Bad I/O and approaches to fixing it

Jonathan Carter
NERSC
Lawrence Berkeley National Laboratory

This work was supported by the Office of Science, Office of Advanced Scientific Computing Research, Mathematical, Information and Computational Sciences Division, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

Overview

- T3E is a shared and highly contested resource
- Standard styles of I/O can give poor or even disastrous performance
- Some case studies based on our experiences

Cray FFIO library

- FFIO is a set of I/O layers tuned for different I/O characteristics
- Buffering of data (configurable size)
- Caching of data (configurable size)
- Available to regular Fortran I/O without reprogramming
- Available for C through POSIX-like calls, e.g. `ffopen`, `ffwrite`

The assign command

- The assign command controls
  - controls which FFIO layer is active
  - striping across multiple partitions
  - lots more
- Scope of assign
  - File name
  - Fortran unit number
  - File type (e.g. all sequential unformatted files)

bufa FFIO layer

- bufa is an asynchronous buffering layer
- Performs read-ahead, write-behind
- Specify buffer size with `-F bufa:`*`bs`*`:`*`nbufs`* where *`bs`* is the buffer size in units of 4Kbyte blocks, and *`nbufs`* is the number of buffers
- Buffer space increases your application's memory requirements

global FFIO layer

- global is a caching and buffering layer which enables multiple PEs to read and write to the same file
- If one PE has already read the data, an additional read request from another PE will result in a remote memory copy
- File open is a synchronizing event
- By default, all PEs must open a global file, this can be changed by calling GLIO_GROUP_MPI(comm)
- Specify buffer size with `-F global:`*`bs`*`:`*`nbufs`* where *`bs`* is the buffer size in units of 4Kbyte blocks, and *`nbufs`* is the number of buffers per PE

Case I

- Application is Computational Chemistry, Relativistic Self-Consistent Field Theory
- The SCF step involves reading in N^4 quantities from files and assembling an N^2 matrix
- Each PE reads a separate file using Fortran sequential unformatted I/O
- Default buffer size is 48 blocks, file is blocked
- Obvious tuning is to go to FFIO layer bufa with a larger buffer. Less I/O requests, read ahead, and no blocking

Case I (cont.)

- Results from calculation on UO_2^{2+} (333 Gb total I/O)

FFIO	*LOGIOReqs*	*IO WAIT (secs)*	*MPP (secs)*
cos:48	1736736	43539	222231
bufa:256:2	333400	2779	142892
new cray libs	330960	3513	156565

Case II

- Astrophysics application
- All PEs read overlapping sections from one file via POSIX I/O
- POSIX I/O can be efficient provided that requests are large, are not buffered at the system level, and are well-formed
  - start on sector boundaries
  - multiple of sector size
  - open with O_RAW and data cache line aligned
- Ill-formed I/O will all go through the system buffer cache

Case II (cont.)

- Replace POSIX I/O with FFIO calls
- FFIO global layer merges application level requests ensuring well-formed requests to system
- Results from test calculation (17Gb total I/O)

FFIO Spec	*LOGIOReqs.*	*IOWAIT (secs)*	*MPP (secs)*
none	299632	49327	58626
global:128:2	44497	992	18103

Case III

- Climate application
- All PEs use netCDF to read same data file
- By default, netCDF uses FFIO bufa:336:2 for each file
- bufa does not do well-formed I/O if application seeks to non-sector boundaries
- Pathological slowdown

Case III (cont.)

- netCDF reworked by R.K. Owen and Steve Luzmoor
- Obeys FFIO specification on a per file basis
- Added support for 'unlimited dimension' on parallel write
- Added API for subset of PEs to open a netCDF files which uses FFIO global layer

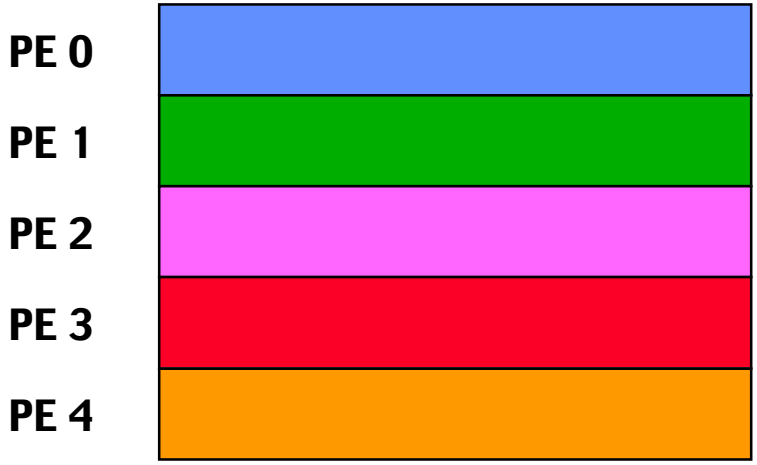
Case III (cont.)

- **Results from example netCDF dataset (64Mb array)**

#PEs	Total I/O (Mb)	LOGIO Reqs.	IOWAIT (secs)	MPP (secs)
bufa:336:2				
4	259	214	16	21
16	1023	850	252	290
32	2042	1698	974	1042
64	4079	3394	3835	4023
global:512:2				
4	65	45	3	17
16	65	81	2	69
32	65	129	5	177
64	65	225	5	702

Case IV

- Linear Algebra (Geology) application
- Write out a large distributed matrix using MPI I/O



MPI I/O

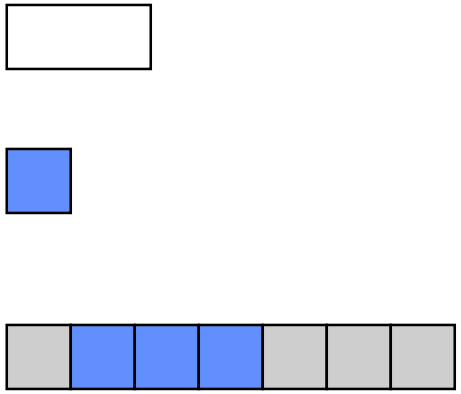
- Data partitioning
- Collective I/O
- Asynchronous I/O
- Portability and interoperability
- T3E implementation
  - Based on ROMIO 1.0.1
  - No shared file pointers
  - No non-blocking collective (split collective)

MPI I/O

- Simplest approach is to use individual and non-contiguous access. Poor performance
- Can use collective I/O to merge requests for non-contiguous data. Better performance
- Use MPI "fileview" to define the non-contiguous access pattern. Best performance

MPI I/O Fileviews

- A fileview is composed of three pieces:
  - a displacement (in bytes) form the beginning of the file
  - an elementary datatype (etype), which is the unit of data access and positioning within the file
  - a filetype, which defines a template for accessing the file. A filetype can contain etypes or holes of the same extent as etypes.



MPI I/O Fileviews (cont.)

- The filetype pattern is repeated, “tiling” the file
- Only the non-empty slots are available to read or write

Fileview (cont.)

- Each process can have a different filetype

Process 0

Process 1

Process 2

Case IV (cont.)

- **Write out a 70400 by 2000 matrix, distributed over 32 PEs**

I/O	*LOGIOReqs.*	*IOWAIT (secs)*	*MPP (secs)*
MPI I/O no fileview	64066	30251	39634
MPI I/O fileview	386	3651	6308
global:512:2	668	340	4076