Non-Contiguous I/O Support for Object-Based Storage

To appear in the Proceedings of ICPP'08, P2S2 Workshop, Portland, OR, September 2008.

Dennis Dalessandro
Ohio Supercomputer Center
1224 Kinnear Rd
Columbus, OH 43212
dennis@osc.edu

Ananth Devulapalli
Ohio Supercomputer Center
1224 Kinnear Rd
Columbus, OH 43212
ananth@osc.edu

Pete Wyckoff
Ohio Supercomputer Center
1224 Kinnear Rd
Columbus, OH 43212
pw@osc.edu

Abstract

The access patterns performed by disk-intensive applications vary widely, from simple contiguous reads or writes through an entire file to completely unpredictable random access. Often, applications will be able to access multiple disconnected sections of a file in a single operation. Application programming interfaces such as POSIX and MPI encourage the use of non-contiguous access with calls that process I/O vectors.

Under the level of the programming interface, most storage protocols do not implement I/O vector operations (also known as scatter/gather). These protocols, including NFSv3 and block-based SCSI devices, must instead issue multiple independent operations to complete the single I/O vector operation specified by the application, at a cost of a much slower overall transfer time.

Scatter/gather I/O is critical to the performance of many parallel applications, hence protocols designed for this area do tend to support I/O vectors. Parallel Virtual File System (PVFS) in particular does so; however, a recent specification for object-based storage devices (OSD) does not.

Using a software implementation of an OSD as storage devices in a Parallel Virtual File System (PVFS) framework, we show the advantages of providing direct support for non-contiguous data transfers. We also implement the feature in OSDs in a way that is both efficient for performance and appropriate for inclusion in future specification documents.

1 Introduction

As high performance computing has transitioned from large shared memory machines to the distributed cluster model commonly used today, new challenges have presented themselves. Internetworking, process communication, and distributed shared memory are some of the most explored areas. Another is the issue of non-contiguous I/O. The need for accessing data that is not sequential in a file has long been known [10]. There has also been much work [2, 16] involving parallel file systems and non-contiguous data access.

However, one area that does not address the need for efficient non-contiguous I/O is in object based storage, also known as OSD. Object based storage is the potential next step in storage device technology. Data is no longer stored as a stream of bytes. Rather data is represented as objects. The object view of data is not new [4, 12, 8] at the file system level; however, an OSD is not a file system level component. An OSD is the actual storage medium. Managing data as objects at the device level is an abstraction that allows for powerful layered semantics.

An object view of data is not the only contribution of OSDs—one of the major performance benefits of using OSDs is their capability to handle the decision of how to lay out data on the disk. Previously this task was the responsibility of the file system. Offloading this work to the storage devices means freeing up more resources on the file system host.

Since OSDs are relatively new, there do not exist any available disks on the commodity market, although storage vendors are experimenting with the technology. In order to facilitate research into OSDs we have previously implemented a software object-based storage device [6], and integrated it with the popular parallel file system PVFS [1]. Using our OSD system we are able to explore further aspects of using OSDs in high performance computing (HPC) environments.

In HPC environments, many applications work cooperatively to solve complex problems. Data generated as well as used for input is often accessed in parallel, in non-contiguous chunks. The *de facto* standard for parallel application programming in high performance computing environments is MPI. Applications are able to take advantage of efficient data access, including non-contiguous access by making use of the I/O component to MPI known as MPI-I/O [14]. The underlying file system needs to support non-contiguous I/O as well, and

when implemented on PVFS using block based storage, it does. However this is not true for many file systems (including NFS), nor is it currently true for object-based storage devices.

As it stands now, the OSD specification [15] does not provide for clients to specify a description of a non-contiguous I/O operation. This means that each read or write must be broken into many smaller operations when the storage medium is an OSD. Since these likely involve messages being exchanged over an interconnect, and each message requires some amount of header data and processing the performance penalty can be quite large. This is especially true in high latency environments like wide-area networks. In order to support real applications running on parallel file systems, native non-contiguous I/O access is required.

As part of our ongoing work with object-based storage, we have drafted and submitted a proposal to SNIA [13], the organizing group for the OSD specification. Based on their feedback, we expect non-contiguous I/O support to be included in the next version of the specification. The rest of this paper shows the performance benefit of adding non-contiguous I/O support to object-based storage.

2 Background

We now provide technical background information on a few topics which are central to the themes in this paper.

2.1 Non-contiguous I/O

Non-contiguous I/O can be defined as any I/O operation which attempts to access data in a way other than how it is logically stored. Data can be both contiguous or non-contiguous in memory as well as in the file. In other words how data is laid out in memory and how it is represented in the file are orthogonal concepts. Non-contiguous I/O operations are referred to as either scatter or gather, depending on the direction of the data flow. Scatter happens when data is written to multiple locations, while gather retrieves data from multiple locations. Figures 1 and 2 show examples of how data blocks can be arranged in memory and in the file. In Figure 1, data exists in a single region of memory and is scattered across the file when written. In Figure 2, data is non-contiguous both in client memory and in the file. The gather of these figures would simply reverse the way the arrows point to indicate a read from the file.

Often times parallel jobs whose processes are cooperating share the same file, but access interleaved byte ranges. This is common practice in applications ranging from simply storing a two-dimensional matrix, to extremely complex database implementations. The common theme is the decomposition of data across multiple processes, resulting in non-contiguous reads and writes when individual processes go to access their portions of the overall data. Utilizing non-contiguous I/O can often yield performance improvements in these cases. MPI Blast is one such application that is in this category [11].

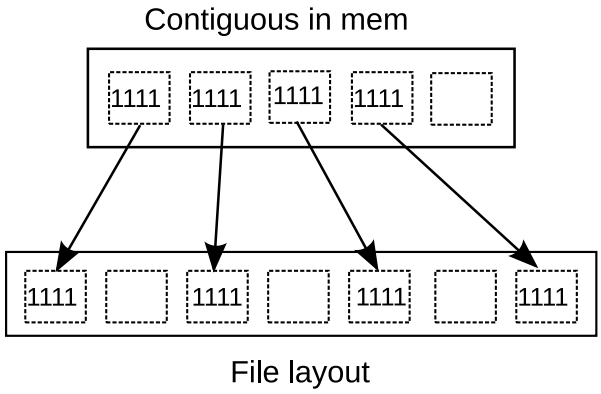

Figure 1. Write from contiguous memory buffer to non-contiguous file.

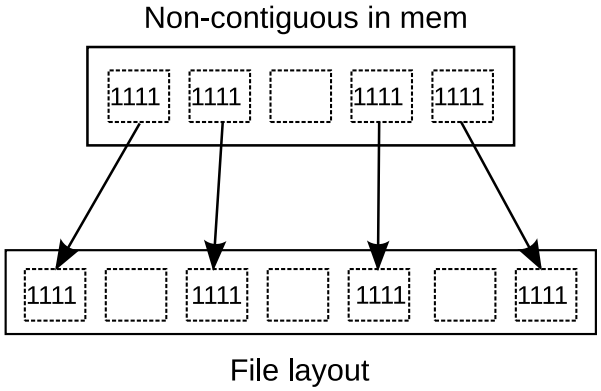

Figure 2. Write from non-contiguous memory buffer to non-contiguous file.

2.2 PVFS

The Parallel Virtual File System (PVFS) [1], is a popular parallel file system used on computing clusters today especially for large scale MPI-based jobs. PVFS, like other parallel file systems, such as Lustre [4], segregate data and metadata operations, in hopes of achieving higher throughput for data. The basic architecture of PVFS includes clients, metadata servers, and I/O servers as shown in Figure 3. While there is only one metadata server shown in this figure, it is possible, and commonplace, to have multiple metadata servers.

In PVFS it is also possible for the same node to be used in multiple roles. In general when a client wants to access data it first queries the metadata servers. The

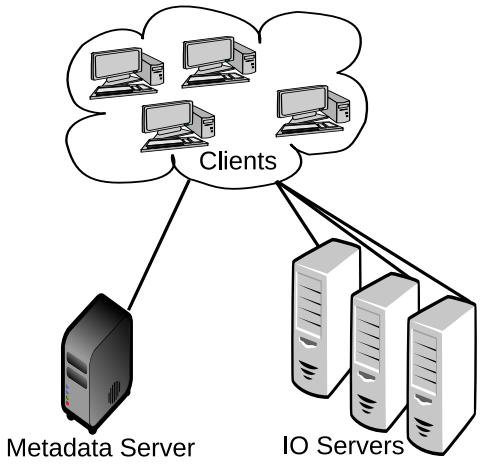

Figure 3. General architecture of a PVFS system.

metadata servers convey the necessary information to find the data on the I/O servers. The client then retrieves data from the I/O servers, or sends the data in the event of a write. Typically a file is striped across multiple I/O servers. This enables multiple network reads/writes to proceed in parallel.

2.3 Object-based Storage

Object-based storage, as the name implies, treats data as objects which have meaningful attributes, rather than as unrelated blocks of data. There are file systems which have aspects of object-based storage such as Lustre [4], but these still rely on normal block devices to actually store the data. Thus OSDs should not be viewed as an alternative to such file systems, rather OSDs are an alternative to the normal block devices that those object-based file systems use.

One of the primary differences between traditional block-based storage and object-based storage is the device's role in determining data layout. Block-based devices rely upon the file system input for data placement, layout and retrieval. Contrary to this, OSDs determine where to put data and how to lay it out; the file system need not involve itself with this aspect. OSDs are able to be more intelligent about data as objects have the important notion of attributes. The OSD specification [15] defines a set of mandatory attributes that go along with each object. These include size, modification time, etc. The power of attributes is further extended by OSD's support for user-defined attributes and powerful semantics to select objects based on attribute values.

Clients interact with OSDs via ordinary SCSI commands. Each command is not only capable of accessing the object's data, it can also set and retrieve attributes on that object as well. Like all SCSI systems, an OSD system is made up of a target and an initiator. These terms are synonymous with the terms server and client (respectively) in networking. Since OSDs are part of the SCSI framework, it is possible to encapsulate commands and ship them over any SCSI network, including iSCSI for use on TCP/IP networks.

2.4 OSD Software Implementation

In previous work [6], we implemented an OSD in software, including the target and libraries to support initiators. In our initiator we make use of the in kernel iSCSI infrastructure. We provide a user space library that handles OSD requests, and calls into the kernel to ship the SCSI commands over the network via iSCSI. As part of the work, we integrated PVFS to use OSD devices directly, rather than sending all data through servers. Not only does this give us familiar territory as far as the file system is concerned, but enables us to use MPI applications out of the box and run them with our OSD system. The OSD initiator stack is shown in Figure 4.

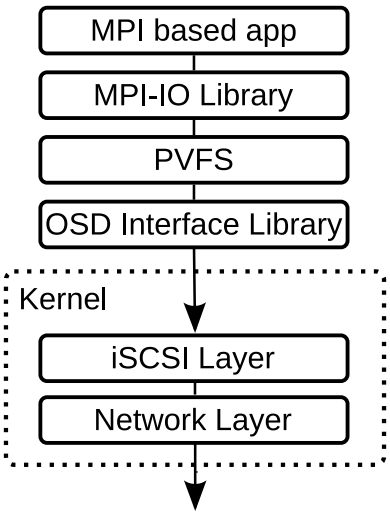

Figure 4. Initiator software stack.

The target, unlike the initiator, can not rely on an in-kernel iSCSI stack as stock Linux does not include one and supporting and adapting an out-of-tree SCSI target is difficult and fruitless. Thus we decided to modify tgt [7], an open source user-space iSCSI target implementation. We use tgt to handle the processing of iSCSI command descriptor blocks and to send responses back to the initiator. We have an OSD command processing layer that fulfills the requests. In order to manage and keep track of attributes on objects, we utilize the light weight embedded database, SQLite [9]; however, this does add some overhead. A related work [5] explores the attribute design and database involvement in more detail.

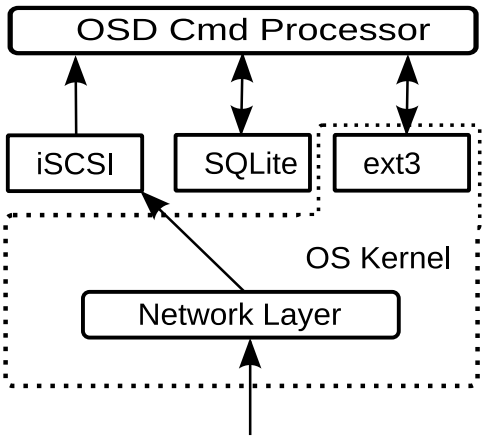

Figure 5. Target software stack.

Figure 5 shows the software stack for the target. A message comes in from the network and gets handled by the iSCSI (tgt) layer. If it is an OSD command, the request is handled by the OSD command processor which will utilize an SQLite database to handle attribute management. Finally the I/O operations are done against the disk using the existing file system interface. As this implementation relies on existing media, one cannot expect to see performance gains over ordinary block devices for simple transfers. The goal instead is to use the software implementation as a research vehicle for OSDs and to guide embedded hardware development in the future.

3 OSD Support for Non-contiguous I/O

The OSD specification [15] does not provide for non-contiguous I/O operations. In other words there is no mechanism for an initiator to supply a list of file offsets and sizes to access multiple sections of data in a single operation. However, for many applications non-contiguous I/O is commonplace. This could be a severe detriment to performance of such applications, especially if the individual data sections are small, because each section requires a full network request and response pair. Even on low latency networks this can prove to be costly.

3.1 Non-contiguous Support Options

There are a number of options we can choose to support in terms of non-contiguous data access. The most simple case, and how things are implemented currently, is to not support non-contiguous access at all. When a client asks to do a non-contiguous data access it will be broken up into multiple contiguous chunks. However, as outlined above this can have severe performance penalties.

Clearly there is a need for some sort of non-contiguous support. The most basic type of non-contiguous support would be to allow initiators to make data requests by specifying a list of offsets and lengths. The advantage is in the simplistic implementation, but the disadvantage is the inefficiency of the request format and its inability to compactly describe large number of regular strided accesses or accesses involving closely segregated regions.

It is also possible to optimize the structure of the request to support regular access patterns of a given type. For instance taking advantage of the derived data-type mechanisms in MPI and PVFS. This is probably the most optimal, but becomes application specific in that applications need to be aware of the ways to format the request and format it appropriately. This also may not be easily implemented in an embedded device.

3.2 Types of Non-contiguous Access

We identify three important patterns of non-contiguous access, called data distribution types (DDT). The first is the scatter/gather list, or SGL, that is familiar from the long standing Unix I/O vector interface. In SGL we specify each segment to read or write by including in a list the offset and the length of the segment from that offset. There is no need for offsets to appear in any particular order, and segments may overlap. The offset and length values are each 8 bytes, matching the values used elsewhere in OSD to specify offsets and lengths. There is also an 8-byte count to specify the total number of segments. Figure 6 shows an SGL header.

Byte	Description
0	Count
8	Offset
16	Length
24	Offset
32	Length
...	...
$8 + 16 \times$ Count	Data

Figure 6. SGL DDT header.

While a scatter/gather list can condense multiple data segment accesses into a single operation, it can also result in a lot of overhead, especially if the byte ranges are extremely small. For instance accessing every other byte would require a scatter gather list that has 16 bytes of overhead for every byte accessed plus an additional 8 bytes for the count.

As a slight modification to the above, if the access pattern is regular, we can use what is called strided I/O. This is one optimization we can make without imposing restrictions on the application. In other words an OSD initiator can make the decision to use strided I/O without being told to do so by the application. It can figure

this out on its own by a simple examination of the user request. In strided I/O, only the stride and length, each of which is an 8-byte field, are specified. The stride is defined to be the distance from the start of one block to the start of another. The initial offset for the operation is specified at the start as a part of standard OSD I/O command. A strided DDT header is shown in Figure 7. Regular strided access occurs in many applications that operate on data that consists of a series of fixed-size chunks. If accesses can be represented using the strided form, the data description can be very compact. Using the every-other-byte example above, this would result in a total of 16 bytes of overhead, regardless of the number of segments.

Byte	Description
0	Stride
8	Length
16	Data

Figure 7. Strided DDT header.

3.3 OSD Implementation

In order to convey this information to the target, the initiator makes use of the data-out buffer of the Command Descriptor Block (CDB). The specification [15] provides an option byte, of which we utilize two bits as follows:

- 00 = Contiguous
- 01 = Scatter/Gather List
- 10 = Strided
- 11 = Reserved for future use

This means that the DDT shares the same buffer as the data being sent to the target. We chose to overload this buffer and specify the DDT at the beginning of the data-out buffer. There are however other possible implementations.

For instance, it is possible to tack on an extra data segment at the end of the data-out buffer. However we chose not to do this because it would require the target to buffer the incoming data in order to get to the DDT to figure out which data to access. If it is a large data access this could prove troublesome, and may be limited in size on embedded devices.

Another alternative would be to specify the DDT as an attribute that gets set on the data operation. The issue with this approach is the way in which OSDs process data and attribute requests. First the data is read from the buffer then attributes are get and set, so again this approach would require buffering the incoming data until the DDT can be figured out from the attributes.

Finally it is also possible to use a separate set attribute operation to specify the DDT, before doing the read or write. Which would be good if the DDT only needs to be set once and can be reused. If each read or write requires a different DDT then an extra command is required to set it which includes an extra network round trip. Moreover this assumes existence of shared OSD attributes, which are not yet accepted by the OSD standard.

3.4 CDB Fields

In supporting non-contiguous I/O, we must clarify the meaning of two important CDB fields: the offset, and data-out length. For the SGL data distribution, the CDB data-out length will be the sum of the data being written and the size of the SGL headers. The strided DDT will have a CDB data-out length field that is the sum of the size of the data being accessed and the 16 byte overhead.

$$\begin{aligned} \textit{SGL Data-out length} &= 8 + 2 \times 8 \times NS \times DS \\ \textit{Strided Data-out length} &= 8 + 8 + DS \end{aligned}$$

The above equations show the corresponding relationships. Here NS stands for number of number of segments and DS for data size. For a read operation in the SGL or strided case the data size will be 0.

The other important CDB field to address is the offset value. The offset in the CDB will be treated as a master offset and applied to all data accesses. Thus for the SGL DDT the first byte accessed will be at CDB offset + SGL offset. Similarly in the strided DDT case, data starts at CDB offset.

4 Experimental Results

We now turn our attention to experimental results that show that non-contiguous access is achievable with OSDs and that there is an increase in performance. These experiments were conducted using our software OSD implementation [6]. The experimental platform consists of a Linux cluster running a 2.6.24 kernel. Each compute node is outfitted with dual Opteron 250 processors and 2 GB of RAM. The disks in use are 80 GB SATA disks, locally attached.

4.1 Performance Model

In order to reason about expected performance and to understand the trade-offs between the different DDTs we offer the following model. Where:

$$\begin{aligned} n &= \textit{number of segments} \\ x &= \textit{segment size} \end{aligned}$$

The overhead due to the particular data distribution type is:

$$\begin{aligned} O_{iterative} &= 0 \\ O_{sgl} &= 16 \times n + 16 \\ O_{strided} &= x \times n + 16 \end{aligned}$$

The amount of data to be transmitted includes the payload and this overhead:

$$\begin{aligned} S_{iterative} &= x \times n \\ S_{sgl} &= x \times n + 16 \times n + 16 \\ S_{strided} &= x \times n + 16 \end{aligned}$$

All formats require at least one round trip, and the iterative (basic) approach requires a round trip per segment. We assume an RTT of 30 μs:

$$\begin{aligned} N_{iterative} &= n \times RTT \\ N_{sgl} &= RTT \\ N_{strided} &= RTT \end{aligned}$$

Finally, the total send time can be expressed, using a network bandwidth (BW) of 800 Mb/s, which is a reasonable throughput on gigabit Ethernet.

$$\begin{aligned} T &= S/BW + N \\ T_{iterative} &= (x \times n)/BW + n \times RTT \\ T_{sgl} &= (x \times n + 16 \times n + 15)/BW + RTT \\ T_{strided} &= (x \times n + 16)/BW + 30 \end{aligned}$$

This model at face value is for write performance. However it can be used to model read performance as well. When reading the same number and size of data transfers happen, there is simply an extra message sent for the read request to start things off. This means a constant equal to the RTT/2 would be added to each case.

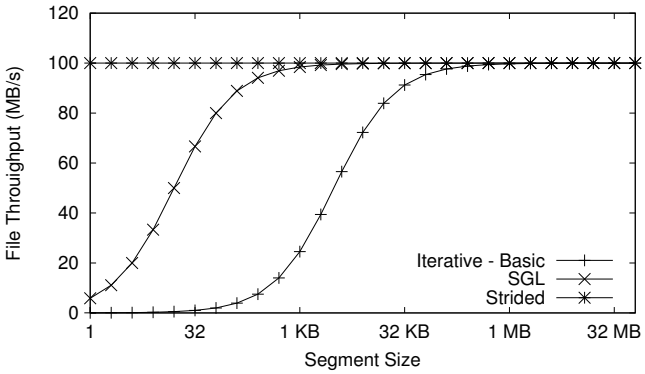

Figure 8. Write performance model.

Figure 8 shows the result of simulating the model for writing a 100 MB file with varying segment sizes. We see that clearly the strided case is best, regardless of the segment size. However strided is a special case of SGL and not always possible. As segment size gets larger throughput gets bigger. The reason is because the amount of data being sent into the network is sufficient enough to keep the network full.

4.2 Basic Performance Results

The following experiments were performed using PVFS configurations consisting of a single I/O server and a single metadata server. Since this experiment is data intensive, the single metadata server will not be a bottleneck. There is nothing to be gained from these experiments by running with multiple I/O servers. All trends will remain the same, overall performance will simply increase with the rate of parallelism.

There are two configurations shown in the following graphs. Write measures how long it takes to get data to the file server. This does not take into account waiting for data to be flushed to disk. The other configuration is read, where data actually comes from the disk.

Figure 9 shows throughput for a contiguous file of varying size. Clearly the performance is about the same for all three DDTs, as is expected. Basically for these figures there is a single write for the full file, up to 400 kB. Above 400 kB we break single operations into multiple operations due to kernel iSCSI buffer restrictions.

Next we consider non-contiguous file accesses. The file in question has a view set up by an MPI-I/O vector type that writes blocks of varying size then leaves a two byte hole before writing the next block. Figure 10 shows the write, and read throughput for this non-contiguous file access. There is some penalty paid for supporting non-contiguous access on the low end, but as the graphs indicate our new non-contiguous codebase greatly improves OSD performance. This figure differs from the previous in that we are now looking at segment size, of which multiple are sent at once. It is important to note that we may be sending 1 MB of data, but the actual size of the file on the other end will be larger, to account for the two byte holes between each segment. This is the likely cause of the slight dip in performance seen for strided I/O, and the reason the others do not reach slightly higher throughput.

4.3 Parallel Sequence-Search Benchmark

The largest application field for computational clusters, as of the spring 2008 IDC report at the HPC Forum, is bio-sciences. One of the basic tasks that computational biologists perform is sequence searching. This amounts to searching a database containing DNA sequences for certain genes or proteins. What this means to computer scientists is searching for approxi-

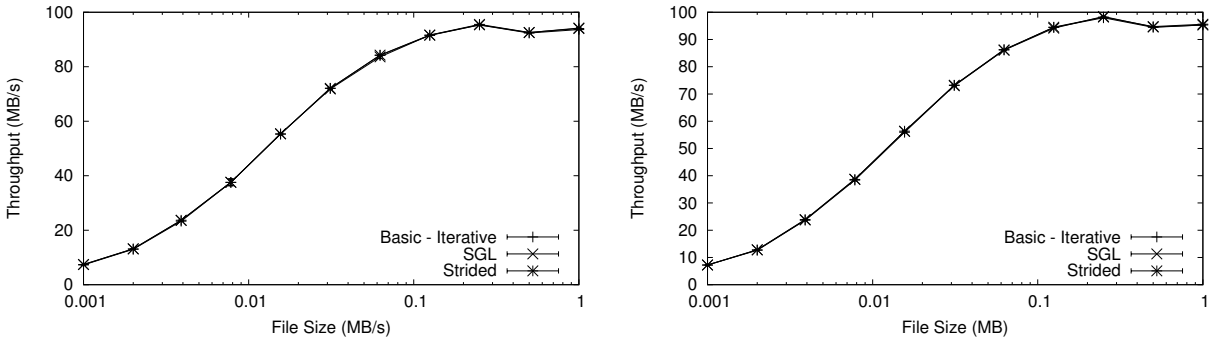

Figure 9. I/O performance with contiguous file view, left: Write; right: Read.

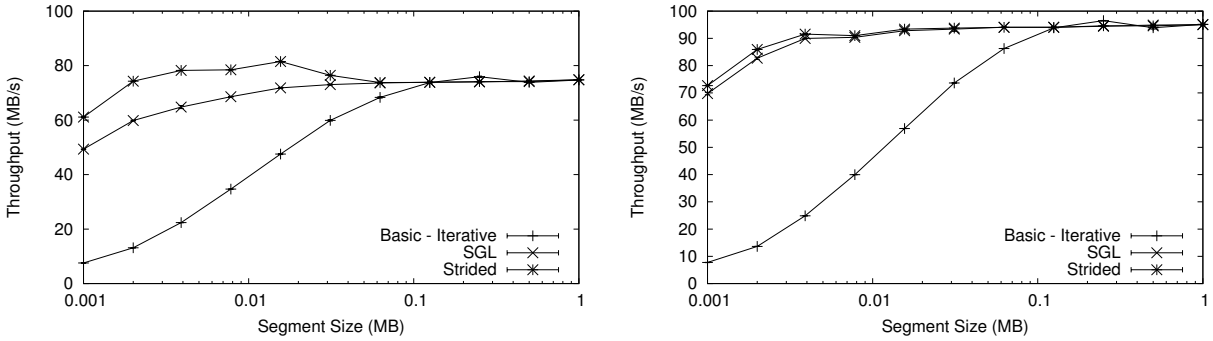

Figure 10. I/O performance with non-contiguous file view, left: Write; right: Read.

mate matches of given substrings in a large file full of A, C, T and G characters. While the computation time currently overshadows I/O time in most applications, growing data sets are making higher demands to I/O performance. In order to explore the costs of I/O under various conditions, the S3aSim tool has been developed [3]. We use this code to show the importance of supporting non-contiguous file access.

Figure 11 shows S3aSim results for one particular configuration. Here we used a PVFS file system consisting of four object-based storage devices as data storage elements and four metadata servers. The number of clients was fixed at 15: one master and 14 worker processes. The time to complete the search queries on a database of a given size is shown in Figure 11. The number of queries as well as the number of results is fixed for all cases. There is a clear difference between supporting non-contiguous access and using the naive or basic method. The reason that the time decreases as database size increases is due to the fact that at small database sizes, there are a large number of queries resulting in a large number of small results, which effectively result in large number of small writes. As the database size increases, with the number of queries and results fixed, the results are simply bigger. This yields the same number of writes, only larger in size, increasing overall throughput.

The impact of supporting non-contiguous data transfers can be clearly seen in this application, which was not modified. It simply sees the advantage of the more capable data transfer mechanism.

5 Future Work

We are pursuing a number of related aspects in using OSDs in parallel file systems, such as extensions for atomic operations and directory support. We will be reworking our current non-contiguous implementation to follow the specification as it evolves, possibly using attributes to specify non-contiguous regions of a file to access.

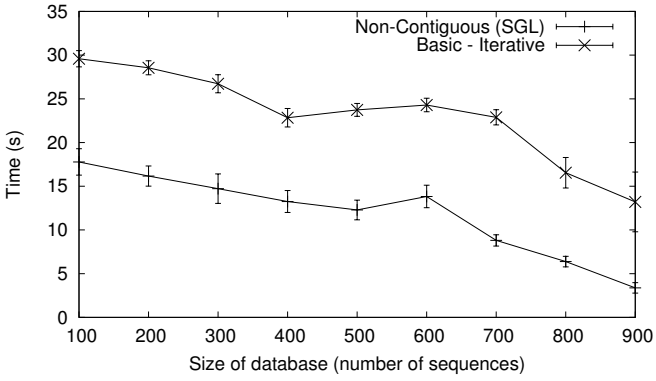

Figure 11. Parallel Sequence-Search Benchmark Example.

6 Conclusion

As most storage protocols do not implement vectored I/O, or scatter/gather operations, this is left up to the application level to handle. These types of operations are critical to the performance of many parallel applications. Some protocols used in high-performance computing do support I/O vectors, including PVFS, but many do not. For the evolving object-based storage approach, there is currently no vectored I/O support. Implementations must resort to making multiple requests for data access that are logically only a single operation.

In this paper we have shown the need for object-based storage devices to support non-contiguous I/O. Using a software implementation of an OSD in a PVFS file system, we show the performance gains by supporting such operations. Our non-contiguous I/O scheme is both efficient and appropriate for inclusion in future OSD specifications.

References

[1] P. H. Carns et al. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, 2000.

[2] A. Ching, A. Choudhary, W. Liao, R. Ross, and W. Gropp. Noncontiguous I/O through PVFS. In *Proceedings of the 2002 IEEE International Confernece on Cluster Computing*, 2002.

[3] A. Ching, W. Feng, H. Lin, X. Ma, and A. Choudhary. Exploring I/O strategies for parallel sequence-search tools with S3aSim. In *Proceedings of the International Symposium on High Performance Distributed Computing (HPDC'06)*, June 2006.

[4] Cluster File Systems, Inc. Lustre: a scalable high-performance file system. Technical report, Cluster File Systems, Nov. 2002.

[5] A. Devulapalli, D. Dalessandro, N. Ali, and P. Wyckoff. Attribute storage design for object-based storage devices. In *MSST'07*, San Diego, CA, Sept. 2007.

[6] A. Devulapalli, D. Dalessandro, P. Wyckoff, N. Ali, and P. Sadayappan. Integrating parallel file systems with object-based storage devices. In *Proceedings of SC'07, to appear*, Reno, NV, Nov. 2007.

[7] T. Fujita and M. Christie. tgt: framework for storage target drivers. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, July 2006.

[8] B. Halevy, B. Welch, J. Zelenka, and T. Pisek. Object-based pNFS Operations. Technical Report draft-ietf-nfsv4-pnfs-obj-00.txt, IETF, Jan. 2006.

[9] D. R. Hipp et al. SQLite. http://www.sqlite.org/, 2007.

[10] D. Kotz and N. Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 640–649, New York, NY, USA, 1994. ACM.

[11] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova. Efficient data access for parallel BLAST. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Mar. 2005.

[12] D. Nagle et al. The Panasas ActiveScale storage cluster—delivering scalable high bandwidth storage. In *Proceedings of SC'04*, Pittsburgh, PA, Nov. 2004.

[13] Storage Networking Industry Association. SNIA: Advancing storage & information technology. http://www.snia.org.

[14] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32, New York, NY, USA, 1999. ACM Press.

[15] R. O. Weber. Information technology—SCSI object-based storage device commands -2 (OSD-2), revision 3. Technical report, INCITS Technical Committee T10/1729-D, Jan. 2008.

[16] J. Wu, P. Wyckoff, and D. Panda. Supporting efficient noncontiguous access in PVFS over Infiniband. In *Proceedings of the 2003 IEEE International Conference on Cluster Computing*, 2003.