size with different numbers of processes, each on a different processor, to a maximum of 64. Figure 12 shows I/O throughput as a function of the number of compute nodes, relative to the throughput achieved on a single node, for benchmark *mpi-io-test*, for both resonant I/O and ROMIO collective I/O. As shown, resonant I/O is as scalable as ROMIO collective I/O. Because the quantity of data requested in a collective-I/O call is proportional to the number of processes, the I/O throughput increases with the number of processes to the limit of the storage system at 32 processes. When the performance of the storage system becomes a bottleneck, efficient use of the disk-based system becomes critical, which explains the performance advantage of resonant I/O over the ROMIO collective I/O when the number of processes is larger than 32. In general, both resonant I/O and ROMIO collective I/O scale well in our experiment. In addition, we note that the program shared the I/O nodes with other concurrently running programs during its execution. As the measurements show, the concurrent I/O requests from other programs do not negate the effects of resonant I/O arranging a preferred access order for a higher I/O throughput. This is because the requests belonging to a collective I/O, implemented as resonant I/O, still arrive at the I/O system in a bursty fashion and so retain their preferred order.

5 Conclusions and Future Work

We have proposed, designed, and implemented a new collective I/O scheme, *resonant I/O*, that makes resonance—a phenomenon describing the increase in performance when there is a match between request patterns and data striping patterns—a common case. Resonant I/O makes the client-side implementation of collective I/O aware of the I/O configuration in its rearrangement of re-

quests without compromising the portability of client-side middleware and the flexibility of server-side configuration. Our experimental results show significant increases—up to 157%—in I/O throughput for commonly used parallel I/O benchmarks. Resonant I/O demonstrated advantages both at scale, and in the presence of competition for I/O services. Finally, resonant I/O has not been observed to substantially degrade performance (relative to ROMIO collective I/O) in any test scenario.

While our experiments have shown that resonant I/O is a promising technique for the alleviation of the increasingly serious I/O bottleneck in high-performance computing, there are some limitations in its implementation and evaluation that will be addressed in future work. First, we will use asynchronous I/O to fully exploit the performance potential of resonant I/O. As current ADIO does not support real asynchronous I/O, we will use additional threads to implement asynchronous I/O. Second, the dedicated cluster used in our experiments is of relatively small scale, and the larger cluster was not available for dedicated use. Our plan includes evaluating resonant I/O on a larger dedicated cluster to obtain more insight into its performance characteristics. Third, we plan to implement resonant I/O on top of other state-of-the-art file systems, including Lustre, to further evaluate its potential.

6 Acknowledgments

This work was supported by the National Science Foundation under grant CCF-0702500. This work was also funded in part by the Accelerated Strategic Computing program of the 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. The authors also thank Dr. Xiaodong Zhang and the Ohio Supercomputer Center for providing computing resources.

References

[1] Argonne National Laborarory. URL *http://www.pvfs.org/*, 2008.

[2] A. Ching, A. Choudhary, W. Liao, L. Ward, and N. Pundit. Evaluating I/O Characteristics and Methods for Storing Structured Scientific Data. In *Proc. of IPDPS '96,Honolulu, Hawaii, April, 1996.*

[3] Argonne National Laboratory. MPICH2. URL *http://www.mcs.anl.gov/research/projects/mpich2/*, 2008.

[4] S. Baylor and C. Wu. Parallel I/O workload characteristics using Vesta. In *Proc. of IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 16-29, Apr. 1995.

[5] P. Crandall, R. Aydt, A. Chien and D. Reed. Input/output characteristics of scalable parallel applications. In *Proc. of Supercomputing '95*, Dec. 1995.

[6] A. Ching, A. Choudhary, K. Coloma, and W. Liao. Noncontiguous I/O Accesses Through MPI-IO. In *Proc. of the IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2003.

[7] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp. Efficient Structured Data Access in Parallel File Systems. In *Proc. of the IEEE International Conference on Cluster Computing*, Dec. 2003.

[8] Cluster File Systems, Inc. Lustre: A scalable, robust, highly-available cluster file system. URL *http://www.lustre.org/*, 2006.

[9] HPIO I/O Benchmark. URL *http://cholera.ece.northwestern.edu/ aching/research-webpage/io.html*, 2008.

[10] Interleaved or Random (IOR) benchmarks. URL *http://www.cs.dartmouth.edu/pario/examples.html*, 2008.

[11] J. Ilroy, C. Randriamaro and Gil Utard. Improving MPI-I/O Performance on PVFS. In *Proc. Euro-Par'01*, 2004.

[12] D. Kotz. Disk-directed I/O for MIMD Multiprocessors. In *ACM TOCS*, Feb. 1997.

[13] R. Latham, N. Miller, R. Ross, and P. Carns. A Next Generation Parallel File System for Linux Clusters. In *LinuxWorld Magazine*, 1, 2004.

[14] T. Latham, W. Gropp, R. Ross, and R. Thakur. Extending the MPI-2 Generalized Request Interface. In *Proc. of the 14th European PVM/MPI Users' Group Meeting*, Sept. 2007.

[15] S. Liang, S. Jiang, and X. Zhang. STEP: Sequentiality and Thrashing Detection Based Prefetching to Improve Performance of Networked Storage Servers. In *Proc. of ICDCS07*, June. 2007.

[16] P. Lu and K. Shen. Multi-layer Event Trace Analysis for Parallel I/O Performance Tuning. In *In Proc. of ICPP '07*, 2007.

[17] J. May. Parallel I/O for High Performance Computing. In *Page ,1 edition, October, 2000. Morgan Kaufmann Press.*

[18] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis and M. Best. File-access characteristics of parallel scientific workloads. In *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075-1089, October 1996.

[19] Noncontig I/O Benchmark. URL *http://www-unix.mcs.anl.gov/thakur/pio-benchmarks.html*, 2008.

[20] Parallel I/O Benchmarking Consortium URL *http://www-unix.mcs.anl.gov/pio-benchmark/*, 2008.

[21] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of the USENIX FAST 2002 Conference*, 2002.

[22] P. Schwan. Lustre: Building a file system for 1000-node clusters. URL *http://off.net/cv/papers/ols2003.ps*, 2003.

[23] K. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proc. of Supercomputing '95*, Dec. 1995.

[24] Sun Microsystems. Lustre 1.6 Operations Manual. URL *http://docs.sun.com/app/docs/doc/820-3681*, 2008.

[25] R. Thakur, W. Gropp and E. Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proc. of the 6th Symposium on the Frontiers of massively Parallel Computation*, Oct. 1996.

[26] R. Thakur, W. Gropp and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *Proc. of the 7th Symposium on the Frontiers of massively Parallel Computation*, Dec. 1995.

[27] R. Thakur, W. Gropp and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *In Proc. of IOPADS '99*, 1999.