Profiling Input/Output of HPC Applications

HPC applications spend time in three phases: computation, communication (between processes) and input/output (I/O) that read/write data to persistent storage. Tools and techniques for profiling computation and communication are widely available as most codes spend time in these phases, but I/O is often neglected. However, with the advent of large HPC systems, I/O is now becoming a bottleneck, hence the need for optimising I/O which can only be done once the I/O characteristics have been profiled.

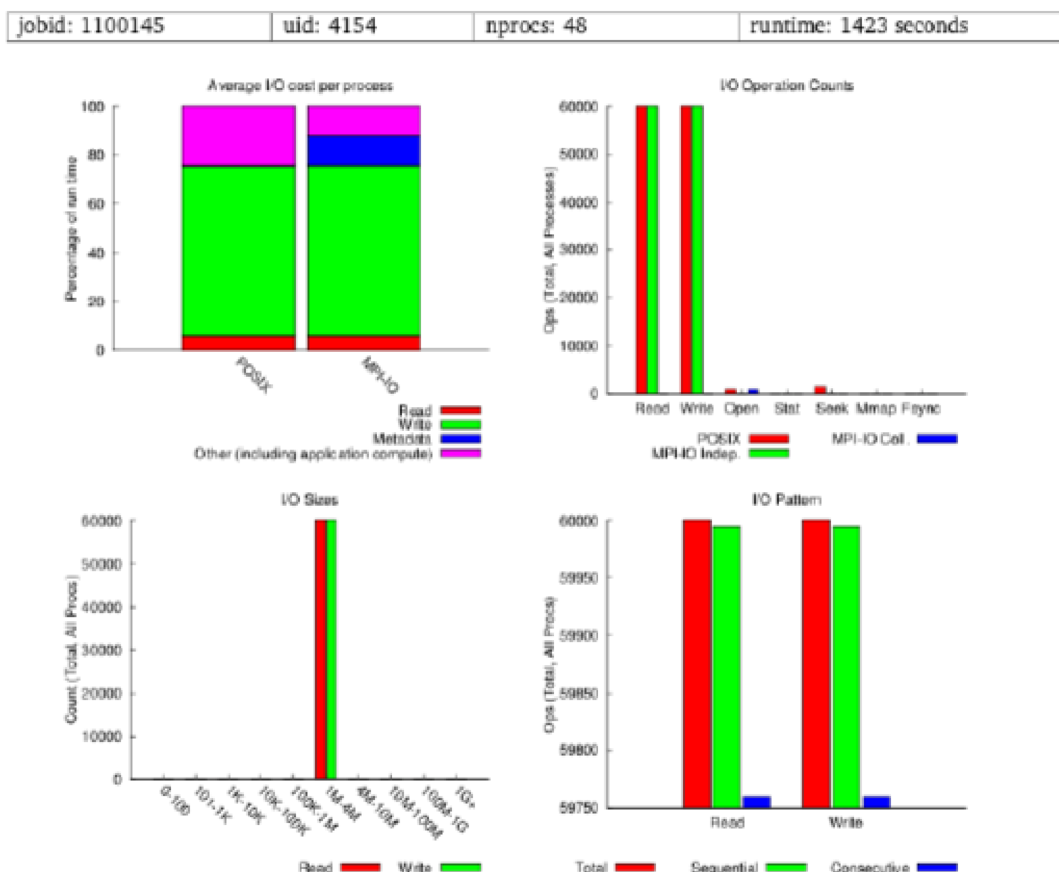
The I/O profiling tool that will be covered here is Darshan[1] which is an open-source lightweight tool for profiling I/O of MPI applications. Darshan is able to profile the following methods: a) POSIX I/O b) MPI-IO c) parallel NetCDF and d) parallel HDF5, and it is only able to profile codes written in C, C++ and Fortran. Darshan is invoked using the `LD_PRELOAD` Linux environment variable just prior to the code execution. For static executables, the application code must statically link the Darshan library. After the code completes Darshan creates a file in the form:

```
<USERNAME>_<BINARY_NAME>_<JOB_ID>_<DATE>_<UNIQUE_ID>_<TIMING>.darshan.gz
```

which will be referred to as <darshan file>.darshan.gz which is a zipped trace file. Once the above output file is created, use the Darshan Perl script to create a summary report:

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

Below is an example summary report created by Darshan for a 48 process MPI run:



---

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

Note that the Darshan report provides a summary report and not a time line report.

The “Average I/O cost per process” bar chart (top left) is showing POSIX I/O and MPI-IO statistics relative to the application run time and in which modes, namely read, write and meta-data. Note that meta-data operations can be costly and should be avoided where possible. Depending on the number of meta-data servers in the parallel file systems, meta-data operations can be serialised resulting in degraded performance.

The “I/O operation counts” bar chart (top right) shows how many I/O operations occurred either by individual MPI processes or collective I/O calls. This example is showing a large number of operations which can be costly. Ideally, an application should read/write a large amount of data with fewer I/O calls, thereby avoiding I/O overheads.

The “I/O Sizes” graph (bottom left) is showing the data sizes in I/O operations which shows that data sizes are within the range of 1 MB to 4 MB which is good as applications should avoid writing small amounts of data where I/O overheads will start to dominate.

The “I/O Pattern” bar chart (bottom right) shows the number of sequential and consecutive I/O operations. Consecutive I/O access means that data blocks are read with no gaps in between and sequential access means I/O access with irregular gaps in between which are costlier.

Darshan has other utilities which provide summary information. The following command gives statistics on every file accessed by the HPC application:

```
darshan-summary-per-file.sh <darshan file>.darshan.gz output-dir/
```

The following command produces a list of the files opened by an application and the amount of time spent performing I/O to each of them:

```
darshan-parser --file-list <darshan file>.darshan.gz
```

The above command prints a unique signature of each file called a *hash*. For statistics on a specific file of interest, use the following commands:

```
darshan-convert --file hash <darshan file>.darshan.gz \
interesting_file.darshan.gz
darshan-job-summary.pl interesting_file.darshan.gz
```

where *hash* is the hash from the previous command. For a complete listing of all data contained within a Darshan trace file in human readable form, use the command:

```
darshan-parser <darshan file>.darshan.gz
```

Darshan is an extremely lightweight and powerful tool for profiling I/O characteristics and stores all profiling data in buffers so it does not skew the profile. The profiling data is flushed to a trace file after `MPI_Finalize()` is called so it will not interfere with your application’s performance.