UNDERSTANDING I/O

ROB LATHAM
PHIL CARNES
Argonne National Laboratory

4:30-5:00pm, August 11, 2016
St. Charles IL
CHARACTERIZING APPLICATION I/O

How is your application using the I/O system, and how successful is it at attaining high performance?

- The best way to answer these questions is by observing behavior at the application and library level.
- In this portion of the training course we will focus on *Darshan*, a scalable tool for characterizing application I/O activity.
DARSHAN: CONCEPT
Goal: to observe I/O patterns of the majority of applications running on production HPC platforms, without perturbing their execution, with enough detail to gain insight and aid in performance debugging.

- Majority of applications – transparent integration with system build environment
- Without perturbation – bounded use of resources (memory, network, storage); no communication or I/O prior to job termination; compression.
- Adequate detail:
  - basic job statistics
  - file access information from multiple APIs
THE TECHNOLOGY BEHIND DARSHAN

- Intercepts I/O functions using link-time wrappers
  - No code modification
  - Can be transparently enabled in MPI compiler scripts
  - Compatible with all major C, C++, and Fortran compilers

- Record statistics independently at each process, for each file
  - Bounded memory consumption
  - Compact summary rather than verbatim record

- Collect, compress, and store results at shutdown time
  - Aggregate shared file data using custom MPI reduction operator
  - Compress remaining data in parallel with zlib
  - Write results with collective MPI-IO
  - Result is a single gzip-compatible file containing characterization information

- Works for Linux clusters, Blue Gene, and Cray systems
The **darshan-job-summary** tool produces a 3-page PDF file that summarizes job I/O behavior.

- Run time
- Performance estimate
- Percentage of runtime in I/O
- Access type histograms
- Access size histogram
- File usage

---

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

<table>
<thead>
<tr>
<th>type</th>
<th>number of files</th>
<th>avg. size</th>
<th>max size</th>
</tr>
</thead>
<tbody>
<tr>
<td>total opened</td>
<td>6000</td>
<td>512M</td>
<td>512M</td>
</tr>
<tr>
<td>read-only files</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>write-only files</td>
<td>6000</td>
<td>512M</td>
<td>512M</td>
</tr>
<tr>
<td>read/write files</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>created files</td>
<td>6000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

**Average I/O cost per process**

**I/O Operation Counts**

**POSIX Access Sizes**

**MPI-IO Access Sizes**

---

**I/O performance estimate (at the MPI-IO layer):** transferred 3072000.0 MiB at 48781.92 MiB/s
This graph (and others like it) are on the second page of the `darshan-job-summary.pl` output. This example shows intervals of I/O activity from each MPI process. In this case we see that different ranks completed I/O at very different times.
HOW TO USE DARSHAN

- Compile a C, C++, or FORTRAN program that uses MPI
- Run the application
- Look for the Darshan log file
- This will be in a particular directory (depending on your system’s configuration)
  - `<dir>/<year>/<month>/<day>/<username>_<appname>*.darshan`
    - Mira: see /projects/logs/darshan/
    - Edison: see /scratch1/scratchdirs/darshanlogs/
    - Cori: see /global/cscratch1/sd/darshanlogs/
- Use Darshan command line tools to analyze the log file

- Application must run to completion and call MPI_Finalize() to generate a log file
- Warning/disclaimer: Darshan does not currently work for F90 programs on Mira
- Opt in (i.e., by loading module or software key) at OLCF, LANL, LLNL and others: see site-specific documentation
AVAILABLE DARSHAN ANALYSIS TOOLS


- Key tools:
  - **Darshan-job-summary.pl**: creates pdf with graphs for initial analysis
  - **Darshan-summary-per-file.sh**: similar to above, but produces a separate pdf summary for every file opened by application
  - **Darshan-parser**: dumps all information into text format

Darshan-parser example (see all counters related to write operations):

```
“darshan-parser user_app_numbers.darshan.gz |grep WRITE”
```

See documentation above for definition of output fields
DARSHAN: EXAMPLES OF FINDING AND ISOLATING I/O PERFORMANCE PROBLEMS
**EXAMPLE: CHECKING USER EXPECTATIONS**

- User opened 129 files (one "control" file, and 128 data files)
- Should be one header, about 40 KiB, per data file
- This example shows 512 headers being written
  - Code bug: header was written 4x per file

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>jobid:</td>
<td>uid:</td>
<td>nprocs: 4096</td>
<td>runtime: 175 seconds</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Average I/O cost per process](image1)

- POSIX
- MPI-IO
- Read
- Write
- Metadata
- Other (including application compute)

![I/O Operation Counts](image2)

- Read
- Write
- Open
- Stat
- Seek
- Mmap
- Fsync
- POSIX
- MPI-IO Independ.
- MPI-IO Collect.

![I/O Sizes](image3)

- Count (Total All Processes)
- Access Size
- 0-100
- 101-1K
- 1K-10K
- 10K-100K
- 100K-1M
- 1M-10M
- 10M-100M
- 1G

![I/O Pattern](image4)

- Read
- Write

**Most Common Access Sizes**

<table>
<thead>
<tr>
<th>access size</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>67108864</td>
<td>2048</td>
</tr>
<tr>
<td>41120</td>
<td>512</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

**File Count Summary**

<table>
<thead>
<tr>
<th>type</th>
<th>number of files</th>
<th>avg. size</th>
<th>max size</th>
</tr>
</thead>
<tbody>
<tr>
<td>total opened</td>
<td>129</td>
<td>1017M</td>
<td>1.1G</td>
</tr>
<tr>
<td>read-only files</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>write-only files</td>
<td>129</td>
<td>1017M</td>
<td>1.1G</td>
</tr>
<tr>
<td>read/write files</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>created files</td>
<td>129</td>
<td>1017M</td>
<td>1.1G</td>
</tr>
</tbody>
</table>

Argonne National Laboratory
PERFORMANCE DEBUGGING OUTPUT

- Combustion physics application
  - Was writing 2-3 files per process with up to 32,768 cores
  - Darshan attributed 99% of the I/O time to metadata (on Intrepid BG/P)

| jobid: 0 | uid: 1817 | nprocs: 8192 | runtime: 863 seconds |

![Average I/O cost per process chart]

<table>
<thead>
<tr>
<th>File Count Summary</th>
<th>type</th>
<th>number of files</th>
<th>avg. size</th>
<th>max size</th>
</tr>
</thead>
<tbody>
<tr>
<td>total opened</td>
<td>16388</td>
<td>2.5M</td>
<td>8.1M</td>
<td></td>
</tr>
<tr>
<td>read-only files</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>write-only files</td>
<td>16388</td>
<td>2.5M</td>
<td>8.1M</td>
<td></td>
</tr>
<tr>
<td>read/write files</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>created files</td>
<td>16388</td>
<td>2.5M</td>
<td>8.1M</td>
<td></td>
</tr>
</tbody>
</table>
SIMULATION OUTPUT (CONTINUED)

- With help from ALCF catalysts and Darshan instrumentation, we developed an I/O strategy that used MPI-IO collectives and a new file layout to reduce metadata overhead.
- Impact: 41X improvement in I/O throughput for production application.

I/O performance with 32,768 cores

![Graph showing I/O performance with 32,768 cores. Initial version: 56 MiB/s, Precreated files: 1,611 MiB/s, MPI-IO shared file: 2,296 MiB/s.]

File Count Summary

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of Files</th>
<th>Avg. Size</th>
<th>Max Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>total opened</td>
<td>8</td>
<td>515M</td>
<td>2.0G</td>
</tr>
<tr>
<td>read-only files</td>
<td>2</td>
<td>2.2K</td>
<td>3.7K</td>
</tr>
<tr>
<td>write-only files</td>
<td>6</td>
<td>686M</td>
<td>2.0G</td>
</tr>
<tr>
<td>read/write files</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>created files</td>
<td>6</td>
<td>686M</td>
<td>2.0G</td>
</tr>
</tbody>
</table>

Average I/O cost per process

![Graph showing average I/O cost per process with POSIX and MPI-IO.]

Other (including application metadata)
PERFORMANCE DEBUGGING: AN ANALYSIS I/O EXAMPLE

- Variable-size analysis data requires headers to contain size information
- Original idea: all processes collectively write headers, followed by all processes collectively write analysis data
- Use MPI-IO, collective I/O, all optimizations
- 4 GB output file (not very large)
- Why does the I/O take so long in this case?

<table>
<thead>
<tr>
<th>Processes</th>
<th>I/O Time (s)</th>
<th>Total Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8,192</td>
<td>8</td>
<td>60</td>
</tr>
<tr>
<td>16,384</td>
<td>16</td>
<td>47</td>
</tr>
<tr>
<td>32,768</td>
<td>32</td>
<td>57</td>
</tr>
</tbody>
</table>
AN ANALYSIS I/O EXAMPLE (CONTINUED)

- **Problem**: More than 50% of time spent writing output at 32K processes. Cause: Unexpected RMW pattern, difficult to see at the application code level, was identified from Darshan summaries.

- What we expected to see, read data followed by write analysis:

- What we saw instead: RMW during the writing shown by overlapping red (read) and blue (write), and a very long write as well.
AN ANALYSIS I/O EXAMPLE (CONTINUED)

- **Solution:** Reorder operations to combine writing block headers with block payloads, so that "holes" are not written into the file during the writing of block headers, to be filled when writing block payloads.

- **Result:** Less than 25% of time spent writing output, output time 4X shorter, overall run time 1.7X shorter.

- **Impact:** Enabled parallel Morse-Smale computation to scale to 32K processes on Rayleigh-Taylor instability data.

<table>
<thead>
<tr>
<th>Processes</th>
<th>I/O Time (s)</th>
<th>Total Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8,192</td>
<td>7</td>
<td>60</td>
</tr>
<tr>
<td>16,384</td>
<td>6</td>
<td>40</td>
</tr>
<tr>
<td>32,768</td>
<td>7</td>
<td>33</td>
</tr>
</tbody>
</table>
EXAMPLE: REDUNDANT READ TRAFFIC

- Scenario: Applications that read more bytes of data from the file system than were present in the file
  - Even with caching effects, this type of job can cause disruptive I/O network traffic
  - Candidates for aggregation or collective I/O

- Example:
  - Scale: 6,138 processes
  - Run time: 6.5 hours
  - Avg. I/O time per process: 27 minutes

- 1.3 TiB of file data
- 500+ TiB read!

<table>
<thead>
<tr>
<th>File Count Summary</th>
<th>type</th>
<th>number of files</th>
<th>avg. size</th>
<th>max size</th>
</tr>
</thead>
<tbody>
<tr>
<td>total opened</td>
<td>1299</td>
<td>1.1G</td>
<td>8.0G</td>
<td></td>
</tr>
<tr>
<td>read-only files</td>
<td>1187</td>
<td>1.1G</td>
<td>8.0G</td>
<td></td>
</tr>
<tr>
<td>write-only files</td>
<td>112</td>
<td>418M</td>
<td>2.6G</td>
<td></td>
</tr>
<tr>
<td>read/write files</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>created files</td>
<td>112</td>
<td>418M</td>
<td>2.6G</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Transfer Per Filesystem</th>
<th>Write</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>File System</td>
<td>MiB</td>
<td>Ratio</td>
</tr>
<tr>
<td>/</td>
<td>47161.47354</td>
<td>1.00000</td>
</tr>
</tbody>
</table>
EXAMPLE: SMALL WRITES TO SHARED FILES

- Scenario: Small writes can contribute to poor performance
  - Particularly when writing to shared files
  - Candidates for collective I/O or batching/buffering of write operations

- Example:
  - Issued 5.7 billion writes to shared files, each less than 100 bytes in size
  - Averaged just over 1 MiB/s per process during shared write phase

![I/O Sizes](image)

<table>
<thead>
<tr>
<th>access size</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3418409696</td>
</tr>
<tr>
<td>15</td>
<td>2275400442</td>
</tr>
<tr>
<td>24</td>
<td>422899948</td>
</tr>
<tr>
<td>12</td>
<td>14725053</td>
</tr>
</tbody>
</table>
EXAMPLE: EXCESSIVE METADATA OVERHEAD

- Scenario: Very high percentage of I/O time spent performing metadata operations such as open(), close(), stat(), and seek()  
  - Close() cost can be misleading due to write-behind cache flushing  
  - Candidates for coalescing files and eliminating extra metadata calls

- Example:  
  - Scale: 40,960 processes for 229 seconds, 103 seconds of I/O  
  - 99% of I/O time in metadata operations  
  - Generated 200,000+ files with 600,000+ write() and 600,000+ stat() calls
METADATA SIDE TOPIC: WHAT’S SO BAD ABOUT STAT()?

- stat() is actually quite cheap on most file systems (and practically free on a laptop)
- But not a large-scale HPC I/O system!

- The usual problem is that stat() requires a consistent size calculation for the file
- To do this, a PFS has two options:
  - Store a precalculated size on the metadata server, which becomes a source of contention
  - Calculate size on demand, which might cause a storm of requests to *all* servers
- No present-day PFS deployments respond very well when thousands of processes stat() the same file at once
I/O UNDERSTANDING TAKEAWAY

- Scalable tools like Darshan can yield useful insight
  - Identify characteristics that make applications successful
    ...and those that cause problems.
  - It’s easy to use, in fact the technically hard part (instrumenting your application) may already be done

- Scalable performance tools require special considerations
  - Target the problem domain carefully to minimize amount of data
  - Avoid shared resources
  - Use collectives where possible

- For more information, see:
  http://www.mcs.anl.gov/research/projects/darshan
I/O PERFORMANCE TUNING
“RULES OF THUMB”

- Make sure you are using the right file system
  - Burst buffers if you have them
  - *Not your home directory*

- Use collectives when possible

- Use high-level libraries (e.g. HDF5 or PnetCDF) when possible

- A few large I/O operations are better than many small I/O operations

- Avoid unnecessary metadata operations, especially `stat()`

- Avoid writing to shared files with POSIX

- Avoid leaving gaps/holes in files to be written later

- Use tools like Darshan to check assumptions about behavior
  - It’s probably already instrumenting your code
THANK YOU!

THIS CONCLUDES “UNDERSTANDING I/O”

NEXT UP: “FUTURE OF I/O”