

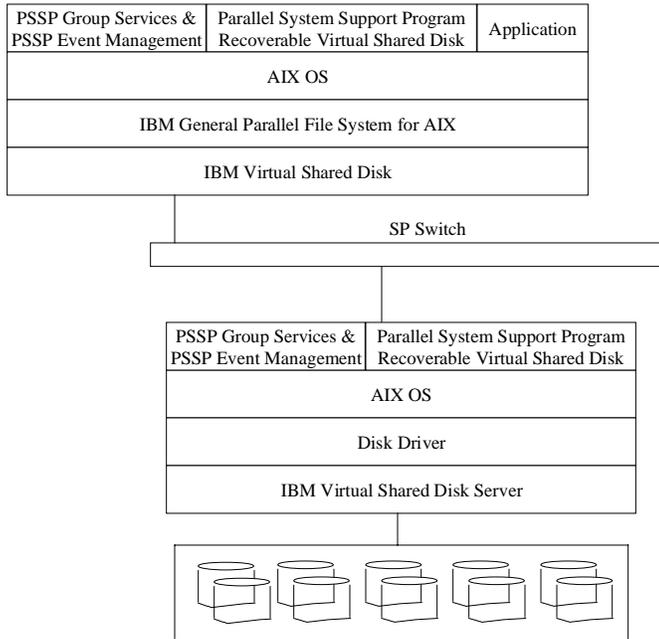
General Parallel File System (GPFS) 1.4 for AIX Architecture and Performance – November 2001*

*Dominique Heger (dheger@us.ibm.com)
Gautam Shah (gautam@us.ibm.com)*

IBM General Parallel File System (GPFS) for AIX® allows users shared access to files that can span multiple disk drives on multiple nodes. GPFS allows parallel applications simultaneous access to either the same or different files from any node in the GPFS nodeset (a nodeset is defined as a group of nodes that run the same level of GPFS). It is designed to provide a common file system abstraction for data that is being shared among all the nodes in a cluster and allows applications to easily access files, utilizing standard UNIX® file system interfaces. Most UNIX file systems are designed for a ‘single server’ environment. In such an environment, adding additional file servers typically does not improve specific file access performance. GPFS is designed to provide high performance by ‘striping’ I/O across multiple disks, high availability through logging, replication, as well as high scalability (by utilizing multiple servers) through the SP™ Switch and the SP Switch2 [2] [3].

New Features in General Parallel File System 1.4 for AIX

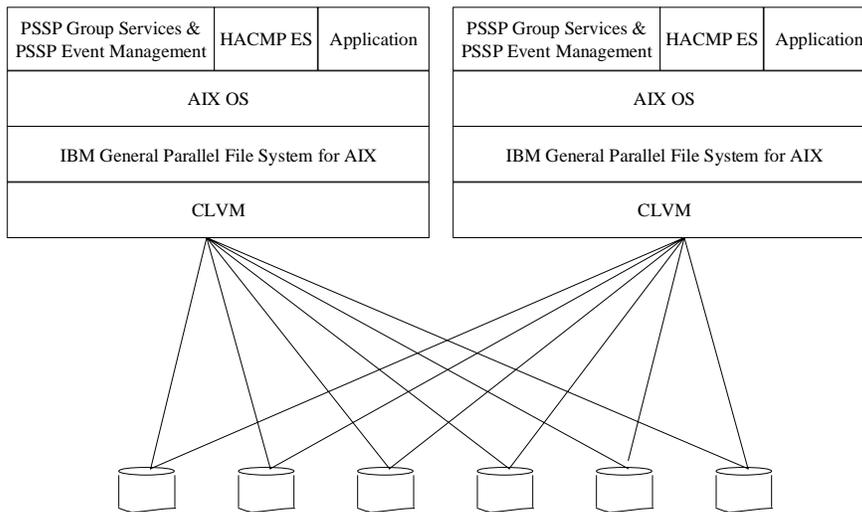
Figure 1: IBM General Parallel File System for AIX in a Virtual Shared Disk Environment



GPFS 1.4 introduces support for concurrent file sharing in a High Availability Cluster Multiprocessing (HACMP) cluster environment. GPFS 1.4 is designed to provide the capability to share data across Serial Storage Architecture (SSA) shared disks or disk arrays directly attached to multiple RS/6000® machines running AIX Version 4 Release 3.3 and HACMP/ES Version 4 Release 4. In a HACMP environment, SSA shared disks or disk arrays are directly attached to each node in the nodeset providing physical access to the disks from every node. The limitations of the SSA adapter constrain the size of a nodeset to a maximum of

8 nodes. GPFS 1.4 has (as do earlier versions of GPFS) the capability to operate in PSSP managed clusters including both the IBM @server Cluster 1600 or IBM RS/6000 SP - building on IBM Virtual Shared Disk and IBM Recoverable Virtual Shared Disk components of IBM Parallel System Support Program (PSSP) for AIX (see Figure 1, the 'IBM Virtual Shared Disk environment'). GPFS 1.4 can also be operated in an IBM Cluster 1600 or an IBM RS/6000 SP environment with HACMP/ES Version 4 and directly attached disks (see Figure 2, the 'non-IBM Virtual Shared Disk environment'). In this environment, SSA shared disks or disk arrays are directly attached to each node in the nodeset. The limitations of the SSA adapter constrain the size of a nodeset to a maximum of 8 nodes.

Figure 2: IBM General Parallel File System for AIX in a non-Virtual Shared Disk Environment



GPFS Characteristics

On each node, the GPFS kernel extension provides the interfaces to the AIX vnode and virtual file system (VFS) interfaces. From a structural perspective, applications issue file system calls to AIX, which presents the calls to the GPFS file system kernel extension. In this sense, GPFS appears to the applications as just another file system. GPFS is implemented as a number of separate software components including a GPFS kernel module and a GPFS daemon (the *mmfsd*). The GPFS kernel extension will either satisfy the requests, utilizing resources that are already available in the system, or send a message to the GPFS daemon to complete the request. The daemon performs the I/O, as well as the buffer management of the GPFS client cache. In addition, the GPFS daemon further initiates read-ahead operations (on I/O patterns GPFS can recognize), as well as conducts write-behind operations to accomplish more efficient pipelining. Many of the services that are necessary for GPFS to operate are provided by the GPFS daemon. The daemon is a multi-threaded process with some threads dedicated to specific functions. This services requiring 'priority attention' are not blocked on other threads that are busy servicing routine work. The daemon communicates with instances of the daemon on other GPFS nodes to coordinate configuration changes, recovery, and parallel updates of the same data structure [1] [2].

GPFS incorporates 'client side caching'. The GPFS cache is located in a dedicated (pinned) area in each of the GPFS application node's memory subsystems. The GPFS cache is labeled as the '*pagepool*'. The *pagepool* is used to cache user data and indirect blocks. It is the GPFS *pagepool* mechanism that allows GPFS to implement *read()* and *write()* requests asynchronously via read-ahead and write-behind

mechanisms. GPFS is multithreaded, so as an application's write buffer has been copied into the *pagepool*, the *write()* request is completed from an application's perspective. Then GPFS schedules a worker thread to manage the *write()* request through to completion by issuing I/O calls to the disk device driver.

One of the major advantages of GPFS over other file system designs is its degree of scalability. GPFS stripes a file across multiple disks. This feature is designed to provide higher (aggregate) read and write throughput, and allows applications to work on large files and large file systems. The file striping mechanism provided by GPFS is designed so that data (as well as metadata) are 'managed' in a distributed manner to avoid hot spots. The token manager server (which is one of the many roles performed by the *mmfsd* daemon) is designed to maintain consistency in GPFS. There is one token manager server per GPFS file system [2] [3]. The granularity for locking may be either a whole file or only portions of a file. The item being accessed (a file) is termed a lock object. The 'per object' lock information is termed a token. The status of each token is stored in two locations, on the token management server and on the token management client holding the token. On every *write()* request, 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 to this file, a 'write token' has to be acquired. The *mmfsd* has to negotiate with the node that holds the token to get write permission. The *mmfsd* first contacts the token manager server to acquire a list of nodes that hold the token, and in a second step, negotiates with the nodes in the list to acquire the token. In order for a node to relinquish a token, the daemon has to surrender the token. First, the daemon has to release any locks that are being held utilizing the token. This process may involve waiting for I/O to complete. Distributing the task to acquire the token to the *mmfsd* reduces serialization at the token manager server level resulting in much better scalability of the whole system. The 'token management' scheme employed by GPFS permits 'byte range locking'. In particular, one task may be granted read or write access to one portion of a file, while other tasks may be granted read or write access to other portions of the same file. This allows read and write request to occur concurrently without suffering any serialization that could otherwise be due to any consistency constraints. In smaller configurations, the load imposed onto the token manager is insignificant. On larger systems, the load on the token server can become significant and it may therefore be desirable to control which node in the configuration will act as the token manager (which is accomplished by creating a preference file).

Benchmarks have shown that GPFS performance is at its peak when manipulating large data objects, but GPFS can also provide a lot of benefits when an application processes large aggregates of small objects [6] [9]. Further, GPFS can be configured with multiple metadata copies which allows continuous operation in the case the path to a disk or the physical device itself is lost. In a direct storage attached configuration, the loss of connectivity from one node to the storage pool does not affect the other nodes in the cluster. In configurations that utilize IBM Virtual Shared Disk component and the SP switch, routing the data through multiple IBM Virtual Shared Disk servers provides the same redundancy. In either configuration, the failure of a system does not result into a complete loss of access to the file system data pool. In a direct attached GPFS configuration, the SSA connection between the storage pool and the actual system determines the data performance. Multiple SSA links can be configured up to the maximum adapter slots available in a particular model. All of these SSA adapters can be linked to a single GPFS file system. On a PSSP managed cluster system, IBM Virtual Shared Disk component and the SP switch provide the low overhead data movement between a node that has physically attached disks and an application node that accesses the data pool. The disks on the I/O node can be spread across adapters (within a server) as well as across multiple server nodes providing scalability and high performance access to the GPFS file system.

GPFS Data Flow (IBM Virtual Shared Disk environment)

In an IBM Virtual Shared Disk environment, GPFS attaches disks utilizing the IBM Virtual Shared Disk component of PSSP. IBM Virtual Shared Disk uses a client/server protocol that is working across the switch. GPFS initiates *read()* and *write()* requests to the (IBM Virtual Shared Disk) client. The requests are sent to the (IBM Virtual Shared Disk) server that owns the physical connection to the disk. Basically, the IBM Virtual Shared Disk layer of GPFS allows a node to locally issue an I/O request (basically a *read()* or *write()* request) that physically occurs on a disk that is attached to a remote node. The IBM Virtual Shared Disk layer communicates either by utilizing the IP layer of AIX or by using the KLAPI subsystem of PSSP.

In the case of IBM Virtual Shared Disk over IP, fragmentation and reassembly functions are required when the *read()* or *write()* request sizes are greater than the size of a single IP packet (IP packets are limited to a maximum size of 64KB). So on a *write()* request, the IBM Virtual Shared Disk layer has to defragment larger request into multiple packets on the client side, and has to reassemble the packets back into a single request on the server side. The reassembly function further handles the cases where IP packets are received in a different order than from how they were sent. Multiple data copies dominate the general overhead of GPFS when using IBM Virtual Shared Disk over IP. Eliminating a data copy was required to reduce the overhead and ultimately improve the GPFS node throughput. Basically, IBM Virtual Shared Disk requires an efficient transport service to handle communication between the (IBM Virtual Shared Disk) clients and the (IBM Virtual Shared Disk) servers. As already elaborated, GPFS maintains a kernel addressable buffer cache (the *pagepool*) to support read-ahead and write-behind functionality. In releases prior to PSSP 3.2, GPFS *read()* and *write()* requests resulted in two data copies. One copy (performed by GPFS) between the application buffer and the GPFS *pagepool*, and a second copy (on the IBM Virtual Shared Disk client) between the GPFS buffer cache and the Communication Sub System (CSS) IP interface managed cluster buffer (performed by the IBM Virtual Shared Disk layer). After the second copy, the switch adapter utilizes DMA to access the cluster buffer. The only data copy in the GPFS path that could be eliminated while preserving the GPFS buffer cache was the data copy done by the IBM Virtual Shared Disk layer. Using IBM Virtual Shared Disk over KLAPI (Kernel LAPI) allows the elimination of this additional data copy. KLAPI provides reliable transport services to kernel subsystems that have to communicate across the SP switch. KLAPI provides semantics for active messages and ‘one sided’ communication, message fragmentation and reassembly, packet flow control, and recovery from lost packets. The KLAPI layer also provides interfaces and the infrastructure for communication to occur without data copies in the communication layer. Functions are provided by KLAPI to ‘prepare’ subsystem buffers for DMA, so that when data is either transmitted or received, the switch adapter is able to DMA directly to or from the subsystem buffers. KLAPI also guarantees the reliable delivery of messages in the absence of node failures. The main advantage of KLAPI over IP is enhanced systems performance by reducing CPU and memory bandwidth utilization as KLAPI effectively utilizes the capabilities of the SP Switch2 adapter. Further, because of flow control in KLAPI, systems utilization will improve under heavy system load.

RAID Configuration

GPFS stripes the data across all the disks that are part of a file system. On a per disk basis, this approach tends to result in an access pattern that has a random placement effect with an I/O size equal to the block size of the GPFS file system. Reasons to use a RAID configuration include tolerating some degree of disk failures as well as increasing GPFS performance for a single block. Studies have shown that in a RAID-5 (4+P) configuration with a GPFS block size of 256KB, the read and the write throughput was measured as 14MB/sec. and 12MB/sec., respectively. These measurements were being conducted with 7500RPM SSA disks (see Appendix D for GPFS performance measurements on different hardware components). In a RAID configuration, the recommendation is that the stripe size matches the logical block size of the GPFS file system (especially for write intensive workloads). SSA disks configured as (n+P) RAID systems have a stripe size of $n \times 64\text{MB}$ (where ‘n’ depicts the number of disks not including the parity disk). Thus for GPFS file systems that have a 256KB block size, the optimal RAID configuration would be 4+P [6].

Performance Impact of small I/O requests

In order to achieve the best possible GPFS performance, data access has to be done by utilizing large block sizes. Selecting an application read or write size that is smaller than the GPFS block size implies that a significant portion of the GPFS block size will be wasted. In the case of truly random I/O requests where the *read()* or *write()* size is smaller than the GPFS block size, this implies that it is necessary to amortize the time to deliver a GPFS block over the amount of data requested by the application [11]. The following analysis demonstrates the impact of small I/O requests, assuming truly random I/O (each I/O request

accesses a different GPFS block), a GPFS block size of 256KB, an application request size of 32KB, and no GPFS overhead.

DR = Delivery Rate for a single GPFS block

DR (for a 256KB block size GPFS file system) = 256KB/(latency + delivery time)

AR = Application Request Size / GPFS Block Size

RR = Rate to deliver a random application request (RR = AR*DR)

For an application request size of 32KB (AR=0.125) and a delivery rate of 128MB/sec. (DR=128MB/sec.), the rate to deliver an application record (RR) equals to 16MB/sec. (RR = AR*DR -> 0.125 * 128MB/sec.). In other words, if the nominal data rate to deliver a 256KB GPFS block equals to 128MB/sec., utilizing a 32KB application request reduces the rate down to only 16MB/sec.

Read, Write, Open, Stat, and Metadata Operations

In GPFS, the creation of new data (either by writing to a new file or extending an existing file) requires updating the file's inode and the on-disk allocation maps for that particular file system. The GPFS file system is designed to incorporate logging when updating the allocation maps, inodes, and indirect blocks [2]. All these operations will result in physical disk I/O. It is imperative to have both, sufficient I/O bandwidth and capacity on the metadata disks when separating metadata from the actual data disks. In the case of parallel write operations to a shared file, GPFS has to acquire the necessary tokens over the requested region on that node in order to ensure 'read/write' ordering semantics. The token acquiring process is normally a small portion of the total processing overheads for I/O requests, but if several application instances on multiple GPFS nodes are writing to the same region of a shared file, token conflicts are unavoidable. The recommendation is that applications that perform fine-grained sharing of the same file consider the utilization of an MPI-IO interface to boost performance. MPI libraries are a popular way of designing parallel applications, as MPI defines a standard for a message-passing paradigm [4] [10]. When encountering random writes to a large GPFS file, the actual data is not being written out to disk until either the block is being reused or the sync daemon runs. The requirement for writing data to disk prior to the reuse of dirty buffers can impact performance on systems that are constricted for *pagepool* resources and have large amounts of random write operations. Increasing the size of the *pagepool* will alleviate some of the pressure put on the GPFS cache and ultimately improve performance.

GPFS *write()* operation

Write() processing is initiated by a system call to the operating system, which calls the GPFS subsystem when the *write()* request involves data in a GPFS file system. GPFS 'moves' the data synchronously (in terms to the application *write()* call) from a user buffer into a file system buffer, but defers the actual *write()* operation to disk. This technique allows GPFS to overlap computation and communication and ultimately improves I/O performance [7]. The file system buffers that are being allocated are part of the GPFS *pagepool*. An actual data block is scheduled to be written to a disk when:

- The user application specifies synchronous write operations
- The system needs the storage space
- A token has been revoked
- The last byte of a block of a file that is being written sequentially is written out
- A *sync()* call is initiated

Until one of the above operations take place, the data remains (cached) in GPFS memory. *Write()* processing encounters three 'levels of complexity' that are basically based on system activity and status, each having its own performance characteristics:

1. The actual buffer is available in memory

2. The necessary token is available locally, but the actual data must be read in
3. The data and the tokens have to be acquired

Metadata changes are flushed under a subset of the same conditions. They can be written either directly (if the node is the metanode) or through the metanode, which consolidates changes from multiple GPFS application nodes. The last scenario normally occurs when threads on multiple GPFS application nodes create new data blocks in the same region of a shared file.

1. The actual buffer is available in memory. The simplest path describes the case where the buffer already exists in memory. This scenario occurs when a previous *write()* call accessed the block and the block is still resident in memory. So the write token already exists from the prior system call. In this case, the data is copied from the application buffer into the GPFS buffer. If it is a sequential *write()* (and the last byte has been written) an asynchronous message is sent to the GPFS daemon to schedule the buffer for ‘page out’ to disk.

2. The necessary token is available locally, but the actual data has to be read in. There are two scenarios in which the token may exist but the buffer does not:

- The buffer has been recently stolen to satisfy other needs for buffer space
- A previous *write()* call obtained a ‘desired range token’ for more space than what was actually needed

In either case, the kernel extension determines that the buffer is not available, suspends the application thread, and sends a message to a (daemon) service thread requesting the buffer. If the *write()* call is for a full file system block, an empty buffer is allocated as the entire block will be replaced. If the *write()* call is for less than a full block and the rest of the block exists, the existing version of the block has to be read from disk. If the *write()* call creates a new block in the file, the daemon searches through the allocation map (for a free block) and assigns a block to the file. At that point, the *write()* call proceeds as described in ‘the actual buffer is available in memory’.

3. The data and the tokens have to be acquired. The third (and most complex) path involving a *write()* operation occurs when neither the buffer nor the token exists on the (local) GPFS node. Prior to the allocation of a buffer, a token has to be acquired for the area of the file that is needed. If the I/O pattern is sequential, a token covering a larger range than what is actually needed will be obtained (if no conflicts exist). If necessary, the token management function will revoke the token from another node holding the token. Note that revoking a token might (if the block is dirty) cause an I/O to occur on the node where the token is being revoked. After acquiring and locking the token, the *write()* call continues as described in ‘*the necessary token is available, but the actual data has to be read in*’.

GPFS *read()* operation

In the case of *read()* operations that involve a regular data pattern (such as pure sequential reads as identified by the GPFS daemon), GPFS attempts to prefetch the data, overlapping the actual execution of an application with data transfers from the disk subsystem [7]. The amount of data to prefetch depends on the response time of the disk subsystem and the rate at which the application is reading, but is always limited by the amount of available *pagepool* space on the GPFS clients. Increasing the size of the *pagepool* can boost performance for applications that show a pure sequential read I/O pattern. If the I/O pattern reflects a random read, GPFS is forced to fetch the data synchronously. In this case, GPFS will only fetch the disk sectors required to satisfy the applications *read()* request. If there is an overlap among the random read operations with each other, increasing the *pagepool* could economize on the number of I/O operations necessary to fetch the data.

The GPFS *read()* function is invoked in response to a *read()* system call, and a call through the operating system’s vnode interface to GPFS. As in the case of *write()* system calls, GPFS *read()* processing falls into three levels of complexity (based on system activity and status), each having its own performance characteristics:

1. The buffer and the necessary locks are available in memory
2. The tokens are available locally but the data has to be read in
3. The data and the tokens have to be acquired

1. *The buffer and the necessary locks are available in memory.* The simplest *read()* operation occurs when the data is already available in memory (either because the data has been ‘prefetched’ or because it has been read recently by another *read()* system call). In either case, the buffer is ‘locally locked’, and the data is copied to the application’s data area. The lock is released when the copy is completed. It has to be pointed out that in this scenario no token communication is required because ‘possession of the buffer’ implies that the node at least possesses a ‘read token’ that includes the buffer. After the memory copy, prefetch operations are initiated if appropriate.

2. *The tokens are available locally but the data has to be read in.* The second (and more complex) type of *read()* operation is necessary when the data is not available in memory. This situation may occur under three different circumstances:

- The token had been acquired on a previous *read()* operation that encountered no contention
- The buffer had been stolen for other use (pressure on the *pagepool*)
- On some random *read()* operations the buffer may not be available

In the first of a series of random *read()* requests the token will not be available locally. In such situations, the buffer is not found and has to be read. No token activity has occurred because the node has a ‘sufficiently strong token’ to lock the required region of the file locally. A message is sent to the GPFS daemon, which is handled by one of the waiting daemon threads. The daemon allocates a buffer, locks the file range that is required if the token cannot be stolen for the duration of the I/O, and initiates the I/O to the device that is holding the data. The originating thread waits for this process to complete and gets posted by the daemon upon completion.

3. The data and the tokens have to be acquired. The third (and most complex) *read()* operation requires that the tokens, as well as the actual data have to be acquired on the GPFS application node. The kernel code determines that the data is not available locally, and sends a message to the GPFS daemon. The daemon thread determines that the necessary tokens to perform the operation are not being held. In this case, a ‘token acquire request’ is sent to the token management server. The requested token specifies the required length (the range of the file) which is needed for this particular buffer. If the file is being accessed sequentially, a desired range of data, starting at the point of the read and extending to the end of the file is specified. In the event that no conflict exists, the desired range will be granted, which eliminates the need for additional token calls on subsequent reads. At the completion of a *read()* request, the prefetching of data is contemplated. GPFS computes a ‘desired read-ahead’ for each open file based on the performance of the disks and the rate at which the application is reading data. If additional prefetch operation is needed, a message is sent to the GPFS daemon that will process the request asynchronously (in terms to the completion of the current *read()* operation).

GPFS *open()* and *stat()* operation

Processing of *open()* or *stat()* system calls (which cause significant access to metadata information) is characterized by the necessary I/O operations to read directory and inode information [7]. The actual read of the required information is unavoidable, but the performance of repeated inquiries can be improved by increasing the *maxStatCache* and *maxFilesToCache* parameters (see Appendix A). The opening of a GPFS file is invoked by the application issuing a system call to the operating system specifying the name of the file. Processing of the *open()* system call involves:

1. Processing the required directory entry identifying the file specified by the application
2. Building the required data structures based on the inode

The kernel extension code will process the 'directory search'. If the required information is not in memory, the daemon will be called to acquire the necessary token(s) for the directory (or part of the directory) that is needed to resolve the lookup operation. Further, the daemon will read the directory entry into memory. The lookup process occurs one directory at a time in response to the calls from the operating system. In the final stage of an *open()* request, the inode for the file is read from disk and linked to the operating system *vnode* structure. This requires acquiring locks on the inode, as well as a lock that indicates 'the presence' to the metanode:

- If no other node has this particular file open, the node that issued the *open()* becomes the metanode
- If another node has a previous *open()*, then that node is the metanode and all the other nodes will interface with the metanode in certain *parallel write()* situations
- If the *open()* involves the creation of a new file, the appropriate locks are obtained on the 'parent directory' and the 'inode allocation file block'. The directory entry is created, an inode is selected and initialized to complete the *open()* processing

The *stat()* system call returns information associated with a file. The call is issued by (for example) the *ls -l* command. The data required to satisfy the *stat()* system call is contained in the inode. GPFS processing of the *stat()* system call differs from other file systems in that it supports the proper execution of *stat()* calls on all nodes without having to funnel the calls through a server. This requires GPFS to obtain tokens that protect the actual metadata. In order to maximize parallelism, GPFS supports the locking of individual inodes. In cases where a 'special I/O pattern' can be detected (such as an attempt to *stat()* all of the files in a large directory) inodes will be fetched in parallel in anticipation of their future use. Inodes are cached within GPFS either as a 'full inode' or a 'limited *stat()* cache'. The 'full inode' is required to perform (data) I/O against the file. The '*stat()* cache' is sufficient to open the file and satisfy a *stat()* system call. As discussed, it is intended to efficiently support commands such as *ls -l* or *du*, as well as certain backup applications that scan entire directory structures analyzing 'modification times' and file sizes. The caches and the requirement for 'individual tokens on inodes' are the reason why the second invocation of a 'directory scanning request' may execute faster than the first one.

GPFS Systems Configuration

In order to configure a system that performs "reasonably well" with GPFS several issues have to be taken into consideration. The expected I/O performance is one of the first points that should be addressed before configuring the system. This will determine the number of GPFS nodes (in direct attached configurations) or IBM Virtual Shared Disk servers (in IBM Virtual Shared Disk environments) that will have to be configured to fulfill the performance requirements. Attempting to transfer more data than what the configuration is capable of will result in performance degradation. A single node's raw I/O performance is impacted by factors such as memory bandwidth, disk adapter performance, as well as physical disk performance. In addition, GPFS performance is influenced by factors such as the size of the *pagepool*, the prefetch depth (and in case of an IBM Virtual Shared Disk environment), as well as the networks bandwidth capabilities into the node. The total performance requirements (in conjunction with the capacity requirements) will help determine the type, as well as the number of disks and adapters necessary to achieve the desired performance level (see Appendix D). As already elaborated, GPFS stripes data across the disks that make up the actual file system. At the disk level, this tends to generate a random placement access pattern with an I/O size equal to the block size of the GPFS file system. Depending on the disk type, the throughput numbers vary, with smaller block sizes resulting in lower throughput numbers. GPFS performance problems can originate within any of the layers in the path to the disks. An in-depth analysis of all the layers has to be considered when addressing a GPFS performance problem. A large number of IBM Virtual Shared Disk retries (in an IBM Virtual Shared Disk environment) is a prime indicator of a performance problem that originated in a layer other than GPFS (use the *statvsd* command to analyze the number of retries). Such retries can be due to an overloaded (IBM Virtual Shared Disk) server node, or can be caused by performance problems on the disk subsystem or switch level.

A GPFS server is being overloaded when the aggregate I/O request rate from all the GPFS client systems exceeds the IBM Virtual Shared Disk server's throughput limit. This can be due to having a high (IBM Virtual Shared Disk) client to server ratio. To alleviate this problem, either the number of GPFS servers has to be increased or the client's I/O rate has to be throttled back. From a disk subsystem perspective, it is imperative to ensure that there is no overload in the disk attachment mechanism. Having a situation with a very high busy rate without a high transfer rate indicates that the disks are being diverted from actually transferring data by recovery operations or congestion problems in the disk attachment path. Such a 'congestion' is normally due to connecting too many disk drives per adapter. A recovery situation is normally logged in the system error logs and points to an actual hardware problem on the disk level. Error logs should be scrutinized to insure a smooth operation. To summarize, the root causes for GPFS performance problems are based on the fact that either (a) the client systems are trying to process I/O rates that exceed the server's throughput capacity, (b) there is an under-configured GPFS/IBM Virtual Shared Disk/KLAPI/IP resource, or (c) there exists an actual disk problem. The first case can be addressed by either increasing the server capacity or by reducing client's I/O demands. The recommendation for the second cause is to check the tuning parameters against the suggested settings in this paper (see Appendix A). The third cause requires analyzing the disk I/O subsystem (*iostat*) and to consult the server log files.

GPFS Performance on POWER3TM SMP Thin, Wide, and High Nodes

The next section in this paper summarizes the results of a GPFS performance study conducted on POWER3 SMP Thin and Wide Nodes, as well as on POWER3 SMP High Nodes. The goal was to establish a baseline for single, as well as multi server performance, and to analyze the scalability and robustness of GPFS 1.4 over KLAPI. The benchmark programs that were being utilized throughout the study simulated the sequential I/O pattern of a parallel application, scaling the number of GPFS application nodes by keeping the workload per application thread static [8]. All the benchmarks reported aggregate throughput numbers in MB/second (1MB=1024² bytes). See Appendix B (POWER3 SMP Thin and Wide Nodes) and Appendix C (POWER3 SMP High Nodes) for the IBM Virtual Shared Disk server configurations, as well as for the raw performance numbers that were being collected throughout the study.

Benchmark

The benchmark program used in this study measures the (aggregate) sequential read/write throughput to/from a shared GPFS file. The simulated access pattern in all the benchmarks was segmented (which results in sequential access for each task, permitting read-ahead and write-behind strategies to take affect). The total 'job time' was calculated as the delta between the time the first task started its I/O activities and the last task concluded its I/O processing. After spawning off all the worker threads (tasks), a barrier was called to insure that all the tasks start their I/O activities approximately as the same time. After setting the barrier, each task issued a call to '*gettimeofday*' to initialize its timing function. After processing the last I/O request, each task again called '*gettimeofday*' to report the end-time. All the 'start and end time' information was being reported to task 0, and summarized in a log for further analysis. Per benchmark, every worker thread executed 2,000 256KB *read()* or *write()* requests to or from the shard GPFS file. There are several reasons why parallel applications need a parallel file system such as GPFS and why such applications have to have access to a shared file. Where I/O performance is the major bottleneck, spreading the *read()* and *write()* requests across multiple disks, ultimately balancing the load to maximize combined throughput increases the aggregate bandwidth of the file system [5] [12]. When dealing with very large data sets, the ability to spawn multiple disk drives with a single file makes the management of the file seamless to the application (and after processing is completed easy visualization of the data is possible).

Benchmark Environment – POWER3 SMP Thin and Wide Nodes

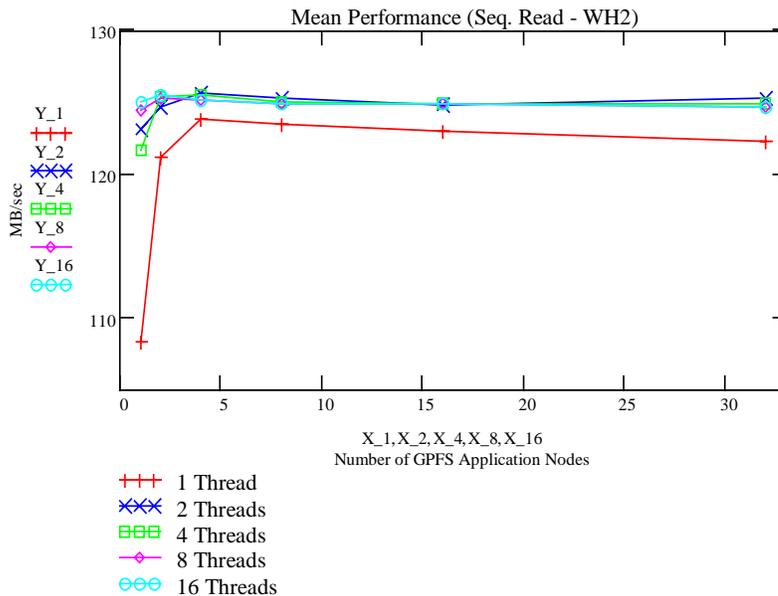
- GPFS server(s): 1 and 2 POWER3 SMP Wide nodes, each node with 4 CPU's and 8GB of memory
- GPFS clients: 40 POWER3 SMP Thin nodes, each configured with 4 CPU's and 4GB of memory
- Each server configured with 2 SSA adapters (1 per PCI bus)
- Each server with 128 18GB SSA disks, 64 per SSA adapter (JBOD), 256KB GPFS block size
- GPFS 1.4, AIX 4.3.3.25, PSSP 3.3.2.10, IBM Virtual Shared Disk over KLAPI
- 80 buddy buffers configured on IBM Virtual Shared Disk servers, 100MB pagepool on GPFS clients
- Maximum aggregate switch bandwidth (SP Switch) for 1 server approximately 130MB/sec

A buddy buffer is being used by a VSD server to handle a disk I/O. The VSD server uses a buddy buffer to temporarily store data for I/O operations that originated at a client node.

Single Server Performance – POWER3 SMP Thin and Wide Nodes

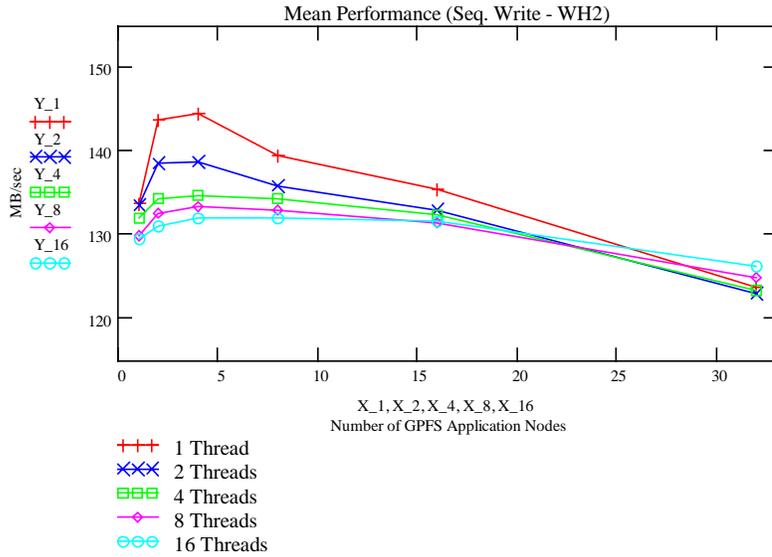
Based on the limitations of the I/O subsystem (the two SSA adapters), the peak bandwidth of the I/O system was anticipated to be around 180MB/second. Further, the maximum aggregate SP Switch bandwidth for the one server configuration was anticipated to be approximately 130MB/second (see Appendix B for the raw performance numbers for the charts shown below).

Figure 3: Sequential Read Performance – 1 IBM Virtual Shared Disk Server



The single server read performance peaked at 125.5MB/sec., whereas the peak write performance was at 144.4MB/sec. The peak read performance was achieved with 2 worker threads and 4 GPFS application nodes (see Figure 3), with an average CPU utilization of 12% (2% user, 10% system) on a GPFS application node, and an average CPU utilization of 43% (all system) on the VSD server. The peak write performance was achieved with 1 worker thread and 4 GPFS application nodes (see Figure 4), with an average CPU utilization of 14% (2% user, 12% system) on a GPFS application node, and an average CPU utilization of 42% (all system) on the VSD server. Increasing the number of application nodes resulted into diminished aggregate write throughput numbers as the number of *client retries* increased substantially.

Figure 4: Sequential Write Performance – 1 IBM Virtual Shared Disk Server



Multi Server Performance – POWER3 SMP Thin and Wide Nodes

Based on the limitations of the I/O subsystem (the two SSA adapters per server), the peak bandwidth of the I/O subsystem was anticipated to be around 360MB/second. Further, the maximum aggregate SP Switch bandwidth for the two IBM Virtual Shared Disk servers was anticipated to be approximately 240MB/second.

Figure 5: Sequential Read Performance – 2 IBM Virtual Shared Disk Servers

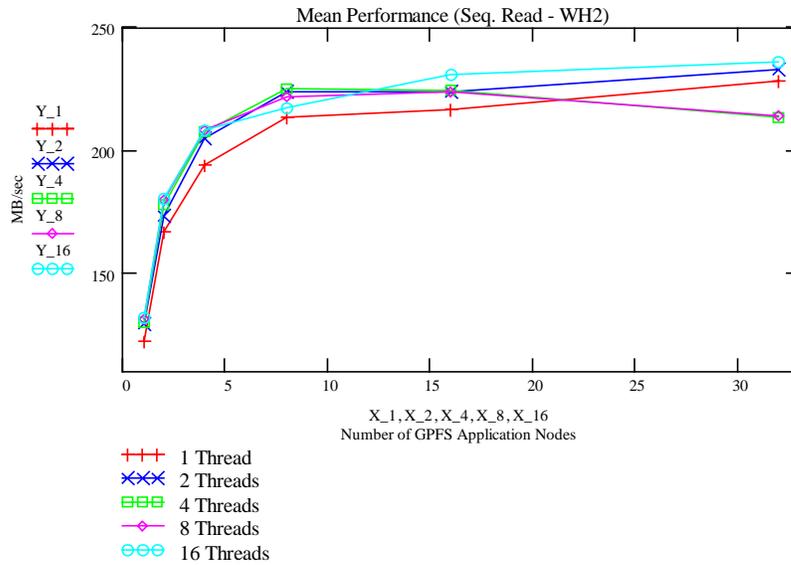
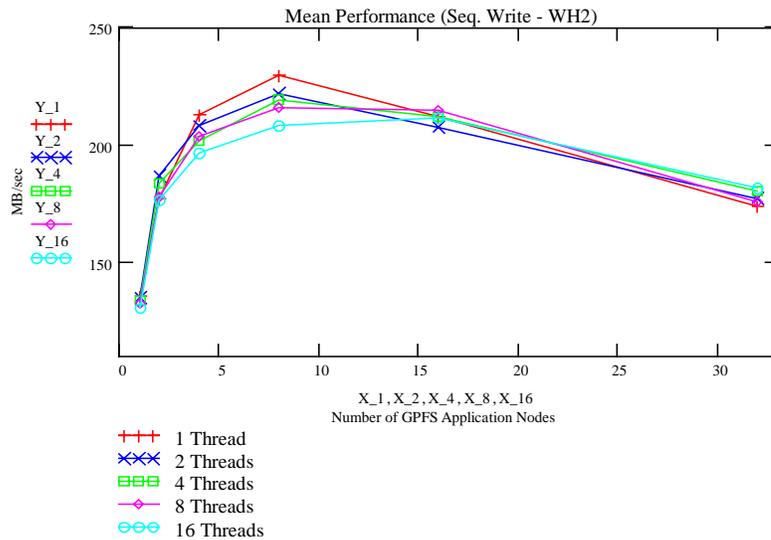


Figure 6: Sequential Write Performance – 2 IBM Virtual Shared Disk Servers



The multi-server read performance peaked at 235.9MB/sec. (see Figure 5), whereas the peak write performance was at 229.2MB/sec. (see Figure 6). The ‘peak read performance’ was achieved with 16 worker threads and 32 GPFS application nodes, with an average CPU utilization of 8% (1% user, 7% system) on a single GPFS application node, and an average CPU utilization of 38% (all system) per VSD server. The ‘peak write performance’ was reported with 1 worker thread and 8 GPFS application nodes, with an average CPU utilization of 9% (1% user, 8% system) on a single GPFS application node, and an average CPU utilization of 38% (all system) per VSD server. The peak aggregate read as well as write throughput numbers are close to the anticipated SP Switch adapter bandwidth of 240MB/sec. Scaling the number of application nodes resulted into a diminished write throughput as the number of *client retries* increased significantly.

Single Server Performance – POWER3 SMP High Nodes

Each SP Switch2 adapter on the SP-Switch2 is capable of sustaining 500MB/sec. (peak in each direction). A single thread can achieve about 350MB/sec. communication bandwidth, which is limited by the CPU copy rate. However, multiple CPU’s can achieve over 425MB/sec. With 2 SP Switch2 adapters on 2 planes (on the SP Switch2) the communication subsystem is capable of over 900MB/sec. of unidirectional bandwidth. It has to be pointed out that while the communication subsystem by itself can achieve over 900MB/sec., that the I/O card on the POWER3 SMP High Node shares the memory bandwidth with the communication card. So the aggregate memory bandwidth available for I/O and communication combined is limited to less than 1,800 MB/sec. Thus one can not expect to sustain more than 900MB/sec. of I/O bandwidth on IBM Virtual Shared Disk servers that are configured with 2 SP Switch2 adapters (see Appendix C for the raw performance numbers for the charts shown below).

Benchmark Environment – POWER3 SMP High Nodes

- GPFS server: 1 POWER3 SMP High Node configured with 16 CPU’s and 16GB of memory
- GPFS server: 6 RIO’s (remote I/O connections), 4 SSA adapters (1 per PCI bus) per RIO
- GPFS server: 96 RAID systems (4+P, 480 spindles, mixed 9.1GB/18.2GB drives, 20 spindles per loop)
- GPFS clients: 30 POWER3 SMP High Nodes, each configured with 16 CPU’s and 16GB of memory
- GPFS 1.4, AIX 4.3.3.25, IBM Virtual Shared Disk over KLAPI, 256KB (GPFS) block size)
- 256 buddy buffers configured on IBM Virtual Shared Disk server, 62 MB pagepool on GPFS clients
- Max. aggregate switch bandwidth (SP Switch2, 2 SP Switch2 adapters) approximately 800MB/sec

Figure 7: Sequential Read Performance – 1 IBM Virtual Shared Disk server

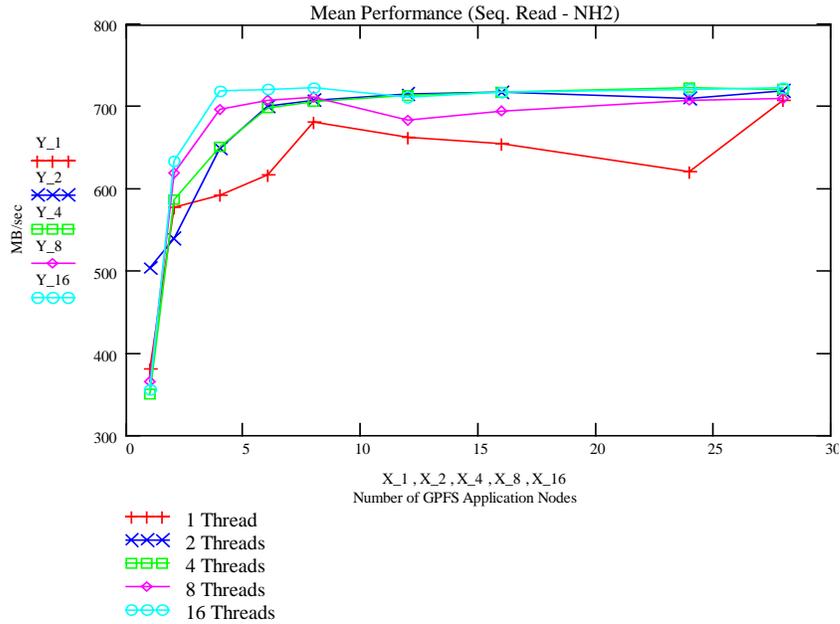
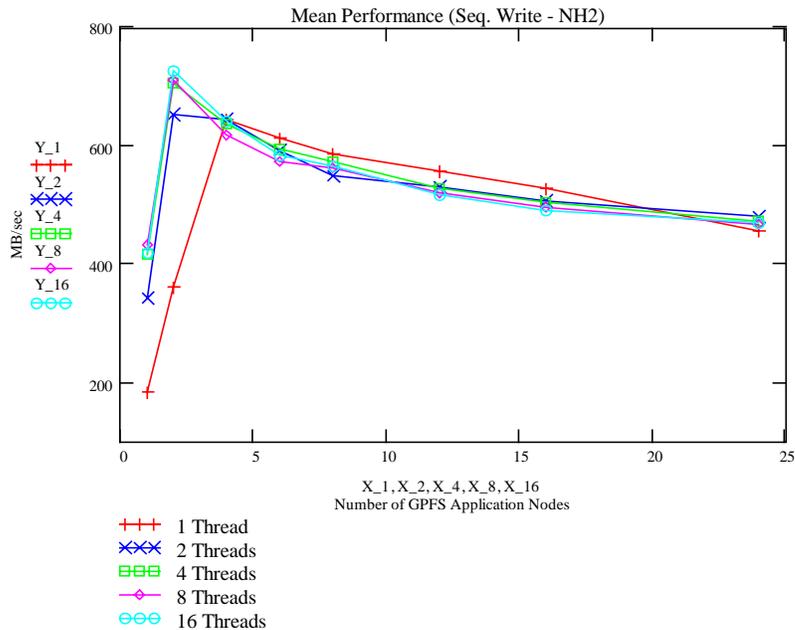


Figure 8: Sequential Write Performance – 1 IBM Virtual Shared Disk Server



The read performance peaked at approximately 715MB/sec. (single IBM Virtual Shared Disk server configuration with 2 worker threads and 16 GPFS application nodes). The read performance was limited by the performance of the communication subsystem (see Figure 7). The write performance (see Figure 8) peaked at approximately 710MB/sec. (8 worker threads and 2 GPFS application nodes). However, adding additional GPFS application nodes caused the write performance to degrade. This is believed to be due to an inefficiency in resource management on zero-copy buffers on the SP Switch2 adapter by the KLAPI subsystem.

Appendix A: GPFS Tuning Parameters

Parameter: **pagepool** (*mmfsadm dump pgallo* to evaluate current setting, *mmchconfig* to adjust it)

Recommendation: Application and memory specific, a 100MB pagepool is recommended

Description: The pagepool is used to cache user data and indirect blocks. The default value is 20MB. It is the GPFS pagepool mechanism that allows GPFS to implement read as well as write requests asynchronously. Basically, increasing the size of the pagepool increases the amount of (pinned) memory that is available to the application. Applications that either reuse data or have a random I/O pattern can normally benefit from a larger pagepool.

Parameter: **maxMBpS** (*mmfsadm dump config* to evaluate current setting, *mmchconfig* to adjust it; GPFS daemon has to be restarted after adjusting it)

Recommendation: Default (of 150) for SP Switch, 450 for SP Switch2 with 1 SP Switch2 adapter and 900 for SP Switch2 with 2 SP Switch2 adapters

Description: This setting determines the amount of prefetching that can be performed. If the default value is not adjusted accordingly it will affect single-client performance. However, in an IBM Virtual Shared Disk environment, the aggregate performance of a server can still be achieved with a sufficient number of GPFS clients.

Parameter: **GPFS Block Size** (*mmlsfs -B* command to evaluate the current setting, *mmcrfs* to adjust it)

Recommendation: Application Specific

Description: The GPFS block size determines the minimum preferred increment for either reading or writing file data. From a performance perspective, the recommendation is to set the GPFS block size to match the application buffers size. Supported block sizes are 16KB, 64KB, 256KB, 512KB, and 1MB. As noted in the section on "Raid Configuration", it is important (from a performance perspective) for the GPFS block size to match the stripe size on the RAID system. If the GPFS block size does not match the RAID stripe size, performance may severely be degraded (especially for *write()* operations).

Parameter: **max_buddy_buffers** (*vsdatafst -n* to evaluate the current setting, *updatevsnode* to adjust it)

Recommendation: I/O throughput and request latency specific.

Description: Buddy buffers are being used by the IBM Virtual Shared Disk servers to handle disk I/Os. The IBM Virtual Shared Disk server utilizes the buddy buffers for temporarily storing data for I/O operations originating at a client. The maximum number of buddy buffers that have to be configured on (a per IBM Virtual Shared Disk server basis) is related to the I/O throughput and the average turn-around time of an individual request. For example, If the throughput on the server is 130MB/second, and the average turn-around time of an individual request is 40 milliseconds, a minimum of 5.2MB (0.04 seconds * 130MB/seconds) of buddy buffer storage space is required. With a maximum buddy buffer size set to 256KB, at least 21 buddy buffers are necessary to handle the load. The recommendation is to at least triple that value to insure an adequate safety margin.

Parameter: **max_buddy_buffer_size** (*vsdatafst -n* to evaluate current setting, *updatevsnode* to adjust it)

Recommendation: Set *max_buddy_buffer_size* to the same as the max GPGS block size

Description: The *max_buddy_buffer_size* parameter sets an upper limit to the size of a buddy buffer. Setting the maximum buddy buffer size lower as the GPFS block size will result into additional overhead in acquiring the additional buddy buffers necessary to hold 'a file system block size' worth of data. Setting the maximum buddy buffer size greater than the GPFS block size will result in an over allocation of memory in regards to the buddy buffer space.

Parameter: **maxFilesToCache** (*mmfsadm dump config* to evaluate current setting, *mmchconfig* to adjust it)
Recommendation: default

Description: The maxFilesToCache parameter specifies the number of inodes to cache for recently used files that have been closed. Storing a file's inode in the cache permits faster re-access to the file. The default is 1000. Increasing this number may improve throughput for workloads with high file reuse. However, on larger configurations it may be preferable to lower the value on at least the compute nodes so that the token manager node does not become the bottleneck.

Parameter: **maxStatCache** (*mmfsadm dump config* to evaluate current setting, *mmchconfig* to adjust it)
Recommendation: default

Description: The maxStatCache parameter specifies the number of inodes to keep in the stat cache. The stat cache maintains only enough information from the inode to perform a query on the file system. The default is $4 \times \text{maxFilesToCache}$.

Parameter: **rw_request_count** (*vsdata1st -n* to evaluate current setting, *updatevsdnode* to adjust it)
Recommendation: 16 per I/O device

Description: The rw_request_count parameter specifies the number of pbufs. These are control structures used on the IBM Virtual Shared Disk server to describe each read and write request that is pending.

Parameter: **max_coalesce** (*lsattr -El hdiskX* to evaluate current setting, *chdev* to adjust it)
Recommendation: 256KB (RAID-5 4+P configuration)

Description: The max_coalesce (SSA) parameter defines the max number of bytes that the SSA disk device driver attempts to transfer to or from a SSA logical disk in one operation. This parameter is very important when dealing with RAID systems.

Parameter: **GPFS comm_protocol** (*mmlsconfig* to evaluate the current setting, *mmchconfig* to adjust it)
Recommendation: Application Specific

Description: The GPFS comm_protocol parameter selects the communication protocol that is being used among the GPFS daemons. The two options are TCP and LAPI. TCP is the default value. LAPI should be chosen for applications that encounter a lot of metadata traffic (for instance small strided *write()* operations, or a lot of file creations and deletions) as the LAPI protocol provides a lower latency than TCP. Choosing LAPI will reduce the number of available switch adapter windows by one, and so may not be the preferred choice for installations that do not want to dedicate an adapter window to GPFS communication (use the *chgcss* command to determine the number of available adapter windows). GPFS has to be stopped and restarted for the communication protocol change to take effect.

Parameter: **IBM Virtual Shared Disk Communication Protocol** (*statvsd* to evaluate current setting, *ctlvsd* to adjust it)

Recommendation: KLAPI

Description: The default IBM Virtual Shared Disk communication protocol is IP. GPFS 1.3 (in conjunction with PSSP 3.1) introduced the KLAPI option. KLAPI is a high-performance communication interface that can be used to avoid additional data copies. Avoiding the data copies reduces CPU utilization. In order to

switch from IP to KLAPI, IBM Recoverable Virtual Shared Disk system has to be stopped and IBM Virtual Shared Disk component has to be un-configured. In the case that IBM Virtual Shared Disk over KLAPI encounters a communication problem, the system will switch to IBM Virtual Shared Disk over IP. Choosing KLAPI **will not** affect the number of user-space windows that are available to user applications.

Parameter: **IBM Virtual Shared Disk adapter choice** (*vsdata1st -n* to evaluate the current setting, *updatevsdnode* to change it)

Recommendation: Configuration Specific

Description: Configurations that have more than one adapter per node have the option to use the ml0 adapter type for IBM Virtual Shared Disk (in that case, IBM Virtual Shared Disk communication can utilize all the adapters on the node). Using the ml0 device (for IBM Virtual Shared Disk) will result in better performance in the case where multiple communication adapters are available on the node.

Parameter: **max_IP_msg_size** (*vsdata1st -n* to evaluate the current setting, *ctlvsd* to adjust it)

Recommendation: 60KB

Description: The *max_IP_msg_size* parameter defines the largest size packet the IBM Virtual Shared Disk software will send between the client and the server (IBM Virtual Shared Disk over IP only).

Parameter: **ipqmaxlen** (*no* to evaluate the current setting, *no* to adjust it)

Recommendation: 512

Description: The *ipqmaxlen* parameter controls the number of incoming packets that can exist on the IP interrupt queue. The default value is 128 (IBM Virtual Shared Disk over IP only).

Parameter: **spoolsize** (*lsattr -El cssX* to evaluate the current setting, *chgcss* to adjust it)

Recommendation: 16MB

Description: The *spoolsize* parameter describes the allocation of memory that is effectively a staging area for information to be sent over the switch (IBM Virtual Shared Disk over IP only).

Parameter: **rpoolsize** (*lsattr -El cssX* to evaluate the current setting, *chgcss* to adjust it)

Recommendation: 16MB

Description: The *rpoolsize* parameter describes the allocation of memory that is effectively a staging area for information to be received over the switch (IBM Virtual Shared Disk over IP only).

Appendix B: GPFS Performance – POWER3 SMP Thin and Wide Nodes

All the performance numbers represent MB/sec. All the test runs were being executed six times, and the performance data in the actual tables represent the mean performance over all the test runs. Every worker thread in the benchmark executed 2,000 256KB read or write operations to a shared GPFS file. The number of worker threads was scaled from 1 per application node up to 16 per application node. The number of application nodes was scaled from 1 to 2, 4, 8, 16, and up to 32.

1 IBM Virtual Shared Disk Server (over KLAPI)

Table 1: Raw Data - Mean GPFS Read Performance - 1 Server (in MB/sec.)

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
1 GPFS Node	108.2	123.0	121.6	124.4	124.9
2 GPFS Nodes	121.1	124.6	125.3	125.2	125.4
4 GPFS Nodes	123.7	125.5	125.4	125.1	125.1
8 GPFS Nodes	123.4	125.2	125.0	124.8	124.8
16 GPFS Nodes	122.9	124.7	124.8	124.8	124.8
32 GPFS Nodes	122.2	125.2	124.8	124.6	124.6

Table 2: Raw Data - Mean GPFS Write Performance - 1 Server (in MB/sec.)

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
1 GPFS Node	133.7	133.4	131.9	129.9	129.4
2 GPFS Nodes	143.6	138.5	134.2	132.5	130.9
4 GPFS Nodes	144.4	138.7	134.7	133.3	132.0
8 GPFS Nodes	139.4	135.7	134.2	132.9	131.9
16 GPFS Nodes	135.3	132.9	132.3	131.3	131.6
32 GPFS Nodes	123.7	122.9	123.2	124.9	126.1

Table 3: 1 Server Environment:

System	POWER3 SMP Thin and Wide Nodes, 1 IBM Virtual Shared Disk Server, SP Switch, GPFS 1.4, PSSP 3.3.2.10
Configuration	100MB Pagepool on GPFS Application Nodes, 80 Buddy Buffers on GPFS Server, KLAPI, 128 SSA Drives on two SSA Adapters (on GPFS Server), JBOD (Just a bunch of disks)
Benchmark	2,000 256KB sequential read or write requests (per worker thread) to or from a single (shared) GPFS file
Date	05/30/2001

2 IBM Virtual Shared Disk Servers (over KLAPI)

Table 4: Raw Data - Mean GPFS Read Performance - 2 Servers (in MB/sec.)

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
1 GPFS Node	121.7	128.9	129.7	131.0	131.5
2 GPFS Nodes	166.4	172.6	177.2	179.2	179.8
4 GPFS Nodes	193.6	204.9	207.3	207.7	207.9
8 GPFS Nodes	213.1	223.3	224.4	221.3	217.1
16 GPFS Nodes	216.5	223.7	224.0	223.5	230.5
32 GPFS Nodes	227.7	232.8	213.1	213.4	235.9

Table 5: Raw Data - Mean GPFS Write Performance - 2 Servers (in MB/sec.)

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
1 GPFS Node	135.3	134.8	133.1	131.8	130.4
2 GPFS Nodes	176.9	186.4	183.1	177.4	176.3
4 GPFS Nodes	212.6	208.1	201.1	203.2	196.5
8 GPFS Nodes	229.2	221.4	218.7	215.9	208.0
16 GPFS Nodes	212.0	206.9	212.0	214.5	211.0
32 GPFS Nodes	173.6	177.0	179.7	175.4	181.6

Table 6: 2 Server Environment:

System	POWER3 SMP Thin and Wide Nodes, 2 IBM Virtual Shared Disk Servers, SP Switch, GPFS 1.4, PSSP 3.3.2.10
Configuration	100MB Pagepool on GPFS Application Nodes, 80 Buddy Buffers per GPFS Server, KLAPI, 128 SSA Drives on two SSA Adapters (per Server), JBOD (Just a bunch of disks)
Benchmark	2,000 256KB sequential read or write requests (per worker thread) to or from a single (shared) GPFS file
Date	06/01/2001

Appendix C: GPFS Performance – POWER3 SMP High Nodes

All the performance numbers represent MB/second. All the test runs were being executed four times. The highest and the lowest values were discarded, and the performance data in the actual tables represent the mean performance over the other two test runs. Every worker thread in the benchmark executed 2,000 256KB read or write operations to/from a shared GPFS file. The number of worker threads was scaled from 1 per application node up to 16 per application node. The number of application nodes was scaled from 1 to 2, 4, 6, 8, 12, 16, and up to either 24 (for *write()* operations) or 24 and up to 28 (for *read()* operations).

1 IBM Virtual Shared Disk Server (over KLAPI)

Table 7:Raw Data - Mean GPFS Read Performance - 1 Server (in MB/sec.)

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
1 GPFS Node	379.2	502.3	349.5	365.2	354.4
2 GPFS Nodes	577.1	538.1	586.9	617.5	633.6
4 GPFS Nodes	590.8	647.6	650.9	695.0	719.2
6 GPFS Nodes	616.2	699.2	698.0	707.7	720.8
8 GPFS Nodes	681.1	707.3	705.0	710.5	723.1
12 GPFS Nodes	660.8	714.8	712.1	683.1	710.4
16 GPFS Nodes	654.8	716.4	716.7	693.8	716.9
24 GPFS Nodes	619.3	708.7	722.4	707.9	720.2
28 GPFS Nodes	706.6	718.8	721.1	708.3	722.0

Table 8:Raw Data - Mean GPFS Write Performance - 1 Server (in MB/sec.)

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
1 GPFS Node	181.6	341.9	416.4	431.7	417.1
2 GPFS Nodes	359.6	651.5	703.7	711.1	726.7
4 GPFS Nodes	643.8	644.8	636.8	615.8	641.6
6 GPFS Nodes	611.4	591.7	593.6	571.7	583.8
8 GPFS Nodes	586.4	549.4	571.3	560.6	564.3
12 GPFS Nodes	556.2	529.5	525.6	518.8	516.6
16 GPFS Nodes	527.8	504.7	502.5	495.8	491.0
24 GPFS Nodes	456.0	477.9	470.7	467.2	467.3

Table 9: 1 Server Environment:

System	POWER3 SMP High Nodes, 1 IBM Virtual Shared Disk Server, SP Switch2, GPFS 1.4
Configuration	62MB Pagepool on GPFS Application Nodes, 256 Buddy Buffers on GPFS Server, KLAPI, 96 RAID-5's (4+P SSA Drives per RAID-5) on 24 SSA Adapters (on GPFS Server)across 6 RIO ports
Benchmark	2,000 256KB sequential read or write requests (per worker thread) to or from a single (shared) GPFS file
Date	06/05/2001

Appendix D: GPFS Performance on Different HW Components

The table below outlines some peak GPFS measurements for various hardware components along the I/O path. The performance numbers are intended as a guide to configure a GPFS system. All measurements were being taken on a GPFS system that was configured with 6 RIO's (evenly balanced across both I/O chips), 4 SSA adapters per RIO (with one SSA adapter per PCI bus). Each adapter had two loops. Each loop had 20 disks configured as 4 RAID-5 systems. For each of the measurements taken, a GPFS file system was configured consisting of all the disks that constituted that component. As an example, for the single SSA loop measurement, the file system was configured with 4 RAID-5 systems that make up the loop, or for the single SSA adapter measurements, all the RAID-5 systems (in the two loops) of the adapter were being used to create the GPFS file system. It has to be pointed out that the performance numbers represent 'best case performance data' measured on a development system running a development version of the PSSP software. So actual GPFS performance may vary depending on an applications I/O characteristic or on how the I/O subsystem is actually configured. The measurements in Table 10 were being taken with 7500RPM SSA drives (a mixture of 4.5GB, 9.1GB and 18.2GB disk). This is especially important for the first row in Table 10. For instance, performance tests have shown that newer disks consistently deliver over 20MB/sec. (measured for reads on a single RAID-5 (4+P) configuration).

Table 10: GPFS Performance along the I/O path

Component	Read Throughput	Write Throughput
4+P RAID	14MB/sec.	12MB/sec.
Single SSA Loop	46MB/sec.	33MB/sec.
Single SSA Adapter	79MB/sec.	65MB/sec.
Single RIO	196MB/sec.	200MB/sec.
High Node (6 RIO's, 1 SP Switch2 Adapter, 1 Port)	340MB/sec.	340MB/sec.
High Node (6 RIO's, 2 SP Switch2 Adapters, 2 Ports)	650MB/sec	650MB/sec.

References

1. Barrios, M., Jones, T., Kinnane, S., Landzettel, M., Al-Safran, S., Stevens, J., Stone, C., Thomas, C., Troppens, U., *Sizing and Tuning GPFS*, IBM Red Book, 1999
2. Barrios, M., et al., *GPFS: A Parallel File System*, IBM Red Book, 1998
3. Curran, R., A GPFS Primer, http://www.ibm.com/servers/eserver/pseries/software/whitepapers/gpfs_primer.html, June 2001.
4. Culler, D., Singh, J., *Parallel Computer Architecture*, Morgan Kaufmann, 1999
5. Hennessy, J., Patterson, D., *Computer Architecture – A Quantitative Approach*, Morgan Kaufmann, 1996
6. IBM Corporation, *GPFS Performance*, IBM Technical Report, February 1999
7. IBM Corporation, *GPFS – Concept, Planning, and Installation Guide for Linux*, June 2001
8. Jain, R., *The Art of Computer Systems Performance Analysis*, Wiley, 1992
9. Jones, T., Koenigs, A., Kim Yates, R., *Performance of the IBM General Parallel File System*, Lawrence Livermore National Laboratory, 1999
10. Koenigs, A., *Industrial Strength Parallel Computing*, Morgan Kaufmann, 2000
11. Paden, R., *GPFS 1.2 and 1.3 Benchmark Tests*, IBM Technical Report, November 2000
12. Pfister, G., *In Search of Clusters*, Prentice Hall, 1998

© International Business Machines Corporation 2001
IBM Corporation
Marketing Communications
Server Group
Route 100
Somers, NY 10589
Produced in the United States of America
12-01 All Rights Reserved

More details on IBM UNIX hardware, software and solutions may be found at:
www.ibm.com/servers/unix

You can find notices, including applicable legal information, trademark attribution, and notes on benchmark and performance at:
www.ibm.com/rs6000/hardware/specnote.html

Please also see the following Webpage:
www.ibm.com/servers/eserver/pseries/hardware/specnote.html

IBM, AIX, RS/6000, POWER3 and SP are registered trademarks or trademarks of the International Business Machines Corporation in the United States and/or other countries. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Limited.

IBM may not offer the products, programs, services or features discussed herein in other countries, and the information may be subject to change without notice. General availability may vary by geography. IBM hardware products are manufactured from new parts, or new and used parts. Regardless, our warranty terms apply. All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Any performance data contained in this document was determined in a controlled environment. Results obtained in other operating environments may vary significantly.

* All performance data contained in this paper was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements quoted in this paper may have been made on development systems. There is no guarantee these measurements will be the same of generally-available systems. Some of the measurements quoted in this paper may have been estimated through extrapolation. Actual results may vary. Users of this paper should verify the applicable data for their specific environment.