

**Revisiting the parallel I/O problem
(was "Another SCSI isn't the solution")**

**Rob Ross
Argonne National Laboratory**

Who am I?

- Formerly a student at Clemson University
 - Walt Ligon was advisor
 - Origin of PVFS work
- Postdoctoral researcher at Argonne
- Involved in three projects
 - MPICH/MPICH2
 - ROMIO
 - PVFS
- Primarily focusing on last two today

Overview

- Topics
 - Distributed locking
 - Redundancy
 - Datatype processing
 - I/O request descriptions
 - Benchmarking
- Open research areas
- Some proposed solutions
- Lots of questions along the way...

Topics

- **Distributed locking**
- Redundancy
- Datatype processing
- I/O request descriptions
- Benchmarking

Distributed locking

- Applicability of page based distributed locking in memory systems is well-established
 - Works with hardware support in SMP and NUMA systems
 - Not practical/scalable in software implementations
- These lessons apply *directly* to I/O systems as we try to scale them
- In spite of this, region-based locks are used in most parallel I/O systems
- Limits scalability, decreases reliability (ties clients closer to servers)

Roles of atomicity in distributed storage

- Locks are primarily used to provide atomicity
- Metadata operations
 - Creating and removing files
 - Updating block lists
- Some data operations
 - POSIX compliant concurrent access semantics
 - MPI-IO atomic mode accesses (maybe)
- Users often guarantee atomic access on their own and don't want or need this service

The common approach

- Basically clients (instances of file system code on nodes) lock blocks on storage in the traditional way
 - Lazy/eager release
 - Single writer/multiple reader
- It's tough to scale this, and high availability issues make it harder
 - Lots of state stored on clients and servers
 - Issues in reliably/quickly detecting failed nodes
- Perhaps there is a simpler building block?

Another shot at atomic access

- Optimistic access in DB systems promotes local modification before "locking" the global copy
- Store conditional operations in processors atomically perform an update only when no changes have been made
- What if we had atomic, conditional operations in distributed storage?
- Many operations could be done with this building block only
- Can build locks out of this if necessary

Implementing atomic conditional access

- Allow clients to obtain "version tags" (vtags) when performing reads
- vtag identifies a region as being in a certain state
- Subsequent writes provide original vtag, fail if region has changed
- Problems
 - Depending on implementation, not all operations can be performed with one such access
 - Naïve implementation uses n^2 operations when n clients all update a single region
 - Not all functionality of locks

More on vtags

- Servers do not need to know state of clients
 - Before they needed to know who held locks in case node died
 - Simplifies recovery from client failure
- What are common access patterns?
 - Is there a lot of contention?
 - Can we design algorithms to work around this?
- Need benchmarks to test...

Topics

- Distributed locking
- **Redundancy**
- Datatype processing
- I/O request descriptions
- Benchmarking

Redundancy

- Two applications
 - Metadata – ensuring that the storage system maintains a consistent state
 - Data – ensuring that stored files remain available in the event of component failure
- Traditional RAID approaches are often applied to solve both problems at a block level
- Performance implications are quite serious
 - Back to locking again
 - Lots of extra I/O

Redundancy requirements

- Metadata really should be redundantly stored at all times
- Users often don't need their data files to be redundantly stored at every moment in time
 - Want completed checkpoints to be redundantly stored
 - Want output from completed runs to be redundantly stored
- There is room for a separate policy for data redundancy

Lazy data redundancy

- What if we allowed users to specify when to enforce redundancy?
 - `MPI_File_replicate()`
 - Could pass a flag at open to force "always redundantly stored" mode
- Advantages
 - Provides underlying storage with an opportunity to optimize replica creation
 - Allows users to trade performance for reliability
- Questions
 - How does this propagate down in storage layers?

Topics

- Distributed locking
- Redundancy
- **Datatype processing**
- I/O request descriptions
- Benchmarking

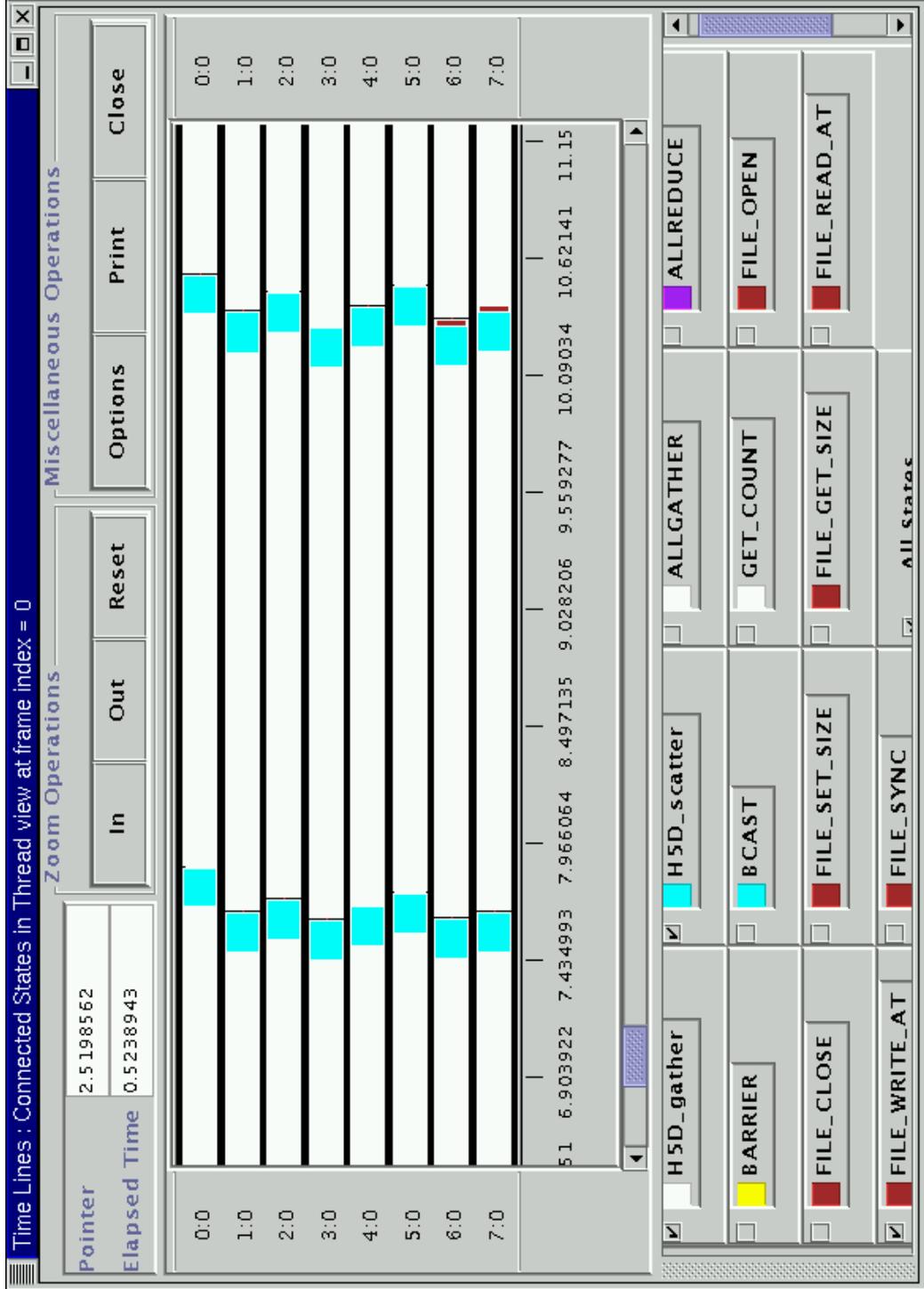
Datatype processing

- Scientific applications operate on complicated datasets
- In message passing and in I/O datatypes are used to simplify handling of these datasets
- Research has shown that many current message passing systems have poor datatype processing implementations
- This performance problem leads application programmers to hand-code processing instead

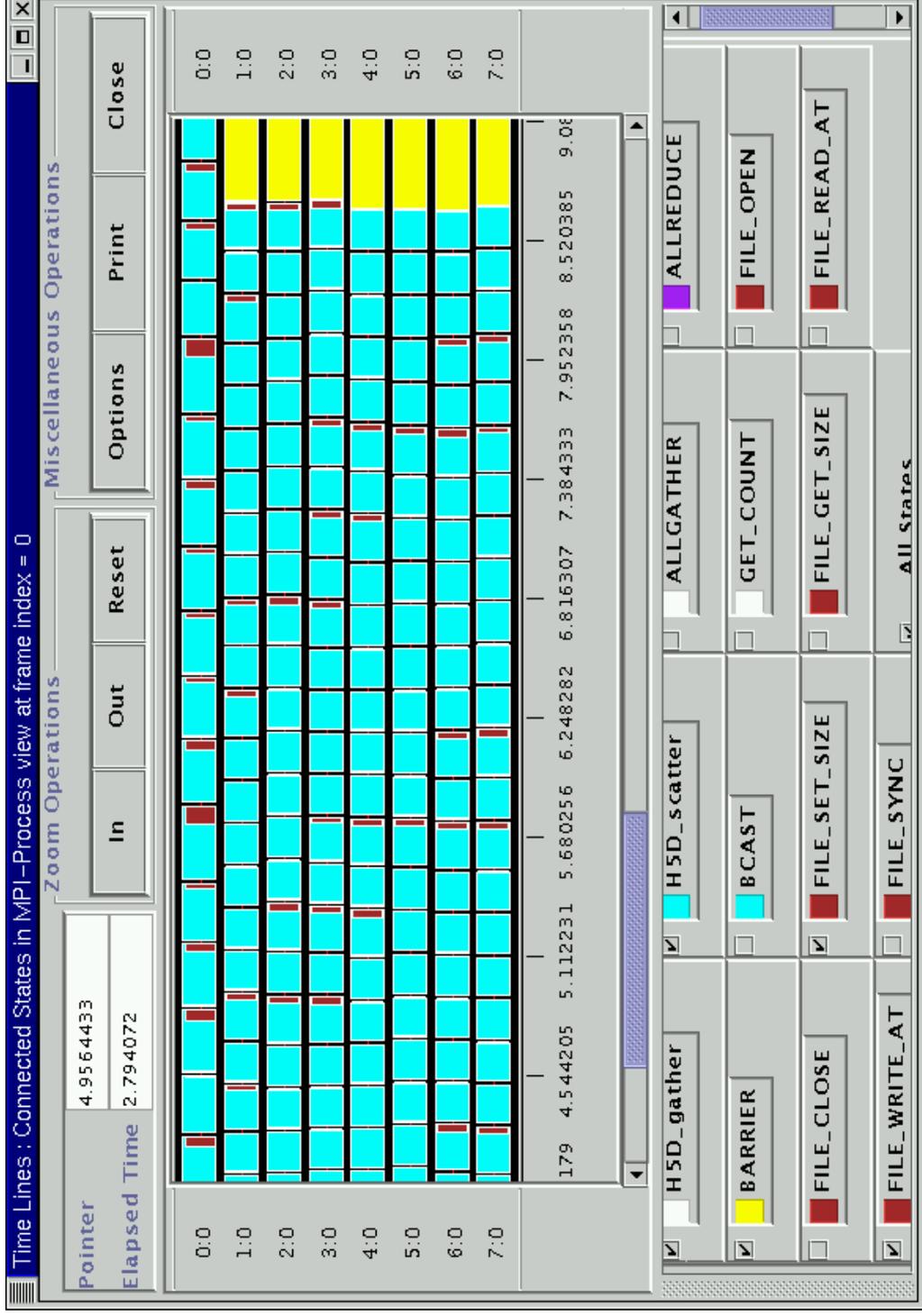
Datatype processing in I/O systems

- Less work has focused on datatype processing in I/O systems
- We can argue that the same datatype processing overhead could exist
- Is the overhead enough to matter in I/O?
- Example application: Flash I/O benchmark
 - HDF, MPI-IO for checkpointing
 - Running on Chiba City using PVFS
 - Instrumenting performed with MPE library

Flash I/O benchmark using datatypes in memory



Flash I/O benchmark with hand-coded packing



More on datatype processing

- Obviously this can be an issue
- Perhaps this became an issue only recently?
- We need to examine the solutions from the message passing world and see what is applicable
- Again...benchmarking...

Topics

- Distributed locking
- Redundancy
- Datatype processing
- **I/O request descriptions**
- Benchmarking

Describing I/O requests

- Applications desire to operate on high-level logical structures
- Application interface needs to pass on as much information as possible
 - Complete description of region to access
 - Desired atomicity
 - Relationships between accesses
- Underlying interfaces should monopolize on this information

MPI-IO implementation

- I'm biased from working on ROMIO...
- Datatypes allow for complete region description
 - Opportunity to make best use of underlying PFS interface
- Collective I/O routines describe relationship between requests of multiple processes
 - Opportunity to perform aggregation
- Missing way of describing patterns of subsequent access (e.g. sequential or random access)

Parallel file system primitives

- Low level interface to PFS should support noncontiguous accesses
- Avoid splitting application accesses
 - Vector of contiguous regions is descriptive enough but not concise
 - Datatype–like descriptions would be ideal
- What about collective operations?
- Inter–request hints?

I/O interfaces and performance portability

- We want people to be able to use datatypes, collective operations to pass on as much information as possible
- Hints can be used to tune how implementations use this information
- Avoids the misapplication of optimizations (e.g. data sieving, aggregation)

Topics

- Distributed locking
- Redundancy
- Datatype processing
- I/O request descriptions
- **Benchmarking**

Benchmarking

- There is no widely used benchmark for parallel I/O performance
- Three categories of quantitative measurements
 - "Interactive experience"
 - Peak performance
 - Application kernels
- Everyone is interested in something a little different
 - Write-dominant, long running applications
 - Read-dominant visualization applications
 - Varied access patterns, file sizes, etc...

Putting together a collection of benchmarks

- Open to suggestions :)
- POSIX tests
 - File creation/deletion
 - Directory traversal
 - Block access
- MPI-IO tests
 - Block access
 - Noncontiguous patterns
 - Collective/independent operations
- Higher level interfaces?

Conclusions

- Incremental technology improvements in I/O devices aren't solving the I/O problem
- Mainstream I/O interfaces and approaches need to be rethought in the face of the scales of systems and types of applications we see today
- Substantial gains can be had simply by improving how we use the hardware we have now
- Comprehensive benchmarks are needed to quantitatively measure the benefits of any improvements