Performance of Parallel IO on ARCHER

David Henty, Adrian Jackson, Charles Moulinec and Vendel Szeremi

June 15, 2015 – Version 1.0

1. Abstract

File input and output often become a severe bottleneck when parallel applications run on large numbers of processors. Simple methods such as performing all IO via a single master process are no longer feasible at scale. In order to take full advantage of the potential of the Lustre file system on ARCHER, IO also needs to be done in parallel.

In this paper we investigate the performance that can be achieved using parallel MPI-IO. We benchmark a simple test case of a three-dimensional distributed dataset . We find that MPI-IO can improve performance by orders of magnitude over naive serial IO, but that this requires some tuning of parameters. First, MPI-IO must be configured to use collective routines; second, the Lustre file system must be configured to use appropriate parallel striping.

2. Typical parallel IO patterns

Saving data to disk can be quite tricky even in serial. In essence, a file is simply a stream of bytes so a user may have to substantially rearrange their program data before writing it to disk. For example, if a weather model has three main arrays storing air velocity, pressure and temperature then it might make sense for all values for a particular grid point to be stored together in memory within the application (e.g. for performance reasons). However, in the file it might be preferable for the velocities for all gridpoints to be stored in sequence, then all the pressure values then the temperatures (e.g. to help in post-processing).

The problem in parallel is that data rearrangement is almost always required if the parallel code is to produce the same file as the serial one. For example, in a general domain decomposition parallel tasks do not own a single contiguous chunk of the global data set. Even in a simple 2D decomposition, the local data comes from many different locations in the file, with each local row coming from a different place.

This rearrangement implies communication between tasks during the IO phases, often in a new pattern that is not used within the computational phases. There are a number of ways to simplify this communication pattern which leads to four common IO strategies. Here we concentrate on the case of writing data: HPC codes typically write much more data than they read, and also writing is a more complicated (and typically slower) operation in parallel than reading.

Multiple files, multiple writers The simplest approach is to avoid the data rearrangement completely, with each task writing its data to a different file. In practice this does not avoid the issue, but simply delays it to a later time: subsequent post-processing or analysis programs will almost certainly have to access multiple files to read the data they require.

Single file, single writer This is the other extreme, where a single master task coordinates the data rearrangement, e.g. receiving rows from many tasks and reconstructing the global data set prior to writing it out. This pattern is also called Master IO.

Single file, multiple writers Here the data rearrangement is achieved by each task writing its data directly to the correct place in the file, e.g each individual row is written to a different location. Although this does not involve transfer of data between tasks, the tasks will still have to communicate to avoid their writes clashing with each other.

Single file, collective writers This sits between the two approaches above where either one or all of the parallel tasks perform IO; here we identify a subset of tasks to perform the IO. These IO tasks must communicate with the computational tasks to receive and rearrange the data, and must coordinate with each other to avoid IO clashes.

Note that, other than the "multiple files, multiple writers", all these methods should produce identical output to each other on any number of processors. However, they may have very different performance characteristics.

3. Lustre filesystem

Fundamentally, Lustre achieves performance by storing a single file across multiple disks; this is called *striping*. By understanding a few basic features of the parallel Lustre filesystem on ARCHER it is possible to identify the potential bottlenecks and limiting factors in the parallel IO strategies outlined above.

3.1. Lustre architecture

The Lustre architecture has four important components:

Object Storage Targets The OST's correspond to the actual storage devices; one OST may contain more than one physical disk.

Object Storage Servers The OSS's are the servers that perform the IO and are directly connected to one or more OST's. They can also communicate efficiently with the compute nodes running user jobs as they are directly connected to the high performance Aries interconnect.

Meta Data Server Each Lustre file system has its own MDS which stores all the information about the file, e.g. its location, access permissions and striping settings.

Lustre Clients Remote clients that can mount the Lustre filesystem, e.g. the compute nodes or the login nodes.

On ARCHER, each OST comprises a RAID6 array with 10 (8+2) disks; each OSS controls 4 OST's. The Lustre filesystem in /work is actually split into three 1.5 PB file systems, /fs2, /fs3 and /fs4, which have 12, 12 and 14 OSS's (48, 48 and 56 OST's) respectively.

The basic point is that Lustre is optimised for very large IO operations on contiguous blocks of data, as these can easily be parallelised across OSS's and OST's.

3.2. Achieving performance

Given the architecture of the Lustre parallel filesystem, the following potential issues can be identified for each of the parallel IO strategies described above (these are fairly typical for all parallel filesystems).

Multiple files, multiple writers This has the potential to achieve high bandwidth if different files are stored on different OST's. However, as individual files are likely to be small it is probably best not to stripe each file across multiple OST's. The major issue with this strategy is that accessing large numbers of files at once will stress the filesystem. For example, simply opening thousands of files may overload the MDS as this is a serial operation. This will limit the scalability of this approach.

Single file, single writer If only one process is writing then it cannot take advantage of striping: IO rates will be limited by the bandwidth achievable from a single process and will not scale.

Single file, multiple writers Although this could in principle benefit from striping if different processes access different regions of the file, in practice this is not the case as the IO system cannot know in advance if there will be any clashes. Typically the file will be locked be each process while it writes, which serialises the IO and introduces locking overheads. This approach would not be expected to scale.

Single file, collective writers As the IO is collective, i.e. the IO system knows that all processes are writing at the same time and can therefore gather global information on the IO patterns, this approach has the potential to achieve high bandwidth. The number of writers can be set to match the available number of OST's. By careful choice of which writers access which sections of the file, the IO system can guarantee that there are no IO clashes and avoid the need for file locking. Therefore, with appropriate file striping, this approach has the potential to scale.

3.3. Controlling striping

Lustre gives simple mechanisms for control over how many OST's a file is striped. For this reason, from now on we will talk exclusively about the number of OST's and not the number of OSS's.

Although striping is essential to achieving high bandwidth from parallel applications, it increases the access time for small files. Most user files are small (source code, object files etc.) and are accessed using serial programs (editors, compilers etc.) so having a very high level of striping for all files would adversely impact users. As a compromise, the default level of striping on ARCHER is set to 4. Files are striped in a round-robin (i.e. block-cyclic) manner with a fixed stripe size (the block size). By default this is set to 1 MiB, and although it can easily be changed we do not investigate the effects of different stripe sizes here.

To set the number of stripes to, for example, 16:

```
user@archer$ lfs setstripe -c 16 outfile.dat
```

will stripe the file `outfile.dat` across 16 OST's. If `setstripe` is used on a directory, it sets the default striping for all files subsequently created in that directory. In practice, this is the easiest way to control file striping.

A stripe count of -1 indicates that a file should be striped across all available OST's (around 50 on ARCHER). Information on the actual striping of a file or directory can be found using:

```
user@archer$ lfs getstripe outfile.dat
```

4. MPI-IO

For these studies we use the MPI-IO routines from the system-supplied Cray MPI library, which have been optimised for the Lustre file system.

4.1. Parallel IO model

The MPI-IO parts of the MPI standard allow users to perform parallel IO in a portable manner across different platforms. There are many ways to use different MPI-IO calls to achieve the same result, but here we choose the highest level of abstraction . The important point is that this approach delegates all the details of how the IO is actually performed to the MPI-IO library. A well-written library can then choose to do the IO in an optimal way.

In this approach, each process describes which portion(s) of the parallel file it owns, i.e. the mapping from the local to the global data layout. This is done using MPI derived datatypes, and in MPI-IO is called setting the file view. Note that, in general, a process will own many disjoint sections of the file. Just as in standard send and receive routines, MPI datatypes can be used to describe what sections of the local data array are sent (or written to file in MPI-IO). This often occurs when the local data arrays are defined with halos, but only the core data should actually be saved.

Having fully described the IO pattern, data can be written to file using two routines:

1. `MPI_File_write()`
2. `MPI_File_write_all()`

Routine 1 is non-collective, with each process acting independently of the others; this corresponds to the "single file, multiple writers" pattern. MPI-IO has to process each request individually, and as argued in Section 3.2. this will not be very efficient. As well as potentially serialising access to the file, each process will typically be making a large number of small IO transactions. For example, in a simple 2D decomposition each local row is written to a different position in the file.

Routine 2 is collective, i.e. we guarantee that all processes call this routine together; this corresponds to the "single file, collective writers" pattern. This allows the MPI-IO library to implement a number of important optimisations:

1. nominate a subset of MPI processes as writers, the number being selected automatically to match the available OSS's and OST's (i.e. the file striping);
2. aggregate data from multiple processes together before writing, ensuring a smaller number of larger IO transactions;
3. ensure that there are no clashes between different writers so that IO transactions can take place in parallel across all the OST's.

4.2. Test case

The test case is very simple: a 3D array of double precision floating-point variables distributed across processes using a 3D block decomposition. To better mimic a real application, the local arrays have depth-1 halos in every dimension. We do weak scaling studies, fixing the local volume so the global volume (i.e. the file size) L^3 scales with the number of processes P. For simplicity we restrict ourselves to cubic local volumes of size n^3 and decompositions with the same number of processes in each dimension.

Even with this simple setup, the IO pattern is surprisingly complex: if IO were done in the "single file, multiple writers" pattern then each process would have to perform n^2 individual writes of size n, distributed across the file.

4.3. Benchio code

The `benchio` code is available from the ARCHER web site - see *IO benchmarking code* at http://www.archer.ac.uk/training/course-material/2014/09/IO_DL/. It is written in Fortran 90, although the performance from an equivalent C code should be the same. As supplied, it writes the array in two ways:

Serial IO corresponding to the "single file, single writer" pattern. Here we simply call Fortran's built-in IO routines using "stream" access to ensure raw binary output. This gives us a baseline figure for the bandwidth achievable from a single process.

Parallel IO corresponding to the "single file, collective writer" pattern using `MPI_File_write_all`. We would expect this to give the highest bandwidth.

The program loops over three subdirectories and writes in serial and parallel to each of them: `defstriped`, `striped` and `unstriped`. The first directory should have the default striping (stripe count of 4) and the others should be set, using `lfs setstripe`, to have 1 stripe and full striping (stripe count of -1) respectively.

The IO pattern can be described very simply using the datatype creation routine `MPI_Type_create_subarray`; this is used to create the `filetype`. The same routine is used to create the `mpi_subarray` type which corresponds to the local data array with all the halos removed.

The data itself is stored in an array `iodata(0:n+1,0:n+1,0:n+1)`. To aid verification, the halos are set to -1 and the core data is set so that, if all the arrays are written correctly, the file should contain the double-precision values $1.0, 2.0, 3.0, \ldots, L^3 - 1, L^3$ in order.

Once the file has been opened with `MPI_File_open` and attached to the file handle `fh`, setting the file view and writing in parallel are deceptively simple using these two datatypes:

```
call MPI_File_set_view(fh, disp, MPI_DOUBLE_PRECISION, filetype, &
                       'native', MPI_INFO_NULL, ierr)
...
call MPI_File_write_all(fh, iodata, 1, mpi_subarray, status, ierr)
```

Notes:

- using `'native'` format ensures raw binary output;
- `MPI_INFO_NULL` means we provide no additional hints to MPI-IO about the best way to perform the parallel IO;

- we write a single `mpi_subarray`, i.e. the core L^3 block of `iodata` excluding halos, collectively from all processes in parallel;
- it is important to check the returned `ierr` values as, unlike the rest of MPI, errors in MPI-IO routines are not fatal by default;
- for simplicity, the serial IO code just writes the data from process 0 a multiple of P times to give an output file of the correct size – the values will not be the same as the parallel file.

5. Results

As described above, the benchio test code performs both *serial* IO (single file, single writer) and *parallel* IO using MPI-IO (single file, multiple writers and single file, collective writers).

5.1. Effects of striping

First we compare serial and parallel collective IO, and investigate how their performance is affected by the three choices of striping: no striping, default striping and full striping.

The results are shown in Figure 1 and Figure 2 for local data sizes of $n^3 = 128^3$ and $n^3 = 256^3$ doubles respectively. These are weak scaling plots: the total file size increases linearly with the number of processes.

As expected, Figure 1 shows that all the serial results are basically the same at around 500 GiB/s, unaffected by the striping. This is also the same value we get for the unstriped parallel IO; here the IO is effectively serialised and we easily saturate the bandwidth of the single target OST. This is a very important point to note: it is **not sufficient** simply to have parallel IO coming from the application; **the filesystem** must also be configured to operate in parallel.

However, having gone to the effort of implementing MPI-IO the bandwidth can now be improved by using striping with no further code changes. With the

default striping of 4, we achieve 4 times the serial bandwidth at 2GiB/s on more than a few hundred processes. With full striping, the bandwidth increases with process count up to a maximum of 14 GiB/s on 4096 processes (the largest case tested here). This corresponds to 64 GiB of data being written in under 5 seconds; using serial IO, or parallel IO without striping, takes around 150 seconds.

Figure 2 shows the same basic pattern with serial and unstriped parallel IO all achieving around 500 MiB/s. However, the effect of striping seems to be reduced by a factor of two compared to the smaller test case: we achieve 1 GiB/s and 8 GiB/s for default and full striping respectively. This might be improved by increasing the stripe size above the default of 1 MiB, although this was not investigated here.

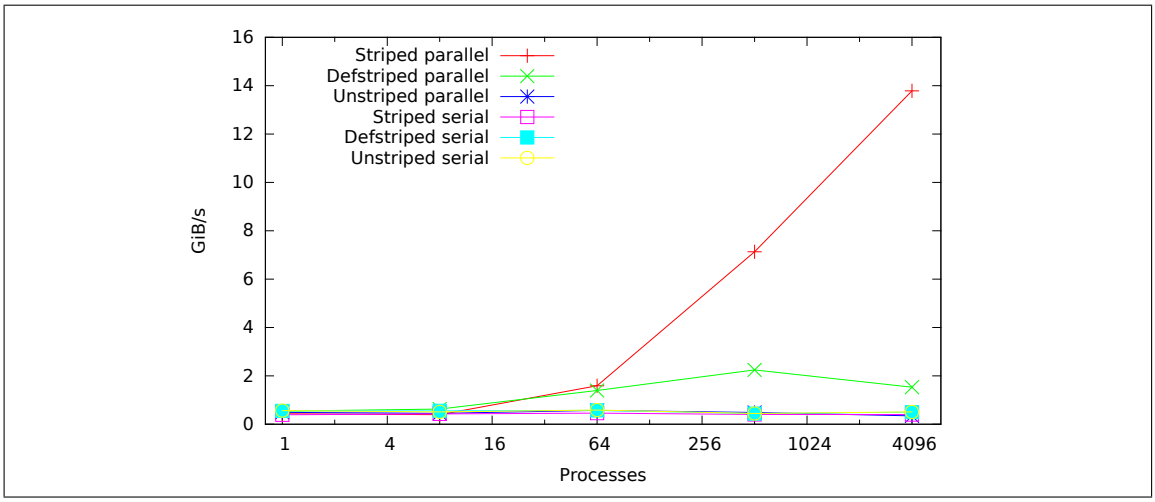

Figure 1: Write bandwidth for local data volume $n^3 = 128^3$

5.2. Collective vs non-collective IO

To investigate the effects of non-collective IO, the call to `MPI_File_write_all()` is simply replaced by `MPI_File_write()`. With the latter form, the MPI-IO library cannot assume that all processes are performing IO at the same time (although we know in practice that they are), so cannot apply any of the optimisations described in Section 4.1..

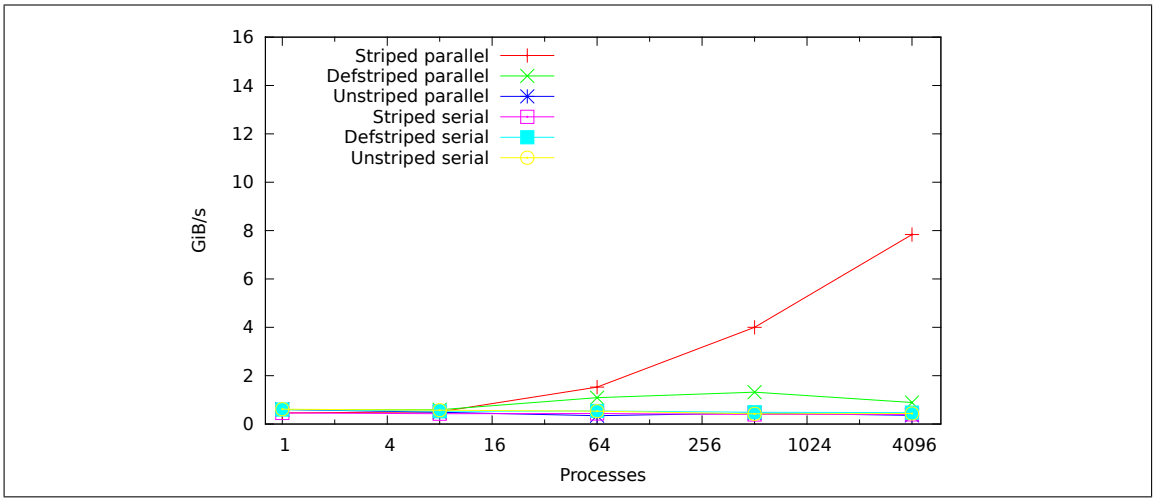

Figure 2: Write bandwidth for local data volume $n^3 = 256^3$

Processes	Individual (MiB/s)	Collective (MiB/s)
1	49.5	441
8	5.9	404
64	2.4	1630

Table 1: Bandwidth of Individual and Collective IO

The effects on performance and scaling are dramatic - see Table 1 where we used the default striping. The performance of non-collective IO is extremely poor, and degrades rapidly with increasing process count (we did not benchmark on more than 64 processes as the time taken was so long).

6. Conclusions

We have shown that it is possible to achieve very good IO rates on ARCHER with the MPI-IO library, of the order of 10's of GiB/s on a few thousand cores. However, this requires three things to be true:

1. IO must be done in parallel from the application;

2. the IO must be done in a collective manner, i.e. use the “single file, multiple writers” model;

3. the Lustre file system must use parallel IO to the disks, i.e. the files must be appropriately striped.

If either the application or the file system are not operating in parallel then IO rates are the same as a serial program: a single OST saturates at around 500 MiB/s. If the IO is not done collectively then bandwidths drops to a few MiB/s on as few as 64 cores

We would expect these results to apply in general to other IO libraries such as NetCDF and HDF5, i.e. good IO bandwidth requires parallel collective IO routines with striped files. We are currently investigating this in detail.

7. Acknowledgements

The authors would like to thank Dr Tom Edwards of the Cray Centre of Excellence at EPCC for his invaluable advice on configuring Lustre on ARCHER.