Outline

- Performance Issues in I/O interface design
- MPI Solutions to I/O performance issues
- The ROMIO MPI-IO implementation

1

Semantics of I/O

- Basic operations have requirements that are often not understood and can impact performance
- Physical and logical operations may be quite different

2

Read and Write

- Read and Write are *atomic*
- No assumption on the number of processes (or their relationship to each other) that have a file open for reading and writing
- Process 1 / Process 2

Process 1	Process 2
read a	…
…	write b
read b	

- Reading a large block containing both a and b (Caching data) and using that data to perform the second read without going back to the original file is *incorrect*
- This requirement of read/write results in *overspecification* of interface in many applications codes (application does not require strong synchronization of read/write).

3

Open

- User's model is that this gets a file descriptor and (perhaps) initializes local buffering
- Problem: no Unix (or POSIX) interface for "exclusive access open".
- One possible solution:
  - » Make open keep track of how many processes have file open
  - » A second open succeeds only after the process that did the first open has changed caching approach
  - » Possible problems include a non-responsive (or dead) first process and inability to work with parallel applications

4

Close

- User's model is that this flushes the last data written to disk (if they think about that) and relinquishes the file descriptor
- When is data written out to disk?
  - » On close?
  - » Never?
- Example:
  - » Unused physical memory pages used as disk cache.
  - » Combined with Uninterruptible Power Supply, may never appear on disk

5

Seek

- User's model is that this assigns the given location to a variable and takes about 0.01 microseconds
- Changes position in file for "next" read
- May interact with implementation to cause data to flush data to disk (clear all caches)
  - » Very expensive, particularly when multiple processes are seeking into the same file

6

Read/Fread

- Users expect read (unbuffered) to be faster than fread (buffered) (rule: buffering is bad, particularly when done by the user)
  - » Reverse true for short data (often by several orders of magnitude)
  - » User thinks reason is “System calls are expensive”
  - » Real culprit is atomic nature of read
- Note Fortran 77 requires unique open (Section 12.3.2, lines 44-45).

7

Tuning Parameters

- I/O systems typically have a large range of tuning parameters
- MPI-2 File hints include
  - » MPI_MODE_UNIQUE_OPEN
  - » File info
    - – access style
    - – collective buffering (and size, block size, nodes)
    - – chunked (item, size)
    - – striping
    - – likely number of nodes (processors)
    - – implementation-specific methods such as caching policy

8

I/O Application Characterization

- Data from Dan Reed's Pablo project
- Instrument both logical (API) and physical (OS code) interfaces to I/O system
- Look at existing parallel applications

9

I/O Experiences (Prelude)

- Application developers
  - » do not know detailed application I/O patterns
  - » do not understand file system behavior
- File system designers
  - » do not know how systems are used
  - » do not know how systems perform

10

Input/Output Lessons

- Access pattern categories
  - » initialization
  - » checkpointing
  - » out-of-core
  - » real-time
  - » streaming
- Within these categories
  - » wide temporal and spatial variation
  - » small requests are very common
    - – but I/O often optimized for large requests…

11

Input/Output Lessons

- Recurring themes
  - » access pattern variability
  - » extreme performance sensitivity
  - » users avoid non-portable I/O interfaces
- File system implications
  - » wide variety of access patterns
  - » unlikely that a single policy will suffice
  - » standard parallel I/O APIs needed

12

Input/Output Lessons

- Variability
  - » request sizes
  - » interaccess times
  - » parallelism
  - » access patterns
  - » file multiplicity
  - » file modes

13

Asking the Right Question

- Do you want Unix or Fortran I/O?
  - » Even with a significant performance penalty?
- Do you want to change your program?
  - » Even to another portable version with faster performance?
  - » Not even for a factor of 40???
- User “requirements” can be misleading

14

Effect of user I/O choices (I/O model)

- MPI-IO example using collective I/O
  - » Addresses some synchronization issues
- Parameter tuning significant

15

Importance of Correct User Model

- Collective vs. Independent I/O model
  - » Either will solve user's *functional* problem
- *Same* operation (in terms of bytes moved to/from user's application), but slightly different program and assumptions
  - » Different assumptions lead to very different performance

16

Why MPI is a Good Setting for Parallel I/O

- Writing is like sending and reading is like receiving.
- Any parallel I/O system will need:
  - » collective operations
  - » user-defined datatypes to describe both memory and file layout
  - » communicators to separate application-level message passing from I/O-related message passing
  - » non-blocking operations
- Any parallel I/O system would like
  - » method for describing application access pattern
  - » implementation-specific parameters
- I.e., lots of MPI-like machinery

17

Introduction to I/O in MPI

- I/O in MPI can be considered as Unix I/O plus (lots of) other stuff.
- Basic operations:

  `MPI_File_{open, close, read, write, seek}`
- Parameters to these operations (nearly) match Unix, aiding straightforward port from Unix I/O to MPI I/O.
- However, to get performance and portability, more advanced features must be used.

18

MPI I/O Features

- Noncontiguous access in both memory and file
- Use of explicit offset (faster seek)
- Individual and shared file pointers
- Nonblocking I/O
- Collective I/O
- Performance optimizations such as preallocation
- File interoperability
- Portable data representation
- Mechanism for providing hints applicable to a particular implementation and I/O environment (e.g. number of disks, striping factor): **`info`**

19

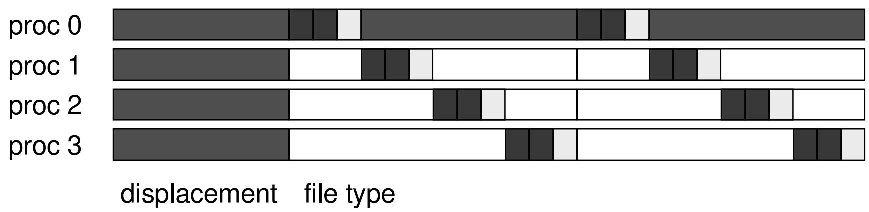
"Two-Phase" I/O

- Trade computation and communication for I/O.
- The interface describes the overall pattern at an abstract level.
- I/O blocks are written in large blocks to amortize effect of high I/O latency.
- Message-passing (or other data interchange) among compute nodes is used to redistribute data as needed.

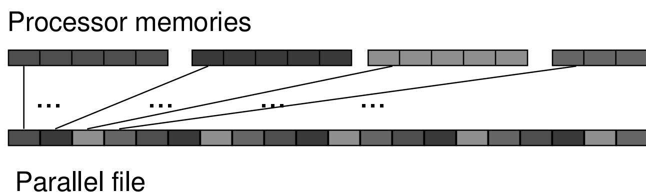
20

Noncontiguous Access

- In memory:



In file:



21

Discontiguity

- Noncontiguous data in both memory and file is specified using MPI datatypes, both predefined and derived.
- Data layout in memory specified on each call, as in message-passing.
- Data layout in file is defined by a file *view*.
- A process can access data only within its view.
- View can be changed; views can overlap.

22

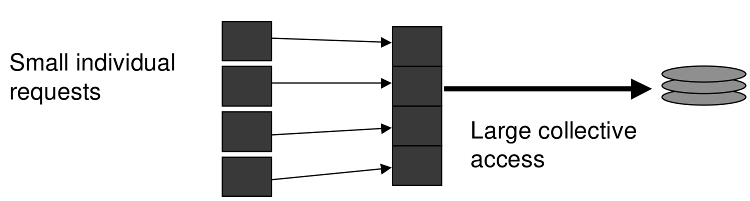
Basic Data Access

- Individual file pointer: **`MPI_File_read`**
- Explicit file offset: **`MPI_File_read_at`**
- Shared file pointer: **`MPI_File_read_shared`**
- Nonblocking I/O: **`MPI_File_iread`**
- Similarly for writes

23

Collective I/O in MPI

- A critical optimization in parallel I/O
- Allows communication of "big picture" to file system
- Framework for 2-phase I/O, in which communication precedes I/O (can use MPI machinery)
- Basic idea: build large blocks, so that reads/writes in I/O system will be large



24

MPI Collective I/O Operations

- Blocking:

```
MPI_File_read_all( fh, buf, count,
                   datatype, status )
```

- Non-blocking:

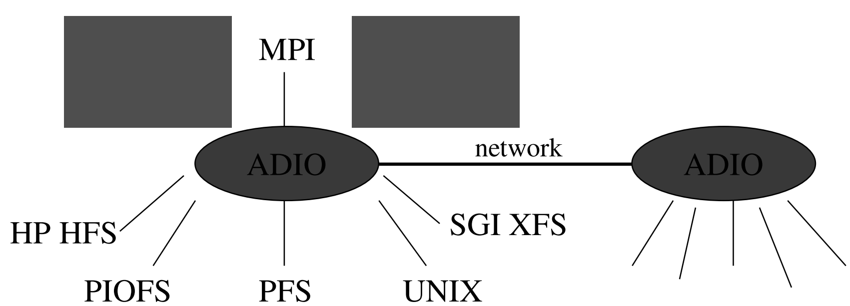
```
MPI_File_read_all_begin( fh, buf, count,
                         datatype )

MPI_File_read_all_end( fh, buf, status )
```

25

ROMIO - a Portable Implementation of MPI I/O

- Rajeev Thakur, Argonne
- Implementation strategy: an abstract device for I/O (ADIO)
- Tested for low overhead
- Can use any MPI implementation (MPICH, vendor)



26

Current Status of ROMIO

- ROMIO 1.0.0 released on Oct.1, 1997
- Beta version of 1.0.1 released Feb, 1998
- A substantial portion of the standard has been implemented:
  - » collective I/O
  - » noncontiguous accesses in memory and file
  - » asynchronous I/O
- Support large files---greater than 2 Gbytes
- Works with MPICH and vendor MPI implementations

27

ROMIO Users

- Around 175 copies downloaded so far
- All three ASCI labs. have installed and rigorously tested ROMIO and are now encouraging their users to use it
- A number of users at various universities and labs. around the world
- A group in Portugal ported ROMIO to Windows 95 and NT

28

Interaction with Vendors

- HP/Convex is incorporating ROMIO into the next release of its MPI product
- SGI has provided hooks for ROMIO to work with its MPI
- DEC and IBM have downloaded the software for review
- NEC plans to use ROMIO as a starting point for its own MPI-IO implementation
- Pallas started with an early version of ROMIO for its MPI-IO implementation for Fujitsu

29

Hints used in ROMIO MPI-IO Implementation

- cb_buffer_size
- cb_nodes
- stripping_unit
- stripping_factor

MPI-2 predefined hints

- ind_rd_buffer_size
- ind_wr_buffer_size

New Algorithm Parameters

- start_iodevice
- pfs_svr_buf

Platform-specific hints

30

Performance

- Astrophysics application template from U. of Chicago: read/write a three-dimensional matrix
- Caltech Paragon: 512 compute nodes, 64 I/O nodes, PFS
- ANL SP 80 compute nodes, 4 I/O servers, PIOFS
- Measure independent I/O, collective I/O, independent with data sieving

31

Benefits of Collective I/O

- 512 x 512 x 512 matrix on 48 nodes of SP

MB/sec	Independent	Collective
Read	5.83	88.4
Write	1.36	70.6

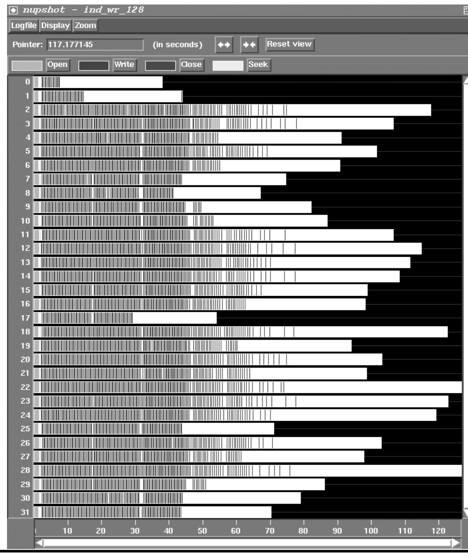
512 x 512 x 1024 matrix on 256 nodes of Paragon

MB/sec	Independent	Collective
Read	4.02	160
Write	1.85	277

32

Independent Writes

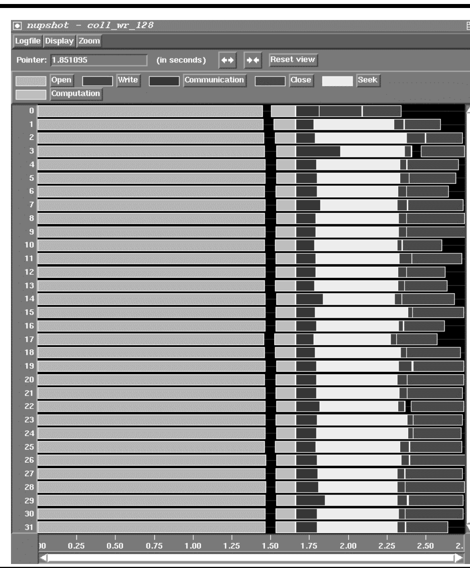
- On Paragon
- Lots of seeks and small writes
- Time shown = 130 seconds



33

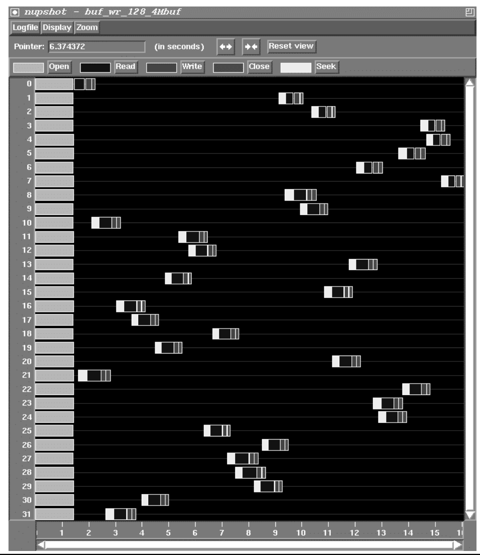
Collective Write

- On Paragon
- Communication and communication precede seek and write
- Time shown = 2.75 seconds



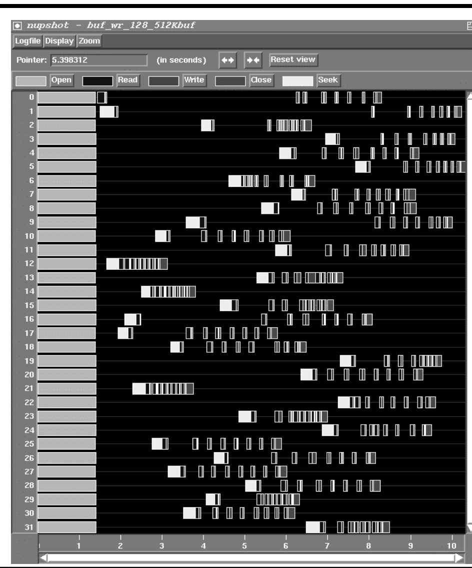
Independent Writes with "Data Sieving"

- On Paragon
- Use large blocks, write multiple "real" blocks plus "gaps"
- Requires lock, read, modify, write, unlock for writes
- Paragon has file locking at block level
- 4 MB blocks
- Time = 16 seconds



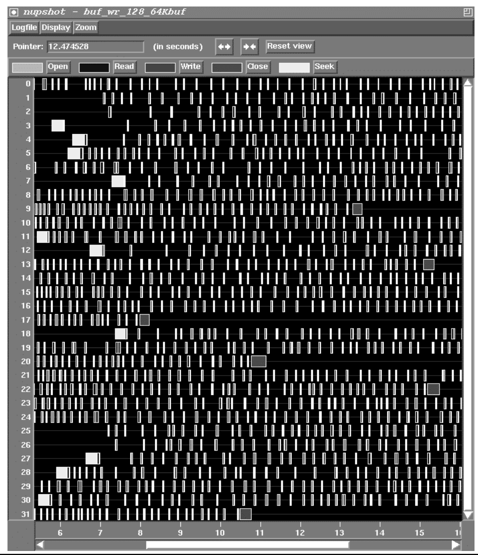
Changing the Block Size

- Smaller blocks mean less contention, therefore more parallelism
- 512 KB blocks
- Time = 10.2 seconds
- Still 4 times the collective time



Data Sieving with Small Blocks

- If the block size is too small, however, then the increased parallelism doesn't make up for the many small writes
- 64 KB blocks
- Time = 21.5 seconds



Conclusions

- OS level I/O operations overly restrictive for many HPC applications
  - » You want those restrictions for I/O from your editor or word processor
  - » Failure of NFS to implement these rules a continuing source of trouble
- Physical and logical (application) performance different
- Application "kernels" often unrepresentative of actual operations
  - » Use independent I/O when collective is intended
- Vendors can compete on the quality of their MPI IO implementation

38