Performance of the IBM General Parallel File System

Terry Jones, Alice Koniges, R. Kim Yates
Lawrence Livermore National Laboratory
rkyates@llnl.gov

Abstract

We measure the performance and scalability of IBM's General Parallel File System (GPFS) under a variety of conditions. The measurements are based on benchmark programs that allow us to vary block sizes, access patterns, etc., and to measure aggregate throughput rates. We use the data to give performance recommendations for application development and as a guide to the improvement of parallel file systems.

1. Introduction

Large-scale scientific computations such as those associated with ASCI[1] stretch the limits of computational power. Parallel computing is generally recognized as the only viable solution to high performance computing problems [1,2]. I/O has become a bottleneck in application performance as processor speed skyrockets, leaving storage hardware and software struggling to keep up. Parallel file systems must be developed that allow applications to make optimum use of available processor parallelism.

To deal with these parallel I/O issues, IBM has developed the General Parallel File System (GPFS) [3,4]. GPFS allows parallel applications to have simultaneous access to a single file or to a collection of files. Each node on an SP has equal access to files using standard POSIX calls. In addition, increased flexibility for parallel applications can be gotten by reading and writing GPFS files via MPI-I/O libraries layered on top of the file system [5].

There are several reasons why parallel applications need such a file system. Where performance is the major bottleneck, the aggregate bandwidth of the file system is increased by spreading reads and writes across multiple disks, and balancing the load to maximize combined throughput. For dealing with very large data sets, the ability to span multiple disks with a single file makes the management of the file seamless to the application. The alternative, writing to a separate file for each process, is not only very inconvenient (the user must keep track of the thousands of files that would be left after every run), it can prevent or complicate reading back the data to a different number or different set of processors, and usually requires an extra post-processing step to coalesce the separate files into a single file for, say, visualization.

[1] ASCI (the Accelerated Strategic Computing Initiative) is a U.S. DOE Defense Programs project to create leading-edge high-performance capabilities in scientific computation (see http://www.llnl.gov/asci/).

1.1 I/O requirements and workload

Recently a computing rate of 2.14 Tflops was achieved on a linear algebra benchmark on the 1464-node RS/6000 SP machine (called "SKY") at LLNL. This machine has a theoretical peak computational rate of about 3.9 Tflops and a total memory size of 2.6 Tbytes. If we use a common rule of thumb that predicts applications will store one byte of information per 500 peak flops, this suggests that an I/O throughput of approximately 7.3 GB/sec is needed. (Note: throughout this paper, 1 MB is 1024^2 bytes, and 1 GB is 1024^3.) Another common rule to estimate how well a system is balanced, based on past experience, says that an application will store half of total memory once per hour, and that this should take no more than five minutes. For SKY this rule suggests an I/O target rate of about 4.4 GB/sec. In the current installation SKY is equipped with two GPFS file systems for each of its three partitions, providing an aggregate throughput of about 6.7 GB/sec to the six separate file systems. (Note: It would have been possible to combine the two GPFS file systems (with a total of 56 servers) on each of SKY's three partitions into a single file system on each partition, but it was thought that two file systems per partition would be more useful. It is not possible for a GPFS file system to span the three partitions that form SKY.)

But peak and sustained performance rates alone are not the only factors. Scientific applications are notoriously complex and diverse in their file access patterns [6,7]. I/O access patterns are generally divided into subgroups [8]:

1. Compulsory
2. Checkpoint/restart
3. Regular snapshots of the computation's progress.

4. Out-of-core read/writes for problems which do not fit the memory.
5. Continuous output of data for visualization and other post-processing.

In the applications with which we are most familiar, writes will need to be performed more often than reads, with categories 2 and 5 dominant.

Finally, we cannot neglect the question of reliability. To achieve gigabyte-per-second performance there must be hundreds or thousands of disks, with dozens of servers and attendant connections. These must all be highly reliable. More importantly, they must be fail-safe, so that the system can continue to function when a component fails. This requires sophisticated and well-tuned software that can compensate for failures in a distributed system.

2. Structure and function of GPFS

The GPFS architecture was designed to achieve high bandwidth for concurrent access to a single file (or, of course, to separate files), especially for sequential access patterns. The intended platform for this file system is IBM's line of massively parallel computers, the RS/6000 SP, and performance is achieved with commodity disk technology. The RS/6000 SP line of machines are general purpose, high end computers which scale to thousands of processors [9]. Each node runs a Unix kernel and is autonomous. A proprietary network technology permits every node to communicate with a corresponding remote node simultaneously. Access is uniform to all remote nodes (there is no notion of a "neighbor" node which has better bandwidth characteristics) [10].

Node-to-node communication is enhanced through the use of the special network fabric present in IBM SP parallel machines. Commonly referred to simply as "the switch," this interconnect provides unidirectional IP at 83 MB/sec for the model installed at LLNL [11].

There has been much research in parallel file systems (e.g., [12,13,14,15,16,17,18,19]). However, as we need production-quality file systems that can deliver gigabyte-per-second throughput, the most relevant systems are Intel's PFS [20] and SGI's XFS [21]. The main difference between GPFS and PFS is that the latter has a non-standard interface and has not shown high performance on concurrent access to a single file. XFS does use the standard POSIX interface and has high performance, but works only for shared memory architectures.

2.1 GPFS architecture

GPFS is implemented as a number of separate *software subsystems* or *services*. Each service may be distributed across multiple nodes within an SP system. Many of the services necessary for GPFS are provided by a persistent GPFS daemon called mmfsd. Among the more important services provided by mmfsd are (see Fig. 1): (1) file system access for nodes which wish to mount GPFS; (2) a *metanode* service which retains file ownership and permissions information for a particular file; (3) a *stripe group manager* service which manages and maintains information about the various disks that make up the file system; (4) a *token manager server* which synchronizes concurrent access to files and ensures consistency among caches; (5) finally a *configuration manager* which ensures that a stripe group manager and token manager server are operational and that a quorum exists.

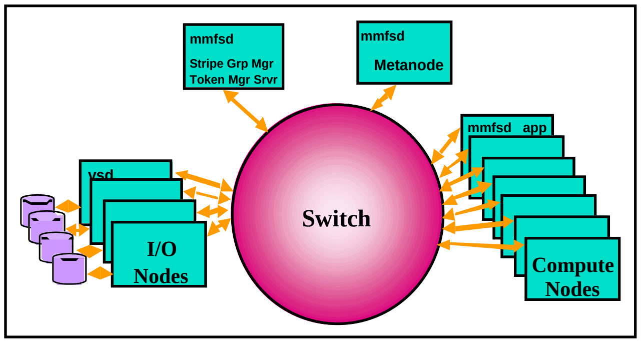

Figure 1. Overall GPFS architecture

Each of the nodes dedicated to running parallel applications has an mmfsd daemon present to mount the file system and perform access. It is responsible for actually performing the reads and writes performed on that node.

The Virtual Shared Disk (VSD) layer of GPFS permits a node to locally issue a write that physically occurs on a disk attached to remote node. The VSD layer therefore consists of VSD clients on the application nodes and VSD servers on the disk-attached I/O nodes.

GPFS is a "client-side cache" design. The cache is kept in a dedicated and pinned area of each application node's memory called the *pagepool* and is typically around 50 Mbytes per node. This cache is managed with both read-ahead (prefetch) techniques and write-behind techniques. The read-ahead algorithms are able to discover sequential access and constant-stride access.

GPFS is multi-threaded. As soon as an application's write buffer has been copied into the pagepool, the write is completed from an application thread's point of view. GPFS schedules a worker thread to see the write through to completion by issuing calls to the VSD layer for communication to the I/O node. The amount of concurrency available for write-behind and read-ahead activities is determined by the system administrator when the file system is installed.

Consistency is maintained by the token manager server of the mmfsd daemon. (There is one such copy of the mmfsd running within the entire SP parallel computer.) The item being accessed (for example, a file) is termed a lock *object*. The per-object lock information is termed a *token*. On every write access, the mmfsd determines if the application holds a lock that permits the right to modify the file. If this is the first write for this node and for this file, a write token must be acquired. The mmfsd negotiates with the node that holds the token in order to get the requested token. It first contacts the token manager server for a list of nodes that have the token, then it negotiates with the tokens in that list to acquire the token. This technique is employed for scalability reasons: distributing the task to the mmfsd reduces serialization at the token manager server. Moreover, in anticipation of sequential access the token manager may extend the range of bytes locked beyond what was actually requested.

GPFS enforces strict POSIX atomicity semantics. That is, if two separate nodes write to the same file, and if the writes are overlapping, the overlapped region must be either entirely from node A or entirely from node B.

2.2 GPFS data paths

It is instructive to study the data flow of reads and writes when analyzing any file system. This is particularly true of file systems with distributed components.

When an application requests read or write access to a file, GPFS first determines if the file already exists via the metanode (which is running on a possibly remote copy of the mmfsd). Any updates to the inode information for the file are negotiated with the metanode. The original node to open the file will become the initial metanode for that file and will have pertinent metadata cached including the original access. The metanode manages all directory block updates. The metanode may change locations in instances where the node fails. The following assumes the application has successfully opened the file for writing. Figure 2 shows the major steps involved with a write:

- The application makes a call with a pointer to a buffer in its space. The mmfsd on the application node acquires a write access token for the byte range involved in the write.
- The mmfsd acquires some of the file's metadata to reflect where the data is to be written, some unused disk blocks for the write, and some buffer space from the pagepool. If no buffer is available one is cleared by writing the oldest buffer to disk.
- The data is moved from the application's data buffer to the GPFS pagepool buffer. A thread is scheduled to continue the write. As far as the application is concerned, the write has completed. This technique is commonly called *write-behind caching*.
- The GPFS worker thread calls the VSD layer to perform the write. This in turn is passed on to the IP layer where the write is broken up into IP message packets (mbufs, typically 60 Kbytes), and the data is copied to the switch communications send pool (spool) buffers. At this point, the data has been copied twice, once into a GPFS pagepool buffer and then to the switch send pool buffer. Both copies are handled by the application's CPU.
- The data is communicated over the switch. Once the data is received at the VSD server receive pool (rpool) buffer, the switch driver forwards each packet to the VSD through the IP layer of AIX.
- Once all packets of a request have been received at the VSD server, a buddy buffer is allocated. The buffer reassembles the large chunk of data from the packets. If a buffer is not immediately available, the request is queued and the data remain in the switch receive pool.
- The VSD server releases all the receive pool mbufs and issues a write via the disk device driver. The device driver may wait a short time (configurable) before issuing the write so that it might be combined with immediately occurring sequential writes in an attempt to write an entire storage block (size determined by the system administrator). On RAID systems this should be the RAID stripe size.
- The VSD server releases the buddy buffer and sends notification of completion to the VSD client.
- The VSD client drives the completion processing. The pagepool buffer is now available for use for another application call.

Reads are similar, with data flowing in the opposite direction. GPFS attempts to guess which data is desired next and prefetch it into the pagepool on reads. For this reason, substantial performance gains are available for sequential read access patterns.

2.3 Unusual features and mechanisms of GPFS

As mentioned earlier, the degree of scalability is probably the most unique feature of GPFS. This design permits a file to be striped across a system-administrator-defined number of server nodes. Not only does this provide higher aggregate read and write performance, it also permits larger files and file systems. Furthermore, each node may stripe its portion across many locally attached disks thus providing additional parallelism. GPFS's file striping mechanisms ensure metadata and data are managed in a distributed manner to avoid hot spots. Traditional

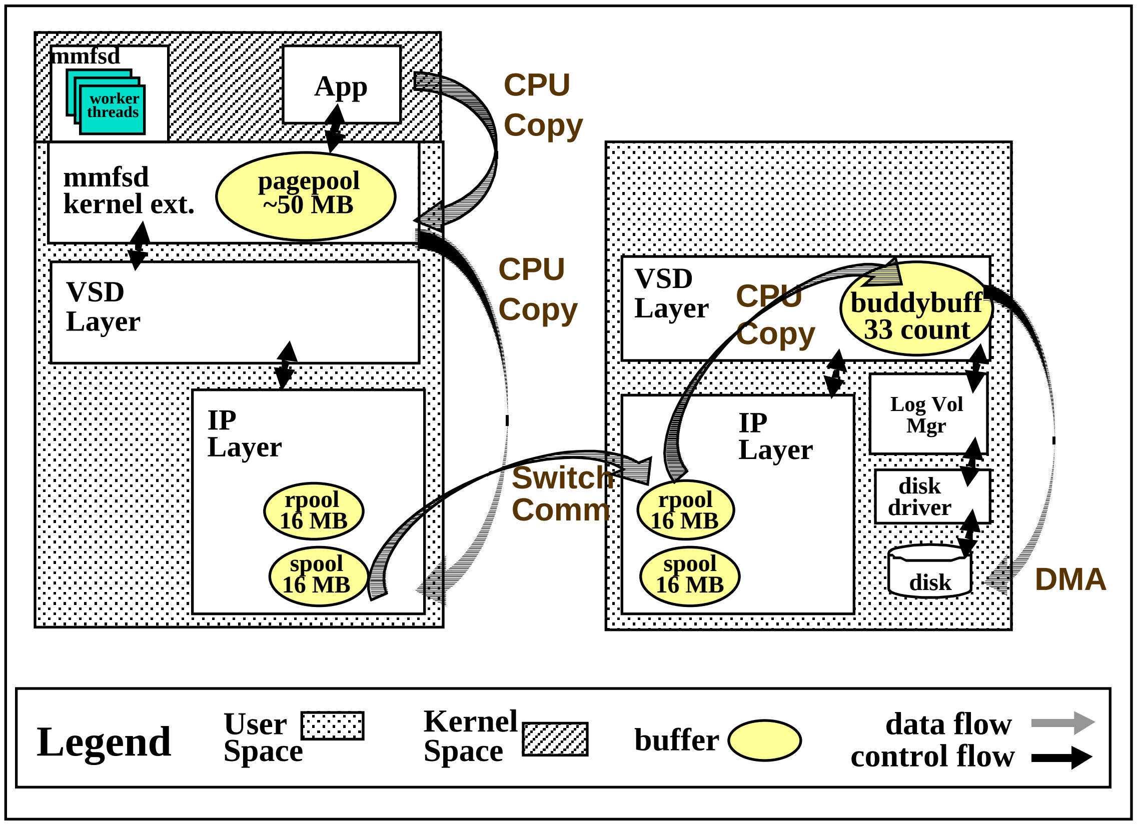

Figure 2. Control and data flow in writing to disk

local or distributed file systems are far more localized in terms of data placement that greatly increases the risks of loading. Together these features permit file systems that are terabytes in capacity and provide over a gigabyte per second bandwidth.

The token management scheme employed by GPFS permits byte-range locking. That is, one task may be granted to write or read access to a portion of a file, and other tasks may be granted read or write access to other portions of the same file. This permits writes and reads to occur concurrently without serialization because of consistency. Unfortunately, traditional UNIX file systems, and most other file systems, do not support parallel access well: the mechanisms they provide for file consistency (file locking) are performed at the entire file level. This is particularly ill-suited for parallel computing where multiple nodes may be writing to different portions of the same file concurrently. Furthermore, the GPFS token management eliminates the possibility of "stale mount points" which commonly occur in NFS. These features are a key advantage to GPFS.

GPFS incorporates extensive reliability and availability measures. GPFS uses the High Availability subsystem provided with every RS/6000 SP for improved fault tolerance. This system, which uses a neighbor ping system to determine the health of every node, is used to check the health of distributed components [22]. The token manager server is usually co-located with the stripe group manager. In the event that the mmfsd providing the stripe group manager service or the token manager service becomes unavailable, the configuration manager will select a pre-determined replacement and a randomly chosen "next in line" node should the new candidate fail. A quorum is required for successful file system mounts. This prevents the file system from getting into an inconsistent state in the event of a partition in the network fabric. Finally, extensive logging is used to commit file system metadata changes in a safe manner. Availability is enhanced through the ability to replicate files, use of RAID arrays, or AIX mirroring.

2.4 Potential problems and bottlenecks

GPFS version 1.2 has some functionality limitations. It does not support memory mapped files, a common non-POSIX way to establish a mapping between a process's address space and a virtual memory object. In addition, since the atime, mtime, and ctime information is maintained in a distributed manner (for performance reasons), some time is required before the most up to date information on an actively changing file is available to all nodes. For our applications, these are not hindrances.

GPFS 1.2 has a performance limitation that can arise when clients send data to the servers faster than it can be drained to disk. For any given GPFS file system, there is an upper bound on how fast the rotating media can actually commit writes or perform reads. With enough application nodes sending information to these disks via the high performance SP interconnect, applications may be able to exceed the ability of the aggregate disks to drain the information. When this happens, current versions of GPFS use an exponential backoff protocol: An application node is delayed a time y, and then it retries. If that write fails, it waits $2y$, then $4y$, $8y$ and so on. We have observed that under extreme conditions this backoff protocol can actually reduce the throughput below what the file system is capable of maintaining.

The data path presented in section 2.2 also describes the potential bottlenecks. For instance, if an application is doing a write and the pagepool is full, the write must block until some information from the pagepool can be committed. Adjusting the size of the various buffers in the data path to permit efficient performance will depend on the type and number of VSD servers in a given GPFS file system, the type and number of disk drives and the connections to these drives, and of course on the application access patterns. In general, it is best to have a balanced configuration in which in all VSD servers have similar numbers of disk drives and similar types of disk drives. The application should make large writes and reads where possible to amortize the system call cost: one write call with a one megabyte buffer is much more efficient than one million calls with a one byte buffer simply because of the CPU limitations on the application node. The GPFS block size should be compatible with the RAID array when RAIDs are employed.

Another potential bottleneck arises from the fact that data is copied twice within the client: once between the application's buffer and the pagepool, and again between the pagepool and IP buffer pool. For writes, this has the advantage that the application can continue as soon as the data is copied into the pagepool. But copying the data twice can use enough memory bandwidth to limit the usefulness of having more than one processor per node write to a file concurrently. However, for all but very small jobs (i.e., those with few processes) this is of little consequence, since the throughput will be limited by the number of servers rather than by the number of clients.

As can be seen from the design of GPFS, and as will become clear in the experimental data, GPFS is biased toward sequential access patterns. This can be a disadvantage for applications in which processes access the file in small pieces that are interleaved with data from other processes. Client-side caching contributes to this effect, as does GPFS's handling of the tokens that ensure atomicity of writes. However, this nonsequential small-block effect should be mitigated somewhat by using a higher-level I/O library to redistribute data into larger blocks before they are sent to GPFS.

3. Experiments

The experiments shown here have been chosen because they show the effects of varying the I/O characteristics of application programs. We measured how aggregate throughput varied depending on the number and configuration of client processes, the size of individual transfers, and access patterns. We also show how GPFS performance scales with system size. In addition, we have also run many experiments to test the effects of changes in GPFS tuning parameters that are fixed when the file system is built, but we do not show these here; some can be found in [3].

3.1 Methodology

We are primarily interested in measuring the aggregate throughput of parallel tasks creating and writing a single large file, and of reading an existing file. To accomplish this we have created a benchmark program (ileave_or_random, written in C using the MPI message passing library) capable of varying a large number of application I/O characteristics. To measure the throughput of writes, the benchmark performs a barrier, then each task records a "wall clock" starting time, process 0 creates the file and all other processes wait at a barrier before opening it (but where noted, some experiments access a separate file for each process), then all processes write their data according to the chosen application characteristics (in the tests shown here, always independently of each other, filling the file without gaps and without overlap); finally, all processes close the file and record their ending time. The throughput is calculated as the total number of bytes written in the total elapsed wall clock time (the latest end time minus the earliest start time). Reads are measured similarly, except that all processes can open the file without having to wait for any other process. This approach is very conservative, but its advantages are that it includes the overhead of the opening and closing and any required seeks, etc., and measures true aggregate throughput rather than, for example, an average of per-process throughput rates. Because most of our experiments were run on production systems in full use, we could not be sure when other jobs were competing for the file system being tested. To address this problem, we ran each test several times and report the best time. Hence the results indicate the peak performance the file system is capable of delivering rather than what a user would see in the presence of other jobs competing for the same resources.

Like all file systems, the performance of GPFS depends heavily on the access pattern of the application. The two access patterns we report on are illustrated in Fig. 3.

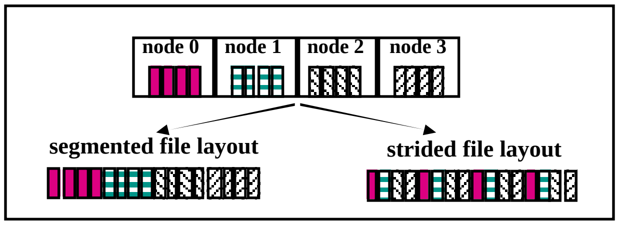

Figure 3. Segmented vs. strided access

What we call the *segmented* pattern is processor-wise sequential, i.e., the file is divided evenly among the client processes, with each process writing a sequence of equal-sized blocks to (or reading from) a contiguous portion of the file. Conversely, in the *strided* access pattern the blocks are interleaved, with process 0 accessing blocks 0, p, 2p, etc., process 1 accessing blocks 1, p+1, 2p+1, etc. The block size is the number of bytes moved by each individual write or read operation, and is not necessarily the same as the stripe width of the file system (which was 256 kB for the systems tested).

As one would expect from its token management and client-side caching as described in Sec. 2, and as demonstrated by the data shown below, GPFS exhibits much better performance for the segmented access pattern. All experiments shown are for the segmented access pattern, except where otherwise noted.

3.2 Relevance to real applications

The I/O performance seen in real applications will depend on complex interactions between the system and the application's runtime behavior. Competition with other applications for I/O resources, wild access patterns, and some randomness in the file system can cause performance to be lowered. Another relevant effect is that GPFS will do better when it is "warmed up," i.e., when its current access patterns resemble most recent patterns. On the other hand, because our benchmarks do not overlap I/O with computation, and because they force all I/O to occur simultaneously, real applications may actually do better.

3.3 Experimental results

For a given GPFS file system, the most important factors affecting performance (aside from the access pattern) are the number of parallel processes participating in the transfers, and the size of the individual transfers. Figure 4 shows that performance is highest when the ratio of client processes to VSD server nodes is near 4:1. (Though the nodes running our experiments have four processors per node, we ran only one client task per node, except where otherwise noted.) In Fig. 4, we wrote/read a 102 GB file using 256 kB blocks in the segmented pattern.

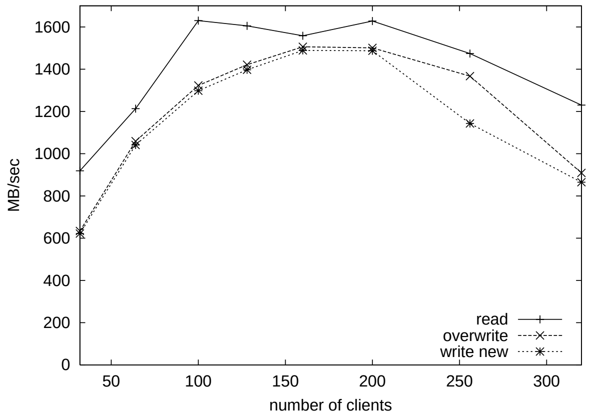

Figure 4. Varying clients vs 38 servers

When the client:server ratio is too low the servers are starved for data; when it is too high the receiving buffers fill up faster than they can be drained, eventually causing packets to be dropped and retries initiated, reducing performance. This points to a need for improved control of the data flow between client and server. In the middle of the curves the throughput is high: around 1500 MB/sec for writes and 1600 for reads; this agrees with our expectation of about 40 MB/sec times the number of servers. Note also that file overwrites are not appreciably faster than new file creation.

The next four Figures (5a,b and 6a,b) show the effects of different transfer block sizes as well as varying the number of clients, this time for a smaller, 20-server GPFS file system. Figure 5a shows the performance of reading a single file, while Fig. 5b shows the result of reading the same amount of data split into separate files, one file for each client process. Figures 6a and 6b show the corresponding results for writing. First of all, note that the size of the individual transfers ("block size" in the plots) doesn't matter very much, except for very small transfers (< 8 kB). (However, as will be seen later, transfer size has a very strong effect in non-segmented access patterns.) Secondly, note that there is not a great deal of

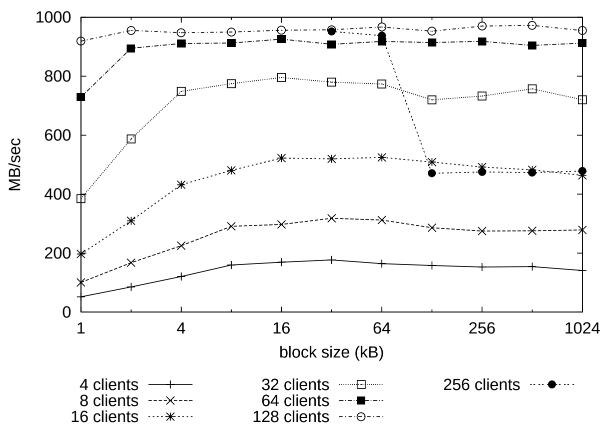

Figure 5a. Reading a single file (20 srvs)

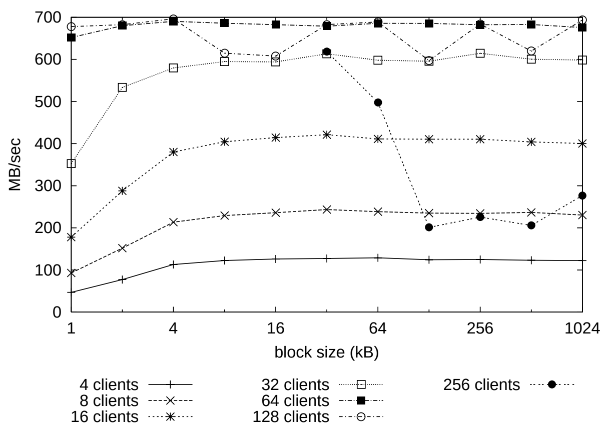

Figure 6a. Writing a single file (20 srvs)

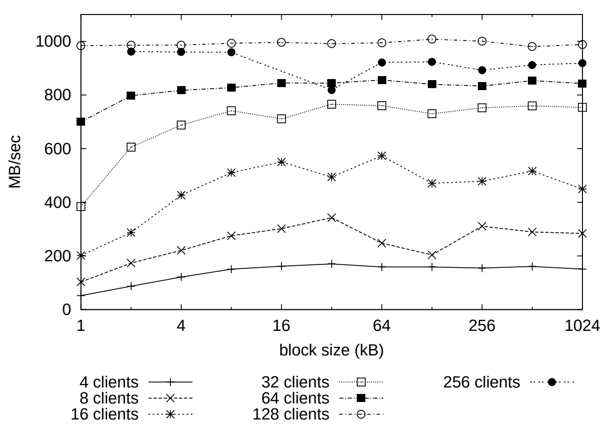

Figure 5b. Reading separate files (20 srvs)

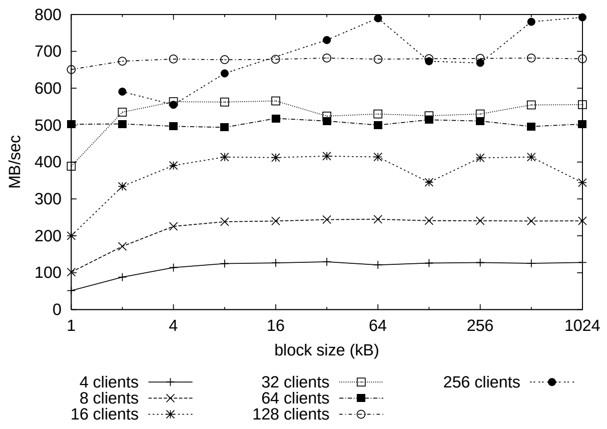

Figure 6b. Writing separate files (20 srvs)

difference in the aggregate performance between accessing a single file or separate files, until one reaches 128 kB blocks with 256 clients, where single-file access drops dramatically.

In the previous experiments only a single processor on each 4-processor client node participated in the file accesses. The following four graphs show what happens when more of the processors are used on each of 4, then 32 nodes (Figs. 7a-7b and 8a-8b, respectively). These data show that there is little to be gained from using more than one processor per node to access GPFS, with the possible exception of reads in small jobs. If the GPFS code in the-client were made to run faster (e.g., perhaps by eliminating the intermediate copying of data between the application's buffer and GPFS's pagepool), one could expect that performing I/O in 2,3, or 4 client processors per node would show increased performance. However, there would be little point in doing so since most jobs will use enough client nodes to saturate the capacity of the servers, even using a single I/O process per node.

The performance of GPFS for different transfer sizes in the round-robin access pattern using 80 clients and 20 servers is shown in Fig. 9. Note first that performance is extremely poor for anything smaller than the stripe width of 256 kB. This is as one would expect, given the client-side caching in GPFS. Application programs should definitely avoid this combination of nonsequential access pattern and small block size, or use a higher-level library such as MPI-IO, which can redistribute the data via collective parallel I/O functions, passing the resultant larger blocks to GPFS in place of the many separate smaller blocks. At larger block sizes, note that the read performance is about 1/3 of the peak speed achievable with the segmented access pattern (e.g., compare to the 20-server point in Fig. 10). Writes fare relatively better at intermediate block sizes, achieving 366 to 553 MB/sec (compared to around 700 MB/sec peak for the segmented pattern), before dropping precipitously with 1 MB blocks.

Figure 10 shows how the peak throughput rates of GPFS scale along with the number of servers. Note that writes scale almost perfectly with the 40 MB/sec "ideal",

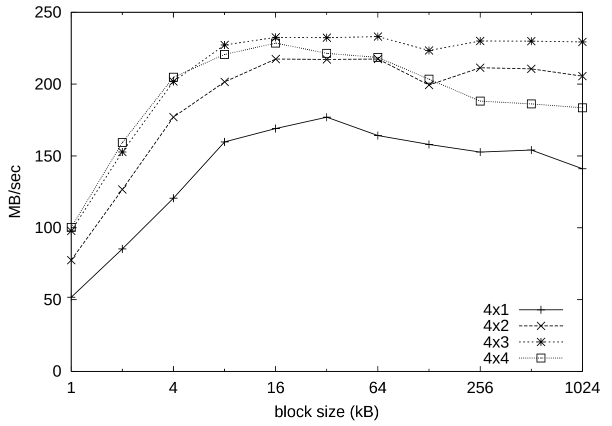

Figure 7a. 4 nodes read, 1-4 tasks / node

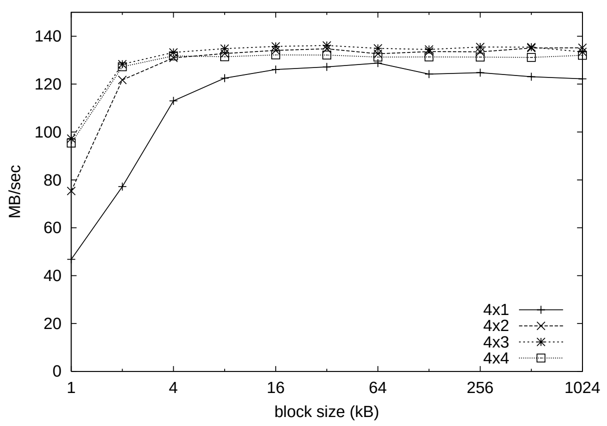

Figure 8a. 4 nodes write, 1-4 tasks / node

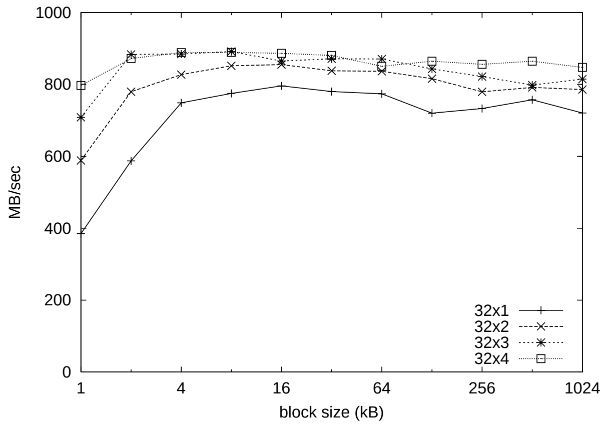

Figure 7b. 32 nodes read, 1-4 tasks / node

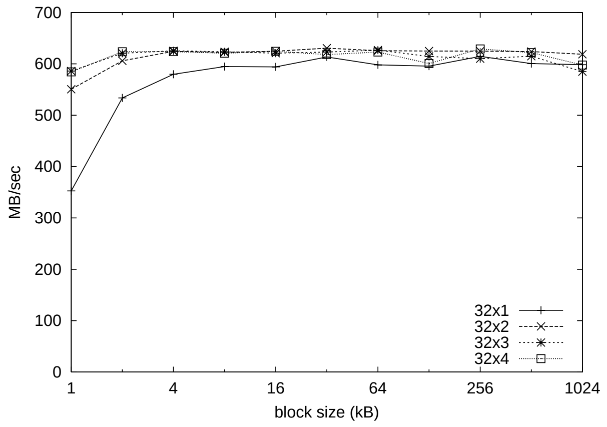

Figure 8b. 32 nodes write, 1-4 tasks / node

demonstrating sustained throughputs over 2 GB/sec at the high end; reads are even better. Of course, these peak rates were obtained with segmented access patterns, and with well-chosen block sizes and client:server ratios. (Note: the data for 58 servers were obtained from IBM [23].) Theoretically, if the proper balance is maintained between computational nodes and I/O server nodes, GPFS should scale up to over 4 GB/sec for the maximum 512-node system, but this has not been demonstrated.

5. Conclusion

We find that GPFS is capable of excellent aggregate throughput for per-process-sequential (i.e., segmented) access patterns, scaling well with the number of servers up to more than 2 GB/sec. Moreover, the familiar standard POSIX interface is adequate to achieve this performance.

To get the best performance from GPFS v1.2, programs should use the segmented access pattern, and should keep the client:server ratio below 6. We expect that improvements to GPFS's control of data flow between clients and servers would eliminate the degradation of performance with higher client:server ratios. At least in its current implementation, GPFS should not be used for nonsequential access patterns when the transfer size is less than the GPFS stripe width (256 kB). In that case, higher-level I/O libraries such as MPI-IO running on top of GPFS should give better performance. Alternatively, one might choose to write a separate file for each process.

For file system designers, we consider GPFS to be a good example of a scalable and trustworthy high-performance parallel file system with a standard user interface. However, we would prefer to see nonsequential access patterns perform better (though not at the expense of lower performance for sequential patterns).

One important improvement we would like to see is in the area of flow control; this would not increase peak throughput rates, but would maintain them at high client:server ratios. Another possible improvement would be to remove or reduce the impact of token management used in enforcing POSIX's atomicity semantics by providing the user the option of turning it off; i.e., the throughput might be improved if the application program could assert that no overlapping writes will occur.

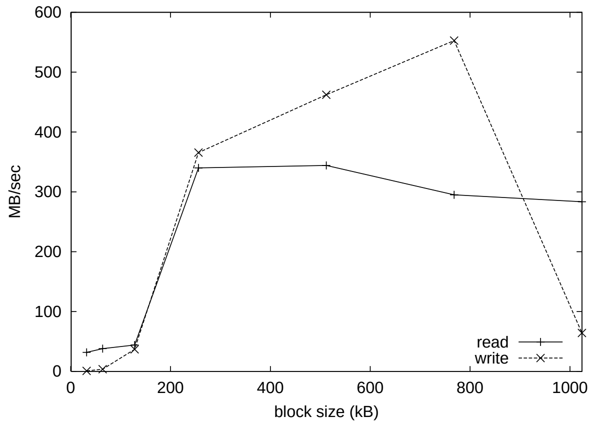

Figure 9. Round-robin pattern

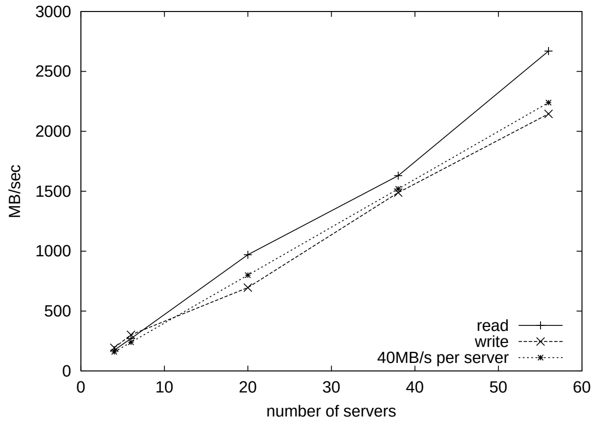

Figure 10. Scaling with number of servers

References

[1] Alice Koniges, Parallel Computer Architecture, in *Industrial Strength Parallel Computing*, Morgan Kaufmann, 2000.

[2] D. Culler and J. P. Singh, *Parallel Computer Architecture: A Harware/Software Approach*. Morgan Kaufmann, 1998.

[3] M. Barrios, T. Jones, S. Kinnane, M. Landzettel, S. Al-Safran, J. Stevens, C. Stone, C. Thomas, U. Troppens, *Sizing and Tuning GPFS*. IBM Corp, SG24-5610-00, 1999, at http://www.redbooks.ibm.com/.

[4] M. Barrios et al., *GPFS: A Parallel File System*. IBM Cor., SG24-5165-00, 1998, http://www.redbooks.ibm.com/.

[5] W. Gropp and S. Huss-Lederman, *MPI the Complete Reference: The MPI-2 Extensions*. MIT Press, 1998.

[6] E. Smirni, R. Aydt, A. Chien, D. Reed, "I/O Requirements of Scientific Applications: An Evolutionary View," HPDC 96

[7] N Nieuwejaar, D Kotz, A Purakayastha, C Ellis, M Best. “File-Access Characteristics of Parallel Scientific Work loads”. *IEEE Tran. Par. and Dist. Sys.*, 7(10), Oct 1996.

[8] Yong Eun Cho, *Efficient Resource Utilization for Parallel I/O in Cluster Environments*, PhD Thesis: U. Illinois, 1999, and references therein.

[9] White, S. W. and Dhawan,S., "POWER2:Next generation of the RISC System/6000 family," *IBM J. Res. Develop.*, 38, No. 5, 493-502, Sept 1994.

[10] C. B. Stunkel, et al, The SP2 High-Performance Switch, *IBM Systems Journal*, 34, No. 2, 1995

[11] Frank Johnston and Bernard King-Smith, “SP Switch Performance”, IBM Corp., Aug 1999. http://www.rs6000 .ibm.com/resource/technology/spswperf.html

[12] S. Baylor and C. Wu. Parallel I/O Workload Characteristics Using Vesta. In R. Jain et al, eds, *Input/Output in Parallel and Distributed Computer Systems*. Kluwer Academic, 1996.

[13] K. Seamons and M. Winslett, “Multidimensional array I/O in Panda 1.0.” *J. of Supercomputing*, 10, 1-22 (1996).

[14] E. Miller and R. Katz, “RAMA: An easy-to-use, high-performance parallel file system”. *Parallel Comp.*, 23, 1997.

[15] N. Nieuwejaar and D. Kotz, “The Galley parallel file system.” *Parallel Comp.*, 23, 447-476 (1997).

[16] G. Gibson et al., “The Scotch Parallel Storage Systems.” Proc. IEEE CompCon, 1995.

[17] S. Moyer and V. Sunderam, “PIOUS: A scalable parallel I/O system for distributed computing environments.” Proc. Scalable High-Performance Comp. Conf., pp. 71-78, 1994.

[18] J. Huber, C. Elford, D. Reed, A. Chien, and D. Blumenthal, “PPFS: A high performance portable parallel file system.” ACM Int. Conf. Supercomputing, 1995.

[19] R. Thakur, A. Choudhary, R. Bordawekar, S. More, S. Kuditipudi, “PASSION: Optimized I/O for parallel applications.” *IEEE Computer*, 29(6):70-78, June 96.

[20] S. Garg, TFLOPS PFS: Architecture and design of a highly efficient parallel file system.” Proc. ACM/IEEE SC98.

[21] M. Holton and R. Das, “XFS:A next generation journalled 64-bit filesystem with guaranteed rate I/O.” SGI Corp.http://www.sgi.com/Technology/xfs-whitepaper.html.

[22] IBM, “RS/6000 HACMP for AIX White Paper.” http:// www.rs6000.ibm.com/resource/technology/ha420v.html.

[23] Jim Wyllie,“SPsort: How to sort a terabyte quickly.” http://www.almaden.ibm.com/cs/gpfs-spsort.html.