Parallel I/O implies execution of multiple input/output operations concurrently. Support for parallel

I/O is essential to the performance of many applications [236]. Therefore, once distributed file systems

became ubiquitous, the natural next step in the evolution of the file system was to support parallel access.

Parallel file systems allow multiple clients to read and write concurrently from the same file.

Concurrency control is a critical issue for parallel file systems. Several semantics for handling the

shared access are possible. For example, when the clients share the file pointer, successive reads issued

by multiple clients advance the file pointer; another semantic is to allow each client to have its own file pointer. Early supercomputers such as the Intel Paragon4 took advantage of parallel file systems to

support applications based on the same program, multiple data (SPMD) paradigm.

The General Parallel File System (GPFS) [317] was developed at IBM in the early 2000s as a

successor to the TigerShark multimedia file system [159]. GPFS is a parallel file system that emulates

closely the behavior of a general-purpose POSIX system running on a single system. GPFS was designed

for optimal performance of large clusters; it can support a file system of up to 4 PB consisting of up to

4, 096 disks of 1 TB each (see Figure 8.6).

Themaximum file size is (263−1) bytes.Afile consists of blocks of equal size, ranging from 16 KBto

1MB striped across several disks. The system could support not only very large files but also a very large

number of files. The directories use extensible hashing techniques5 to access a file. The systemmaintains

user data, file metadata such as the time when last modified, and file system metadata such as allocation

maps. Metadata, such as file attributes and data block addresses, is stored in inodes and indirect blocks.

Reliability is a major concern in a system with many physical components. To recover from system

failures, GPFS records all metadata updates in a write-ahead log file. Write-ahead means that updates

are written to persistent storage only after the log records have been written. For example, when a new

file is created, a directory block must be updated and an inode for the file must be created. These records

are transferred from cache to disk after the log records have been written. When the directory block is

written and then the I/O node fails before writing the inode, then the system ends up in an inconsistent

state and the log file allows the system to recreate the inode record.

The log files are maintained by each I/O node for each file system it mounts; thus, any I/O node

is able to initiate recovery on behalf of a failed node. Disk parallelism is used to reduce access time.

Multiple I/O read requests are issued in parallel and data is prefetched in a buffer pool.

Data striping allows concurrent access and improves performance but can have unpleasant sideeffects.

Indeed, when a single disk fails, a large number of files are affected. To reduce the impact of

such undesirable events, the system attempts to mask a single disk failure or the failure of the access path

to a disk. The system uses RAID devices with the stripes equal to the block size and dual-attached RAID

controllers. To further improve the fault tolerance of the system, GPFS data files as well as metadata

are replicated on two different physical disks.

Consistency and performance, critical to any distributed file system, are difficult to balance. Support

for concurrent access improves performance but faces serious challenges in maintaining consistency.

In GPFS, consistency and synchronization are ensured by a distributed locking mechanism; a central

lock manager grants lock tokens to local lock managers running in each I/O node. Lock tokens are also

used by the cache management system.

Lock granularity has important implications in the performance of a file system, and GPFS uses

a variety of techniques for various types of data. Byte-range tokens are used for read and write

operations to data files as follows: The first node attempting to write to a file acquires a token

covering the entire file, [0,∞]. This node is allowed to carry out all reads and writes to the file

without any need for permission until a second node attempts to write to the same file. Then the range

of the token given to the first node is restricted. More precisely, if the first node writes sequentially

at offset f p1 and the second one at offset f p2 > f p1, the range of the tokens for the two tokens are

[0, f p2] and [ f p2,∞], respectively, and the two nodes can operate concurrently, without the need for

further negotiations. Byte-range tokens are rounded to block boundaries.

Byte-range token negotiations among nodes use the required range and the desired range for the

offset and for the length of the current and future operations, respectively. Data shipping, an alternative

to byte-range locking, allows fine-grained data sharing. In this mode the file blocks are controlled by

the I/O nodes in a round-robin manner. A node forwards a read or write operation to the node

controlling the target block, the only one allowed to access the file.

A token manager maintains the state of all tokens; it creates and distributes tokens, collects tokens

once a file is closed, and downgrades or upgrades tokens when additional nodes request access to a file.

Token management protocols attempt to reduce the load placed on the token manager; for example,

when a node wants to revoke a token, it sends messages to all the other nodes holding the token and

forwards the reply to the token manager.

Access to metadata is synchronized. For example, when multiple nodes write to the same file, the

file size and the modification dates are updated using a shared write lock to access an inode. One of the

nodes assumes the role of a metanode, and all updates are channeled through it. The file size and the last

update time are determined by the metanode after merging the individual requests. The same strategy

is used for updates of the indirect blocks. GPFS global data such as access control lists (ACLs), quotas,

and configuration data are updated using the distributed locking mechanism.

GPFS uses disk maps to manage the disk space. The GPFS block size can be as large as 1 MB,

and a typical block size is 256 KB. A block is divided into 32 subblocks to reduce disk fragmentation

for small files; thus, the block map has 32 bits to indicate whether a subblock is free or used. The

system disk map is partitioned into n regions, and each disk map region is stored on a different I/O

node. This strategy reduces conflicts and allows multiple nodes to allocate disk space at the same time.

An allocation manager running on one of the I/O nodes is responsible for actions involving multiple

disk map regions. For example, it updates free space statistics and helps with deallocation by sending

periodic hints of the regions used by individual nodes.

A detailed discussion of system utilities and the lessons learned from the deployment of the file system

at several installations in 2002 can be found in [317]; the documentation of the GPFS is available

from [177].