

## Performance of the IBM General Parallel File System

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

Presented by Michael Black  
CMSC714, Fall 2005

## Purpose of paper

IBM has developed the GPFS.

How well does it really perform?

- Bandwidth?
- Scalability?
- Problems to be solved in future development?

## Presentation Outline

- What is the General Parallel File System?
- How does it work?
- How well does it work?

## What is the GPFS?

Parallel file system:

- hundreds of computers (nodes)
  - massive scalability!
- nodes can:
  - run applications
  - manage RAID disks
- each node has access to all files on all disks

## Features of GPFS

- Single files can span several disks on several nodes
  - All processes can write output to same file
  - No need for hundreds of output files
- Processes can write to different parts of same file at same time
  - High bandwidth for concurrent access
- File access uses standard Unix POSIX calls

## Comparable File Systems

- Intel PFS:
  - Nonstandard interface
  - Poor performance on concurrent access to same file
- SGI XFS:
  - Standard interface, good performance, *but* only works on shared memory architectures

## Platform for GPFS

IBM's RS/6000:

- scales to thousands of processors
- autonomous nodes run Unix kernel
- proprietary interconnect - "The Switch"
  - provides 83 MB/s one-way
  - allows all nodes to talk in pairs simultaneously
  - uniform access time

## How is GPFS Implemented?

- Set of services located on some nodes
  - services provided by *mmfsd* - a GPFS daemon
- *mmfsd* provides:
  - file system access (for mounting GPFS)
  - metanode service (file permissions & attributes)
  - stripe group manager (info on the disks)
  - token manager server (synchronizes file access)
  - configuration manager (ensures last 2 services are working)

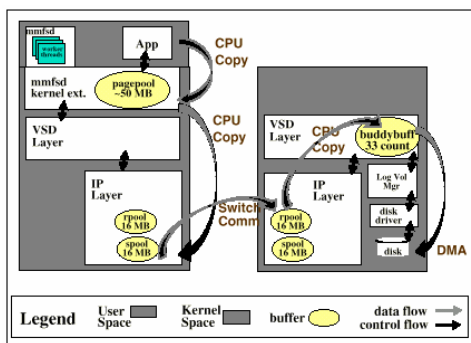
## Layers on each node:

- Application or Disk
- *mmfsd* daemon
  - allows node to mount file system
  - performs reads/writes on that node's disks
- Virtual Shared Disk
  - handles reads/writes to remote disks
  - contains *pagepool* - ~50 MB disk cache
    - allows "write-behind" - application need not wait for actual write
- IP layer

## Consistency

- Maintained by token manager server
- *mmfsd* determines if node has write access to file
  - if not, must acquire token
  - gets list of who has tokens from token manager
  - talks to node with token, tries to acquire it
- tokens guard bytes in files, not whole files
  - allows consistent concurrent access to different parts of same file

## Steps in remote write



## Write steps continued

- Application calls *mmfsd* with pointer to buffer
- *mmfsd* acquires write token
- *mmfsd* checks metanode to see where disk is
- *mmfsd* copies data to *pagepool* (application can now continue)
- VSD copies from *pagepool* to IP, breaks into packets
- Data copied through Switch to VSD receive buffer
- VSD server reassembles data to buddy buffer
- VSD releases receive buffer, writes to disk device driver
- VSD releases buddy buffer, sends ack to client
- Client releases *pagepool*

## Architecture issues with GPFS v1.2

- clients can send faster than server can save to disk
  - exponential backoff used to slow down client
- data copied twice in client
  - memory -> pagepool
  - pagepool -> IP buffer

## Analyzing performance: How much bandwidth is needed?

- Rule of thumb:
  - At peak, 1 byte every 500 flops
  - --> 7.3 GB/s on RS/6000
- Rule of thumb:
  - Store half of memory every hour on average
  - Should take 5 minutes ideally
  - --> 4.4 GB/s on RS/6000
- Also must take varying I/O access patterns and reliability into account

## Experimental objectives

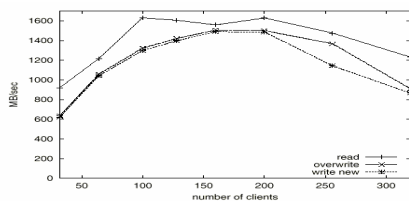
Looking at multiple tasks writing to 1 file:

- How does throughput vary based on:
  - # clients
  - amount of data being transferred
- How does GPFS scale?
- How do I/O access characteristics affect performance?
  - large sequential writes to same file
  - small interleaved writes to same file

## Methodology

- Benchmark: *ileave\_or\_random*
  - written in C, uses MPI
- Measuring throughput:
  - time for all processes to write fixed amount of data to single file
- Effect of I/O access patterns:
  - benchmarks are adjusted for highly sequential (“segmented”) or highly interleaved (“strided”)

## Test: Vary client:server ratio

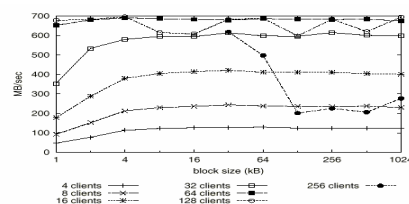


Optimal client:server ratio is 4:1

why?

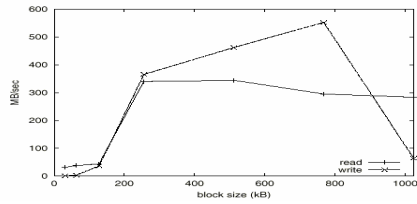
- when too low, server starves
- when too high, server buffers overflow

## Test: Vary transfer block size



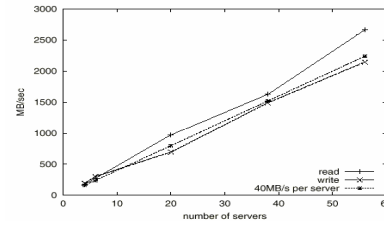
- block size makes no difference
- # clients makes no difference up to 256

### Test: Round-robin file access



- GPFS performs poorly with strided access if block size < 256kB
- Reflects token management overhead

### Test: Scalability with constant client:server ratio



- Linear up to 58 servers

### Conclusions

- Good throughput as long as client:server ratio is less than 6
  - could be increased if data flow is improved
- Programs must use segmented access patterns
  - suggested improvement: allow token management to be turned off (user must manage consistency)