Preprint
UCRL-JC-135828

Performance of the IBM General Parallel File System

T. Jones, A. Koniges and R.K. Yates

This article was submitted to
International Parallel and Distributed Processing Symposium
Cancun, Mexico
May 1-5, 2000

September 27, 1999

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

This report has been reproduced
directly from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62, Oak Ridge, TN 37831
Prices available from (423) 576-8401
http://apollo.osti.gov/bridge/

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.,
Springfield, VA 22161
http://www.ntis.gov/

OR

Lawrence Livermore National Laboratory
Technical Information Department's Digital Library
http://www.llnl.gov/tid/Library.html

Performance of the IBM General Parallel File System

Terry Jones, Alice Koniges, R. Kim Yates
Lawrence Livermore National Laboratory

Abstract

Experimental performance analysis is a necessary first step in input/output software tuning and real-time environment code performance prediction. We measure the performance and scalability of IBM's General Parallel File System (GPFS) under a variety of conditions. The measurements are based on a set of benchmark codes 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.

[Note to reviewers: the text is under the 12-page limit, but the page count is long because of the many large figures. We could reformat to fit in fewer pages if needed.**]**

Introduction

Large-scale scientific computations such as those associated with ASCI[1] and other projects continue to stretch the limits of computational power. I/O has become a bottleneck in application performance as processor speed skyrockets often leaving storage hardware and software struggling to keep up. Parallel computing, whether it be on large-scale systems such as the IBM SP/6000 and the Cray T3E or on a heterogeneous computational grid, is generally recognized as the only viable solution to high performance computing problems. Thus, parallel file systems must be developed that allow the applications to make optimum use of their available processor parallelism. Additionally, hybrid architectures, more complicated than simple Distributed Memory Parallel (DMP), that have Shared Memory Parallel[2] (SMP) boxes connected as nodes of a DMP machine are becoming the norm rather than the exception[1,2]. Such configurations lead to more interesting combinations of overlapping communication, I/O, and computation. To deal with these hybrid architectures and other 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 file system calls. In addition, increased flexibility for parallel applications can be obtained 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

[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/).

[2] SMP is often used to denote Symmetric MultiProcessor instead of shared memory parallel [3,4].

* This work was performed under the auspices of the U.S. Department of energy by Lawrence Livermore National Laboratory under contact number W-7405-Eng-48.

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. I/O requirements and workload characterization

Recently a computing rate of 2.14 Tflop/s (10^{12} floating-point operations per second) was achieved on a linear algebra benchmark calculation 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 capability of approximately 7.3 GB/sec (where 1 GB is 1024^3 bytes) is needed. Another common rule to estimate how well a system is balanced, based on past experience, says that an application which fills the entire machine will store half of total memory once per hour, and that this should take no more than five minutes in every hour. 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[3] file systems.

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. Out-of-core read/writes for problems which do not fit the memory
4. Regular snapshots of the computation's progress
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.

Moreover, characterization of a file system workload is subject to many outside influences such as scheduling queues, file size limits, etc. To get an idea of the random nature of the file system load in a production environment, consider the following snapshots from a 3-week period on the LLNL Blue machine. This machine is a 336-node IBM RS/6000 SP with four 332 MHz processors per node. The local file system has 3 Tbytes and the GPFS file system has 20 Tbytes spread over 240 RAID disks. The data on GPFS disk activity are collected from the AIX operating system calls to the UNIX function *iostat* summed over 5 minute intervals. These numbers are then plotted as a function of time of day in Fig. 1. (The sporadic access may be due to the novelty of the GPFS file system for our users, and may not be a fair representation of what will be a normal workload in the future.) Since the GPFS file system stripes data across the disks, the pattern is fairly representative of the entire file system. Similar patterns were observed when the aggregate performance of 12 disks served by a single node is plotted. For this particular disk, the maximum read/write

[3] 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 separate systems per partition would be more useful. It is not possible for a GPFS file system to span the three partitions that form SKY.

rates sustained over a five minute period were 3.84 and 2.75 Mbytes/sec respectively. We can extrapolate the data to the full set of 240 disks to estimate aggregate peak performance, although such estimates might vary greatly based on traffic across the switch.

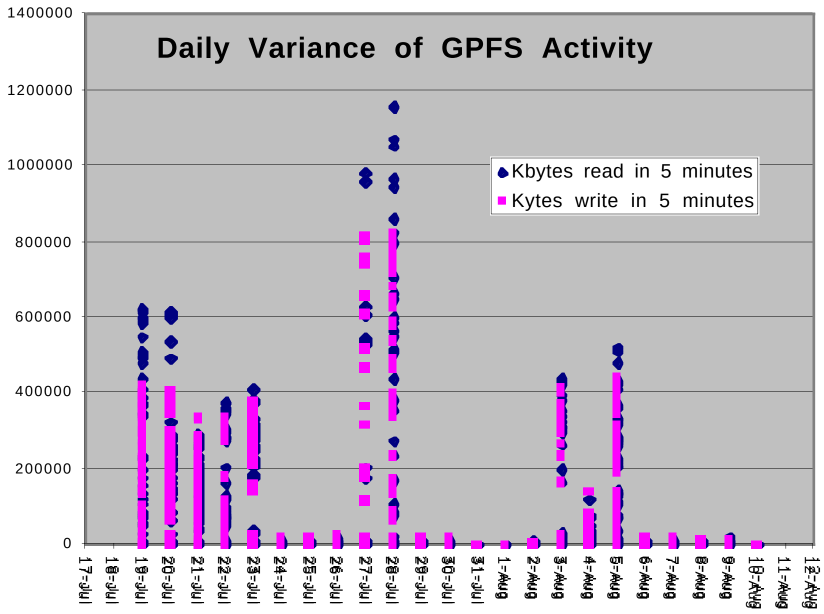


Figure 1

Finally, we cannot neglect the question of reliability, since if a file system breaks its throughput is 0. To achieve gigabyte-per-second performance in a file system 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 whole file 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 full Unix kernel and is autonomous. The nodes are connected via a proprietary network technology that permits each 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]. Two factors play heavily into the GPFS architecture. First, the design assumes that nodes which have the file system mounted will have high-throughput connections to nodes which have the disks attached. Second, the GPFS design employs a great deal of parallelism. 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]

But the primary feature that sets GPFS apart from other file systems is the degree of parallelism in its design. Like distributed file systems (e.g. NFS, AFS, DFS), multiple compute nodes may mount the file system. Unlike distributed file systems, GPFS permits the output to be striped over a number of I/O nodes. By striping across multiple nodes and multiple disks, the GPFS designers sought to provide a truly scalable file system.

There has been much interesting 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 immediately 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, on the other hand, 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. Much of the services necessary for GPFS are provided by a persistent GPFS daemon called mmfsd. Among the more important services provided by mmfsd are: (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* service which ensures that a stripe group manager and token manager server are operational and that a quorum exists.

These services are distributed among the nodes of an RS/6000 SP system in the way illustrated by Figure 2. Note that some services are replicated throughout the machine, whereas other services are implemented within a single mmfsd instance.

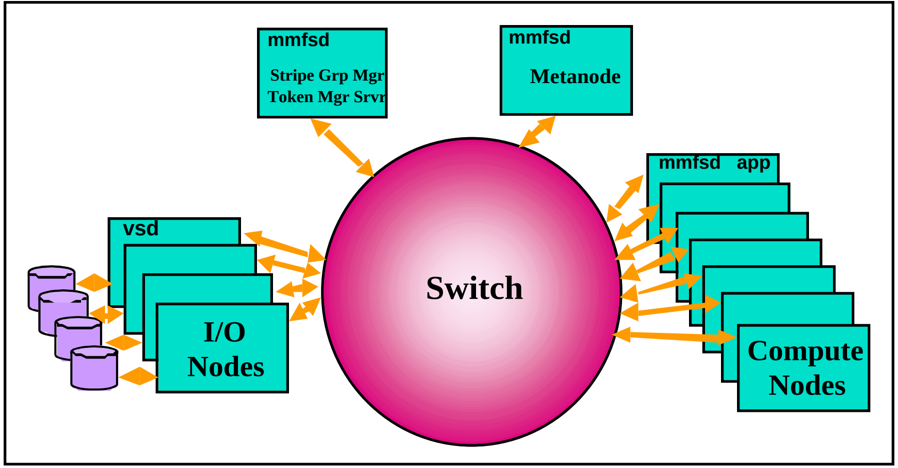


Figure 2

Each of the nodes dedicated to running parallel applications will have an mmfsd daemon present to mount the file system and perform access. This mmfsd is responsible for actually performing the reads and writes performed on that node. There is one mmfsd instance per SMP 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. 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 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.

As alluded to earlier, token management is performed as a distributed service by a token manager. 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 read and write 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 100% from node A or 100% from node B: the overlapped region cannot contain a mish-mash of contents from both nodes scrambled together

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 fails. The following assumes the application has successfully opened the file for writing.

Figure 3 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 token which permits write access 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, a buffer is made available by writing out the oldest buffer to disk

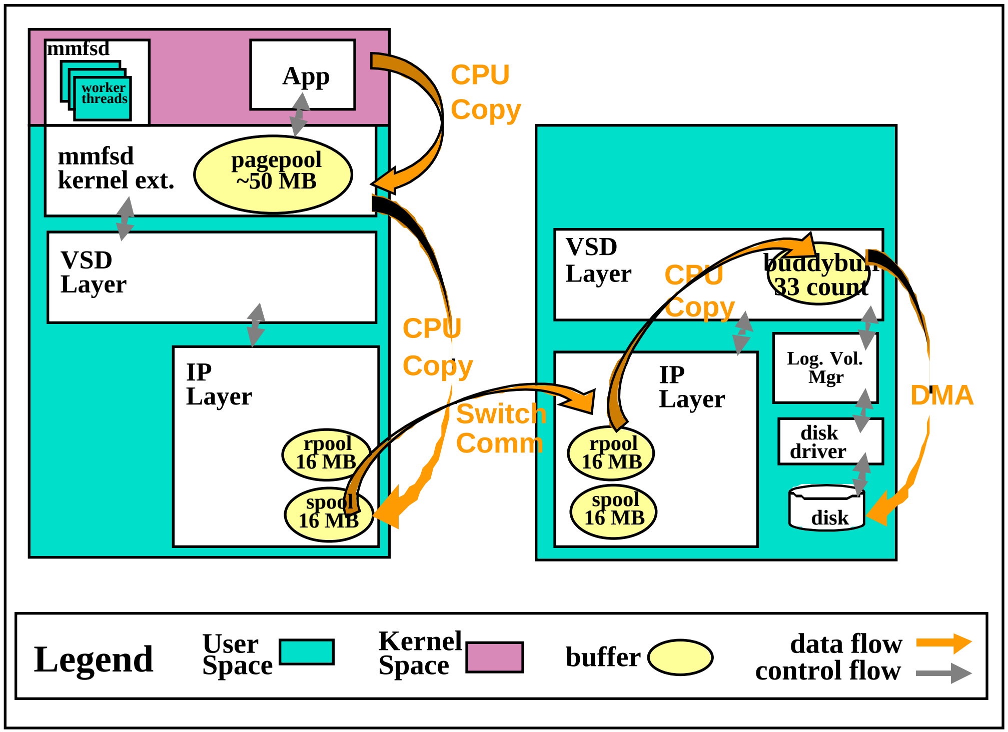


Figure 3

- 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 a second time from the pagepool buffer 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 buddy buffer is used to reassemble the large chunk of data from the packets. If a buddy buffer is not immediately available, the request is queued and the data remains 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.

Data flow for reads is similar. Of course the data travels in the opposite direction. A second difference is that reads must block until the data completes the entire path from disk to application buffer, whereas writes can continue once the data has been copied into the local pagepool. Finally, 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 nodes. Not only does this scalability 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 and eliminating serialization. GPFS's file striping mechanisms ensure metadata and data are managed in a distributed manner to avoid hot spots. Traditional 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 (mmap, munmap, and msync). 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 version 1.2 also 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. For example, if a GPFS file system is constructed with ***1000*** disks and each disk can write at a given bandwidth ***x***, the maximum bandwidth of the GPFS file system by disk limitation is ***1000x.*** 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 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 maximum 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. That is, given a small number of GPFS file systems, 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 for brevity's sake 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 process 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 aggregate 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 in a similar fashion, 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 here are illustrated in Fig. 4. 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.

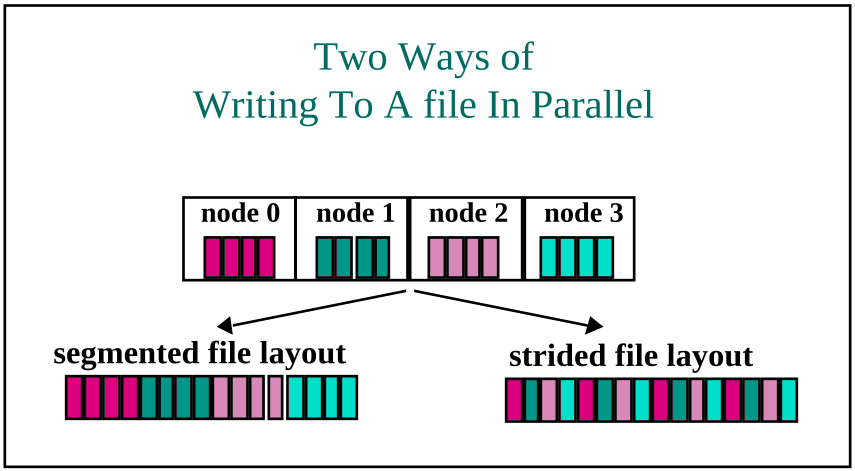


Figure 4

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 5 shows that performance is highest when the ratio of client processes to VSD server nodes is near 4:1. (Though the nodes of the SP/6000s running these experiments have four processors per node, we ran only one client task per node, except where otherwise noted.) When the client:server ratio is too low the servers are starved for data; when the ratio 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. However, note that in the middle of the curves the aggregate throughput is quite high: around 1500 MB/sec [4] for writes and 1600 MB/sec 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, at least for large files.

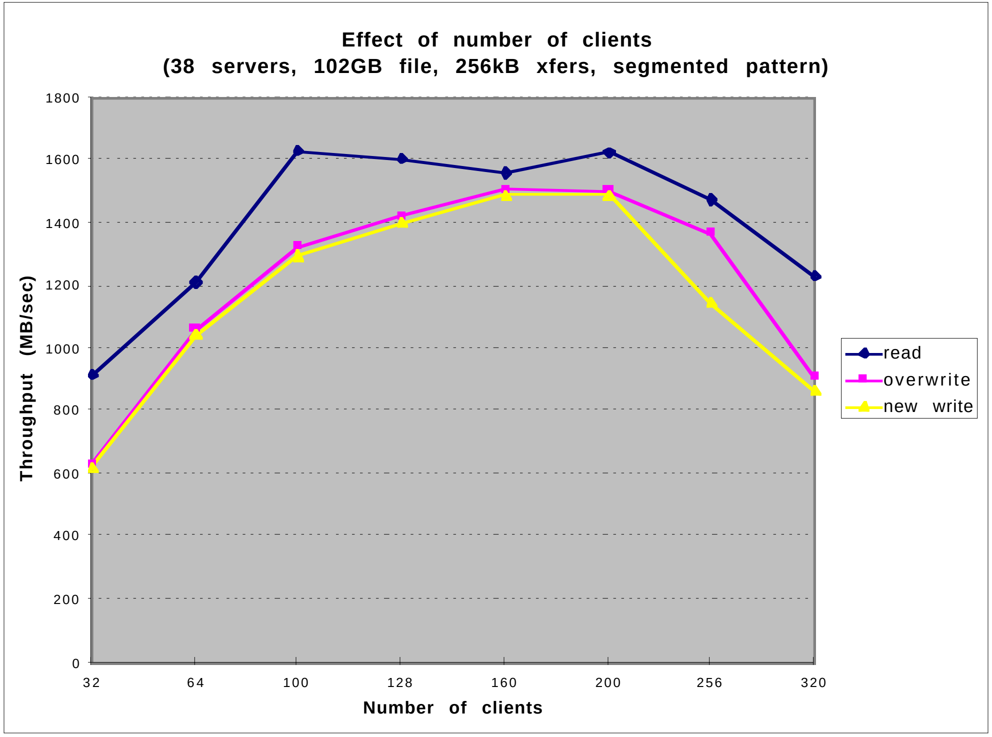
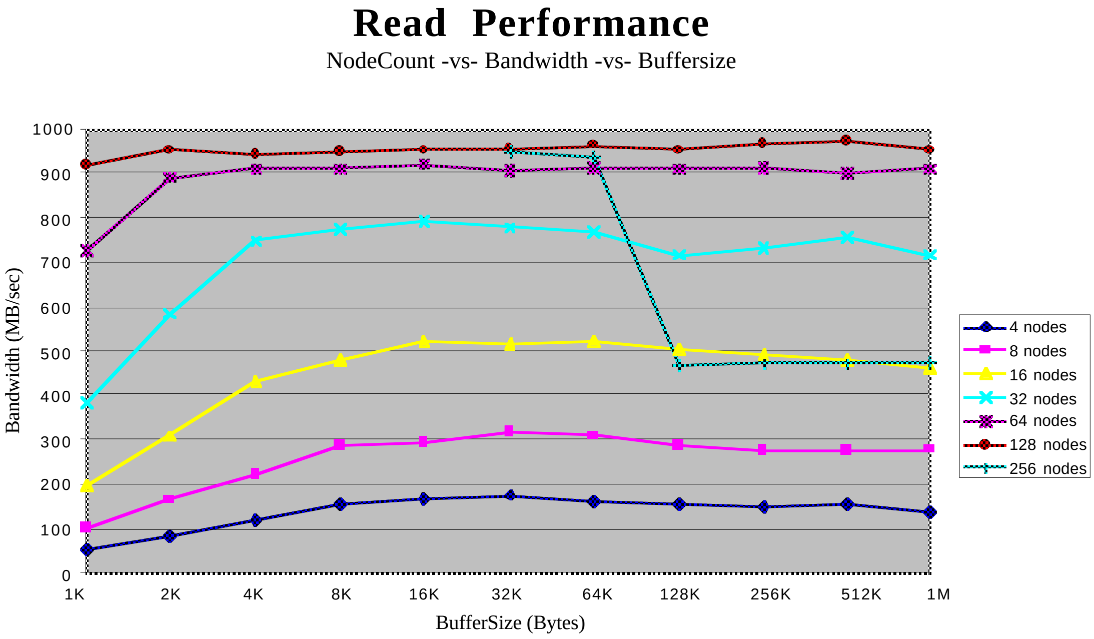


Figure 5

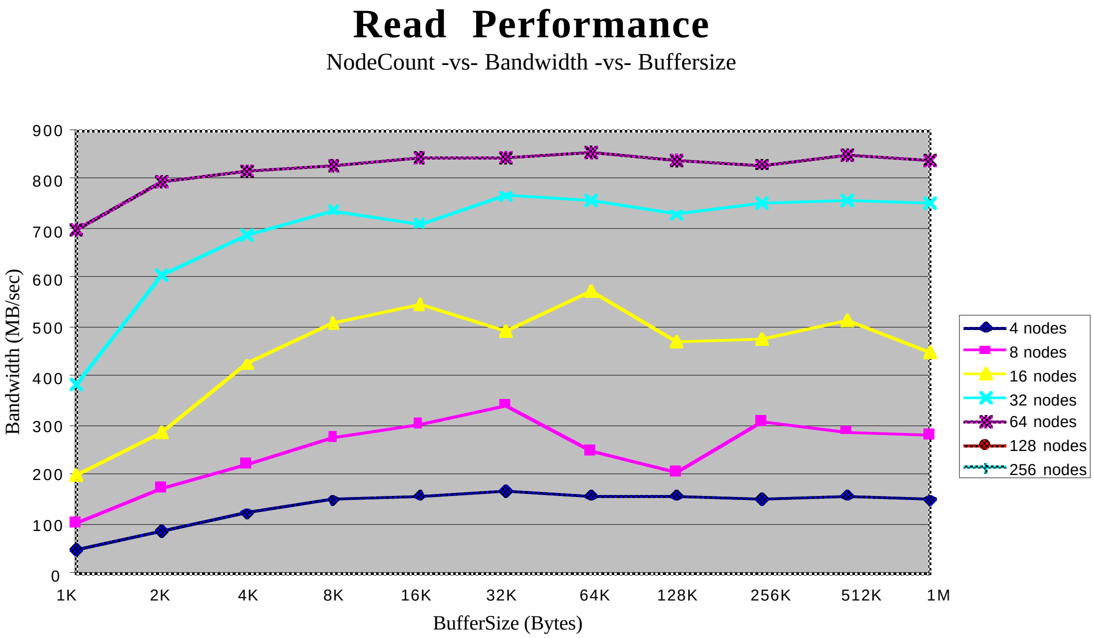
The next four Figures (6a,b and 7a,b) show the effects of different transfer block sizes as well as varying the number of clients, this time for a smaller, 20-server GPFS file system. Figure 6a shows the performance of reading a single file, while Fig. 6b shows the result of reading the same amount of data split into separate files, one file for each client process. Figures 7a and 7b show the corresponding results for writing. First of all, note that the size of the individual transfers (“buffer 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 difference in the aggregate performance between accessing a single file or separate files, with single-file access being somewhat faster, especially for writing with a large number of clients. Figures 6a and 7a also show the effect of poor flow control for large client:server ratio and large transfer sizes. **[Note to reviewers:** the data for 128 and 256 clients is missing from Figs. 6b and 7b. We expect to generate that data before the final deadline, with no alteration of the overall conclusions.**]**

[4] In our notation, 1k=1024, 1M=1024k, and 1G=1024M.



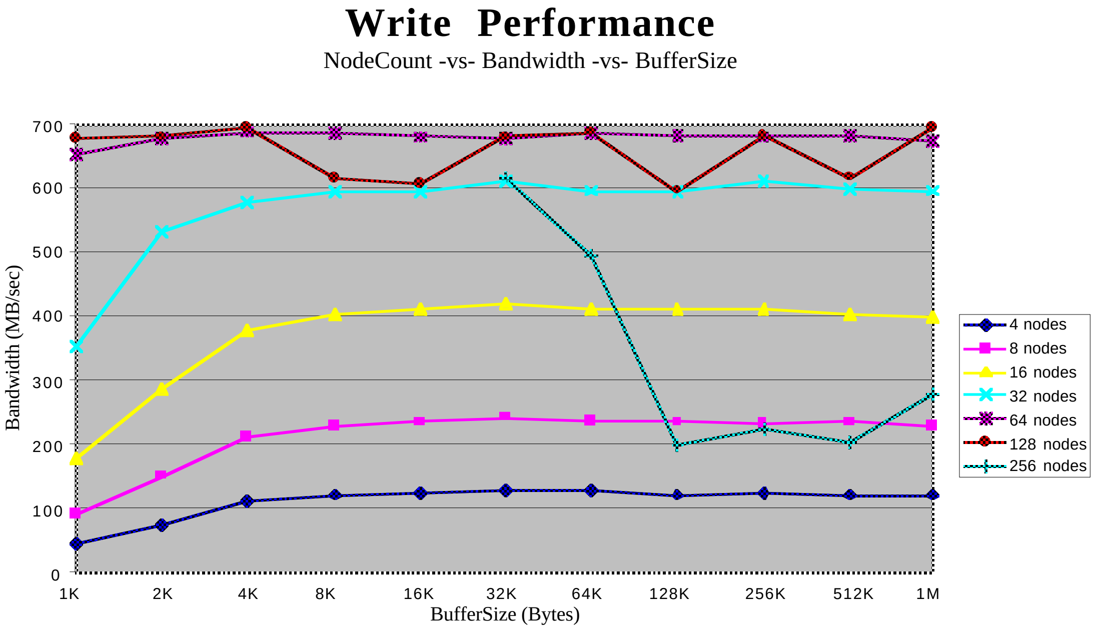
Filesize: 1GB per Task (Segmented) HW Configuration: Twenty Silver VSD Servers with 2 Campbell
GPFS: version=1.2.0.3, with PTF 5+efixes adapters with 6x(4+P RAID5) per adapter
Program: ior_gpfs (redbook version) SW Configuration: ppool=50, mpool=16, buddy=130,thrds=24,6,12

Figure 6a (reading a single file)



Filesize: 1GB per Task (Segmented) HW Configuration: Twenty Silver VSD Servers with 2 Campbell
GPFS: version=1.2.0.3, with PTF 5+efixes adapters with 6x(4+P RAID5) per adapter
Program: ior_gpfs (redbook version) SW Configuration: ppool=50, mpool=16, buddy=130,thrds=24,6,12

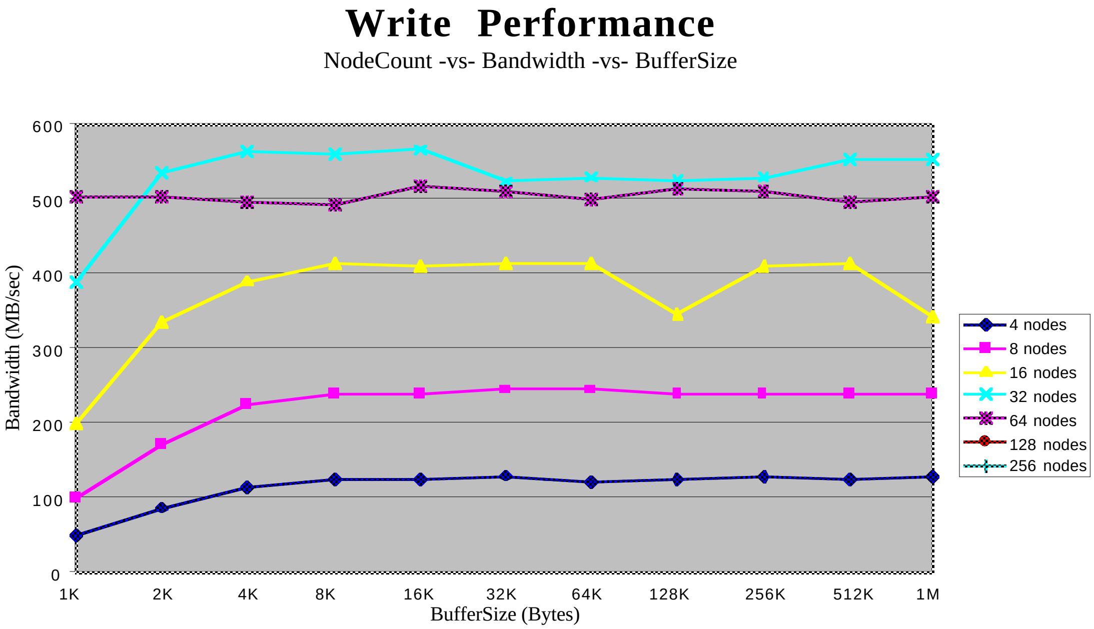
Figure 6b (reading separate files)



Filesize: 1GB per Task (Segmented)
GPFS: version=1.2.0.3, with PTF 5+efixes
Program: ior_gpfs (redbook version)

HW Configuration: Twenty Silver VSD Servers with 2 Campbell adapters with 6x(4+P RAID5) per adapter
SW Configuration: ppool=50, mpool=16, buddy=130,thrds=24,6,12

Figure 7a (creating & writing a single file)

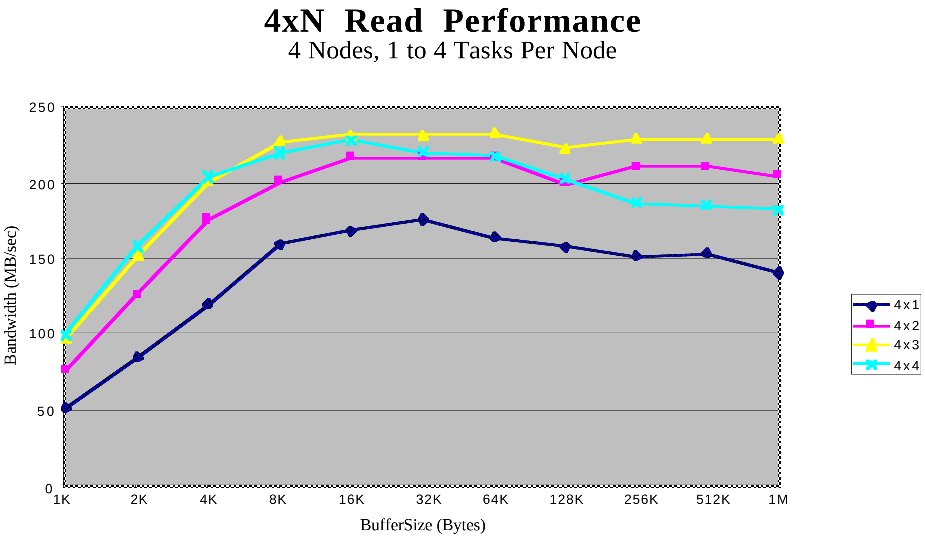


Filesize: 1GB per Task (Segmented)
GPFS: version=1.2.0.3, with PTF 5+efixes
Program: ior_gpfs (redbook version)

HW Configuration: Twenty Silver VSD Servers with 2 Campbell adapters with 6x(4+P RAID5) per adapter
SW Configuration: ppool=50, mpool=16, buddy=130,thrds=24,6,12

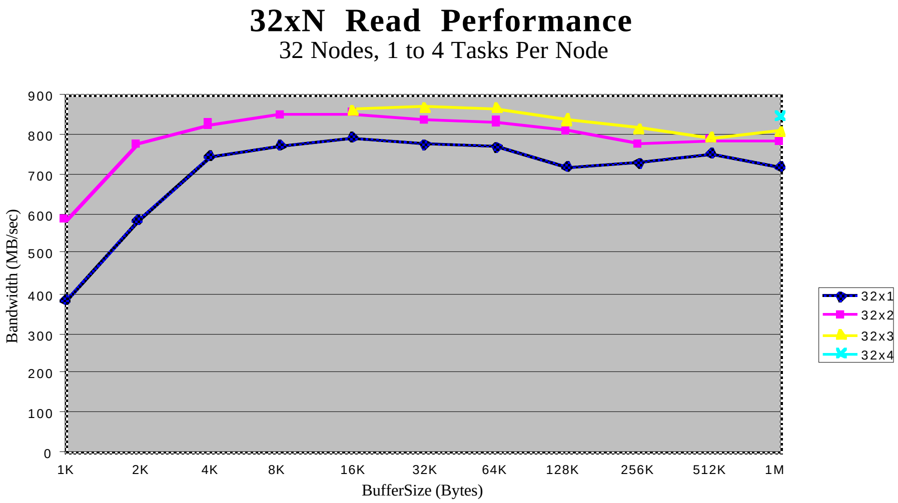
Figure 7b (creating and writing separate files)

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. 8a-8b and 9a-9b, 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.



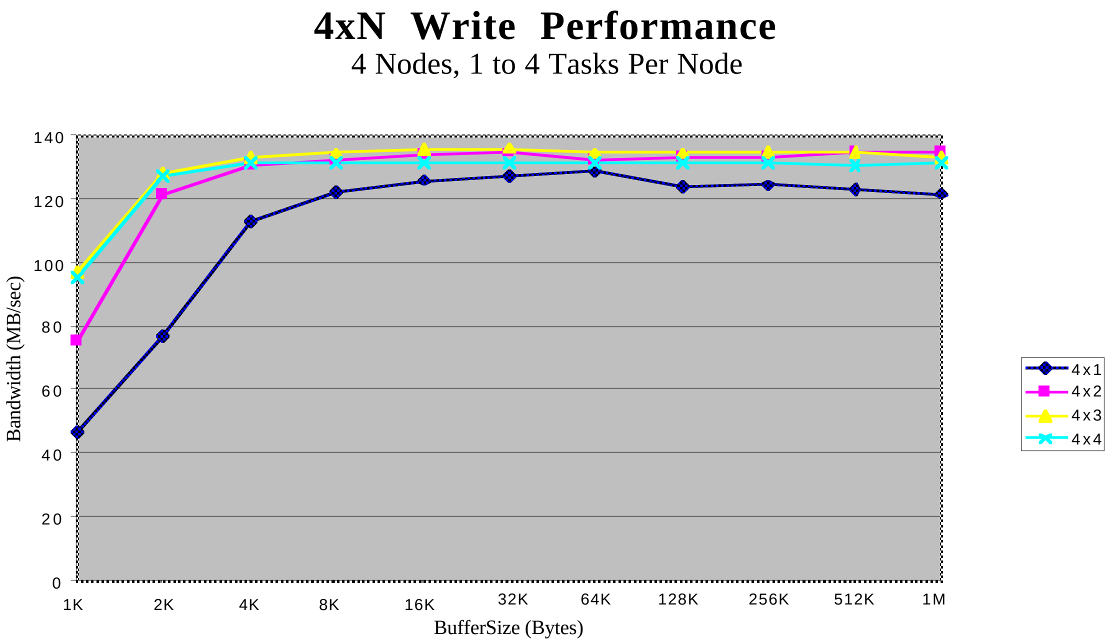
Filesize: 1GB per Task (Segmented) HW Configuration: Twenty Silver VSD Servers with 2 Campbell
GPFS: version=1.2.0.3, with PTF 5+efixes adapters with 6x(4+P RAID5) per adapter
Program: ior_gpfs (redbook version) SW Configuration: ppool=50, mpool=16, buddy=130,thrds=24,6,12

Figure 8a



Filesize: 1GB per Task (Segmented) HW Configuration: Twenty Silver VSD Servers with 2 Campbell
GPFS: version=1.2.0.3, with PTF 5+efixes adapters with 6x(4+P RAID5) per adapter
Program: ior_gpfs (redbook version) SW Configuration: ppool=50, mpool=16, buddy=130,thrds=24,6,12

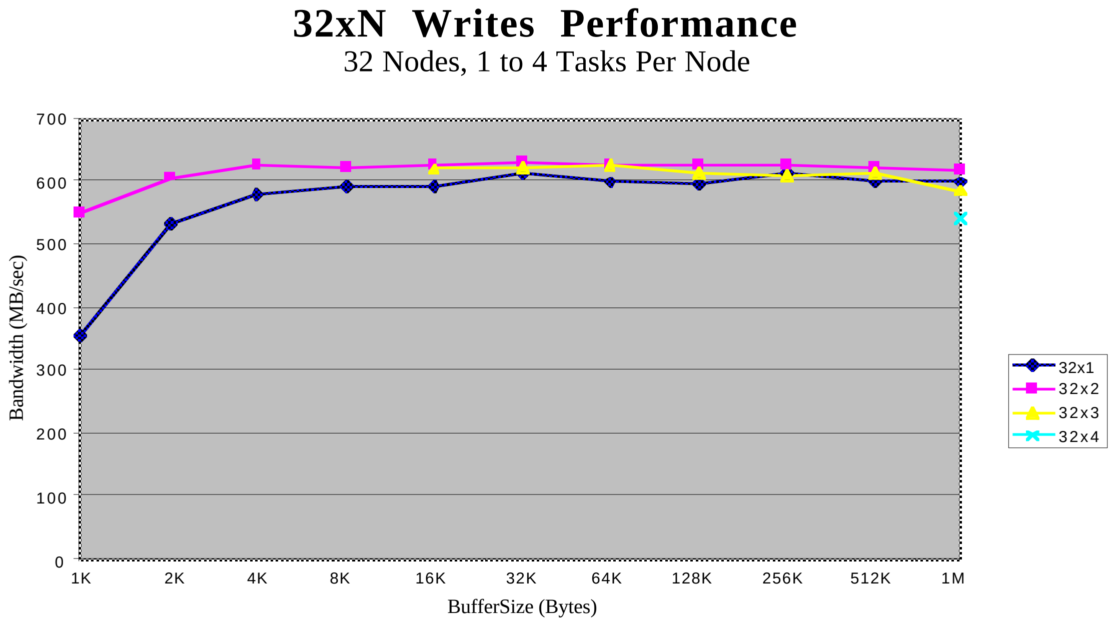
Figure 8b



Filesize: 1GB per Task (Segmented)
GPFS: version=1.2.0.3, with PTF 5+efixes
Program: ior_gpfs (redbook version)

HW Configuration: Twenty Silver VSD Servers with 2 Campbell adapters with 6x(4+P RAID5) per adapter
SW Configuration: ppool=50, mpool=16, buddy=130,thrds=24,6,12

Figure 9a



Filesize: 1GB per Task (Segmented)
GPFS: version=1.2.0.3, with PTF 5+efixes
Program: ior_gpfs (redbook version)

HW Configuration: Twenty Silver VSD Servers with 2 Campbell adapters with 6x(4+P RAID5) per adapter
SW Configuration: ppool=50, mpool=16, buddy=130,thrds=24,6,12

Figure 9b

The performance of GPFS for different transfer sizes in the round-robin access pattern is shown in Fig. 10. The best performance is half or less of what would be expected using a segmented pattern. More importantly, the performance is reasonable only when the transfer size is the same as or double the GPFS stripe width. Performance is extremely poor for anything smaller than the stripe width, especially for writes. 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.

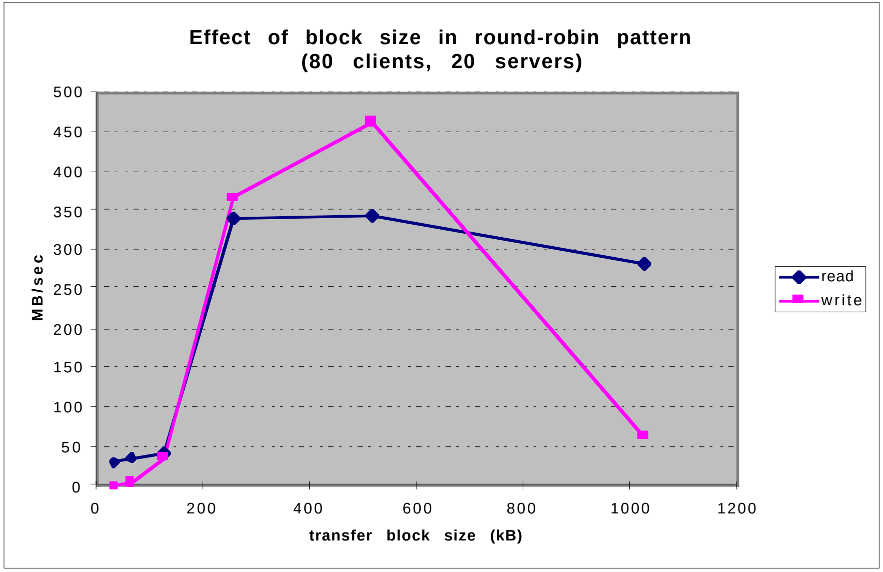


Figure 10

Figure 11 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” line shown all the way from 4 to 58 servers, 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].)

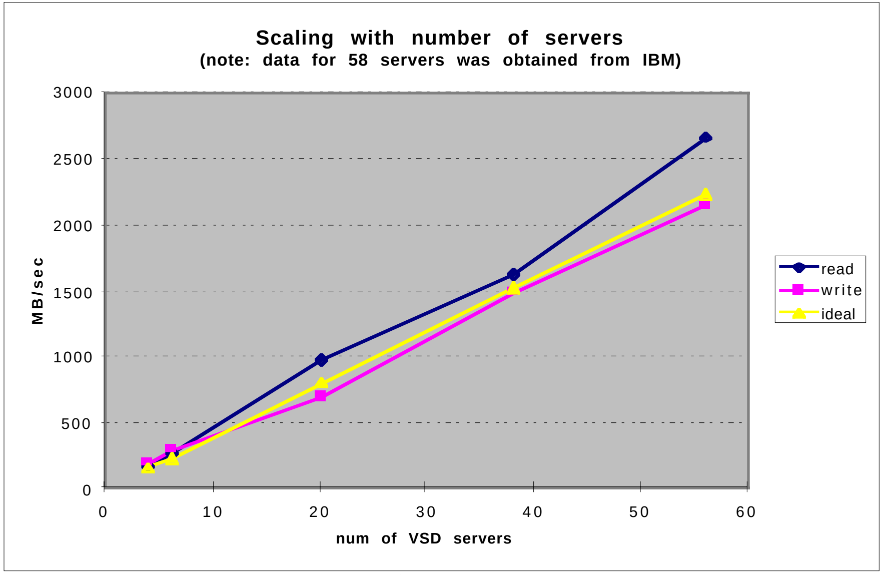


Figure 11

Finally, Fig. 12 is one example of a set of experiments that were done to see how GPFS's various tunable parameters affect performance. In this case we varied the number of write-behind threads on a 38-server system. Many other parameters are tunable, but we do not show them here. For more information, see [3,4].

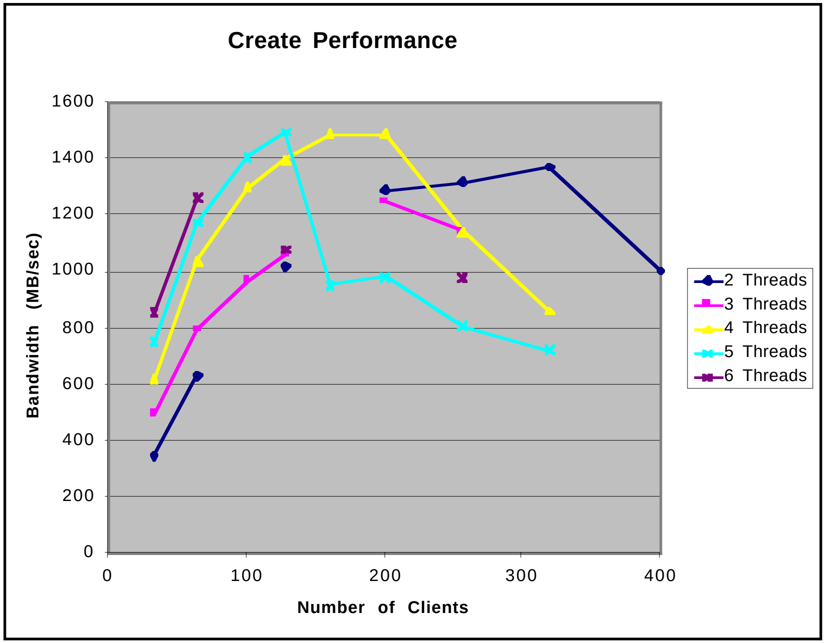


Figure 12

4. 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 user interface is adequate to achieve this performance.

To get the best performance from GPFS, developers of application 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 an excellent 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). And though we have seen GPFS scale well up to 2 GB/sec, it's not clear how much higher it can go. One important improvement we would like to see, as already remarked, 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.

References

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

[2] David E. Culler and Jaswinder Pal Singh, *Parallel Computer Architecture: A Harware/Software Approach*. Morgan Kaufmann, 1998.

[3] M. Barrios, Terry Jones, Scott Kinnane, Mathis Landzettel, Safran Al-Safran, Jerry Stevens, Christopher Stone, Chris Thomas, Ulf Troppens, *Sizing and Tuning GPFS*. IBM Corp, SG24-5610-00, 1999, available at http://www.redbooks.ibm.com/.

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

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

[6] Evgenia Smirni, Ruth A. Aydt, Andrew A Chien, Daniel A. Reed, "I/O Requirements of Scientific Applications: An Evolutionary View," HPDC 96

[7] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. Ellis, and M. Best. “File-Access Characteristics of Parallel Scientific Workloads”. *IEEE Tran. on Par. and Dist. Sys.*, 7(10):1075-1089, 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, Bernard King-Smith, “SP Switch Performance”, IBM Corp., Aug 1999. (available as 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, J. Werth, and J. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, chapter 7, pages 167-185. Kluwer Academic Publishers, 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, 419-446 (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. 1998 ACM/IEEE SC98 Conf.

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

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

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