Parallel I/O: Improving Parallel Access from Clusters

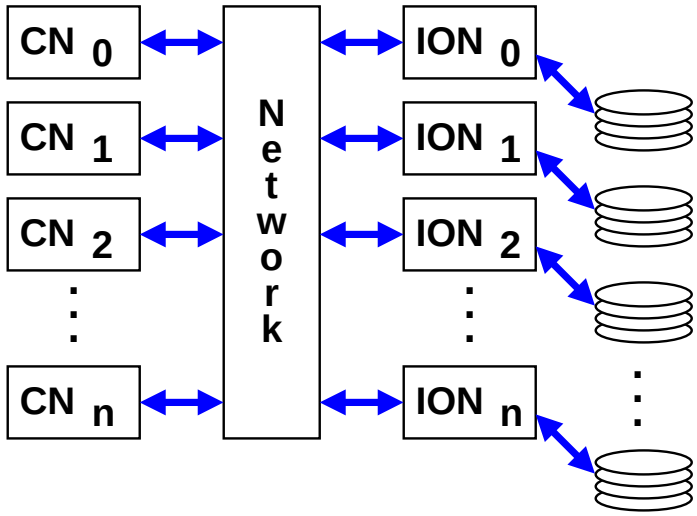
Rob Ross and Bill Gropp
Mathematics and Computer Science Division
Argonne National Laboratory

Alok Choudhary and Wei-Keng Liao
Department of Electrical and Computer Engineering
Northwestern University

Outline

- Parallel I/O on Clusters via PVFS
  - Overview of PVFS
  - Peak performance
- Example Application: FLASH
  - Application characteristics
  - Initial performance problems
  - Current performance comparison
- Areas of Development
  - Data distribution
  - Metadata storage
  - I/O interfaces

Parallel Virtual File System (PVFS)



- File System – allows users to store and retrieve data using common file access methods (open, close, read, write)
- Parallel – stripes data across multiple nodes with separate network connections and avoids bottlenecks in data path
- Virtual – exists only as a set of user-space daemons

PVFS Components

- “Portable” core file system implementation
  - UNIX daemons store data and metadata using local file systems
  - Data is transferred using TCP
  - Library of function calls implement API
  - All usable without root access or kernel hooks
- Kernel-specific client module
  - Allows for “mounting” of PVFS file systems
  - Provides transparent access through kernel
  - Currently implemented for Linux 2.2/2.4

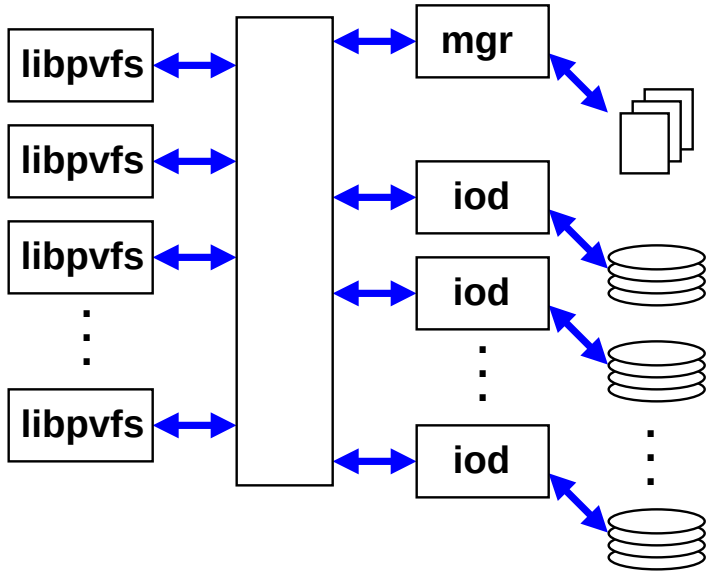
PVFS Core Architecture

Two server (daemon) types:

- mgr – file manager, handles metadata for files
- iods – I/O servers, store and retrieve file data

Client-side library:

- libpvfs – UNIX-like API



- Hides details of data transfer from application tasks

- Can be linked to directly or used by higher-level APIs

Chiba City – The Argonne Scalability Testbed

Software

- Linux 2.4 SMP
- PVFS 1.5.1
- MPICH 1.2.1

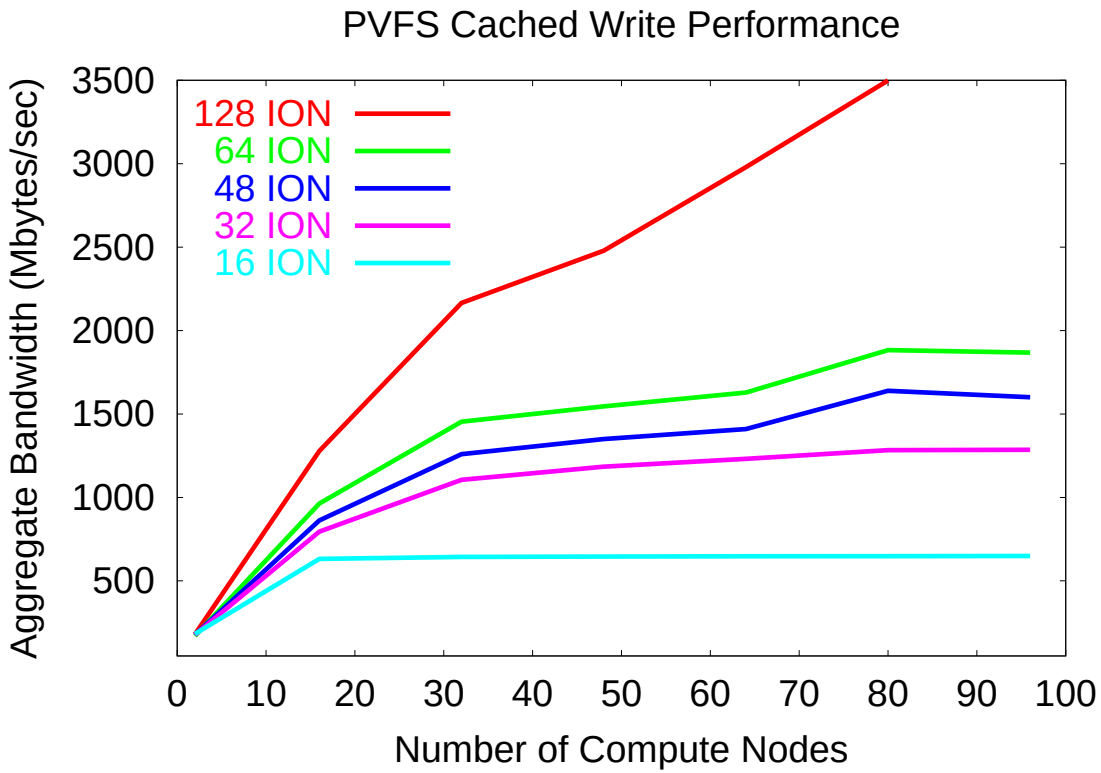
256 Compute Nodes

- 2x500 MHz Pentium IIIs
- 512 Mbytes RAM
- Myrinet (Rev. 3)
- 9 Gbyte SCSI disk

8 Storage Nodes

- 1x500 MHz Pentium III
- 512 MBytes RAM
- Myrinet (Rev. 3)
- IBM EXP-15 SCSI Enclosure

Concurrent Write Performance Through libpvfs



- Using compute nodes for storage in these tests
- Peak at around 30-35 MBytes/sec per I/O server

PVFS and MPI-IO

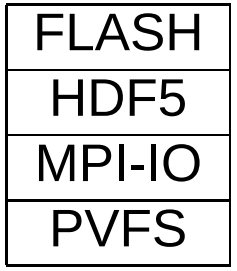
- MPI-IO interface to PVFS files is provided via ROMIO
- ROMIO MPI-IO implementation performs all I/O through an abstract device interface (ADIO)
  - One ADIO implementation hooks ROMIO to PVFS through libpvfs
  - Provides user-space access to PVFS files
  - MPI hints can be used to set some physical distribution parameters
- ROMIO will be covered in more detail in next talk

ASCI FLASH and the FLASH I/O Benchmark

- Adaptive-mesh code used to simulate astrophysical thermonuclear flashes
- Scales to thousands of processors, won Gordon Bell award in 2000
- Interesting from the I/O standpoint:
  - Runs last for days – 0.5 Tbytes of data created
  - Includes both checkpoint and visualization data
  - HDF5 is used for portable storage of output
- FLASH I/O Benchmark extracts the I/O pattern from the FLASH code
- Examining the performance of this benchmark

Software Components for FLASH I/O

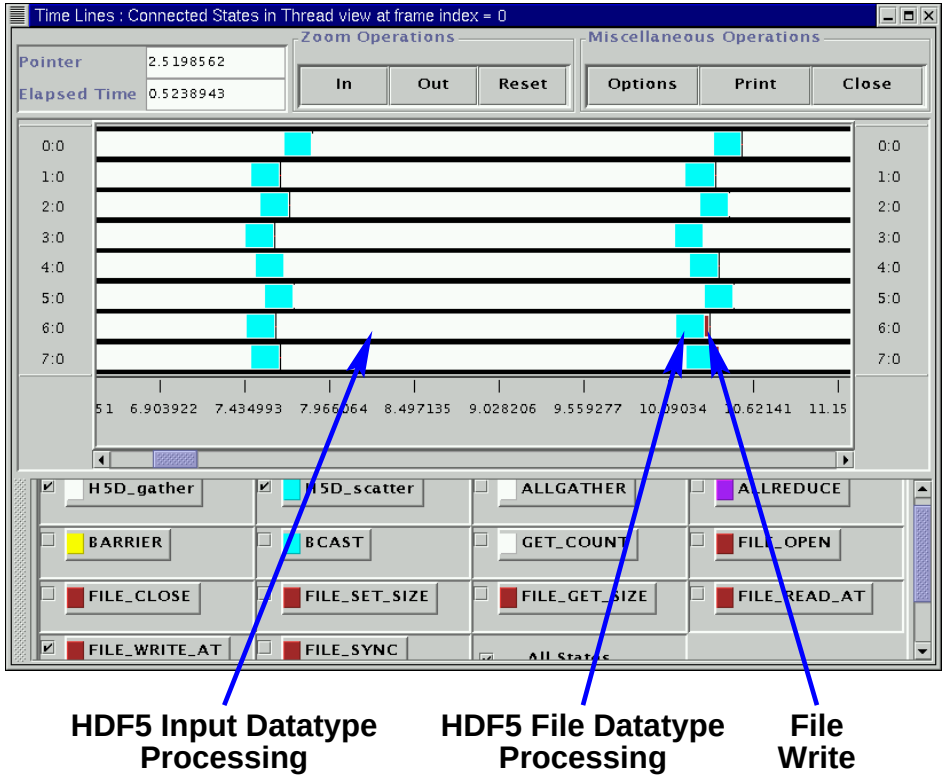
- HDF5 and ROMIO MPI-IO have abstraction layers to allow the use of multiple underlying interfaces
- HDF5 abstraction layer interfaces to MPI-IO
- ROMIO abstraction layer interfaces to PVFS
- HDF5 stores its metadata in the same file as the data



Hints:

- FLASH passes hints through HDF5 property lists
- Hints can include physical distribution info such as “chunking”
- HDF5 propagates hints to MPI through Info parameters

FLASH Checkpoint Performance Problems



- Visualization performed using Jumpshot
- Input datatype processing took over 60 times as long as write!
- Can avoid input datatype processing via packing in app. code

FLASH Checkpoint Performance Comparison

- Chiba storage nodes used as PVFS I/O nodes for these tests
- With packing, reach 23% of peak throughput at 256 processes
- The file datatype processing (internal to HDF5) still present

# of Procs	Size (MB)	Chiba City	ASCI Red	LLNL Frost
32	243.1	26.0 MB/sec	—	—
64	486.2	41.7 MB/sec	0.91 MB/sec	40.8 MB/sec
128	972.4	44.4 MB/sec	—	—
256	1945.1	57.4 MB/sec	0.99 MB/sec	129.1 MB/sec

Lessons from FLASH I/O Experience

- Well-defined interfaces are mandatory for vertical integration of components
- Cross-component visualization is useful for identifying problem spots
- Datatype processing overhead can kill performance in I/O systems too
- Storage of metadata in the data file can perturb data alignment
  - Need to store this metadata outside the data stream

Development Focus: Data Distributions

- Algorithmic mappings describe data distribution in a concise way
- PVFS implements the row-major mapping of file locations to storage
- Row-major assignment results in poor spatial locality in most cases
- Other orderings result in better spatial locality in the average case
- Need hints from application to help us choose!

Row-Major Mapping

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Hilbert Curve Mapping

0	1	14	15
3	2	13	12
4	7	8	11
5	6	9	10

Development Focus: Metadata Storage

- Scientific applications and high-level interfaces create both data and metadata
- In some cases the appropriate location for this metadata is in a database or some other external storage
- Other times (e.g. HDF5 metadata) storage as extended attributes, alongside the file, might be more appropriate
- Tradeoff between ease of search and speed of access
- We are investigating schemas and interfaces for capturing metadata and accessing it within the context of PVFS

Development Focus: I/O Interfaces

- Standards are great, but they aren't necessarily the end-all
- The use of derived datatypes in MPI-IO allows for much richer descriptions of I/O patterns
- More flexible file system interfaces can better utilize these descriptions than POSIX interfaces (e.g. write(), writev())
- Implementing PVFS support for noncontiguous access, both in memory and file
- Integrating this support back into ROMIO