Exercises: Parallel IO Performance on ARCHER

David Henty (EPCC), Tom Edwards (Cray UK)

1 Introduction

The aim of these exercises is to investigate the way that parallel IO performance on ARCHER is affected by a number of factors including:

- interaction with other users;
- the Lustre file system settings;
- the choice of MPI-IO routines;
- data aggregation within the Cray MPI-IO library.

Running the exercises should also give you some feeling for the kinds of IO rates achievable on ARCHER, enabling you more easily to judge whether the IO phases of your own codes can or should be improved.

2 Test code

The test code is called `benchio` and is a simple Fortran program that performs parallel IO using exactly the same MPI-IO routines covered in the first part of the course. The major differences between this code and the one developed previously in the MPI-IO exercises are:

- the code writes data rather than reads it;
- it distributes a 3D array across a 3D process grid rather than being in 2D;
- both serial (Fortran `write`) and parallel (MPI-IO) output are implemented;
- all IO is timed and data transfer rates reported in MiB/s (one MiB is 1024^2 bytes).

3 Setup

Copy the file `benchio.tar` file from the ARCHER course web page to your workspace directory on ARCHER (`/work/y14/y14/guestXX/`) and unpack it. As well as the source code, Makefile etc. you should see three directories call `unstriped`, `striped` and `defstriped`. The code performs serial and parallel IO to each of these directories in turn to investigate the impact of different Lustre settings; note that the program removes each file immediately after it written to save disc space.

Before running the code, you should configure these Lustre directories as follows:

- remove the striping on `unstriped` by using `lfs setstripe` with a count of 1;
- set the striping on `striped` to the optimal (i.e. system-selected) striping by setting a count of -1;
- the `defstriped` directory should be left at its default settings for comparison.

Compile using `make` and submit the supplied batch script `benchio.pbs` to the reserved course queue. As distributed, this selects 3 nodes (i.e. 66 cores) and runs the code three times on 1, 8 and 64 cores.

Note that the program measures weak scaling, i.e. the array dimensions specified are those for the local array. This is simply replicated across processes, and hence the total data size (i.e. the size of the file) increases linearly with process count. The precise distribution of processes in the 3D grid is chosen automatically at runtime. See the Appendix for a more complete description of the code.

4 Initial exercise

You should try and answer the follow questions:

1. How do the IO rates change as the number of processes is increased? How is this affected by the choice of striping, or the use of serial or parallel IO? Can you explain this behaviour in terms of the way that MPI-IO and Lustre are interacting with each other?
2. Run the code several times: how variable are the IO rates?
3. Run on 512 processes (i.e. 22 nodes): can you explain these IO rates compared to 64 processes?
4. Find out the actual numbers of stripes being used by Lustre for default and "optimal" striping by issuing `lfs_getstripe` on a file in the appropriate directory. Note that files must have been created *after* the Lustre settings on the directory were last changed. You can either create a test file, or remove some of the `fdelete` calls in `benchio` so data files are not removed automatically.
5. Replace the collective MPI-IO call `MPI_File_write_all` in `mpiio.f90` with the individual call `MPI_File_write` and rerun the program. How do the parallel IO rates compare to those previously observed?

5 Extra exercises

As explained in the lectures, good parallel IO performance requires that data is written simultaneously to many IO devices and that this is done using a small number of large, contiguous IO transactions.

To achieve this, MPI-IO libraries attempt to aggregate data from multiple processes together into local buffers before writing to file. The Cray implementation can report quite detailed statistics about its internal operations which allow us to assess how successful its aggregation strategy is in practice.

1. In the PBS batch script, set the environment variable `MPICH_MPIIO_STATS=1` to turn on MPI-IO reporting. Re-run the parallel IO tests above and look at how these statistics vary depending on the Lustre settings. Does the observed performance correlate with these statistics?
2. Repeat this study using collective rather than individual IO. Again, does observed performance correlate with the statistics?

Conclusions

Hopefully these exercises have illustrated that fast parallel IO requires *all* the IO layers to be operating efficiently together: a single break in the chain can dramatically compromise the IO performance.

In particular, simply introducing MPI-IO into your code is not enough. Even having chosen the most efficient routines, it may initially have no substantial impact on IO performance. However, having implemented collective MPI-IO, adjusting the Lustre settings can then potentially increase performance by more than an order of magnitude. Most crucially, the benefit now grows with the number of cores as additional parallelism is utilised by both MPI-IO and Lustre.

Appendix: Details of benchio code (from README file)

The code does IO from a very simple configuration: a large 3D array of double-precision numbers distributed across a 3D grid of processes. It is written in Fortran. It was originally written as an MPI function to be called from a coarray code, then changed to pure MPI.

The *local* data size is specified by (n1, n2, n3). The process grid (p1, p2, p3) is chosen using the standard MPI function "MPI_Dims_create", with p1*p2*p3 = size, except that the order is reversed as MPI usually assumes a C-like convention (tending to prefer larger numbers of processes in the first dimension) whereas this code is in Fortran. The global data array is of size (l1, l2, l3) where l1=p1*n1 etc.

The local data arrays have halos on them to make it more realistic, i.e. they are declared as:

```
  double precision :: iodata(0:n1+1, 0:n2+1, 0:n3+1)
```

The data is written out in a decomposition-independent format, so there is a substantial amount of data rearrangement required. It would of course be easier to write out in processor order, i.e. all the data from rank 0, then rank 1 etc., but this is not what we want to do in a real program.

The array is initialised such that each non-halo element has a unique value in the range [1, l1*l2*l3]. Entries are set according to their *global* position so that, if the IO works correctly, the file should contain the numbers 1.0, 2.0, 3.0, ..., l1*l2*l3 in order.

To do this, the subroutine "mpiiowrite" is called with the local dimensions passed explicitly and the process decomposition described by a cartesian communicator. The IO within "mpiiowrite" is quite straightforward:

A subarray type called "filetype" is declared which describes which (n1, n2, n3) subsection of the global (l1, l2, l3) array each process owns; a process simply has to query its position in the cartesian communicator. This is then used as the filetype in "MPI_File_open", at which stage each process has defined what subsets of the global file it is going to access. This info is stored in the file handle "fh".

A subarray type called "mpi_subarray" is declared to describe what portion of the local data to write to the file. This is simply the core (n1, n2, n3) section excluding the halos.

Once this simple setup is done, all that is needed is a single collective call:

```
  call MPI_File_write_all(fh, iodata, 1, mpi_subarray, status, ierr)
```

and *all* the hard work is done by MPI-IO.

We can check the output using "od", e.g. to look at the double precision entries in the file skipping the first million doubles (i.e. 8 million bytes):

```
od –t fD –A d –j 8000000 striped/mpiionative.dat | head
08000000                   1000001                   1000002
08000016                   1000003                   1000004
08000032                   1000005                   1000006
08000048                   1000007                   1000008
08000064                   1000009                   1000010
08000080                   1000011                   1000012
08000096                   1000013                   1000014
08000112                   1000015                   1000016
08000128                   1000017                   1000018
08000144                   1000019                   1000020
```

As expected, the entries are one million+1, million+2, ...