Parallel I/O Profiling using Darshan

Technical Talk
Manchester & Oxford
15/NOV/2017

Wadud Miah
HPC Application Analyst

nag®

Experts in numerical software and
High Performance Computing

Agenda

- Introduction to parallel I/O;
- Using the Darshan I/O profiler;
- Case studies that used Darshan;
- Recommendations on how to optimise file I/O.

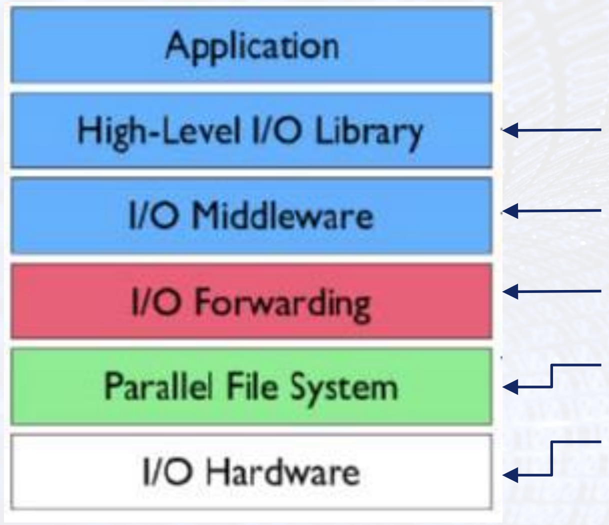
Persistent Storage

- HPC applications have a need for persistent storage;
- Reading initial conditions from disk as well as writing final solution to disk;
- Solution can then be analysed and/or visualised for scientific discovery and data sharing;
- In addition, data is also written to disk periodically during simulation in the event of a node failure;
- If a node crashes, simulation is resumed from previously saved data.

Fault Resiliency in HPC

- With the advent of large HPC clusters with many thousands of nodes, the risk of node failure increases;
- There is also risk of runtime library crashes, which increases the frequency and the need of checkpointing;
- This subsequently increases the load on the parallel file system;
- Just as computation and communication can affect application performance, file I/O can also reduce the application’s performance.

Parallel I/O Stack



Parallel NetCDF and parallel HDF5;

MPI-IO - Part of MPI 2.0 standard;

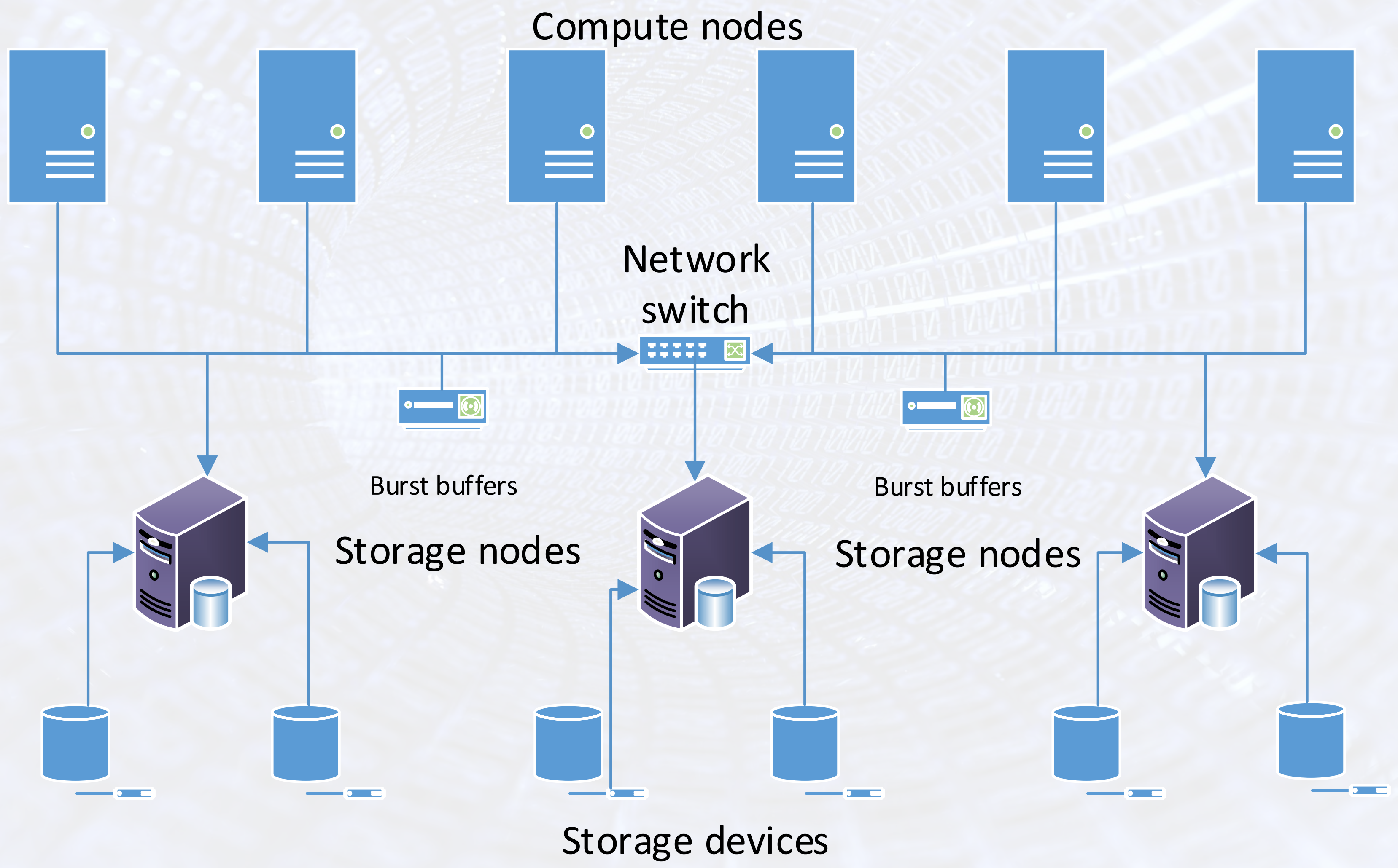
CIO (IBM), DVS (Cray) - aggregates I/O;

GPFS, Lustre, PanFS;

Hard drives and RAID devices. Includes SSD, NVRAM and traditional spinning disks.

- Deep hierarchy to support parallel I/O;
- Science is also becoming increasingly data intensive, hence the importance of high performance I/O;
- File I/O in computational science tends to be write-once and read-many.

Parallel Storage Architecture



Types of Storage Nodes

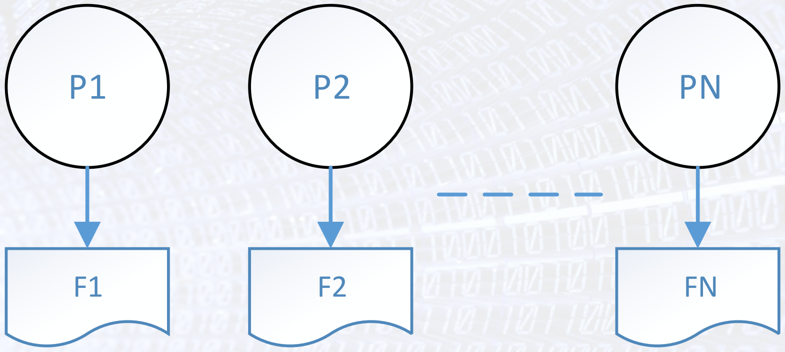
- There are two types of storage nodes: *meta-data* nodes and *data* nodes;
- Meta-data nodes store information such as file owner, access time - Linux inode data;
- Data nodes actually store the file data. There are more data nodes than meta-data nodes;
- Lustre and Panasas have dedicated meta-data nodes whereas GPFS strides the meta-data across storage nodes;
- *Parallel file systems are bandwidth bound.*

Parallel I/O Factors

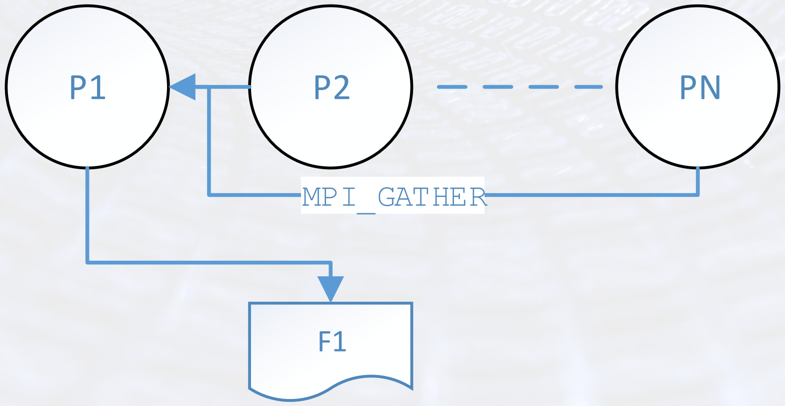
- The number of MPI processes;
- The total amount of data to read or write;
- The size of the files involved;
- Number of files involved;
- Stripe count - number of storage nodes available;
- Stripe size of the parallel file system - block of data that is written to a storage node.

Parallel I/O Models (1)

- One file per MPI process (N:N):

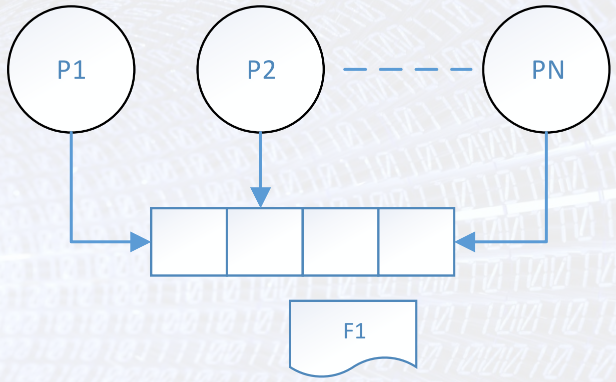


- Single file (N:1) model:

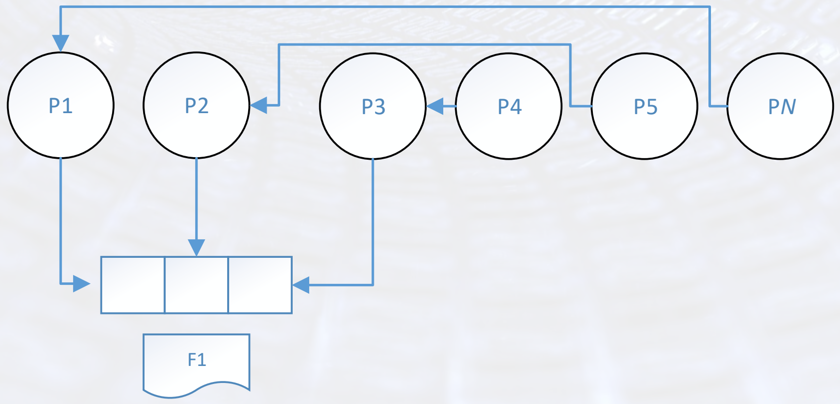


Parallel I/O Models (2)

- Shared file (N:1) model:

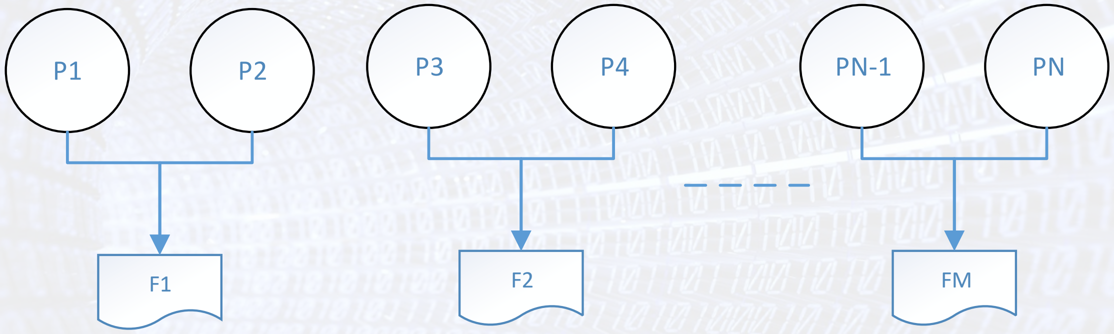


- Alternative shared file (M:1) model (M < N):



Parallel I/O Models (3)

- Hybrid model (N:M) where M < N:



File I/O Profiling (1)

- Just as computation and communication can be profiled, so can file I/O be profiled;
- Subsequently, file I/O can also be optimised;
- Following I/O methods are used in HPC applications: POSIX, MPI-IO, parallel NetCDF and parallel HDF5;
- Darshan [1] is able to profile all four methods and *can only profile MPI codes.* Must call `MPI_FINALIZE`, so Darshan will not work if `MPI_ABORT` is called;

[1] http://www.mcs.anl.gov/research/projects/darshan/

File I/O Profiling (2)

- Serial codes can be profiled but with `MPI_INIT` and `MPI_FINALIZE`;
- Hybrid MPI + OpenMP is also supported;
- Darshan only profiles codes written in C, C++ and Fortran. Has been used for MPI4PY (Python) but not fully supported and tested;
- Darshan instruments I/O - not statistical sampling. Thus profiles are accurate;
- Each I/O call is intercepted by the library;

File I/O Profiling (3)

- Each MPI process collates I/O metrics and collected when an `MPI_FINALIZE` call is made;
- The memory footprint of each MPI process is around 2 MiB, so it is minimal.

Invoking Darshan (1)

- No code changes are required to use Darshan. Some HPC systems switch on Darshan profiling for all their jobs, so it is very lightweight;
- Darshan provides a summary of I/O statistics;
- Darshan can be loaded as a dynamic library if profiling a Linux dynamic executable:

```
LD_PRELOAD=/lib/libdarshan.so mpirun -n 128 \
./wrf.exe
```

Invoking Darshan (2)

- If using static executable, code will have to be rebuilt using Darshan MPI wrappers which are unique for every MPI implementation;
- For profiling MPI4PY, you can *only* use the `LD_PRELOAD` variable method as Python is a dynamic language - no static linking is allowed.

Processing the Darshan Trace File

- After application execution, the trace file can be found in the Darshan log directory. The filename has the following naming format:

```
<user>_<experimentID>_<executable>_id<JOB_ID>_
<timestamp>.darshan.gz
```

- Darshan can create a PDF report from the trace file:

```
darshan-job-summary.pl <trace file>
```

- Or individual statistics can be viewed in text format:

```
darshan-parser <trace file>
```

Processing the Darshan Trace File

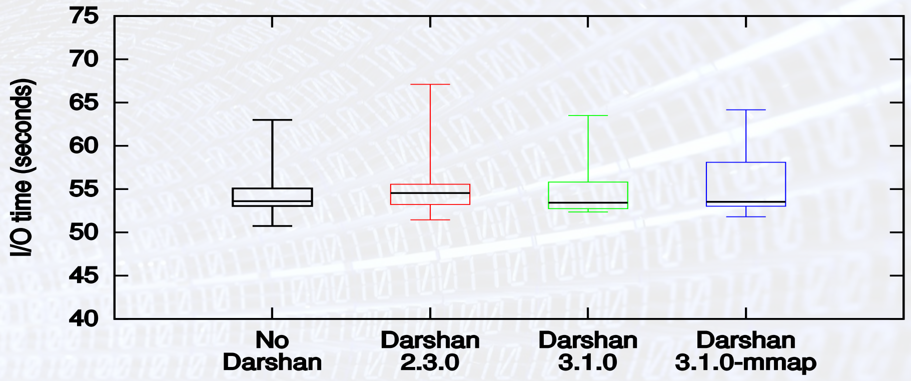
- PDF reports on individual files can also be created using:

```
darshan-summary-per-file.sh <trace file> \
<output-directory>
```

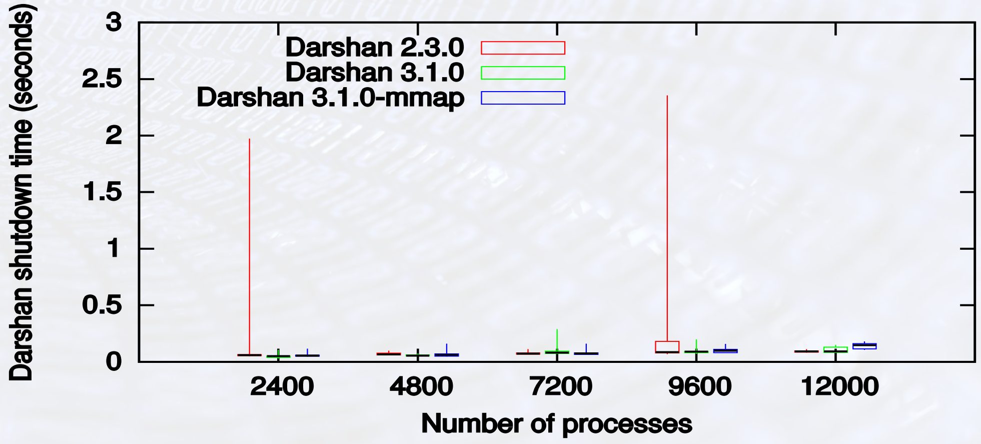
- This is useful to focus on performance metrics on specific input files or output files;
- The trace files are in binary format and are compressed with the zlib compression library.

Overhead of Darshan (1)

- Darshan overhead of a 6,000 MPI process job with one file per process [1]:



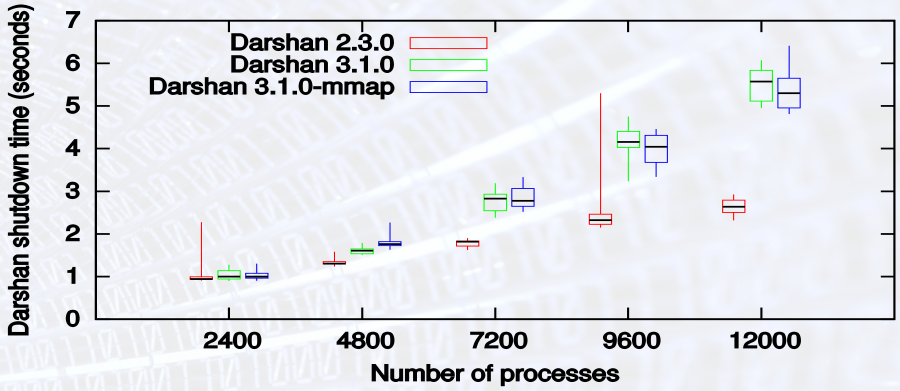
- The shutdown time for a shared file of Darshan is nearly constant with increasing MPI process counts.



[1] "HPC I/O for Computational Scientists: Understanding I/O", P. Carns, et al. ATPESC 2017

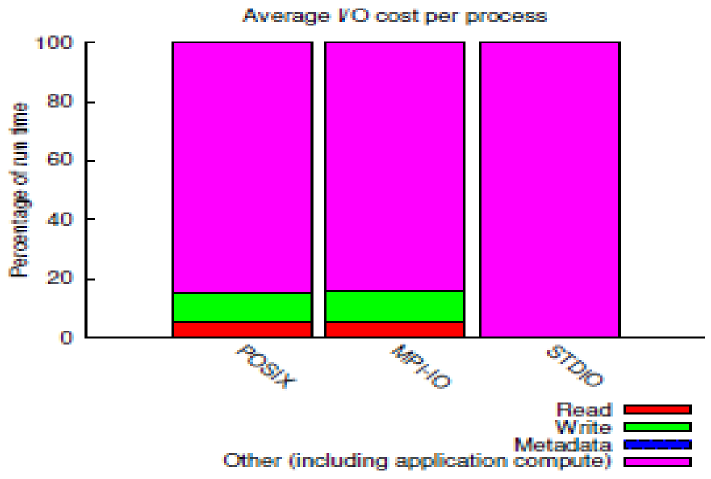
Overhead of Darshan (2)

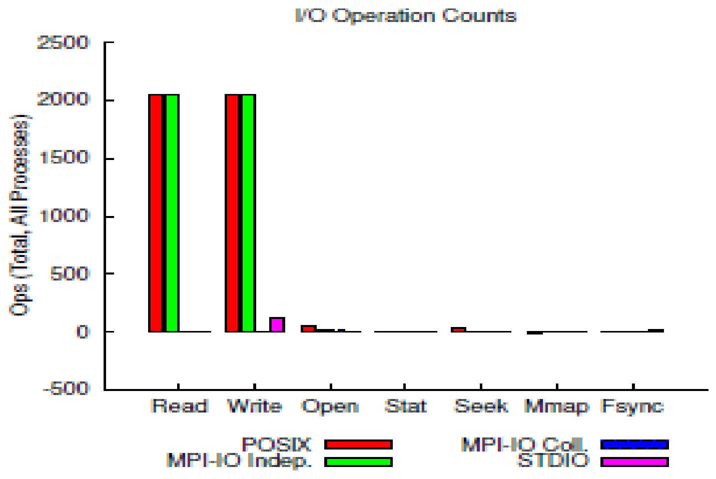
- The shutdown time for one file per process of Darshan scales linearly with increasing MPI process counts;

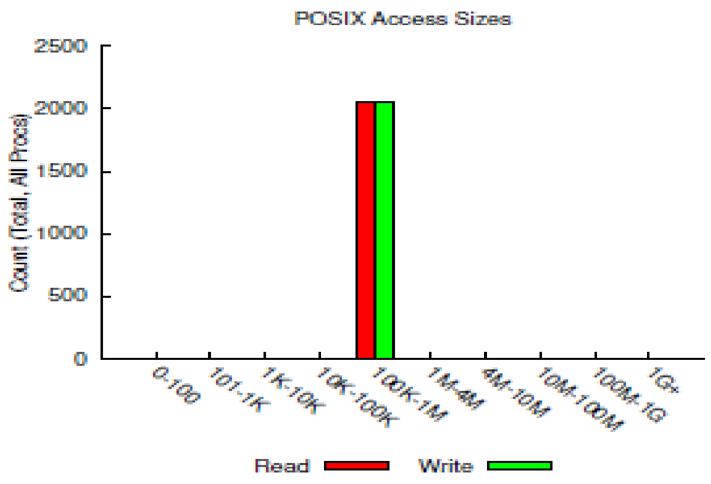


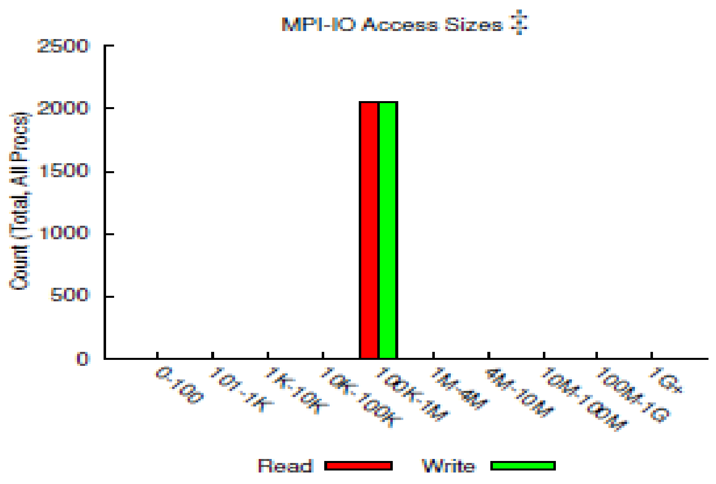
- Relative to the application shutdown time, the Darshan overhead is minimal. Some applications can take a number of minutes to shutdown at very large MPI process counts.

Darshan Report Graphs (MPI-IO)



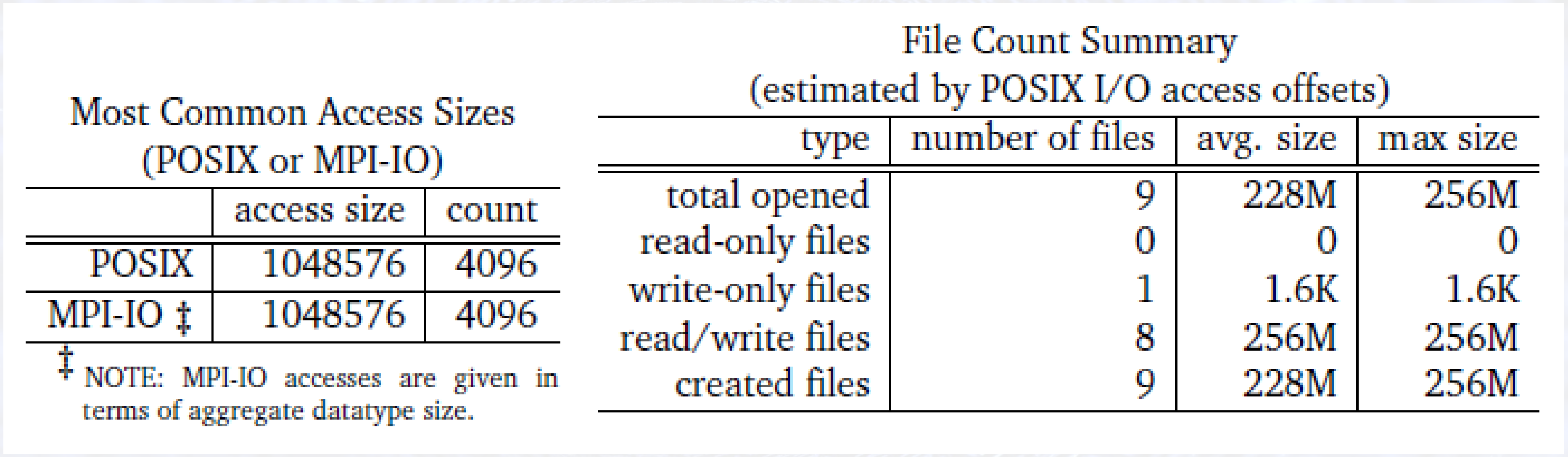






Darshan Report Tables (MPI-IO)

- Table below shows file statistics:



Most Common Access Sizes
(POSIX or MPI-IO)

	access size	count
POSIX	1048576	4096
MPI-IO ‡	1048576	4096

‡ NOTE: MPI-IO accesses are given in terms of aggregate datatype size.

File Count Summary
(estimated by POSIX I/O access offsets)

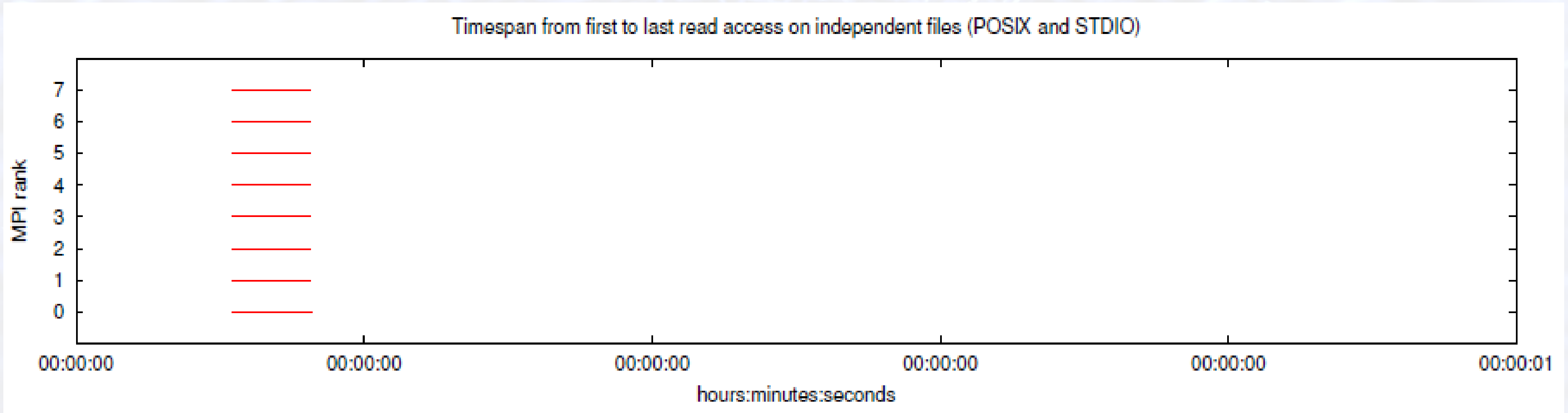
type	number of files	avg. size	max size
total opened	9	228M	256M
read-only files	0	0	0
write-only files	1	1.6K	1.6K
read/write files	8	256M	256M
created files	9	228M	256M

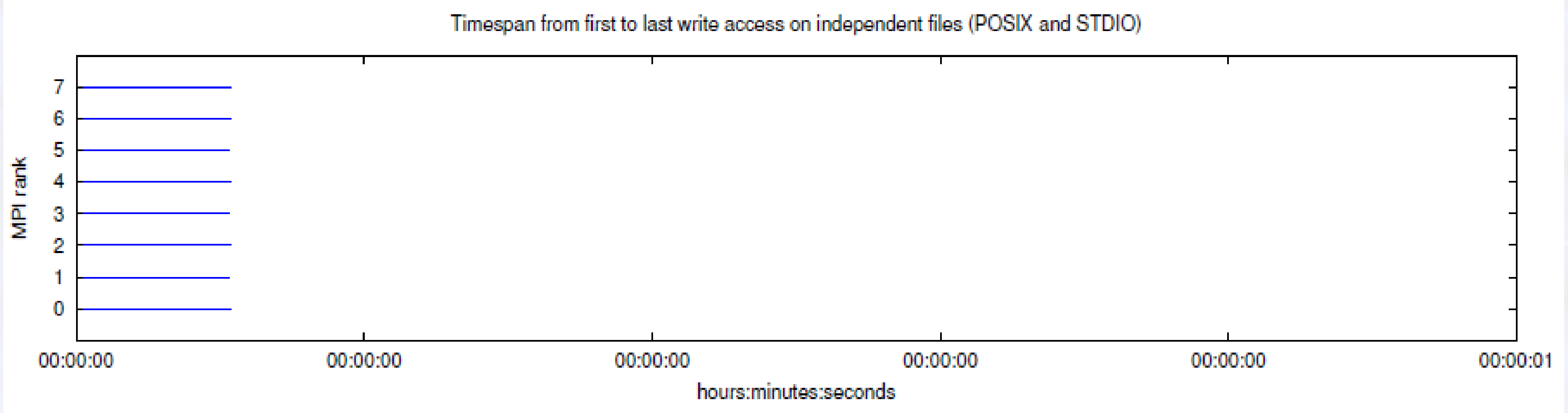
- Opening a large number of small files and/or a large number of I/O operation counts could be a cause of performance problems;

Darshan Report Tables (MPI-IO)

- Ideally, the code should have large access sizes as parallel file systems are bandwidth bound;
- Parallel NetCDF and parallel HDF5 appears as MPI-IO.

Darshan Timeline





Raw I/O Profiling Data (1)

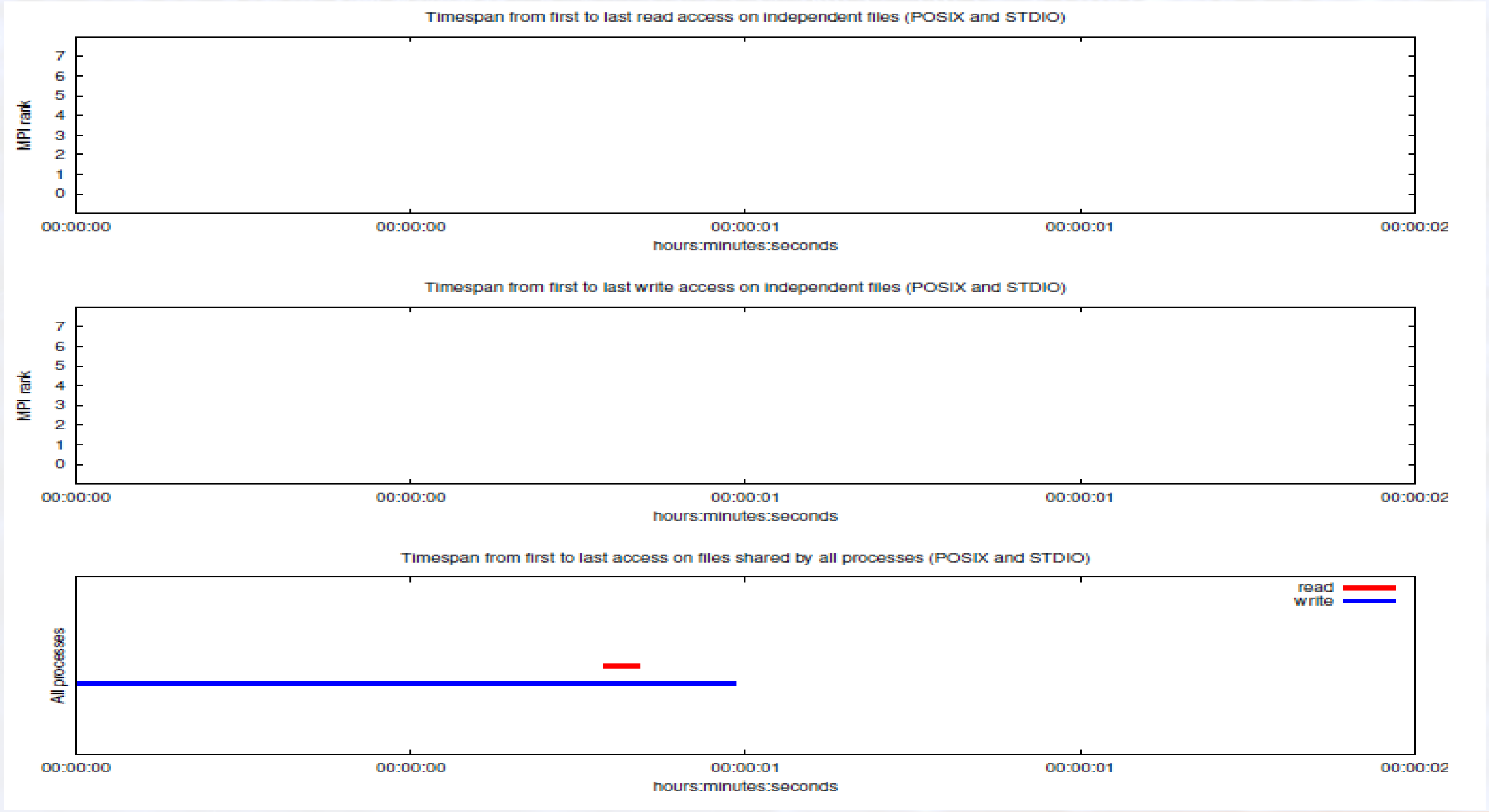
- The `darshan-parser` tool can be used to dump the raw I/O profiling data in text format;
- Number of POSIX read, write, open, seek, stat operations can be obtained for all MPI processes. Total bytes read/written by all MPI processes;
- Number of MPI read, write, open, independent/collective operations. Total bytes read/written by all MPI processes;

Raw I/O Profiling Data (2)

- Lustre file system metrics such as stripe size, stripe width (number of storage nodes over which the file is striped) and the storage node index;
- Plus a plethora of additional I/O metrics.

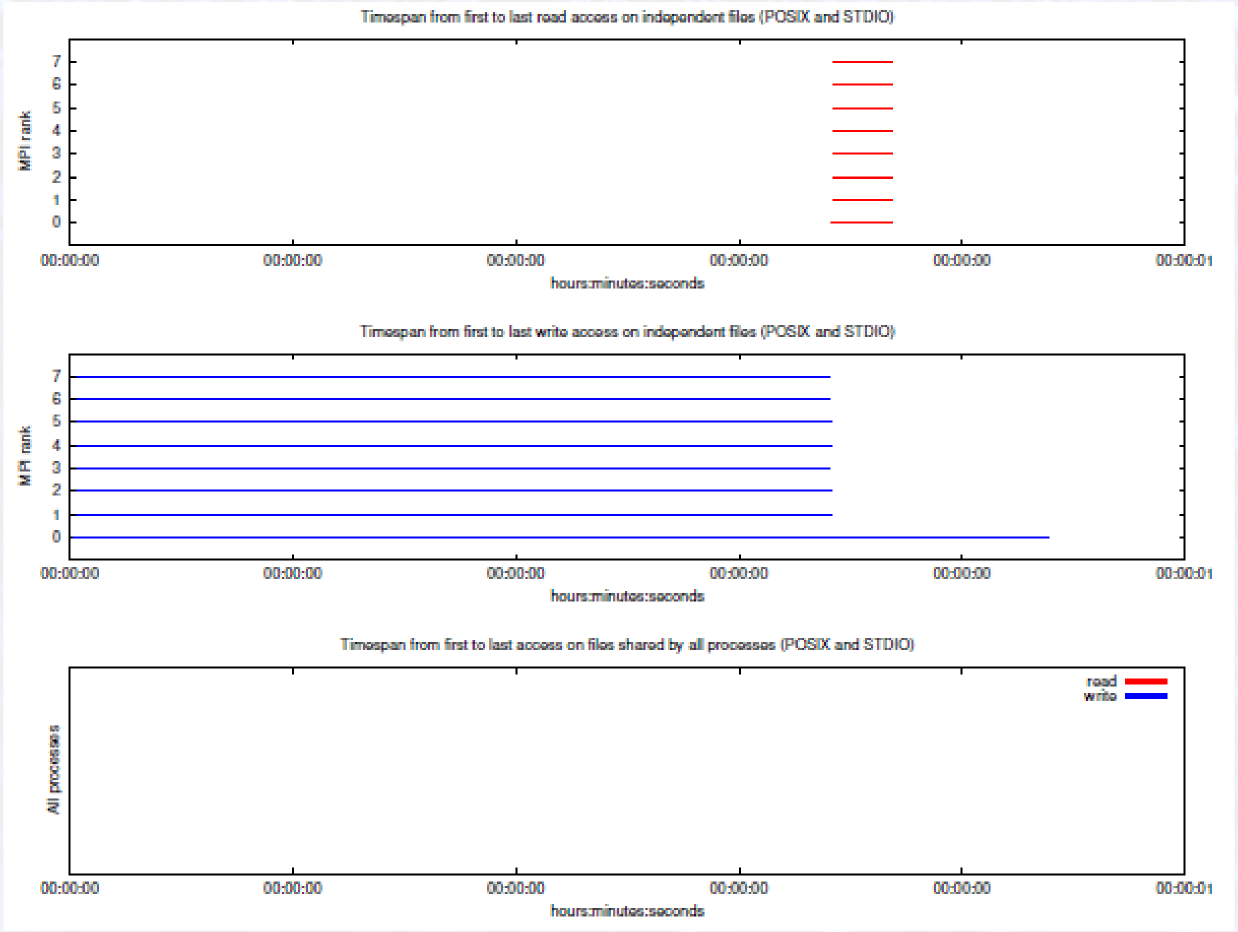
Shared File I/O Profiling (1)

- When multiple MPI processes are writing to a single shared file (N:1), timeline will not show per-process I/O:



Shared File I/O Profiling (2)

```
export DARSHAN_DISABLE_SHARED_REDUCTION=1
```



Extended Tracing (1)

- Darshan summarises profiling data;
- From version 3.1.3, the tool can also provide fine grained profiling metrics in plain text using the Darshan Extended Tracing (DXT):

```
export DXT_ENABLE_IO_TRACE=4

LD_PRELOAD=/lib/libdarshan.so mpirun -n 128 ./wrf.exe

darshan-dxt-parser <trace file>
```

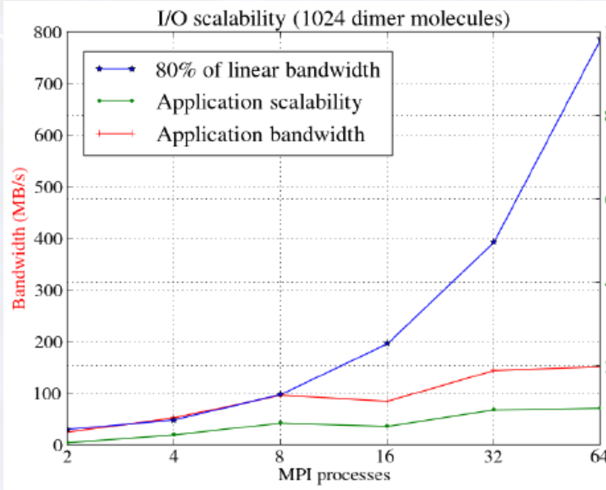
- The last command will print detailed metrics on every I/O segment for every file for every MPI process.

Extended Tracing (2)

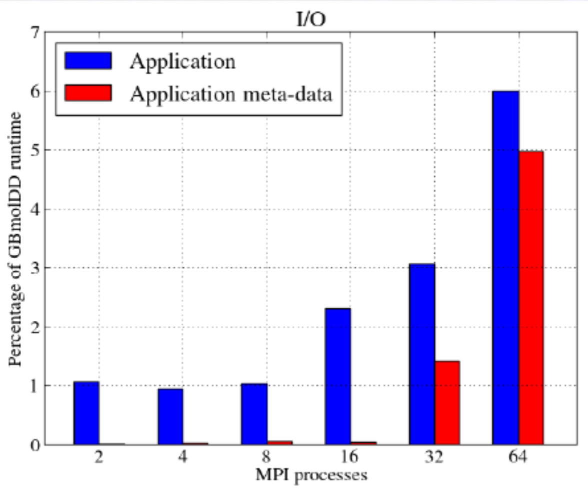
- Extended tracing also shows which I/O nodes were used for each segment;
- This information can be used to check that the I/O servers are load balanced evenly.

GBmolDD - Computational Chemistry POP Audit (1)

- GBmolDD simulates coarse-grained systems of isotropic and/or anisotropic particles. Uses the Lennard-Jones potential function to approximate interaction;
- Molecules' position, energy and temperature is written using one file per MPI process using POSIX I/O (N:N);



(a) I/O scalability



(b) I/O percentage of runtime

GBmolDD - Computational Chemistry POP Audit (2)

- For 64 MPI processes onwards, the I/O is spending more time in file metadata mode than in data mode;
- This is because a Linux inode has to be created for each file (64 MPI process run created 192 files);
- Parallel file systems have fewer metadata servers, so this quickly becomes a bottleneck;
- Recommendation was to use a parallel file format such as MPI-IO, parallel NetCDF or parallel HDF5.

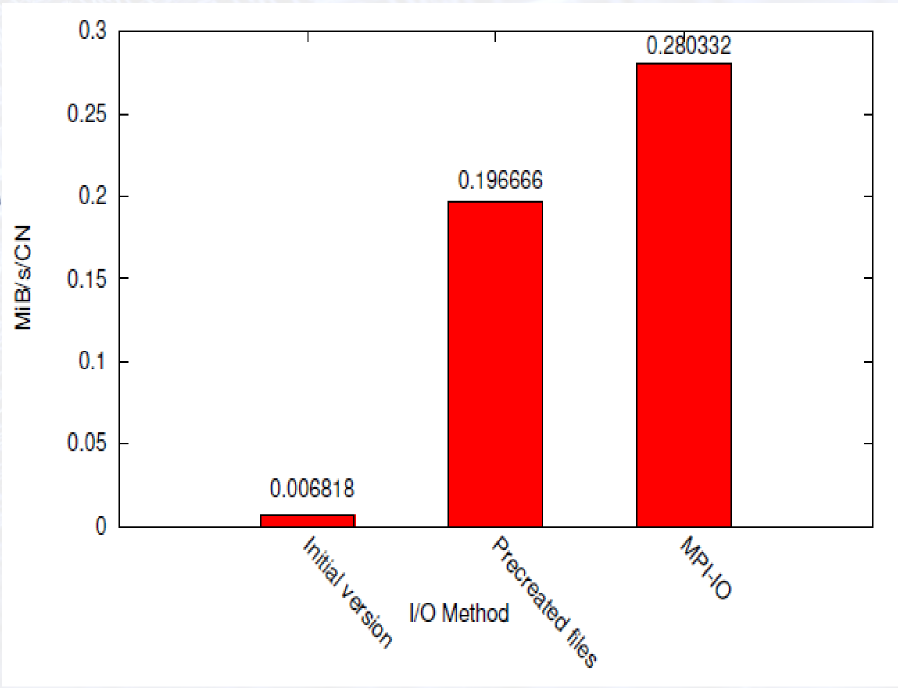
Combustion Physics Code (1)

- Combustion code [1] was checkpointing at intervals using one file per MPI rank (N:N);
- Writing 2^9 mesh points and creating two 20 GiB checkpoint files;
- The file creation time (Linux inode) was considerable and reduced the overall I/O bandwidth;
- Each checkpoint took 728 seconds to complete. The checkpoint files were pre-created prior to the simulation which reduced the I/O to 25 seconds;

[1] "Understanding and Improving Computational Science Storage Access through Continuous Characterization", P. Carns, et al. Proceedings of 27th IEEE Conference on Mass Storage Systems and Technologies (MSST 2011), 2011

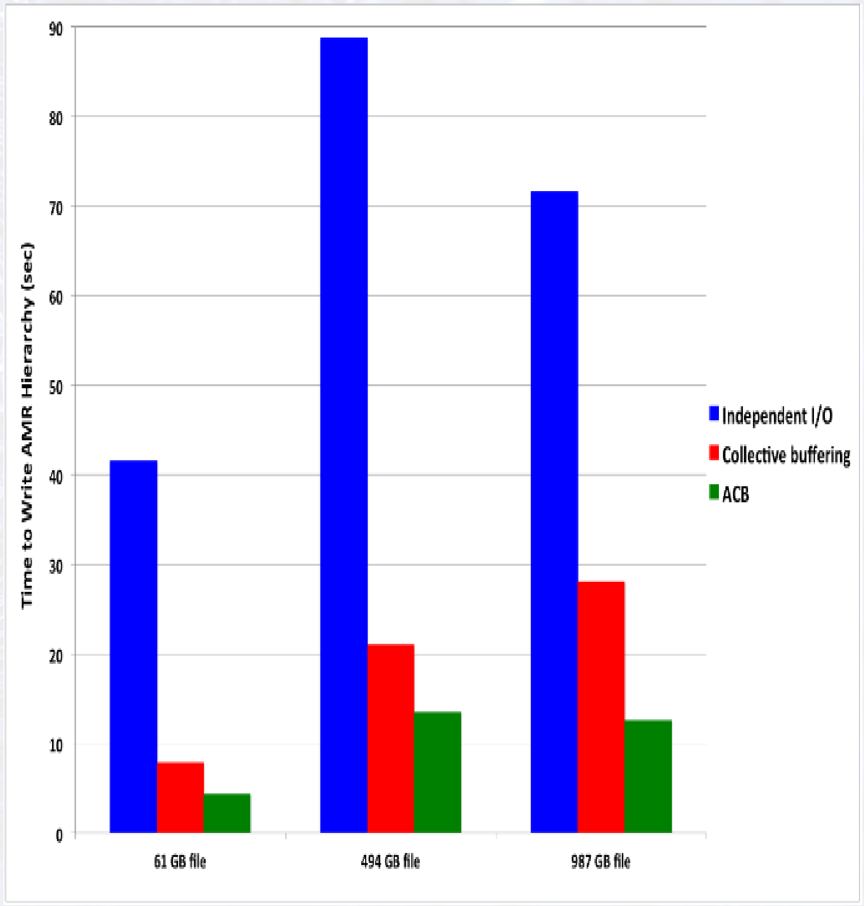
Combustion Physics Code (2)

- The code was profiled with Darshan to measure bandwidth per compute node and a shared file MPI-IO version was implemented;
- The code also used MPI collectives to aggregate write operations with block alignment to increase bandwidth;
- Writes of 1 to 4 MiB were aggregated to 16 MiB writes;
- Number of write operations was reduced from 16k to 4k.



Chombo - AMR PDE Solver (1)

- Chombo is PDE solver on adaptive meshes (AMR);
- Each MPI process writes its own box, resulting in a large number of independent write operations;
- Uses aggregated collective buffering (ACB). Performance was compared with MPI-IO collective buffering [1];



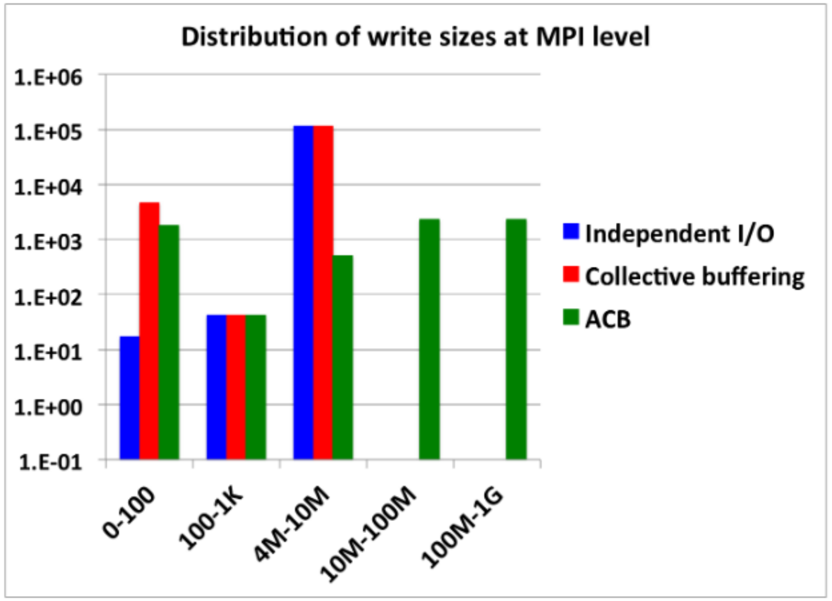
[1] "Collective I/O Optimizations for Adaptive Mesh Refinement Data Writes on Lustre File System", D. Devendran, et al. CUG 2016

Chombo - AMR PDE Solver (2)

- Darshan profiling showed the following characteristics:

	Ind. I/O	Coll. I/O	ACB. I/O
MPI-IO #writes	115268	119808	6912
Max access size	4 MB	4 MB	8 MB

- ACB is buffering:



Programming Tips (1)

- Parallel file systems are bandwidth bound, so try to write/read large amounts of data with fewer operations;
- Reduce the number of new files created. File creation is expensive;
- Avoid POSIX I/O - acceptable for configuration files but not for large data files;
- Write data contiguously to avoid expensive file seek operations;
- Avoid opening and closing files multiple times. Open it once, read/write the data and close it at the end;

Programming Tips (2)

- Use either MPI-IO, parallel NetCDF or parallel HDF5 for data. High level abstractions of MPI-IO and offer a convenient API;
- Use I/O aggregation for small writes;
- Configuration files should be read by a single process and then broadcasted to other MPI processes;
- Parallel I/O offers further optimisation opportunities using MPI-IO hints using:

```
MPI_INFO_SET( hints, key, value, ierr )
```

Programming Tips (3)

- If a shared file model is not suitable for your parallel file system, e.g. because of file lock contention, then try an N:M approach. N is the number of MPI processes and M is the number of files where M < N;
- What is the best approach? N:M, M:1 or N:1? Depends on the size of the file and number of MPI processes and how that is striped across I/O nodes;
- For very large MPI processes, create two communicators: (N - M) processes do computation and M processes do I/O asynchronously or use the M:1 model.

Programming Tips (4)

- Always profile your code! This should be included as part of acceptance testing;
- This should be done prior to every release to ensure that code changes/improvements have not slowed down the performance of your parallel code.

Experts in High Performance Computing, Algorithms and Numerical Software Engineering

www.nag.com | blog.nag.com | @NAGtalk