Bridging The Gap Between Parallel File Systems and Local File Systems: A Case Study with PVFS

Peng Gu, Jun Wang
University of Central Florida
{penggu,jwang}@cs.ucf.edu

Robert Ross
Argonne National Laboratory
rross@mcs.anl.gov

Abstract

Parallel I/O plays an increasingly important role in today's data intensive computing applications. While much attention has been paid to parallel read performance, most of this work has focused on the parallel file system, middleware, or application layers, ignoring the potential for improvement through more effective use of local storage. In this paper, we present the design and implementation of Segment-structured On-disk data Grouping and Prefetching (SOGP), a technique that leverages additional local storage to boost the local data read performance for parallel file systems, especially for those applications with partially overlapped access patterns. Parallel Virtual File System (PVFS) is chosen as an example. Our experiments show that an SOGP-enhanced PVFS prototype system can outperform a traditional Linux-Ext3-based PVFS for many applications and benchmarks, in some tests by as much as 230% in terms of I/O bandwidth.

1 Introduction

Recent years have seen growing research activities in various parallel file systems, such as Lustre [3], IBM's GPFS [19], Ceph [24], the Panasas PanFS File System [17], and PVFS [8]. In many cases, parallel file systems use a local file system or object store to serve as a local data repository. Because of the interfaces used to access these local resources, local storage systems are unaware of the behavior of high-level parallel applications that talk directly to parallel file systems. Likewise, because of interface limitations the parallel file system itself is not aware of the underlying local storage organization and operation. In other words, there is an information gap between local file system and parallel file system, and as a result access locality from applications often gets lost. Specifically, the local storage system can only see accesses to separate pieces of large parallel files. Emerging Object Storage Device (OSD) interfaces, used by parallel file systems [3, 24, 17], do appear to present a more appropriate interface for local storage resources, but at this time the OSD interface does not address this knowledge gap.

As an example, if a large matrix is stored on I/O servers in a row-major pattern, while a parallel program needs to conduct column-based processing of this matrix, then requests become noncontiguous at the local storage system level (Figure 1). This results in a non-sequential access pattern and reduces the chances of prefetching being appropriately applied. If a sequence of operations, such as visualizing the dataset from multiple viewpoints, will perform column-based operations on the matrix, then a similar pattern of noncontiguous accesses will be repeated each time the matrix is accessed. If we could reorganize the on-disk data on the fly such that the future accesses become sequential accesses, or keep a copy of the data in this more optimal organization, we could significantly improve the observed local storage bandwidth, and we could do so without changes to the rest of the I/O system.

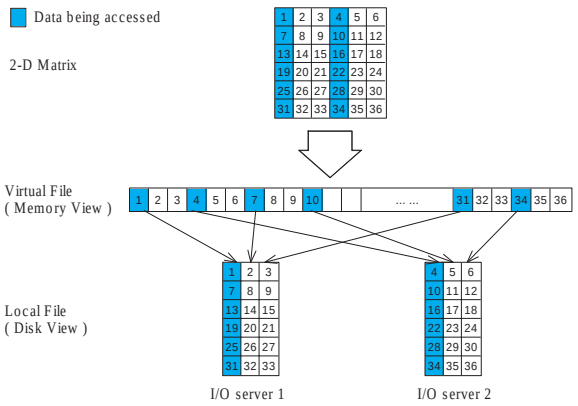


Figure 1. Strided Access Pattern

We note that *partially overlapped accesses* are becoming more common in many emerging scientific and engineering applications [6]: these applications access the same data regions more than once during their execution. Although similar noncontiguous patterns were reported in HPC community one decade ago [18], parallel file system and local file system architectures have not been focused on addressing these patterns. Examples of this pattern of access are common in mapping, Geographic Information Systems(GIS), and visualization applications. In these applications, when a user zooms in or out or changes viewpoint, some foci regions remain in the display window, but new regions are often accessed to be displayed in the new view (Figure 2).

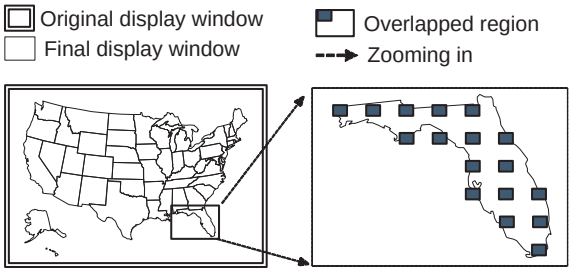


Figure 2. When a user zooms in on a particular region, data in that region is reused. If data must be re-read, the accesses will overlap.

When the data is huge and cannot be held in the memory, such as in computational science visualization applications [4], the application is typically re-reading many of the same regions for every new view.

Researchers have pursued two approaches to address the information gap between parallel file systems and local storage systems for better performance: application-directed prefetching hints [7] and language and compiler techniques that automatically insert speculative I/O access statements when compiling application codes [14, 15]. Except for MPI-IO hints, prefetching hints provided by applications are typically limited to specific compilers and thus are not portable. In addition, any application that would benefit from this technique needs to be re-written to accommodate the new feature. Compiler inserted prefetching seems to be more promising. Kotz [12] proposed a prefetching technique for parallel file systems as well, based on access patterns studied decades ago. While we believe that this approach is still relevant, we do not believe that prefetching alone is adequate to address the challenges of these applications.

In this paper we describe an on-disk grouping and prefetching technique named SOGP to bridge the gap between the local storage system and parallel the file system. The main ideas behind SOGP are to store a copy of data that is often accessed in a more efficient organization by grouping noncontiguous file I/O requests and storing these groups (called *segments*) on a local disk partition, and to use this more efficient organization to improve the performance of prefetching at the local storage level, better catering to the needs of parallel file system. By using several synthetic parallel I/O benchmarks, we see that our SOGP-enhanced PVFS scheme outperforms an EXT3-based PVFS by 39% to 230% in terms of aggregate I/O bandwidth in a testbed cluster system.

2 Related Work

Kotz *et.al.* proposed several techniques to improve parallel I/O performance including disk-directed I/O techniques [11] and practical prefetching techniques [12]. The first work proposes a new technique, disk-directed I/O to allow the disk servers to determine the flow of data for maximum performance by issuing large data requests. The second work developed a local pattern predictor and a global pattern predictor to catch the I/O access patterns in parallel file systems. Our work differs from both of these in that we use a combination of grouping and prefetching to aggregate I/O into large requests and to overlap computation and I/O.

A number of research projects exist in the areas of parallel I/O and parallel file systems, such as PPFS [10] and PIOUS [16]. PPFS offers runtime/adaptive optimizations, such as adaptive caching and prefetching, but does not use segment based on-disk grouping to maximize disk bandwidth utilization. PIOUS focuses on I/O from the viewpoint of transactions, not from that of scientific computing. In addition, these parallel file systems, as well as I/O optimizations on them, are mostly research prototypes, while our work is done on PVFS, a production-ready parallel file system widely used on Linux clusters.

3 SOGP Design and Implementation

In this paper, we present the design and implementation of a segment structured on-disk grouping and prefetching technique to improve I/O system performance for parallel file server based parallel file systems. The system works as a cache, so the original data format on local storage is preserved. At this time all metadata for SOGP is stored in memory for performance reasons, so a node failure will cause all the grouping information and on-disk segment cache to be lost, but all data will still be present as stored by the parallel file system, so the reliability of the parallel file system is unaffected by our enhancements. In our current implementation, we choose PVFS as our development and test platform.

3.1 SOGP Architecture

SOGP consists of the following major components: a segment-structured disk storage subsystem, an in-memory segment lookup table, a locality-oriented grouping algorithm and an in-memory segment cache. These components are shown in Figure 3. The disk storage subsystem is adopted to mainly implement segment I/O, which will be elaborated in Section 3.3. The in-memory segment lookup table is a data structure to index and manage all in-memory and disk segments in SOGP.

3.2 Augmenting PVFS with SOGP

In PVFS, the storage management module is named *trove*. At the time of this writing, the only implementation of trove is called *DBPF* (*i.e.*, DataBase Plus Files). This version uses Berkeley DB for metadata storage and files on a local file system (e.g. ext3) for storing file data. We modified the DBPF code so that SOGP is able to take over the requests dispatched to DBPF. DBPF employs a set of file service functions to handle file I/O requests, among which the most representative ones are *dbpf_bstream_read_list* and *dbpf_bstream_write_list*. These functions are main entries for read/write operations initiated by the upper layers of

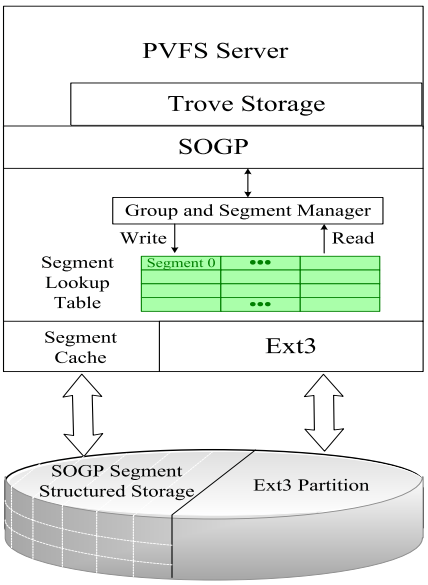


Figure 3. PVFS/SOGP software architecture

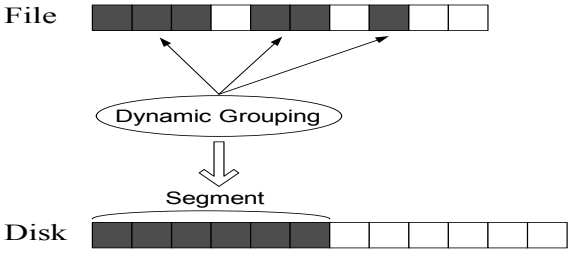


Figure 4. Noncontiguous data accessed together are grouped into a segment

PVFS. SOGP hooks into these service functions to intercept the requests and process them before they reach the local file system. SOGP monitors the requests received by DBPF and transforms the requests into its own segment format. For reads, if the segment is cached in memory in SOGP, then the request is satisfied immediately. Otherwise SOGP checks its in-memory segment lookup table to see if the segment is cached on the raw disk partition. If the data is stored in segment form on disk, SOGP uses the POSIX *read/write* system calls to access the raw disk partition as a device special file. If the request is neither cached nor resident on disk, the request is handed over back to DBPF for service. More details on our I/O interception implementation are provided in Section 3.4.

3.3 Segment I/O in SOGP

An I/O technique called *segment I/O* is used to facilitate large, sequential disk I/O operations even when noncontiguous accesses are present. This technique employs a dynamic grouping algorithm to organize related data, potentially from multiple files, and to save this data on the disk as a contiguous unit called a *segment*. Figure 4 illustrates a common scenario — accessing multiple large file portions by the segment I/O. This grouping of data into segments improves locality of access when these data are accessed again.

3.4 SOGP Data Flow

3.4.1 Read handling

When SOGP receives a read request, it works in the following steps (See Figure 5(a)). Note that in Steps 4 and 5, the read operation fetches not only the requested data, but also other data from the same or related files in the same segment group. In other words, prefetching is implicitly performed in these steps.

1. Receive a DBPF read request and translate the request into SOGP format, i.e., a SOGP read request.

2. Determine if this file belongs to any groups identified by its dynamic grouping algorithm. If so, continue to next step; otherwise, hand over the original request back to DBPF and then rebuild the locality group when necessary.

3. Check the in-memory lookup table to determine if this group is resident in SOGP in-memory cache or on SOGP storage. If in memory, satisfy the request immediately by copying data from the cache.

4. If the data is on SOGP storage, perform raw partition read to retrieve data into the segment cache and return the data.

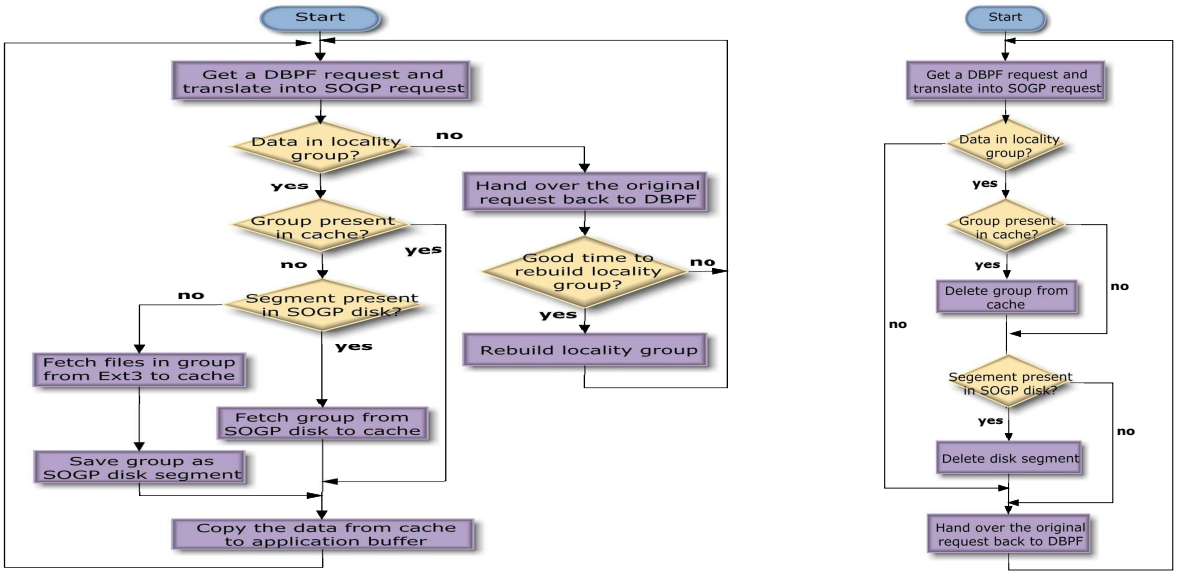
5. Otherwise, fetch the data from the local file system to the segment cache. Since the requested file belong to a segment group which is not on SOGP, we also save the group on SOGP as a segment for future reuse before the request is returned to the user.

3.4.2 Write handling

In our prototype implementation, SOGP does not perform optimizations on write requests. We employ write-invalidation as the segment cache consistency policy, so if write data is found to be on one of SOGP segments, then the entire segment is invalidated from both the disk and the memory (if present). After clearing any old data out of SOGP, write requests are handed over back to DBPF. The detailed steps for writes are described in Figure 5(b).

3.5 Locality-based Grouping and Prefetching

The purpose of SOGP is to group noncontiguous file I/O requests into segments *at runtime*, helping to improve data locality for the parallel file system and higher level applications. Segments are stored as physically continuous chunks on raw disks or partitions. These segments are our atomic



(a) Read request handling

(b) Write request handling

Figure 5. SOGP read/write data flow

storage unit to read, prefetch, write, and invalidate. The segment size is 16 MB in our current design trying to match the disk cylinder group size [21]. By co-locating related data in the same cylinder group, we reduce the number of small I/Os and help maximize read performance. The success of the technique is dependent on our ability to detect and exploit locality in parallel file access.

3.5.1 Grouping in SOGP

We notice that there are temporal and spatial access locality in existing parallel applications. Many compute nodes may concurrently read the same large data file, e.g., a 3-D object database, but in different portions. Researchers have noted several representative access patterns existing in today's scientific computing applications, such as simple strided, nested strided, random strided, sequential, segmented, tiled, and unstructured mesh accesses [20]. For example, Figure 1 illustrates a nested strided access pattern resulting from column based access to a 2-D matrix. Figure 2 shows the effect of zooming in on an image. In these examples, the innate access locality between adjacent column elements disappears at both the file level and the local storage level.

The problem is that parallel file systems may understand these access patterns but do not have control of the disk data organization; while local file systems do have control of the data layout, but they lack knowledge of the higher level parallel I/O access patterns. SOGP works around the knowledge gap at the local storage level by speculating on the parallel I/O access pattern and controlling the local disk layout to better suit this pattern, bridging the gap between parallel file systems and local file systems.

3.5.2 Grouping algorithm

We found that *Probability-based Successor Group Prediction* [5] appears to be a good candidate for our grouping purpose. Unfortunately, it is not a practical solution due to its spatial complexity: it requires unbounded memory to hold the entire online I/O trace in order to calculate the probabilities of one file being a successor of another.

Since we are working on data grouping which requires ideal accuracy, we choose to use Recent Popularity algorithm to build the relationship graph. By adjusting the parameters j and k in best-j-out-of-k algorithm, we can control the accuracy of the prediction algorithm. Once the graph is built, we need to divide the nodes into groups for prefetching. For the purpose of prediction accuracy, we adopt the most strict graph partitioning algorithm — Strongly Connected Component algorithm [9].

3.5.3 Scheduling grouping

It is critical that grouping should not compete with normal I/O operations. In our design, we choose storage system idle periods to perform data grouping. In order to do this, we modified the block device driver to allow reporting the length of its request queue. SOGP periodically checks this to make sure the queue is empty before sending grouping requests. If an idle period is detected, then the grouping request is sent. In our current implementation, we perform this queue length query once per second so that the overhead of this query is kept at a very low level. In addition, the idle period threshold is set to five seconds.

3.5.4 Discarding prefetched group items

The prefetched items are located together in memory with regular cached items. It is possible that prefetched items

get expunged before related requests arrive. In a parallel file system, prefetching and caching can both improve the I/O performance. However, prefetching is more effective in parallel scientific computing domain [13]. In the typical situation, it would be desirable to have a large space for prefetching. With the help of SOGP, such demand is largely alleviated, because grouped segments are sequentially stored on disk, and, to retrieve them from disk again, large I/Os requesting many adjacent blocks are issued to the disk, fully utilizing the maximum disk bandwidth.

4 Evaluation

Our evaluation was performed on two clusters. The first cluster, the Computer Architecture and Storage System (CASS), is a departmental storage cluster servicing multiple research groups at the University of Central Florida. The CASS cluster consists of 16 Dell PowerEdge 1950 nodes. Each node has two dual-core Intel Xeon 2.33 GHz processors, 4 Gbytes of DDR2 533 memory, and two SATA 500 Gbyte hard drives or two SAS 144 Gbyte hard drives. Each node has two Gigabit Ethernet ports, one for management and one for data transfer. These nodes are connected with a Nortel 5510-48T non-blocking 48 port high speed network switch and running the Red Hat Enterprise Linux operating system. PVFS 2.7.0 is installed on each of these nodes with the same configuration. Eight nodes out of the 16 nodes are configured as dedicated PVFS storage nodes and each of them assumes multiple roles: PVFS server and PVFS client. The remaining eight nodes are configured solely as compute nodes. All PVFS files were created with the default 64 KByte strip size, summing up to a 512 KByte stripe across all the eight server nodes. In addition, MPICH2 version 1.0.6p1 is installed as the MPI library. Unless stated otherwise, the tests on the CASS cluster are performed with 16 PVFS servers and 8 PVFS clients. Each test is repeated five times and the average is presented. The caching effect is deliberately avoided by rebooting the server nodes between runs.

To further test the scalability, we also ran some parallel I/O benchmarks on the Chiba City cluster at Argonne National Laboratory. The Chiba City cluster is a 512 CPU cluster running Linux. The cluster also includes a set of eight storage nodes. Each storage node is an IBM Netfinity 7000 with 500 MHz Xeons, 512 MBytes of RAM, and 300 GBytes of disk. The interconnect for high performance communication is 64-bit Myrinet. All systems in the cluster are on the Myrinet. The software stack is the same as CASS.

4.1 Software Configuration

We choose to compare the SOGP solution with an installation using the Ext3 file system, the most popular native file system for Linux-based clusters where PVFS resides. For brevity, in the rest of this paper, PVFS with Ext3 support is referred as PVFS2/Ext3, while PVFS with SOGP as PVFS/SOGP. When performing experiments, we alternated between PVFS/ext3 and PVFS/SOGP. Two independent PVFS "storage spaces" were configured, with the PVFS/SOGP configuration using Ext3 for DBPF data and a separate local disk partition for cached SOGP data. Next we describe our benchmarks in detail and compare the results of both I/O systems.

4.2 Benchmarks

We use three popular parallel I/O benchmarks to evaluate the benefit of our design over the traditional PVFS/Ext3 approach. We use mpi-tile-io to simulate visualization application behavior, and we use noncontig and IOR to simulate zooming behavior in various visualization applications.

As far as the I/O intensive parallel application is concerned, the most important thing users would be interested in is the aggregate I/O bandwidth (*I/O bandwidth* in brief for the rest of the paper) of the entire parallel file system, which is the sum of the I/O bandwidth of all storage nodes. As a result, we choose the I/O bandwidth as the major metric during the evaluation.

Noncontig is a publicly available parallel I/O benchmark from Parallel I/O Benchmarking Consortium [2]. It is designed for studying I/O performance using various I/O methods, I/O characteristics and noncontiguous I/O cases. This benchmark is capable of testing three I/O characteristics (region size, region count, and region spacing) against two I/O methods (list I/O and collective I/O) in four I/O access cases (contiguous memory contiguous disk, noncontiguous memory contiguous disk, contiguous memory noncontiguous disk, and noncontiguous memory noncontiguous disk).

In real world visualization applications, image zooming is an important and common behavior [22]. For a very large dataset, the user might want to see a single picture that represents the entire data at first. After a quick preview, the user might want to focus on a particular region of interest, i.e., zooming to see more detail. Zooming is simulated by increasing veclen while reducing elmtcount so that the product of them is kept constant.

Mpi-tile-io is another synthetic benchmark from the Parallel I/O Benchmarking Consortium benchmark suite [2]. It has been widely used in many parallel I/O related studies [23, 26, 25]. The application implements tile access on a two-dimensional dataset, with overlapped data between adjacent tiles. The size of the tiles and the overlap ratio is adjustable. Collective I/O support is optional in this application. We studied both cases with and without collective I/O support in our experiments.

IOR is developed at Lawrence Livermore National Laboratory [1]. It is designed for benchmarking parallel file systems using POSIX, MPI-IO, Parallel netCDF, or HDF5 interfaces. To test the scalability of our SOGP design, we run the IOR benchmark on the Chiba City cluster by varying the number of clients from 8 to 256. At the same time, we also turned on the use file view option in IOR and changes the view during runs to keep the total size of working set constant while simulating the partially overlapped access pattern.

4.3 Prefetching Accuracy

In order to better understand the source of benefit — group access locality, we further investigate SOGP application-level buffer cache — group cache behaviors. In our experiments, we enable a small amount of logging code in SOGP to collect the group cache utilization statistics while PVFS/SOGP server is running. The results are then written into a server log file. The storage system layer configuration at the time of testing is described in Table 1. We ran noncontig benchmark five times with different parameters to simulate the image zooming effect. The aggregated working set is approximately 100 GB. After the test, we extract the data from the server log file, as shown in Table 2. There are 52917 segment groups in total built during the test and 98.27% of them are accessed more than once.

SOGP segment cache	SOGP partition	Ext3 partition
1 GB	20 GB	180 GB

Table 1. Storage system layers configuration

Number of processes	Hits	Misses	Total	Hit rate(%)
4	4090515	654311	4744826	86.21
16	4144802	726278	4871080	85.09
64	4421932	864930	5286862	83.64
256	4652093	1051098	5703191	81.57

Table 2. Group access locality analysis

4.4 I/O Bandwidth

For the noncontig benchmark, which exhibits noncontiguous file accesses in some phases, we expect that PVFS will significantly benefit from SOGP in these phases. For contiguous file access phases, the corresponding disk accesses may still become non-contiguous because of the gap between the file system and the disk, and therefore PVFS can still possibly benefit from the group access feature of SOGP.

In addition to the performance implications of SOGP, we notice that the results for collective I/O method and non-collective I/O method exhibit some differences. We ran the test on CASS cluster and collected results for both collective and non-collective methods, as shown in Figure 6 to allow a side-by-side comparison.

In Figure 6, PVFS/SOGP exhibits a read performance gain of 92% to 230% over the PVFS/Ext3 baseline system. These results suggest that, by combining highly related accesses into groups, SOGP can boost the I/O performance dramatically. The simple strided pattern of noncontig benchmark issues I/O accesses to the same data repeatedly, which translates into better group access locality that SOGP is able to take the best advantage of.

We were concerned that these results might result from a particularly good match between the vector length used

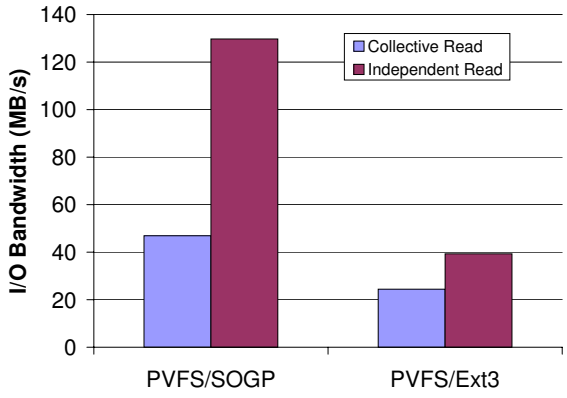


Figure 6. noncontig I/O performance comparison

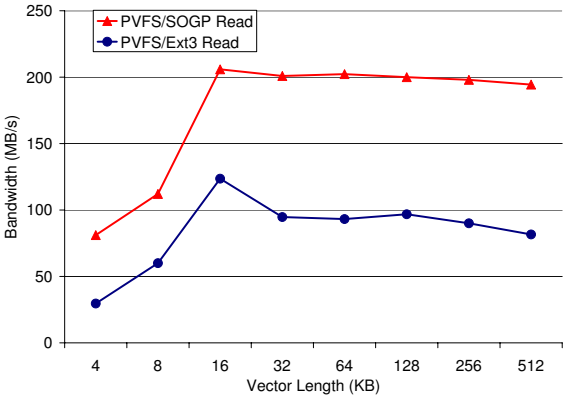


Figure 7. Read performance for noncontig, varying vector length

in the test, which determines the number of regions and region spacing, and our SOGP configuration. Figure 7 explores the impact of vector length on performance for the range between 4 Kbytes and 512 Kbytes. We see that at very small sizes there is a drop-off in performance, possibly due to the general inefficiency of servicing a very large number of small and noncontiguous regions, but otherwise the performance improvement is consistent.

Figure 8 presents the I/O bandwidth results collected on CASS when running mpi-tile-io on PVFS/Ext3 and PVFS/SOGP, respectively. In this test, the total request size was 128 GBytes. To show that the performance impact of collective I/O is orthogonal to that of SOGP, we plot the results for both collective I/O and independent I/O (non-collective) method in this figure. In both cases, PVFS/SOGP exhibits approximately 50% higher I/O bandwidth than PVFS/Ext3. This derives from PVFS/SOGP grouping multiple small I/Os into larger ones for read access and prefetching. With this total size, the entire dataset fits into the SOGP partitions on the servers, allowing for better

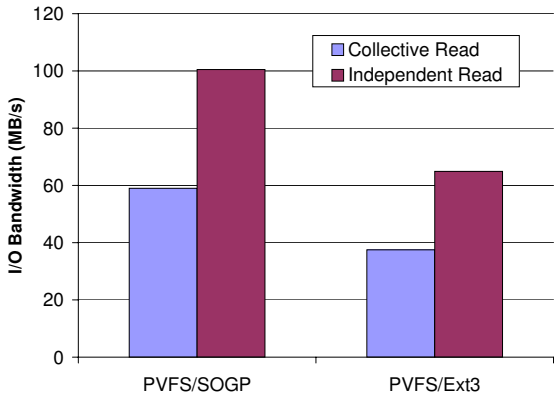


Figure 8. I/O bandwidth with mpi-tile-io

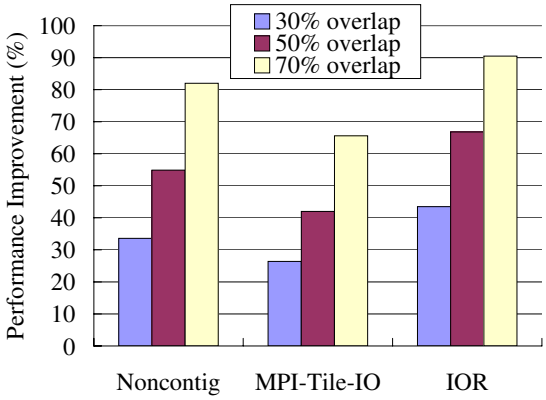


Figure 9. Impact of access overlap on I/O aggregate bandwidth

locality of access on reads. A possible reason why the performance gain is not that "conspicuous" lies in that mpi-tile-io, as a read-once/write-once dominant application, does not make good use of the grouping feature of SOGP in a repeated fashion.

4.5 Overlapped Access

In this section we analyze the percentage of overlapped access in several applications such as mpi-tile-io. These applications are known to have overlapped accesses. For mpi-tile-io, the tiles to be accessed can be specified by the vertical and horizontal spread of the tile, so it is easy to specify to what degree accesses will overlap. Since the resulting I/O accesses are not exactly the same, in many cases they are only partially overlapped. We defined the partially overlapped access as the *number of bytes* accessed more than once. We use this number to the total number of bytes accessed to obtain the percentage of overlapped access.

In Figure 9 we present the impact of overlapped access over performance gains of SOGP. As we would expect, the larger the percentage of overlapped accessed region, the larger the benefit of grouping.

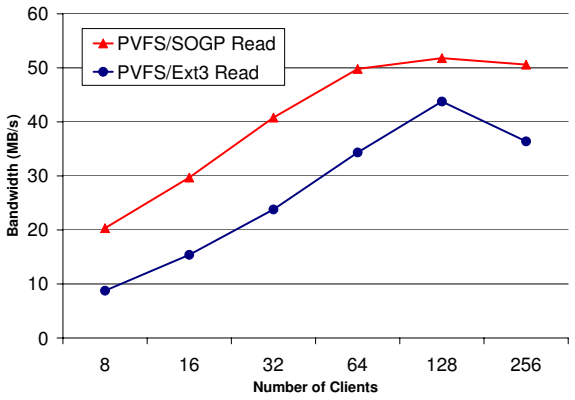


Figure 10. IOR benchmark bandwidth comparison

4.6 Scalability study using IOR

From the results shown in Figure 10, one can observe that both PVFS/SOGP and PVFS/Ext3 show performance gains in proportion to the number of clients (i.e., processes) within a certain range (less than 128). The I/O bandwidth of both systems degrades at 256 clients on this test system, likely because the PVFS servers are saturated. On the other hand, PVFS/SOGP outperforms PVFS/Ext3 by up to 132% in terms of absolute read performance (in unit of I/O bandwidth). Even when the PVFS becomes saturated, PVFS/SOGP performance exceeds that of PVFS/Ext3 by 39%. Another observation is that the benefit gain of PVFS/SOGP over PVFS/Ext3 from group access locality is not decreased by increasing the number of processes that issue the read requests. These results indicate that for read workloads PVFS/SOGP retains the scalability properties of PVFS while providing significantly higher bandwidth at a given number of clients.

5 Conclusions

In this paper, we present the design and implementation of the first version of SOGP for a state-of-the-art parallel file system — PVFS. SOGP employs a segment I/O technique and a grouping based prefetching methodology to resolve some of the limitations existing in today's parallel file systems in how they interact with local storage and the impact of this method of interaction in the face of overlapping noncontiguous access, such as seen in many computational science post-processing and visualization applications. Using several parallel I/O benchmarks in different Linux-based cluster testbeds, we conclude that an SOGP-enhanced PVFS prototype system can significantly outperform a Linux-Ext3-based PVFS by up to 230% in terms of I/O bandwidth. This work also identifies possible enhance-

ments for object storage systems as well, because this issue of lack of communication is also true for object storage systems. More generally, this work points out the advantages of storing application data in more than one organization when ready-heavy workloads are anticipated. We plan to investigate how storing multiple representations of application data may be best managed, using SOGP and our grouping technology as a starting point.

Acknowledgment

This work is supported in part by the US National Science Foundation under grant CCF-0621526,by the US Department of Energy Early Career Principal Investigator Award DE-FG02-07ER25747, and in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. Work is also supported in part by NSF through grants CNS-0551727 and CCF-0325934, and DOE with agreement No. DE-FC02-06ER25777. The authors greatly appreciate Rajeev Thakur, Rob Latham, and Sam Lang from Argonne National Laboratory for their collaboration and support, and the reviewers for their constructive comments and suggestions.

References

[1] IOR Benchmark. http://sourceforge.net/projects/ior-sio/.

[2] Parallel I/O Consortium. http://www-unix.mcs.anl.gov/∼rross/pio-benchmark/.

[3] Lustre. http://www.lustre.org, June 2004.

[4] AKIBA, H., MA, K.-L., CHEN, J. H., AND HAWKES, E. R. Visualizing multivariate volume data from turbulent combustion simulations. *Computing in Science and Engg. 9*, 2 (2007), 76–83.

[5] AMER, A., LONG, D. D. E., AND BURNS, R. C. Group-based management of distributed file caches. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)* (Washington, DC, USA, 2002), IEEE Computer Society, p. 525.

[6] BRYANT, R. E. Data-intensive supercomputing: The case for DISC. Tech. Rep. CMU-CS-07-128, Carnegie Mellon University, May 2007.

[7] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst. 14*, 4 (1996), 311–343.

[8] CARNS, P. H., LIGON III, W. B., ROSS, R. B., AND THAKUR, R. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference* (Atlanta, GA, 2000), USENIX Association, pp. 317–327.

[9] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN., C. *Introduction to Algorithms*, second edition ed. MIT Press and McGraw-Hill, 2001, ch. Section 22.5, pp. 552–557.

[10] JAMES V. HUBER, J., CHIEN, A. A., ELFORD, C. L., BLUMENTHAL, D. S., AND REED, D. A. PPFS: a high performance portable parallel file system. In *ICS '95: Proceedings of the 9th international conference on Supercomputing* (New York, NY, USA, 1995), ACM, pp. 385–394.

[11] KOTZ, D. Disk-directed I/O for MIMD multiprocessors. *ACM Trans. Comput. Syst. 15*, 1 (1997), 41–74.

[12] KOTZ, D., AND ELLIS, C. S. Practical prefetching techniques for parallel file systems. In *PDIS '91: Proceedings of the first international conference on Parallel and distributed information systems* (Los Alamitos, CA, USA, 1991), IEEE Computer Society Press, pp. 182–189.

[13] LEE, Y.-Y., SEO, D.-W., AND KIM, C.-Y. Table-comparison prefetching in via-based parallel file system. In *CLUSTER '01: Proceedings of the 3rd IEEE International Conference on Cluster Computing* (Washington, DC, USA, 2001), IEEE Computer Society, p. 163.

[14] LUK, C.-K., AND MOWRY, T. C. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Trans. Comput. 48*, 2 (1999), 134–141.

[15] MOWRY, T. C., DEMKE, A. K., AND KRIEGER, O. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation* (1996), USENIX Association, pp. 3–17.

[16] MOYER, S. A., AND SUNDERAM, V. S. Characterizing concurrency control performance for the PIOUS parallel file system. *J. Parallel Distrib. Comput. 38*, 1 (1996), 81–91.

[17] NAGLE, D., SERENYI, D., AND MATTHEWS, A. The panasas activescale storage cluster – delivering scalable high bandwidth storage. In *SC'2004 Conference CD* (Pittsburgh, PA, Nov. 2004), IEEE/ACM SIGARCH.

[18] NIEUWEJAAR, N., KOTZ, D., PURAKAYASTHA, A., ELLIS, C. S., AND BEST, M. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems 7*, 10 (October 1996), 1075–1089.

[19] SCHMUCK, F., AND HASKIN, R. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST-02)* (Jan., 2002), pp. 231–244.

[20] SHORTER, F. Design and analysis of a performance evaluation standard for parallel file systems. Master's thesis, Clemson University, 2003.

[21] SHRIVER, E., SMALL, C., AND SMITH, K. A. Why does file system prefetching work? In *ATEC'99: Proceedings of the Annual Technical Conference on 1999 USENIX Annual Technical Conference* (Berkeley, CA, USA, 1999), USENIX Association, pp. 6–6.

[22] WANG, C., AND SHEN, H.-W. Lod map - a visual interface for navigating multiresolution volume visualization. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1029–1036.

[23] WANG, Y., AND KAELI, D. Profile-guided I/O partitioning. In *Proceedings of the 2003 International Conference on Supercomputing (ICS-03)* (New York, June 23–26 2003), ACM Press, pp. 252–260.

[24] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 307–320.

[25] WU, J., WYCKOFF, P., AND PANDAC, D. Pvfs over infiniband: Design and performance evaluation. In *Proceedings of the 2003 International Conference on Parallel Processing (32th ICPP'03)* (Kaohsiung, Taiwan, Oct. 2003), IEEE Computer Society, pp. 125–132.

[26] YU, W., LIANG, S., AND PANDA, D. K. High performance support of parallel virtual file system (PVFS2) over quadrics. In *ICS* (2005), Arvind and L. Rudolph, Eds., ACM, pp. 323–331.