# Review

Venkatasudheerkumar Mupparaju
UBITName: vmuppara(37318317)

This paper “GPFS: A Shared-Disk File System for Large Computing Cluster” describes the overall architecture of GPFS (General Parallel File System) which is IBM's parallel shared-disk file system for cluster computers, paper describes its approach to achieving parallelism and data consistency in cluster environment, it details some of the features that contribute to its performance and scalability, describes the design for fault-tolerance and presents data on its performance.

GPFS achieves its extreme scalability through its shared-disk architecture. SAN provides Shared Disks, but SAN itself does not provide a Shared File System. If you have several computers that have access to a Shared Disk and try to use that disk with a regular File System, the disk logical structure will be damaged very quickly. Disk Space Allocation inconsistency and File Data inconsistency makes it impossible to use Shared Disks with regular File Systems as Shared File Systems. Cluster File Systems are designed to solve the problems outlined above. GPFS is one such parallel File System for cluster computers that provides as closely as possible the behavior of a general-purpose POSIX file system running on a single machine.

Cluster File System allows scaling I/O throughput beyond what a single node can achieve. For exploiting this capability requires reading and writing in parallel from all nodes in the cluster which can be enabled through Data striping across disks and using some buffer pool for storing data. To read a large file from a single-threaded application GPFS prefetches data into its buffer pool (Prefetch), issuing I/O requests in parallel, to as many disks as necessary to achieve the bandwidth of which the switching fabric is capable. Similarly, dirty data buffers that are no longer being accessed are written to disk in parallel (Write-behind).

Striping is ensured by allocation maps. Striping works best when disks have equal size and performance, otherwise there is a trade-off between throughput and space utilization.

GPFS uses extensible hashing to organize directory entries within a directory.

In a large File System it is not feasible to run a file system check (fsck) to verify/restore File System consistency, so GPFS records all meta-data updates that affect file system consistency in a shared disk, which later can be used by any other node to do recovery on behalf of failed node.

Also preserving File System consistency and POSIX semantics requires synchronizing acess to data and metadata from multiple nodes, this process limits the parallelism. There are two approaches to achieving the necessary synchronization: distributed locking or centralized lock management. Lock granularity also impacts performance of parallelism achieved.

GPFS supports fully parallel access both to file data and meta-data. It performs its administrative functions in parallel as well. GPFS employs a variety of techniques to manage these different kinds of data: (i) byte- range locking for updates to user data, (ii) dynamically elected "metanodes" for centralized management of file metadata, (iii) distributed locking with centralized hints for disk space allocation, and (iv) a central coordinator for managing configuration changes.

GPFS Distributed Lock Manager: It uses a centralized global lock manager running on one of the nodes in the cluster, in conjunction with local lock managers in each File System node. Lock tokens play a role in maintaining cache consistency between nodes. A token allows a node to cache data it has read from disk, because the data cannot be modified elsewhere without revoking the token first.

i. byte- range locking for updates to user data (Parallel Data Access): During the Byte-range token negotiation process, nodes predicts the desired range they want to write which includes likely future access and get lock on that portion. In the absence of concurrent write sharing, byte-range locking in GPFS behaves just like whole-file locking and is just as efficient, because a single token exchange is sufficient to access the whole file. An experiment was performed to determine the effectiveness of Byte-range token protocol. In the experiment a single-file was partitioned into 'n' large, contiguous sections, one per node and each node was reading or writing (in place updates) sequentially to one of the sections. The write thorugh-put showed similar scalability as nodes and disks are increased which shows the effectiveness of byte-range token protocol. But when sharing becomes finer grain (each node writing to multiple, smaller regions), the token state and corresponding message traffic will grow (GPFS uses byte-range token to synchronize data block allocation and therefore rounds byte range tokens to block boundaries), in such case GPFS allows switching to data shipping mode (centralized management). File blocks are assigned to nodes in a round-robin fashion. GPFS forwards read and write operations originating from other nodes to the node responsible for a particular data block. Fine-grain sharing is more efficient  than distributed locking, because it requires fewer messages than a token exchange, it also avoids the overhead of flushing dirty data to disk (I/O activity) when revoking a token. Experiment was conducted to see the performance of data shipping and Byte-range token, it showed data-shipping supports finer-grain access. In the experiment throughput was measured while updating fixed size records. For updates with record size smaller than one block BR token requires twice as much I/O because of the required read-modify-write (data flushing to disk because of token conflicts). It is evident that drop in throughput of BR locking was infact  due to additional I/O activity and not an overload to server. The throughput curve for data shipping shows that it also incurred the read-modify-write penalty plus some additional overhead for sending data between nodes, but it out-performed BR in finer-grain.

ii. dynamically elected "metanodes" for centralized management of file metadata (Synchronizing Access to File Meta data): Write operations to inode in GPFS uses a shared write lock allowing concurrent writers on multiple nodes. Shared write lock only conflicts with operations that require exact file size and/or mtime (modification time). One of the nodes accessing the file is designated as the metanode for the file; only the metanode reads or writes the inode from or to disk. Each writer updates local copy and forwards its inode updates to the metanode periodically or when the shared write token is revoked (read() or stat() call by other node).  Byte range token mechanism ensures that only one node will allocate storage for any particular data block. Therefore I/O to the inode and indirect blocks on disks is synchronized using a centralized approach. This allows multiple nodes to write to the same file without lock conflicts on metadata updates and without requiring messages to the metanode on every write operation.

iii. distributed locking with centralized hints for disk space allocation (Allocation Maps): For proper striping a write operation must allocate space for a particular data block on a particular disk, but given the large block size used by GPFS it is not as important where on the disk the data block is written. The disks are divided into regions and the total number of regions is determined at file system creation time based on the expected number of nodes in the cluster. Allocation manager which is responsible for maintaining free space statistics initializes by reading the allocation map when the file system is mounted. The statistics is loosely kept up to date via periodic messages in which each node reports the net amount of disk space allocated or freed during the last period. Nodes ask the allocation manager for a region to try whenever a node runs out of disk space in the region it is currently using. To the extent possible allocation manager prevents lock conflicts between nodes by directing different nodes to different regions. Allocation manager periodically distributes hints about which regions are in use by which nodes to facilitate shipping deallocation requests when deleting a file. The experiment conducted shows the effectiveness of allocation manager hints and metanode algorithms in the experinment write throughput for in place updates to an existing file against creation of new file scaled nearly linearly in the number of nodes, due to the extra work required to allocate disk storage, throughput file creation (allocation manager hints) was slightly lower then for in place updates ( metanode algorithms).

Token Manager Scaling: It seems that token manager might become a bottle neck in large cluster or that the size of the token state might exceed the token managers memory capacity. The distribution of token state (hash of the file inode number,  byte range token for each data block) among several nodes in the cluster might fix the problem. The most likely reason for high load on token manager is lock conflicts. The cost of the disk I/O caused by token conflicts dominates the cost of token manager messages. The effective way to reduce token manager load and improve overall performance is to avoid lock conflicts. GPFS uses number of optimizations like when revoking a token it collects replies from the nodes and forwards as a single message to token manager (I.e acquiring token never requires more than 2 messages to the token manager). It also supports token prefetch and token request batching, which allow acquiring multiple tokens in a single message to the token manager. And when a file is deleted on a node, the node does not immediately relinquish token associated with that file (The next file created can reuse). The experiment conducted shows that token manager is not a bottle neck and effectiveness of the token optimization. Measurements of the CPU load on the token server indicated that it is capable of supporting between 4000 and 5000 token requests per second, so the peak request rate in the experiment consumed only small fraction of token server capacity. Which indicates token server is not the bottle neck.

Fault-Tolerance: 3 kinds of failures are possible Node Failures, Communication failures, Disk failures
GPFS detects a failed node by sending periodic heartbeat messages. When a node fails, GPFS must restore metadata, release tokens, appoint replacements for special roles (metanode,allocation manager)
As recovery logs are stored on shared disks, any other surviving node can perform log recovery on behalf of the  failed node. After log recovery, token manager releases tokens held by failed node. Also other nodes can acquire metanode tokens held by the failed node. If either token manager or allocation manager fails, another node will take the responsibility and reconstructs the state by querying all the surviving nodes.
A communication failure such as a bad network adapter can cause a network partition. GPFS invokes primitives available in the disk subsystem to stop accepting I/O requests from the other nodes in the minority group.
GPFS supports replication , implemented in the file system. GPFS allocates space for two copies of each data/metadata block. If some part of the disk becomes unreadable (bad blocks), metadata replication in the file system ensures that only a few data blocks will be affected, rather than rendering a whole set of files inaccessible.

**Discussions**: In the absence of concurrent write sharing, byte-range locking in GPFS behaves just like whole-file locking and is just as efficient (performance wise), because a single token exchange is sufficient to access the whole file.
Metadata is stored in shared disk so that in the case of node failure any node can perform operations on behalf of failed node.
Who ever first acquires lock on inode will act as Meta node. In case of that node failure new node will be elected. Everything is distributed in GPFS and no single point of failure.
GPFS Version-1(2002) supports upto 4PB.
In the case of communication failure GPFS fences nodes that are no longer members of the group from accessing the shared-disk.
GPFS supports RAID. It also supports data replication(total data).

Smaller size disks have better performance than larger disks, goal is to provide speed and not disk space.
There was decrease in the use of GPFS in top supercomputers as the cost associated with it is higher than other parallel File systems and it only supports IBM hardware.