Parallel IO Benchmarking

Jia-Ying Wu

August 17, 2016

MSc in High Performance Computing with Data Science

The University of Edinburgh

Year of Presentation: 2016
Abstract

The project is aimed to investigate the parallel IO performance on ARCHER, motivated by the significance of file I/O in terms of the execution time in applications that require to work with large datasets. It extends the previous work, benchio [1][2], but works on a 2D data array rather a 3D dataset. The investigation covers a range of aspects, not only the effects of process counts in the IO, but also the effects of process decompositions that decide how processes are arranged in a process grid, distributions that describe the memory layout of the local buffer and that in the file visible on each processes, approaches that implement the distributions, file striping setting on Lustre file system, IO libraries, etc.

Each experiment conducted in the project may consist of multiple factors mentioned above, they are given to the benchmark by setting to the command arguments. In addition, an experiment may be carried out to investigate the IO performance for a factor with various values, for example, to study whether bandwidth is improved with increasing process counts, the provided Bash and PBS scripts allows the experiment to loop over each process count in a single job, a certain continuous executions of benchmark per iteration.

The experiment results show that MPI-IO can scale to at least 1024 processes with full striping on Lustre. With fixed process counts, simple decompositions where processes are arranged in vertical stripes ($1xP$ for $P$ processes) gives better performance than the other ones. With the same decomposition, block distributions can achieve better performance than block-cyclic distributions. NetCDF read can achieve as good performance as MPI-IO read does, but NetCDF write tends to be constant with increasing process counts. Non-collective IO is poor the read for very simple decompositions such $1x1024$ and $2x512$.

The experiment results also give the following recommendations: IO needs to be performed in a collective manner. Simple process decomposition and block distribution are required. MPI-IO is preferred to do parallel IO, but the dataset NetCDF coping with is portable. File striping on Lustre needs to be configured in a way that the process count matches the OSTs available to maintain the scalability.
Contents

Chapter 1 Introduction ................................................................. 1

Chapter 2 Background Theory ......................................................... 3

2.1 benchio ....................................................................................... 3

2.2 Parallel IO Libraries ................................................................. 3

2.2.1 MPI-IO .................................................................................... 4

2.2.2 NetCDF ................................................................................... 7

2.2.3 Summary ............................................................................... 10

2.3 Lustre® File System ................................................................. 11

2.3.1 Background Information of Lustre File System ......................... 11

2.3.2 Configuring the Lustre File System ......................................... 13

2.4 Achieving Performance with Collective IO on Lustre ....................... 13

2.5 IO Patterns ............................................................................... 14

2.5.1 Process Decomposition ......................................................... 15

2.5.2 Block Distribution .............................................................. 16

2.5.3 Block-cyclic Distribution ....................................................... 17

2.5.4 Summary ............................................................................. 18

Chapter 3 Experimental design ....................................................... 20

3.1 Machines and Environments .................................................... 20

3.1.1 ARCHER ............................................................................. 20

3.1.2 Lustre File System ............................................................. 20

3.2 Origin of the Code ................................................................... 20

3.3 Code Design ........................................................................... 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.1 Input Data and Output Data</td>
<td>22</td>
</tr>
<tr>
<td>3.3.2 Compilation and Execution</td>
<td>23</td>
</tr>
<tr>
<td>3.3.3 Design Overview</td>
<td>23</td>
</tr>
<tr>
<td>3.4 Testing</td>
<td>28</td>
</tr>
<tr>
<td>3.5 Choosing Data for Graphing</td>
<td>29</td>
</tr>
<tr>
<td>3.6 Test Cases</td>
<td>31</td>
</tr>
<tr>
<td>Chapter 4 Scalability of Process Count</td>
<td>32</td>
</tr>
<tr>
<td>4.1 Effects of Numbers of Processes</td>
<td>32</td>
</tr>
<tr>
<td>Chapter 5 Effects of Process Decompositions with Block Distribution</td>
<td>34</td>
</tr>
<tr>
<td>5.1 Approaches of Block Distribution</td>
<td>34</td>
</tr>
<tr>
<td>5.1.1 Subarray Approach</td>
<td>35</td>
</tr>
<tr>
<td>5.1.2 Darray-block Approach</td>
<td>36</td>
</tr>
<tr>
<td>5.1.3 Indexed-block Approach</td>
<td>37</td>
</tr>
<tr>
<td>5.2 Experiment Results</td>
<td>38</td>
</tr>
<tr>
<td>5.3 Summary</td>
<td>40</td>
</tr>
<tr>
<td>Chapter 6 Effects of Process Decompositions with Block-cyclic Distribution</td>
<td>41</td>
</tr>
<tr>
<td>6.1 Approaches of Block-cyclic Distribution</td>
<td>41</td>
</tr>
<tr>
<td>6.1.1 Darray-cyclic Approach</td>
<td>41</td>
</tr>
<tr>
<td>6.1.2 Indexed-cyclic Approach</td>
<td>44</td>
</tr>
<tr>
<td>6.2 Experiment Results</td>
<td>45</td>
</tr>
<tr>
<td>6.2.1 Various Block Sizes for 32x32 Decomposition</td>
<td>45</td>
</tr>
<tr>
<td>6.2.2 Fixed Block Size for Various Decompositions</td>
<td>46</td>
</tr>
<tr>
<td>6.3 Summary</td>
<td>48</td>
</tr>
<tr>
<td>Chapter 7 Effects of IO Libraries</td>
<td>49</td>
</tr>
</tbody>
</table>
7.1 Various numbers of processes ................................................................. 49
7.2 Various Decompositions over 1024 Processes ....................................... 50
7.3 Time on IO Routines ............................................................................... 51
7.4 Summary ................................................................................................. 54
Chapter 8 Effects of File Striping ................................................................. 55
  8.1 Effects of Striping .................................................................................. 55
  8.2 Effects of Collective and Non-collective IO .......................................... 57
  8.3 Summary ............................................................................................... 60
Chapter 9 Conclusions ............................................................................... 61
  9.1 Reflection .............................................................................................. 62
  9.2 Future Work .......................................................................................... 63
Appendix A  Provided Source .................................................................... 64
Appendix B  Running the Benchmark ........................................................... 65
References .................................................................................................... 70
List of Tables

Table 1 Command Arguments and Corresponding Fortran Variables..........................25
Table 2 The start and stride arguments for block and block-cyclic distributions implemented in NetCDF. .................................................................27
Table 3 Process decompositions over 1024 processes and the sizes of the local arrays. ..................................................................................................................34
Table 4 The buffer view and file view implemented by the Subarray approach. .......36
Table 5 The file view implemented by the Darray-block approach.............................37
Table 6 The file view implemented by the Indexed-block approach.............................38
Table 7 The file view implemented by the Darray-cyclic approach............................42
Table 8 How the block size is determined in the Darray-cyclic approach..................42
Table 9 The buffer view implemented by the Darray-cyclic approach........................42
Table 10 How aiBuffSubSizes is determined in the Darray-cyclic approach..............43
Table 11 An example of the Darray-cyclic approach..............................................43
Table 12 The file view implemented by the Indexed-cyclic approach.........................44
Table 13 How aiBlocklengths and aiDisplacements are determined in the Indexed-cyclic approach ..........................................................................................44
Table 14 IO bandwidth of collective and non-collective MPI-IO over 1024 processes, measured with different process decompositions, Subarray approach; defstriped. ......58
Table 15 IO bandwidth of collective and non-collective NetCDF over 1024 processes, measured with different process decompositions, Subarray approach; defstriped. ......59
Table 16 MPI-IO Reporting for MPI-IO Read ..........................................................59
Table 17 Key Files of the benchmark .....................................................................64
List of Figures

Figure 1 Architecture of Lustre File System; extracted from [8] ............................................. 12
Figure 2 File Striping on Lustre; extracted from [8] ............................................................... 12
Figure 3 The Coordinate System ............................................................................................ 15
Figure 4 Possible Process Decompositions for 4 processes ...................................................... 16
Figure 5 Block distribution ....................................................................................................... 17
Figure 6 Block-cyclic distribution with various block sizes ...................................................... 17
Figure 7 High Level Flow Chart ............................................................................................... 24
Figure 8 Test image - a 2-D image of size 480x216 .................................................................... 29
Figure 9 Process 0 reads block-cyclic distributed arrays stored in the test image, and writes in a block mapping to a common output ................................................................. 29
Figure 10 Write bandwidth produced by collective parallel MPI-IO with default file striping. Memory layout in both the local and global arrays is described by the Subarray approach of distribution. Data selected by the maximum, average and median of 20 runs .......................................................................................................................... 31
Figure 11 MPI-IO IO performance of different numbers of processes, using Subarray approach of distribution; defstriped ............................................................... 33
Figure 12 Write bandwidth for local volume n3 = 1283. Extracted from the benchio paper [1] ............................................................................................................................... 33
Figure 13 Describing the memory layout of the local array local on process 0 and the global dataset by block distribution ................................................................. 35
Figure 14 Read performance of Subarray, Darray-block, Indexed-block approaches of block distribution, over 1024 processes with different process decompositions; defstriped ............................................................................................................. 39
Figure 15 Write performance of Subarray, Darray-block, Indexed-block approaches of block distribution, over 1024 processes with different process decompositions; defstriped ............................................................................................................. 40
Figure 16 Comparison of IO performance over 1024 processes arranged in a 32x32 process grid, with Darray-cyclic and Indexed-cyclic approaches using various block
sizes; defstriped. For Subarray approach (block size 512x640; not shown in the figure), the read rate sits at 2.55 GiB/s and the write rate sits at 2.76 GiB/s

Figure 17 Read performance of different process decompositions over 1024 processes, using Subarray, Darray-cyclic, and Indexed-cyclic approaches; defstriped. Subarray is block distribution; the others are block-cyclic distribution with block size 16x16

Figure 18 Write performance of different process decompositions over 1024 processes, using Subarray, Darray-cyclic, and Indexed-cyclic approaches; defstriped. Subarray is block distribution; the others are block-cyclic distribution with block size 16x16

Figure 19 IO Performance of different IO libraries over different process counts, using Subarray approach; defstriped

Figure 20 IO performance of different IO libraries over 1024 processes with different decompositions using Subarray approach

Figure 21 Read time allocation for MPI-IO, measured on various numbers of processes and different striping settings, Subarray approach

Figure 22 Write time allocation for MPI-IO, measured on various numbers of processes and different striping settings, Subarray approach

Figure 23 Read time allocation for NetCDF, measured on various numbers of processes and different striping settings, Subarray approach

Figure 24 Write time allocation for NetCDF, measured on various numbers of processes and different striping settings, Subarray approach

Figure 25 Read performance affected by difference numbers of processes and different striping settings, Subarray approach

Figure 26 Write performance affected by difference numbers of processes and different striping settings, Subarray approach

Figure 27 Same as Figure 25 with defstriped and striped mpiwrite removed

Figure 28 Work Plan
Acknowledgements

I owe great gratitude to Dr David Henty, for consistent guidance, feedback and help throughout the project.
Chapter 1

Introduction

The factors regarding the performance of an application include the FLOP/s of CPU, memory bandwidth, network bandwidth, as well as file IO. For data intensive applications running on a large number of processes, the file I/O could outweigh the other factors and be considered a significant bottleneck in high performance computing.

Improvement in IO performance has been proven to achieved by using parallel IO libraries. Previous work has been done in developing a benchmark, benchio [1][2], using parallel MPI-IO [3], NetCDF [4] and HDF5 [5]. The benchio measures the bandwidth of write operations on a 3D dataset that is distributed across processes using a 3D block distribution. The measurement is a weak scaling model, where the local problem size on each process is fixed and so the global volume scales with the number of process. The target platform is ARCHER where parallel Lustre filesystem is configured with various file stripings in order to investigate the performance improvement achieved by collective writes.

Apart from how process count and stripe count over a file could affect the bandwidth, this project extends the previous work to investigate the other factors that could also affect IO performance: the process decompositions, and both block and block-cyclic distributions that describe how the global data is partitioned into processes. However, for simplicity, a regular 2D dataset is chosen as the problem in the project. Both read and write bandwidths are measured, and we do strong scaling studies to investigate how much the performance can be improved by collective IO performed by increasing the number of processes. MPI-IO and NetCDF libraries as well as file striping are also taken into account to investigate how well they could cope with individual decomposition and distribution and file striping on the Lustre file system on ARCHER.

Chapter 2 begins with an introduction to benchio and how it can achieve good write rates, followed by discussing the differences between benchio and this project. Next, the IO libraries adopted in the project are then introduced, an overview of Lustre File System is given and then why the IO libraries can attempt to achieve good performance on Lustre File System on ARCHER is explained. Finally, a review of data distributions is included to explain why block distributions can achieve better performance than cyclic distributions.
Chapter 3 gives a brief description of the ARCHER architecture and the file stripings on the Lustre chosen in the project. The software design of the benchmark is then presented. The chapter also explains how the code implementation of block and cyclic distributions is ensured correct. The reason of the maximum value of multiple trials is chosen to plot graphs is also explained here, and in the end, the available cases of experiments carried out in the benchmark is briefed.

Next, the experiment results are presented in following Chapters. Chapter 4 investigates the scalability of process counts, Chapter 5 studies the effect of process decompositions and the block distribution, while Chapter 6 extends the investigation of cyclic distribution. The effects of IO libraries and file striping are then discussed in Chapter 7 and Chapter 8 respectively. Finally, Chapter 9 draws conclusions.
Chapter 2

Background Theory

In this chapter we start with outlining the previous work and what it achieved, and briefing the background of the dissertation as well as explaining why it is relevant to achieve good performance.

An important point to note is that, to make it clear, the terms in the report listed below consistently mean:

1. **Decomposition**: a process decomposition represents how a group of processes in a communicator is organised to a 2D logical process grid.
2. **Distribution**: a distribution describes how a block of global data is divided and distributed across processes; discussed in the report are block and block-cyclic distributions.
3. **(Distribution) Approach**: the implementation of the data distribution using MPI derived datatypes to describe the data layout of the local array on each process (the buffer view) and that in the global file visible on each process (the file view).

2.1 **benchio**

benchio [1][2] is a parallel IO benchmark for MPI-IO [3], NetCDF [4], and HDF5 [5], writing a 3D array of double precision floating-point data across processes in a collective manner. The benchmark can run on any number of processes. It uses MPI_Dims_create() to split into a 3D grid; this will give the same number in each dimension for cubic numbers like 64, but will work in the general case as well.

The study shows that it is possible to achieve very good write rates (10’s GiB/s) on a few thousand cores on ARCHER with MPI-IO library, as long as the IO is done in parallel and in a collective manner, and the Lustre file system is configured with proper file striping. IO bandwidth resulted from the NetCDF and HDF5 is not much more than 2 GiB/s.

benchio is based on the previous work [13] which is a simple block distribution on a 1D dataset.

2.2 **Parallel IO Libraries**

In this section, we explain why parallel IO and IO libraries are important to achieve good performance and give an overview of the libraries relevant to the benchmark of the project.
For parallel applications, performance of calculation normally comes from using many processors at the same time; however, when it comes to doing IO, the master IO approach is often used. Opposite to the ideal of working in parallel, the master IO approach means that a single master process is in charge of the read/write from/to a single file and rearranging the IO data transferred to/from all other processes. Doing master IO is easy, applications need not consider unexpected IO clashes that would be a concern for parallel IO, but it does not perform very well as the extra time spent on interprocess communication is unavoidable and all the other processes tend to idle while the data rearrangement and the IO on the master process is in progress. Moreover, the master IO does not scale or take advantage of parallel file system as we can see in Figure 12 extracted from the benchio paper [1] that compares the write rates for both parallel and serial IO with different file striping settings on the Lustre file system [7]. The figure shows that the serial IO results basically remain unchanged, not benefiting from the process counts and file striping. IO systems are very complicated and performing parallel IO on a single file using conventional POSIX I/O approach is difficult, so the user wants high-level abstraction where the library takes care of all details to transfer the data in the correct places between the global file or the local memory on each process.

Both MPI-IO [3] and NetCDF [4] approaches hide the system specific details and provide abstract interfaces by which applications only need to provide a global description of the processes decomposition. Performing parallel IO using MPI-IO is virtually as simply as performing sequential IO using POSIX I/O except that each process describes global data layout using MPI derived datatypes and communicate them to MPI-IO by the call `MPI_File_set_view()`. NetCDF which is built on top of MPI-IO is more complicated due to the nature of its dataset that consists of the data itself and the metadata placed in advance to describe the data. In addition, each process delivers the global data layout to NetCDF directly by the read/write function `nf90_get_var()/nf90_put_var()` through the arguments `start`, `count`, `stride`, the use of which restricts the IO patterns available in NetCDF. Further details in the restriction will be explained in Section 3.3.3.4.2.

The details of MPI-IO and NetCDF are introduced in Section 2.2.1 and Section 2.2.2.

### 2.2.1 MPI-IO

MPI-IO [3] introduced in MPI-2 allows distributed-memory applications to take advantage of parallelism to increase IO bandwidth that can be hard to achieve if working at the level of POSIX I/O interface. MPI-IO supports not only independent but also collective and shared IO methods that enable applications to promisingly achieve good parallel IO performance with those methods properly chosen.

As MPI-IO library hides the details of how IO is actually performed in the POSIX I/O layer and instead provides with platform-independent interface, a portable application can maintain its portability with the use of MPI-IO, and the code implementing the IO is concise by using MPI-IO calls.
For an application that desires to perform the IO using MPI-IO, the following IO calls are typically used (this is the Fortran interface):

1. `MPI_File_open(comm, filename, amode, info, fh, ierror)`
2. `MPI_File_close(comm, fh, ierror)`

Routine 1. opens a file specified by the `filename` argument on the processes specified by the `comm` argument. The call is collective which means all processes participate in the IO must call this routine. Similar to its POSIX counterpart, the routine can set the file access mode specified in the `amode` argument to, for example, `MPI_MODE_CREATE`, `MPI_MODE_RDONLY`, `MPI_MODE_RDWR`, `MPI_MODE_CREATE`, etc. The `info` argument provides users to pass hints to the MPI-IO. Hints could be how many disks a file is striped, stripe unit and so on [6]. One of the problems of the `info` is that it is not portable, and if an implementation of MPI-IO does not understand a particular hint an application provides, it will be ignored and the application could not work as expected that may be difficult to be identified. In our benchmark, we do not pass any hints to control the stripe count of a file, instead we delegate this to the Lustre utility `lfs` [7] that is explained in Section 2.3.2 in detail. Specifying `MPI_INFO_NULL` indicates no hints are passed. The `ierror` argument exclusively for Fortran indicates the error status of the routine. Applications tend not to check error of MPI communication routines such as `MPI_Send()` as they terminate once `MPI_ERRORS_ABORT` occurs; however, it is very important to check the `ierror` argument for MPI-IO routines because the default error handler is `MPI_ERRORS_RETURN` that does not cause applications fatal by default. The `fh` argument is returned as the handle corresponding to the opened file for the subsequence calls to access the file.

Routine 2. closes the file associated with the `fh` argument. It is also a collective routine that must be called by all processes specified in the `comm` argument. All subsequent accesses to the file corresponding to the `fh` argument are invalid once `MPI_File_close()` has been called.

Below are the most commonly used routines for applications to read data from a file:

3. `MPI_File_read(fh, buf, count, datatype, status, ierror)`
4. `MPI_File_read_share(fh, buf, count, datatype, status, ierror)`
5. `MPI_File_read_all(fh, buf, count, datatype, status, ierror)`

Routine 3. is non-collective, each process behaving independently of the others. This routine reads data from the file associated with the `fh` argument to the local memory addressed by the `buf` argument. The `count` specifies the number of elements to be read, and the `datatype` represents the type of the elements. The status argument is the same as that used in MPI communication functions such as `MPI_Send()` or `MPI_Recv()`. Likewise, `MPI_STATUS_IGNORE` could be used when the caller is not interested in the status. The start location in the file from which data are read can be set
by MPI_File_seek() as one normally does with POSIX fseek(). Alternatively, MPI_File_read_at() can be used to perform MPI_File_seek() followed by the call MPI_File_read().

Routine 4. is shared, where the file pointer is shared by all processes in the communicator specified in the comm argument pass to MPI_File_open(). When a process calls this routine, the start location in the file from which data to be read (in the subsequent call by any process in the comm) is updated by the amount of elements read.

Routine 5. is collective, this ensuring itself to be called by all processes in the comm group together. When a process calls MPI_File_read_all(), MPI-IO exactly knows which other processes will call the routine, which allows MPI-IO itself to perform the read operations requested by the processes at the same time, as a result, if each process describes a non-overlapped portion in the file to be read, the time spent on reading the entire file can ideally be reduced by a factor of \( P \) where \( P \) is process count; note that this is only true if the increasing \( P \) gives accesses to more IO bandwidth, for example, increasing the number of IO clients.

MPI-IO also provides routines for application to write data to a file; the most commonly used are:

6. MPI_File_write(fh, buf, count, datatype, status, ierror)
7. MPI_File_write_share(fh, buf, count, datatype, status, ierror)
8. MPI_File_write_all(fh, buf, count, datatype, status, ierror)

The write routines are conceptually the same as the read counterparts, except the data transferring direction is inverse, from the local memory (buf) to the opened file (fh).

As mentioned above that a good IO performance can be achieved by all processes defined in the MPI_File_open() reading or writing a non-contiguous piece of the global file at the same time, each process needs to describe a portion of global data by using MPI derived datatypes to specify non-contiguous data layout in memory and file; this is achieved by the MPI_File_set_view() call. The details will be described in Section 3.3.3.4.1.

9. MPI_File_set_view(fh, disp, etype, filetype, datarep, info, ierror)

Routine 9. defines the map of the data layout of the local memory on each process to that in the global data. The fh argument represents the file handler. disp represents the start location (in bytes) in the file from which the data to be read or written. etype specifies the unit of data access. The filetype is the description of which part of the file is selected by the calling process for the later IO calls; it can be any MPI basic or derived data. etype and filetype could be any MPI basic or derive datatype. The datarep
defines in what datatype and in what order the data is stored in the file; the value can be native, internal and external32. native indicates that the data representation in the memory is the same as that in the file; however, it is not portable. By contrast, external32 format is portable but data pack/unpack is additionally required and thus much less inefficient. internal sits in between and is portable within the MPI-IO implementation. For performance native is used in the benchmark. The info argument is used to pass hints information, as used in MPI_File_open().

2.2.2 NetCDF

NetCDF library is a freely distributed [4] software package that allows applications to access array-oriented scientific data in a netCDF file. The library supports C, C++, Fortran and Java. netCDF dataset includes not only a series of raw data but also metadata in advance of the data; it is self-descriptive, machine-independent and portable. On the contrary, in the project, the IO data for MPI-IO is a stream of raw binaries, and it could not be understood by the other applications provided no additional documentation is supported.

Like MPI-IO library, NetCDF hides complex details of data transfer between the local memory and the global file and instead provides applications with interfaces to efficiently achieve good IO bandwidth.

To create or read a netCDF file means we need to understand the format of a netCDF dataset and the NetCDF library interfaces dealing with the dataset. A netCDF dataset contains a sequence of raw data as well as information, or so called header that describes the raw data. The header contains metadata such as dimensions, variables, and attributes, and each has a name and ID number acting as its identification.

Creating a netCDF file needs to enter two modes, define mode and then data mode. A dataset’s groups, dimensions, variables and attributes are defined in the defined mode, and data is written to the file during the data mode. To read a netCDF dataset needs to firstly open the dataset, get whose dimensions, variables and attributes, and finally read the data.

NetCDF library provides the following functions to start and end the sequence of a file access on an MPI parallel file system; this is the Fortran interface:

1. nf90_create_par(path, cmode, comm, info, ncid)
2. nf90_open_par(path, mode, ncid, bufrsize, cache_size, cache_nelems, cache_preemption, comm, info)
3. nf90_create(ncd)

Function 1. creates a netCDF dataset as parallel accesses, with the file name specified in the path argument, and enters the define mode. The cmode argument represents the creation mode flag that includes nf90_noclobber, nf90_netcdf4, nf90_mpiio, nf90_classic_model, etc. The mode flag nf90_mpiio passed to the function causes parallel IO to be enabled. Setting the nf90_HDF4 flag can cause a
netCDF-4/HDF5 file format to be created. The comm argument indicates which processes participate in the file creation. The info argument provides users to pass MPI info to the library as it does in MPI-IO; passing MPI_INFO_NULL means no hints are passed. The netCDF ID is returned in the ncid as a handler for the subsequent calls.

Function 2. opens a netCDF dataset as parallel accesses, with the file name specified in the path argument. The mode argument specifies the data access behaviour, which could be nf90_nowrite (for read-only access), nf90_write (for read-write access), and nf90_share that allows multiple accesses to the dataset concurrently. The mode also supports nf90_netcdf4, nf90_mpiio as the cmode in nf90_create_par() does. The function returns a netCDF ID to the ncid argument which can be used to refer to the dataset in all subsequent calls. The buffer size for the open dataset is specified by the bufrsize argument when the dataset is opened in classic format or opened as a 64-bit offset file. When the file is opened as a netCDF4/HDF5 file, the optional cache_size, cache_nelems, and cache_premtion arguments can be passed to specify the cache size in bytes, number of elements and the pre-emption value of the HDF5 chunk cache. The comm and info arguments are the same as what they are defined in the nf90_create_part().

Function 3. closes an open netCDF dataset. Note that if the dataset is in define mode, the function must be called right after nf90_enddef() has been called. nf90_enddef() is explained below in function 8.

NetCDF provides the following functions for a dataset in define mode to define dimensions, variables, and attributes for data write accesses:

4. nf90_def_grp(parent_ncid, name, new_ncid)
5. nf90_def_dim(ncid, name, len, dimid)
6. nf90_def_var(ncid, name, xtype, dimids, varid)
7. nf90_def_var_fill(ncid, varid, no_fill, fill_value)
8. nf90_put_att(ncid, varid, name, values)
9. nf90_enddef(ncid, h_minfree, v_align, v_minfree, r_align)

NetCDF supports hierarchical groups within netCDF datasets, and a group can be added by function 4. that creates a group named by the name argument and adds it to the group specified by the parent_ncid argument. The ncid of the new group is returned in the new_ncid argument.

Function 5. allows to add a new dimension to an open netCDF dataset; the dimension name is specified in the name argument. The len argument indicates the number of values stored in the dimension. The function returns a dimension ID in the dimid argument.

Function 6. adds a new variable to an open netCDF dataset. The variable name is specified in the name argument. The datatype of the variable is defined in the xtype
that should be of any predefined netCDF external data types. The \texttt{dimids} is a vector of dimension IDs each of which is created by \texttt{nf90_def_dim()}. The variable ID is returned by the function and stored in the \texttt{varid} argument.

Function 7. sets the fill parameter for a variable specified by the \texttt{varid} argument in the open netCDF dataset specified by the \texttt{ncid} argument. Setting non-zero value to the \texttt{no\_fill} argument enables the \texttt{no\_fill} mode that avoids NetCDF library writing fill values to the variable in a netCDF file corresponding to the data buffer to be written to the file, which can degrade the output performance as NetCDF library performs two writes. The Fill value is specified by the \texttt{fill\_value} argument that must be the same type as the variable; it will be ignored if NULL is passed or \texttt{no\_fill} is enabled.

Function 8. sets an attribute associated with the variable specified by the \texttt{varid} argument or a global attribute of the open dataset specified by the \texttt{ncid} argument. The \texttt{name} and \texttt{values} arguments are the attribute name and an array of attribute values to be set respectively.

Function 9. takes an application from the define mode to the data mode where applications actually start to write data to the open netCDF dataset. The netCDF file format has three sections arranged in the order: header, variables of fixed size and variables of unlimited dimension. [4]. No space between sections by default, but can be added by the \texttt{nf90\_enddef()} call. To do this, use the \texttt{h\_minfree} and \texttt{v\_minfree} arguments to define the size of the pad in bytes between the first two and the last two sections respectively. The \texttt{v\_align} and \texttt{r\_align} arguments specify the alignment of the beginning of the second and last sections. The default value of the two alignments are 4 bytes, or they can be set to \texttt{NF90\_ALIGN\_CHUNK} so the NetCDF library will use the chunk size as the align parameter.

In addition to the functions to set groups, dimensions, variables and attributes of a variable in an open netCDF dataset, NetCDF also provides functions to return the metadata:

10. \texttt{nf90\_inq\_ncid(ncid, name, grp\_ncid)}
11. \texttt{nf90\_inq\_dimid(ncid, name, dimid)}
12. \texttt{nf90\_inq\_varid(ncid, name, varid)}
13. \texttt{nf90\_get\_att(ncid, varid, name, values)}

Function 10. returns ncid of a group in the \texttt{grp\_ncid} argument according to the given ncid of the parent group (or a netCDF dataset) and the group name specified in the \texttt{name} argument.

Similarly, function 11. returns a dimension ID in the \texttt{dimid} argument according to the netCDF data set ID (ncid) and the dimension name (name) it receives, function 12. returns the variable ID (varid) of an open netCDF dataset (ncid) according to the variable name given by the \texttt{name} argument, and function 13. returns the attribute values (values) according to the attribute name (name) of a variable (varid).
Once the metadata of a netCDF dataset is set or available, an application can then write or read the raw data to/from the dataset by the following functions:

14. `nf90_var_par_access(ncid, varid, access)`
15. `nf90_get_var(ncid, varid, values, start, count, stride, map)`
16. `nf90_put_var(ncid, varid, values, start, count, stride, map)`

Accesses are collective by default when a netCDF file is created by `nf90_create_par()` or opened by `nf90_open_par()`, but can be changed to independent accesses by function 14. The `ncid` and `varid` arguments are used to refer to the netCDF dataset and whose variable on which the access to be changed. The access argument specifies the whether the access will be a collective operation (`nf90_collective`) or an independent operation (`nf90_independent`).

Function 15. reads data (`values`) from the variable (`varid`) of the open netCDF dataset (`ncid`). The portion of the global data to be selected by the calling process to be read is described in the `start`, `count` and `stride` arguments. The `start` argument is an array of indexes specifying the start location along each dimension in the file from which data are read, and the `count` argument is an integer array of integers specifying the data length in each dimension. The `stride` argument specifies an array of sampling intervals along each dimension of the variable.

Function 16. writes data (`values`) of the variable (`varid`) to the open netCDF dataset (`ncid`). Identical to the `nf90_get_var()`, the arguments `start`, `count`, and `stride` passed to the call define the part of the global file that is visible for the calling process.

### 2.2.3 Summary

Both MPI-IO and NetCDF libraries provide abstraction that allows parallel applications to fulfill parallel IO without caring about how file is separated across IO servers, and they can attempt to achieve good performance by creating a small number of large contiguous reads/writes. However, MPI-IO is more general in describing the map of its local memory to that in the file and to do read/write operations are quite different. MPI-IO achieves this by the MPI derived datatypes passed to `MPI_File_set_view()`, while NetCDF uses the `start`, `count`, and `stride` arguments passed to `nf90_get_var()`/`nf90_put_var()`. The mechanism of the former allows to describe data layout produced by simple block distribution or complicated block-cyclic distribution with any block sizes; however, that of the latter could work with block distribution and cyclic distribution (where block size is 1x1). Another difference between the use of the two libraries is that the data for MPI-IO is a raw binary stream but a self-descriptive dataset for NetCDF. Users who care the IO performance may prefer to work with MPI-IO, but those who intend to share their data would prefer NetCDF. The similarities and differences of the libraries motivate us to investigate whether and how
IO libraries can benefit IO bandwidth, and whether they produce similar or different IO performances.

### 2.3 Lustre® File System

Lustre® file system is an open-source, parallel file system that allows high-performance computing (HPC) clusters to achieve good IO bandwidth by striping a single file across multiple disks [1][7]. Lustre file system is a component of the Lustre software that contains on management server (MGS) and one or more Lustre file systems interconnected with Lustre networking (LNET) [7].

The section will give a brief introduction to the background information of the architecture of Lustre file system and how to configure a Lustre file system.

#### 2.3.1 Background Information of Lustre File System

The architecture is shown in Figure 1. A Lustre file system mainly consists of five components [7]:

- **Metadata Servers (MDS):** Each Lustre file system contains a MDS that allows Lustre clients to access metadata of the file system stored in the MDT through but not limited to Ethernet or InfiniBand network.
- **Metadata Targets (MDT):** Each MDT stores metadata on the storage (such as filenames, directories, permission, striping settings, etc.) For Lustre release 2.3 and later, each file system has one MDT; for release 2.4 and later, Lustre allows to have additional MDS nodes (each has its MDT) to the primary MDT that represent the file system root.
- **Object Storage Servers (OSS):** Each OSS node directly connects to one or more OST nodes, serves and handles file I/O requests to its local OSTs from the Lustre clients (compute nodes).
- **Object Storage Target (OST):** The OSTs correspond to the actual storage devices; each OST contains one or more physical disks. User file data is stored in one or more objects on individual OST.

Lustre Clients: The remote clients able to access the Lustre file system, for example, mount the file system, read a file, create a file, etc. They may be a compute node or a login node.
Figure 1 Architecture of Lustre File System; extracted from [8].

Lustre’s performance comes from striping files over multiple OSTs.

Figure 2 File Striping on Lustre; extracted from [8].

Figure 2 illustrates how a file is striped across multiple OSTs. Lustre utilises simple algorithms to allocate objects (files) to OSTs. It determines the number of OSTs to which a file is striped in a round-robin manner when the free space over OSTs is well balanced, or using a weighted algorithm when the free space across OSTs differs more than a
specific amount (17% by default). Striping on Lustre can allow very large files even when a single OST does not have enough free space as long as the aggregate space across OSTs do. Striping can also bring high IO performance which comes from the aggregate bandwidth from multiple clients accessing a single file.

Although striping is useful for parallel applications to achieve high IO bandwidth, it could also drawbacks: striping small files increases overhead which comes from more locks and extra network operations during the common operations such as `stat` and `unlink`, and from the contention between OSTs of OSSs. Moreover, striping may also increase risk of data lost resulted from one of the OSTs to which a single file is striped across breaks down. As a result, it is recommended to have lower striping level for small files on Lustre.

For a Lustre client to access a file on the Lustre file system, a request is sent to and received on the MDS node which then returns back the metadata including OST information of the file. Data transfer then performs directly between the clients and OSTs.

2.3.2 Configuring the Lustre File System

Lustre utility, `lfs`, allows users to set a file with specific level of striping and the size on each OST, display the striping pattern of existing files, set or display filesystem quota, etc. the project, however, only investigate the level of striping.

The default level of striping on ARCHER is set to 4 [9]; however, user can configure the number of objects per file is separated simply using the `lfs setstripe` command.

To set the number of stripes, use with -c option, for example, 8:

```
  lfs setstripe -c 8 output.dat
```

the file `output.dat` will be striped across 8 OSTs. The command can also be applied to a directory, and the striping setting is also applied to all files subsequently created within the directory. Setting the number to -1 results in striping the target file across all available OSTs on the file system.

To get the number of stripes of a file or directory, use the `lfs getstripe` command:

```
  lfs getstripe output.dat
```

2.4 Achieving Performance with Collective IO on Lustre

As mentioned in Section 2.2.1 and Section 2.2.2, high performance could come from collective IO patterns provided by MPI-IO and NetCDF libraries and striping on Lustre file system. In the section we discuss how they collaborate to benefit bandwidth in large scales.
In addition to the fact that both IO libraries can attempt to achieve good performance by creating a small number of large contiguous reads/writes, when collective access is chosen, MPI-IO and NetCDF library are aware of which processes are performing IO concurrently, they can therefore gather global information on the IO pattern. By carefully assigning which process to read/write which parts of the file that guarantees no sections are overlapped, good performance can be achieved by avoiding the overhead brought by file locking. On Lustre on ARCHER, each node is an IO client, and if we run an application on 1024 processes, we only have 43 IO clients (one node contains 24 cores), so we could reduce the contention. Each client accesses a different stripe that accesses a different OST, then we hope to get a scalable IO.

2.5 IO Patterns

Although working to investigate parallel IO performance on ARCHER as well as benchio had done, instead of using a 3D dataset as the test data, the project works on a regular 2D dataset, and processes are organised as a 2D grid accordingly. Furthermore, another difference between benchio and the benchmark in the project is that our benchmark measures both read and write bandwidths, and we are interested in not only the scalability of process count and how file striping on the Lustre can affect the IO bandwidth, but also the effect of process decompositions, IO patterns produced by various distributions, and IO libraries.

In the project, all experiments are carried out upon a 2D data array, and therefore, before discussing how data layout on each process is described to the IO system, we firstly define the coordinate system used in the project. Figure 3 illustrates the memory layout of an array. Memory locations are consecutive with consecutive i values in Fortran. That is, reading the whole array starts at the bottom-left data point, \(x(1, 1)\), moves along the i dimension until all array elements are accessed, then moves upwards to \(x(1, 2)\) and repeats the process until all data points are accessed.
2.5.1 Process Decomposition

To process a 2D data array in an MPI parallel application, processes can be arranged as a 2D logical grid, as shown in Figure 4, where 4 processes are arranged as a $1 \times 4$, $2 \times 2$, and $4 \times 1$ logical grid in a Cartesian coordinate, respectively. Each process then can be assigned an equal portion of the global data according to the process decomposition chosen. The benchmark in the project basically utilise two types of distributions: block and block-cyclic, the details are introduced in Section 2.5.2 and Section 2.5.3 respectively. An important point is that although the addresses of memory locations are consecutive in the $i$ direction in Fortran, MPI arranges processes in $j$ direction.
2.5.2 Block Distribution

Block distribution assigns blocks of consecutive data to each process. In this project, the block distribution we mention is that all processes are assigned an equal chunk of data from the global array, and the data distributed to each process may be contiguous or more or less non-contiguous in the global array, according to how processes are arranged in a Cartesian grid. As shown in Figure 5, the global array of 16 elements is partitioned into horizontal stripes for the 1x4 process grid, 2x2 data blocks for the 2x2 process grid, and vertical stripes for the 4x1 process grid. The numbers in the data arrays represent the addresses of the data in the global array and in the file, and the colours denote to which process the data in the global array is distributed by which process the data in the file is accessed.

It is evidently that when the process decomposition organises processes a 1x4 grid, each process is assigned a consecutive piece of data from the global array, and if we want each process to transfer its local data to/from the file in a way that the data order in the global array is the same as that in the file, each process also can access contiguous memory locations in the file. By contrast, the 4x1 process decomposition yields the most complicated distribution pattern because the data elements assigned to each process are completely non-contiguous in the memory of the global array, and each process must access non-contiguous memory locations in the file to ensure the data order in the global array is consistent with that in the file. We would expect doing collective parallel IO with the 1x4 decomposition collaborated with block distribution could achieve better performance than the other decompositions do with block distribution. The 4x1 decomposition may be the most difficult decomposition as data in consecutive memory
locations in the global array and file are accessed completely by distinct processes. The $2 \times 2$ decomposition may sit in between; however, it may be competitive to the $1 \times 4$ decomposition for large problems.

2.5.3 Block-cyclic Distribution

Apart from block distribution, the global data can also be distributed in a block-cyclic manner. The $1 \times 4$ decomposition chosen in Figure 5 gives no difference between block distribution and the block-cyclic distribution where the unit for cyclic is a $4 \times 1$ data block. Similarly, in $4 \times 1$ decomposition, the block-cyclic distribution width block size $4 \times 1$ can
produce the same results as the block distribution does. Figure 6 shows the how block-
cyclic distribution works with the 2x2 decomposition where the block size is chosen as
1x1 and 2x2. We can see even the same decomposition has various distributions that thus
yield different maps of the memory locations between the array local to each process and
the global array. Even in the same 2x2 decomposition with the same block size 2x2
chosen, the memory layout on a process generated by block distribution may be different
from that generated by block-cyclic distribution, and the file view may also be different.

2.5.4 Summary

The memory layout of the 1x4 decomposition illustrated in Figure 5 could be recognised
as a faster IO pattern compared with the other cases in the same figure and those in Figure
6, because the processes arranged in a 1x4 grid access contiguous pieces of data in the
file yet the other cases do not and may need rearrangement of data before/after
transferring data between the local array and the global file. We can easily observe the
various levels of complexity of different process decompositions and data distributions
from the simple example of 4 processes arranged in a 2D logical grid. There is already a
large number of possible 2D decompositions to investigate, instead of following the
benchio studying on a 3D dataset that has more variables and is more complicated, a 2D
dataset is chosen in the project for simplicity. The possible 2D decompositions of 1024
processes are illustrated in Table 2 that will be used in Chapter 5.

The same number of processes can be decomposed in various manners, even the small 4
processes can as well. Furthermore, in the same decomposition, the global data can be
split up to processes in a block or block-cyclic manner, and block-cyclic distribution with
different block sizes can produce different results. As a result, for a fixed process count,
processes can have a number of use cases describing the memory layout of the local data
and in the global array. For example, 4 processes arranged in a 2D grid have at least 5
mappings of data layout between the local array and the global array, as shown in Figure
5 and Figure 6, not mention to 1024 processes to which a large 16384x10240 data array
is distributed as shown in Table 2 in Chapter 5.

Different distributions may produce the same file view for processes; however, the code
complexity of implementation may vary. In general, block distribution is simpler to be
implemented than block-cyclic distribution as applications require more data
rearrangements to describe memory mapping for block-cyclic distribution. Even the
same distribution has different implementations. The implementations are explained in
Chapter 5 and Chapter 6 in detail when doing IO with MPI-IO and in Section 3.3.3.4.2
when with NetCDF. The various combinations of decompositions, distributions, block
sizes for a fixed process count, combined with the implementations of distribution
collaborating with IO libraries give us motivations to investigate how well IO libraries
can cope with and how they affect IO performance.

Figure 5 and Figure 6 show that complicated IO patterns can arise from unbalanced
decompositions (e.g. 4x1) with simple distributions (block) or balanced decompositions
(2x2) with complicated distributions (cyclic). We therefore investigate both aspects here,
extending the work of benchio [1][2] which only looked at balanced decompositions and simple distributions.
Chapter 3

Experimental design

3.1 Machines and Environments

3.1.1 ARCHER

ARCHER is the UK National Supercomputing System. At the time of writing this report, ARCHER consists of 4920 compute nodes, each of which contains two 12-core Intel Ivy Bridge series processors. Memory on each node is 64GB, while nodes with large memory can have 128 GB.

3.1.2 Lustre File System

ARCHER supports a high-performance Lustre file system that allows multiple IO clients to read or write a single file in parallel. The Lustre also supports file striping that allows a single file to be stored across multiple disks. In the project, three striping settings are used to benchmark the IO performance, which we call in the project as unstriped (stripe count 1), defstriped (4 stripes, which is default), and striped (optimal striping where the stripe count is 56).

3.2 Origin of the Code

Here we will describe the initial code. The code implementing the benchmark is the extension of work of the MSc course – Message Passing Programming (MPP). In the MPP coursework, an image recovery from the given edge image is implemented by repeating operations through the equation:

\[
(1) \quad new_{i,j} = \frac{1}{4} \left( old_{i-1,j} + old_{i+1,j} + old_{i,j-1} + old_{i,j+1} - edge_{i,j} \right)
\]

where \(edge\) is the edge image, and the \(old\) and \(new\) are the image value at the beginning and end of each iteration. The value of the \(old\) is taken as 255.0 in the first iteration. The iteration continues until the change between the results between two iterations is less than some amount.

The image recovery application takes PGM file as the input, which is a text file containing the image dimensions \(M\) and \(N\), followed by \(M \times N\) integer values representing the pixel values of \(edge_{i,j}\). The reconstructed image is stored in the same format as well.

Code is implemented as a parallel program, each process holds an equal and non-overlapping portion of the global buffer \(masterbuf\) of size \(M \times N\), and processes are
arranged as a 2D logical grid by MPI_Cart_create() to match the problem. The 2D decomposition can be done in strips (a \(1 \times P\) or \(P \times 1\) grid, where \(P\) is the process count) or in blocks (a \(X \times Y\) grid, where both \(X\) and \(Y\) are greater than 1) as referred in Section 2.5.1, for simplicity, however, the \(1 \times P\) grid is preferred if available and it also allows the internal buffer local to each process has a favourable memory layout for Fortran. The process grid is designed to be periodic in the horizontal direction and non-periodic in the vertical direction that is controlled by the period argument passed to MPI_Cart_create() is given as (\(/.true., .false./\)). For simplicity the MPP application can only process the image each of whose dimension is exactly divided by the process count in the corresponding dimension of the logical process grid. Messages are displayed and the program terminates if the condition is not met.

The MPP coursework used Master IO instead of MPI-IO. The input file is read to the master buffer masterbuf and split up by MPI_Ssend() on the master process (rank 0), and the other processes receive an equal and non-overlapping portion of the masterbuf by MPI_Recv() stored to the local array edge. Assuming that processes are arranged as a \(1 \times P\) logical grid, the dimensions of the image size processed on each process is reduced to \(M_p \times N_p\) where \(M_p = M / P\) and \(N_p = N / P\). However, depth-1 halos are applied to the local array new, old and edge to represent the boundary values from the neighbouring processes, the array size is \((M_p + 1) \times (N_p + 1)\) instead. To take edge as an example, the image data is stored in edge(1:Mp, 1:Np), while halos are located in edge(0, 0:Np+1), edge(Mp+1, 0:Np+1), edge(1:Mp, 0), and edge(1:Mp, Np+1). Halos are exchanged between processes in each iteration by non-blocking MPI_Issend() and blocking MPI_Recv(), that allows to overlap communication and calculation of Equation (1).

The application terminates the iteration when the reconstructed image is considered sufficiently accurate where the value of the \(\Delta\) - the maximum absolute change of any pixel in the image - is less than the criterion 0.1. The \(\Delta\) is computed locally on each process and summed up by the collective communication MPI_Allreduce(). Once the stopping criterion is reached, the masterbuf array on the master process is updated by MPI_Recv() that gathers the new array on the other processes sent out by MPI_Ssend(). Similar to read, the output file is generated by a single write on the master process. Halos in the local arrays are involved in the iterative image reconstruction, but are not written to the output file in the end. Derived datatypes MPI_Type_vector and MPI_Type_contiguous are used to describe the halos in halo swapping and to describe the memory layout of the image data local to each process in the transfer between the masterbuf and the edge before the iteration and between the new and the masterbuf after the iteration.

3.3 Code Design

The code of the MPP coursework was written entirely by the author and the user is already familiar with serial IO operating on a 2D dataset, it is then chosen as a basis for IO benchmarking.
Like the initial code, processes are arranged in a 2D logical grid by `MPI_Cart_create()`. The grid is also periodic in the horizontal direction and non-periodic in the vertical direction. The $IxP$ decomposition is also preferred if available.

The data operated in the initial code is a text file, but the benchmark uses a raw binary file for MPI-IO and a netCDF dataset for NetCDF, and as a result, input data is regenerated for both libraries in the project. The MPI point-to-point communications for data transfer between the `masterbuf` on the master process and the `edge` on the other processes and for halo swapping are removed, while halos are still reserved in the local array. The calculations of stopping criterion and the reduction of global sum are also removed. Rather than the serial IO performed in the initial code, the benchmark does collective parallel IO that changes the use of IO calls and removes the global `masterbuf` as well as the local `new` and `old` arrays, only `edge` is reserved.

The program is written using Fortran and compiled using Crayfn 2.4.2 on ARCHER. MPI-IO and NetCDF interfaces are used for collective and parallel IO access. As the local array on each process is designed to have depth-1 halos in each dimension, each process uses various MPI derived types to describe non-contiguous data layout in both the file and the local array to avoid halos being written to the file.

### 3.3.1 Input Data and Output Data

The testing data in the project is a 16384x20480 single-precision floating point array, around 1.3 GiB.

To benchmark different IO libraries, we prepare different file formats – a stream of bytes for MPI-IO, and netCDF format NetCDF library. The input file for MPI-IO is generated from a sequence of 16384x20480 single-precision floating point values written by the Fortran intrinsic `write()`, while the input file for NetCDF is produced by NetCDF IO routines referred in Section 2.2.2. Both input files are generated to contains continuous single-precision floating point values each of which represents the index it locates in the array for the later validation of the benchmark; this simplifies the verification as the file should contain the numbers 1.0, 2.0, etc. in order. The code of input generator is not provided in the project as it is not a part of the benchmark and as it is easy to do on a single process. The output file of the benchmark is exactly the same as the input file if the same IO library and the same approach for distribution are used in both the read and write stages.

Note that the dimensions of the global data array must be specified in the input file name, for example, `input16384x20480.dat`. When MPI-IO is chosen to do read, the input file is taken as raw data and the benchmark is designed to obtain the dimensional information in this way. The same mechanism is applied to NetCDF as well to simply the implementation, although dimensions are defined in the metadata of the netCDF dataset.
3.3.2 Compilation and Execution

The benchmark needs to be built with the system-supplied Crayftn compiler on the login nodes on ARCHER. The Makefile provided in the project is highly recommended to be used, because it compiles each source file and deals with the order of object files entering the linker. Building the code is simply typing `make` command in the directory that contain the Makefile. Note that the module `cray-netcdf-hdf5parallel` is required to be loaded in advance by typing the command:

```
> module load cray-netcdf-hdf5parallel
```

The benchmark is run on the compute nodes on ARCHER and it is highly recommended to use the supplied PBS and Bash scripts to submit jobs. Both executables need to be executed accompanied with arguments, and to ensure successful executions, arguments are all mandatory and need to be supplied in order as illustrated in Table 1. The project provides two pairs of scripts to specify the command arguments so the users can run with different use cases without re-compiling the code\(^1\). Refer to Appendix B for the details.

3.3.3 Design Overview

Figure 7 gives a high level flow illustrating how the benchmark performs IO, measures and outputs IO bandwidth. The program starts by initialising the MPI environment. Command line arguments are then read and parsed individually; the file names of both the input and output are known in this stage, and so are the IO libraries for read and write, process decomposition, and IO patterns. The process decomposition may be specified in the command arguments given as the process count both dimensions of a 2D logical grid, or be determined by the program as the corresponding arguments are specified as 0’s. Once the process decomposition is determined, the 2D Cartesian topology can be created by `MPI_Cart_create()`, and the size of the local array on each process can also be calculated and used to allocate a block of memory for the local array. Given the IO patterns, each process can create corresponding MPI derived datatypes to describe the layout of both the local and global memories. The input file then is opened for read, and each process defines the part of the global file that is visible for itself and then read the file data to the local array in a collective parallel manner. Similarly, in the write stage, a single file is created by all processes, and they define mapping of the data layout of the local memory to that in the global file before transferring data to the output file. The approach for process to describe which part of the global file is visible varies depending on which IO library is in use, the approach for NetCDF is described in Section 3.3.3.4.2, and that for MPI-IO are detailed in Section 5.1 and Section 6.1. The local array not only stores the data read form the input file, but also contains depth-1 halos on the edge; and

\(^1\) To make the benchmark run in non-collective IO mode, the collective IO routines are required to replaced with non-collective ones in the source code, so recompilation is required.
therefore correct array indexing for the local buffer array passed to the read/write routines is required to avoid data are read to the halos or halos are written to the output.

![High Level Flow Chart](image)

**Figure 7 High Level Flow Chart.**

### 3.3.3.1 Reading Command Arguments

The benchmark obtains the file names of the input and output from the command-line arguments passed to it. In addition, the arguments in collective also determine the type of measurement of IO bandwidth to be run. The arguments are passed as a serious of character strings separated by whitespaces, and the program reads each of them by the Fortran intrinsic routine `get_command_argument()` before proper data type conversion and storing to a corresponding variable. Ten arguments are expected and they must be passed in order. The arguments and corresponding Fortran variables in the program as shown in Table 1. Note that the labels listed in the *Argument* column come from the variables in the PBS scripts `benchmark.pbs`, more details are explained in Appendix B.
<table>
<thead>
<tr>
<th>Order</th>
<th>Argument</th>
<th>Fortran Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SinLib</td>
<td>iInIOLib</td>
<td>IO library for read.</td>
</tr>
<tr>
<td>2</td>
<td>SoutLib</td>
<td>iOutIOLib</td>
<td>IO library for write.</td>
</tr>
<tr>
<td>3</td>
<td>$filename</td>
<td>filename</td>
<td>Input file name.</td>
</tr>
<tr>
<td>4</td>
<td>$striping</td>
<td>striping</td>
<td>Stripping setting of both input and output files.</td>
</tr>
<tr>
<td>5</td>
<td>$pSizeM</td>
<td>aiDims(1)</td>
<td>Process count in the first dimension of the logical process grid.</td>
</tr>
<tr>
<td>6</td>
<td>$pSizeN</td>
<td>aiDims(2)</td>
<td>Process count in the second dimension of the logical process grid.</td>
</tr>
<tr>
<td>7</td>
<td>$inDistrib</td>
<td>iInDistribType</td>
<td>Data distribution for read.</td>
</tr>
<tr>
<td>8</td>
<td>$outDistrib</td>
<td>iOutDistribType</td>
<td>Data distribution for write.</td>
</tr>
<tr>
<td>9</td>
<td>$bSizeM</td>
<td>aiBlockSizes(1)</td>
<td>Block size of cyclic distribution in the first dimension.</td>
</tr>
<tr>
<td>10</td>
<td>$bSizeN</td>
<td>aiBlockSizes(2)</td>
<td>Block size of cyclic distribution in the second dimension.</td>
</tr>
</tbody>
</table>

Table 1 Command Arguments and Corresponding Fortran Variables.

3.3.3.2 Determining Decomposition

The dimensions of the 2D process logical grid are chosen either by the values of the command-line arguments $pSizeM and $pSizeN that are passed as positive integer values or by the benchmark itself if the both arguments are passed as zero. In the latter case, the program iterates loops over the number of processes $P to find out the first combination of $i and $j that satisfies $P = i \times j, where $i is process count in the horizontal direction and $j in the vertical direction, an example for 4 processes in available in Figure 4. The $i and $j must exactly divide the width $M and the height $N of the global array respectively. The $i and $j here represent the array variables, $aiDims(1) and $aiDims(2) shown in Table 1. The Fortran array $aiDims specifies the $dims arguments passed to the $MPI_Cart_create() to create a communicator of the 2D logical process grid. The grid is periodic in the first dimension but is not in the second dimension, the same as being arranged in the initial code referred in Section 3.2; however, this boundary condition will not affect the IO bandwidth.

3.3.3.3 Data Distribution

The project adopts two distributions, block and block-cyclic, as mentioned in Section 2.5. Each distribution has varieties implemented by the MPI derived datatypes Subarray, Darray and Indexed when doing IO with MPI-IO; the approaches to implement the block distribution are defined as Subarray, Darray-block, and Indexed-block and are available in Section 5.1, and the approaches that implement the block-cyclic distribution are defined as Darray-cyclic and Indexed-cyclic that can be found in Section 6.1. However, when NetCDF is used to do IO, implementing the distributions is different and achieved by its read/write routine that is explained in Section 3.3.3.4.2.
The choice of data distribution in the benchmark is determined by the command-line arguments $\text{inDistrib}$ and $\text{outDistrib}$; the former decides how the global file is divided and distributed among the processes, and the latter decides how the local memory on each process is gathered to output file. Values for the arguments are integers 1 to 5 individually representing Subarray, Darray-cyclic, Darray-block, Indexed-cyclic, and Indexed-block.

### 3.3.3.4 Choice of IO Libraries

The command-line arguments $\text{inLib}$ and $\text{outLib}$ shown in Table 1 individually determines which IO library is used to perform read and write operations. Values for the arguments are 1 for MPI-IO and 2 for NetCDF.

#### 3.3.3.4.1 Use of MPI-IO

These routines, `mpiread()` and `mpiwrite()`, are designed to implement collective parallel IO using MPI-IO library. File is opened by `MPI_File_open()` where the `mode` argument is passed as `MPI_MODE_RDONLY` for read and `MPI_MODE_WRONLY+MPI_MODE_CREATE` for write. `MPI_INFO_NULL` is used in the `info` argument for both operations. The Cartesian communicator is passed in the `comm` argument.

The processes’ view of data in the file is set by `MPI_File_set_view()` in which the `etype` argument is `MPI_REAL` as the input data is an array of single-precision floating point values, and the `filetype` argument is determined in advance (Section 5.1 and Section 6.1 in detail) according to the command-line arguments $\text{inDistrib}$ and $\text{outDistrib}$. “native” is chosen in the `datarep` argument to perform reads/writes of raw binary data.

File is read by `MPI_File_read_all()` and written by `MPI_File_write_all()` to ensure processes read/write collectively. The `datatype` argument defining the memory locations available on the calling process is also determined in advance and explained in Section 5.1 and Section 6.1. The local buffer specified in the `buf` argument is sized $(Mp+1)\times(Np+1)$ including depth-1 halos as described in Section 3.2, where the IO data is arranged in $(1:Mp, 1:Np)$, so `buf(1,1)` is passed to exclude halos from being read from the input file or written to the output file.

#### 3.3.3.4.2 Use of NetCDF

The routines, `netcdfread()` and `netcdfwrite()`, implements the parallel collection IO using NetCDF library.

To write a netCDF dataset, the file is opened by `nf90_create_par()` to ensure parallel write to be enabled and performed by processes in the Cartesian communicator specified in the `comm` argument. The `cmd` argument is given as
ior(nf90_netcdf4,nf90_mpiio) to create a netCDF-4 file in a parallel manner. MPI_INFO_NULL is used in the info argument.

In the define mode, the dimensions of the global netCDF dataset is the dimid argument passed to nf90_def_dim(), where the label set as “x” and “y” to the name argument. The counterpart of the etype in MPI-IO MPI_File_set_view() is defined in the variable of metadata by the xtype argument in nf90_def_var() with value NF90_FLOAT.

nf90_def_var_fill() is called right after to avoid writing fill values to the variable that could double the write time. To exit the define mode, nf90_enddef() is called, and nf90_var_par_access() is called with the access argument set to nf90_collective to ensure the parallel writes to be collective.

Data variables is written by nf90_put_var(), and the local buffer given to the values argument is also restricted to range (1:Mp, 1:Np) to avoid writing halos as is done in MPI-IO. Unlike MPI-IO where the filetype and datatype are created by the use of MPI derived datatypes, the mapping of the memory layout between the local buffer and the global dataset is defined in the start, count, stride arguments passed.

A very important point to note that the approach (start, count, stride) provided by the NetCDF limits the implementations of memory mapping between the local and global data for the NetCDF in the project. The possible distributions are the block distribution analogous to the Subarray implementation and cyclic distribution with block size set to 1x1 analogous to the Indexed-cyclic and/or Darray-cyclic implementation used for MPI-IO. It is not possible to have block sizes larger than 1x1 for the block-cyclic in NetCDF.

Assuming that the global data of MxN is operated on P processes, and they are arranged in a 2D grid of dimension dims(1)x dims(2). The count is (Mp, Np) for each process. Both the start and stride arguments for block and cyclic distributions are:

```
For block,
start(:) = (M_p x coords(1) + 1, N_p x coords(2) + 1 / ),
stride(:) = 1;
```

For cyclic,
```
start(:) = coords(:) + 1,
stride(:) = dims(:),
```

**Table 2** The start and stride arguments for block and block-cyclic distributions implemented in NetCDF.
The *coords* is the coordinate of the process in the 2D logical process grid. *Mp* and *Np* are the dimensions of the local memory excluding depth-1 halos.

To read a netCDF dataset, file is opened by `nf90_open_par()` and the dimensions and variable of the dataset are got by `nf90_inq_dimid()` and `nf90_inq_varid()` respectively. The arguments passed to these functions are the same as that passed to the write counterparts. Collective reads are also guaranteed by `nf90_collective` passed in the `nf90_var_par_access()` call. As well as the `start`, `count`, and `stride` arguments in `nf90_put_var()`, the data is read by `nf90_get_var()` and distributed to the local buffer in the range *(1:*Mp, 1:*Np)* specified in the `value` argument to let halos unaffected by reads.

### 3.4 Testing

To validate the correctness of our code working on the block and cyclic distributions implemented by the approaches mentioned in Section 5.1 and Section 6.1, a raw binary dataset is used to the input file, in which each data is set to the value that represents its index in the global array. The source can be built as a test program simply by turning on the `-DTEST` option in the Makefile; the option is not enabled by default. The test program can read and write in the same ways as the benchmark does, except that each process validates the content of its local data buffer by comparing with the index values calculated in the `validateBlockRead()` and `validateCyclicRead()` routines. Error messages will be outputted to the log file when any incorrect content is detected.

The Darray-cyclic and Indexed-cyclic approaches can also be proven correct in a graphical test. Figure 8 is a 2-D image of size 480 columns and 216 rows, in which each 2-digit number takes 60 columns and 36 rows. The image file is obtained from the practice of the MSc course Advance Parallel Programming.

In the test case, we run the benchmark over 12 processes, and arrange them as a 4 x 3 logical grid in column-major order, thus each process is assigned a non-overlapping image of size 120 columns and 216 rows. Describing the location of the local array of each process with Darray-cyclic or Indexed-cyclic, with a distribution argument of 60 in column and 36 in row, each process is assigned four 2-digit numbers in a round-robin fashion. We take the advantage of the content of the image where the digits have been chosen to enable us to identify where different parts of the array come from, for example, 65 is at position (6, 5). The local buffer on each process storing the data is then written to a common output, this time, however, in a block-mapping fashion implemented by the Subarray approach, as shown in Figure 9 which illustrates that the distributed blocks of the test image are assigned to the local array on process 0 and written in a whole to a common output using block distribution. In this way, the output image can be a graphic proof showing the correctness of the Darray-cyclic and Indexed-cyclic arrangements made in the benchmark code.
3.5 Choosing Data for Graphing

The experiment results presented in the report are obtained by selecting the maximum values across 20 runs. We explain the reason by Figure 10 that shows the write bandwidth produced by parallel collective MPI-IO, with each data point selected by the maximum,
average and median of multiple runs. The example measures the read bandwidth produced by different process counts, the simplest decomposition is chosen and the Subarray approach of distribution is used to describe the memory layout of the local data and in the global file that is striped across 4 OSTs.

It is clear that for all representations, the write rate basically remains the same at around 800 MiB/s as that achieved by the serial IO. The benefit brought by the file striping takes effect when the process count grows to 64 that is equivalent to 3 compute nodes and therefore the write operations are parallelised across 3 OSTs. However, the three representations give different results after 64 processes. Both the average and median present a slightly increasing trend in the performance with process count increased up to 1024, with obvious fluctuations during the range. Significantly different from the average and median presentations, the figure seen in the maximum shows what would be expected: the write rate starts to saturate at 128 processes where 6 compute nodes run the job, in other words, we saturate all 4 OSTs.

We understand that while on ARCHER compute nodes do not impact each other, file system is shared between users, and we suspect this is the factor of why considerable variations are shown between each single run of the same process count. In addition, we focus on not only the throughputs of specific points, but also the effects and the change in performance discovered by various process counts, process decompositions, data layout of the local and global arrays, IO libraries, file striping settings on Lustre file system, etc. Although the ARCHER users are probably interested in average IO speeds, that is affected by the load on the system. We are interested in trying to understand what each IO library is doing internally and how this interacts with the Lustre file system, to do this we need to try and eliminate the effects of system load so fastest rate or shortest time of multiple runs is chosen to represents graphs in the report.
3.6 Test Cases

The benchmark provides a range of use cases to measure IO bandwidth by selecting process count, process decompositions, distributions, IO libraries and by specifying the file name on which the stripe count on the Lustre file system is configured. Distribution approaches and IO libraries can be chosen individually for both read and write, while decomposition and striping setting are the same for both of them.

In the project, the experiments carried out are arranged into the following cases. The IO bandwidth obtained by various process counts (Chapter 4), the effect of various decompositions over 1024 processes with data distributed in a block manner (Chapter 5). In Chapter 6, we concentrate the effect of the block-cyclic distribution in two aspects, the first experiment focuses on how various block sizes applied can affect the IO performance, where the process decomposition is fixed to a 32x32 grid; the second experiment fixes the block size to 16x16, while various process decompositions are adopted. The process decompositions for 1024 processes selected in the project is listed in Table 3. MPI-IO library are used in the previously-mentioned cases, while the comparison of MPI-IO and NetCDF is discussed in Chapter 7, in terms of various process counts and various decompositions over 1024 processes. Finally, effects of file striping on Lustre is discussed in Chapter 8. To reproduce the experiments carried out in the project, please refer to Appendix B.
Chapter 4

Scalability of Process Count

We now investigate how the IO rates change as the number of processes is increased. In this experiment, the simplest decomposition ($1xP$ for $P$ processes) with block distribution is chosen for each process count and the data layout of the local and global arrays is described using the Subarray approach (explained in Section 5.1.1). MPI-IO is used for both collective reads and writes and file striping is set to default (4 stripes).

4.1 Effects of Numbers of Processes

Figure 11 gives a comparison of IO bandwidths for different numbers of processes. The bandwidths, in general, increase with increasing numbers of processes involved in MPI-IO collective operation. The performance in both read and write can be improved by around three times by using more than 128 and 256 processes, respectively, compared to 1 process. The write can reach to around 2 GiB/s on 1024 processes, the same as the result of benchio shown in Figure 12 (see the data for Defstriped parallel). It is interesting that write is faster than read. When reading, IO clients actually access disks and wait for response; however, when writing, data may be buffered and IO clients do not need to access disks all the time. The good performance achieved on more than a hundred processes can be explained by the file striping. File is striped over 4 OSTs, which means the benchmark can take full advantage of OSTs on 4 compute nodes (the same number of IO clients on Lustre), so both bandwidths would saturate at more than 96 processes. The observed results show that for write, it saturates somewhere between 64 and 128 processes, but for read, it saturates between 128 and 256 processes.
Figure 11 MPI-IO IO performance of different numbers of processes, using Subarray approach of distribution; defstriped.

Figure 12 Write bandwidth for local volume n3 = 1283. Extracted from the benchio paper [1].
Chapter 5

Effects of Process Decompositions with Block Distribution

We also benchmark IO rates of different process decompositions over 1024 processes arranged as a $X \times Y$ logical grid where $X$ and $Y$ must exactly divide 1024 and result in an equal size of local array for each process. The process decompositions are listed in Table 3.

<table>
<thead>
<tr>
<th>Process decomposition ($X \times Y$)</th>
<th>Size of the local arrays (without halos)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 x 1024</td>
<td>16384 x 20</td>
</tr>
<tr>
<td>2 x 512</td>
<td>8192 x 40</td>
</tr>
<tr>
<td>4 x 256</td>
<td>4096 x 80</td>
</tr>
<tr>
<td>8 x 128</td>
<td>2048 x 160</td>
</tr>
<tr>
<td>16 x 64</td>
<td>1024 x 320</td>
</tr>
<tr>
<td>32 x 32</td>
<td>512 x 640</td>
</tr>
<tr>
<td>64 x 16</td>
<td>256 x 1280</td>
</tr>
<tr>
<td>128 x 8</td>
<td>128 x 2560</td>
</tr>
<tr>
<td>256 x 4</td>
<td>64 x 5120</td>
</tr>
<tr>
<td>512 x 2</td>
<td>32 x 10240</td>
</tr>
<tr>
<td>1024 x 1</td>
<td>16 x 20480</td>
</tr>
</tbody>
</table>

Table 3 Process decompositions over 1024 processes and the sizes of the local arrays.

Block distribution is chosen to describe the memory layout of the local and the global file that is striped across 4 OSTs (defstriped). Three different approaches are used to implement the block distribution to investigate how much is the MPI-IO IO performance dependent on how the distribution is described. Before looking into the experiment results, we explain the approaches implementing the block distribution in MPI-IO in Section 5.1.

5.1 Approaches of Block Distribution

The approaches to implement the block distribution in the project are named as Subarray, Darray-block and Indexed-block according to the MPI derived datatypes used. We use Figure 13 as the example to explain how these approaches are implemented. Figure 13 shows that processes are arranged as a 3x2 logical grid, and the global data of $M \times N$ single precision floating point values is divided into equal pieces of $M_p \times N_p$ and distributed across processes. In this example, $M_p = M/3$ and $N_p = N/2$. To be short, we use “file
view” to denote which parts in the global data array are visible on the calling process, and use “buffer view” to describe which parts in the local data array represent the IO data excluding halos.

Figure 13 Describing the memory layout of the local array local on process 0 and the global dataset by block distribution.

5.1.1 Subarray Approach

The Subarray approach is designed in the createSubarrayType() subroutine in the benchmark. The Subarray approach is designed to describe block distributions. The file view and buffer view is generated by the MPI derived Subarray datatype created by the MPI routine:

\[
\text{MPI Type create_subarray}(\text{ndims, array of sizes, array of subsizes, array of starts, order, oldtype, newtype, ierror})
\]

The ndims argument represents process grid dimensions and is passed as 2 for a 2D process grid; it also determines the array size of the following three arguments. The array_of_sizes defines the dimensions of the entire data array, while the array_of_subsizes defines the dimensions of the portion of the data we are interested; the unit for both is the same as the oldtype argument that is set to
The location of the portion of interest is defined in the `array_of_starts` argument. To take Figure 13 as an example, these three arguments to describe the file view and the buffer view are set as below:

For file view,
\[
array_{of\_sizes}(::) = (M, N) \\
array_{of\_subsizes}(::) = (M_p, N_p) \\
array_{of\_starts}(::) = (\text{coords}(1) \times M_p, \text{coords}(2) \times N_p)
\]

For buffer view,
\[
array_{of\_sizes}(::) = (M_p + 2, N_p + 2) \\
array_{of\_subsizes}(::) = (M_p, N_p) \\
array_{of\_starts}(::) = 0
\]

Table 4 The buffer view and file view implemented by the Subarray approach.

Note that the `array_off_starts` is set to (0, 0) rather than (1, 1), because the starting coordinate in each dimension is set in the `buf(1,1)` in the `MPI_File_read_all()` and `MPI_File_write_all()` call as mentioned in Section 3.3.3.4.1.

The `order` denotes the array storage order and is given as `MPI_ORDER_FORTRAN`, and the `newtype` is the created Subarray datatype to represent the file view or buffer view.

### 5.1.2 Darray-block Approach

Another implementation for the same block distribution is introduced here and called Darray-block approach.

The `createDarrayType()` subroutine in the benchmark generates the file view by the MPI derived Darray datatype. This routine is designed for general block-cyclic distributions, but can of course also be used for simple block distributions. The effect of using more complicated distributions is investigated later in Section 6.1.1.

The file view is created by the MPI routine:

\[
\text{MPI\_Type\_create\_darray}(\text{size}, \text{rank}, \text{ndims}, \\
array_{of\_gsizes}, \array_{of\_distrib}, \array_{of\_dargs}, \\
array_{of\_psizes}, \text{order}, \text{oldtype}, \text{newtype}, \text{ierror})
\]
The **size** and **rank** argument represent the process count and process rank in the communicator, respectively. The **ndims** is also the process grid dimensions and determines the array size of the following 4 arguments. The **array_of_gsizes** denotes the number of elements of the **oldtype** (**MPI_REAL**) of the global array. The **array_of_distrib**s is the distribution type, the **array_of_dargs** is the distribution argument, the **array_of_psizes** is the process count in each dimension in the process grid. The **array_of_xxx** arguments to describe the file view shown in Figure 13 are set as below:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>array_of_gsizes:</strong></td>
<td>((M, N))</td>
</tr>
<tr>
<td><strong>array_of_distrib:</strong></td>
<td>((\text{MPI_DISTRIBUTE_BLOCK, MPI_DISTRIBUTE_BLOCK}))</td>
</tr>
<tr>
<td><strong>array_of_dargs:</strong></td>
<td>((\text{MPI_DISTRIBUTE_DFLT_DARG, MPI_DISTRIBUTE_DFLT_DARG}))</td>
</tr>
<tr>
<td><strong>array_of_psizes:</strong></td>
<td>(\text{dims}())</td>
</tr>
</tbody>
</table>

**Table 5 The file view implemented by the Darray-block approach.**

The **MPI\_DISTRIBUTE\_BLOCK** denotes a block distribution and **MPI\_DISTRIBUTE\_DFLT\_DARG** specifies a default distribution. **dims** is obtained from the dimensions of the Cartesian communicator using **MPI\_Cart\_get()**.

The **newtype** argument is the created Darray derived datatype to represent the file view.

Unlike the file view, the buffer view is generated by the MPI derived Subarray datatype in the same way as the file view is generated in the Subarray approach.

### 5.1.3 Indexed-block Approach

The last approach in the benchmark to implement the block distribution is called Indexed-block.

There is an MPI routine called **MPI\_Type\_create\_indexed\_block()** which is specific for block distributions, but we use **MPI\_Type\_indexed()** because it is more general datatype for distributions, that could describe both block and irregular distributions.

While the buffer view is produced in the same way as done in the Subarray approach, the **createIndexedType()** subroutine in the benchmark generates the file view by the MPI derived Indexed datatype that is a very general datatype and can be used to describe completely irregular distributions. The file view is created by the MPI routine:
MPI_Type_indexed(count, array_of_blocklengths, array_of_displacements, oldtype, newtype, ierror)

The MPI_Type_indexed() divides the global array into chunks of data of contiguous memory. In this approach, each chunk is of size \( Mp \times l \). The count argument represents the number of chunks and defines the size of the 1D arrays array_of_blocklengths and array_of_displacements. The array_of_blocklengths specifies the number of elements of oldtype (MPI_REAL) per chunk, and the array_of_displacements specifies the displacement of each chunk. The created Indexed datatype is returned in the newtype.

The arguments describing the data layout are set as below:

For file view,

\[
\text{count} = \frac{M \times N}{M_p} \times \frac{1}{l}
\]

\[
\text{array_of_blocklengths}(i) = M_p
\]

\[
\text{array_of_displacements}(i) = (i - 1) \times M + 1
\]

<table>
<thead>
<tr>
<th>Table 6</th>
<th>The file view implemented by the Indexed-block approach.</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>( \frac{M \times N}{M_p} \times \frac{1}{l} )</td>
</tr>
<tr>
<td>array_of_blocklengths</td>
<td>( M_p )</td>
</tr>
<tr>
<td>array_of_displacements</td>
<td>( (i - 1) \times M + 1 )</td>
</tr>
</tbody>
</table>

The \( i \) is from \( l \) to \( N_p \), and each displacement is added by \( l \) because the global data array starts at \( l \) in Fortran. Note that the blocks here are always the same size.

5.2 Experiment Results

The experiment in the section is conducted to compare the MPI-IO performance affected by the block distribution implemented by the Subarray, Darray-block and Indexed-block approaches. As can be seen in both Figure 14 and Figure 15, in general, both MPI-IO reads and writes are faster at the first few decompositions and slow down at the last few decompositions. This relates to how the data are arranged in the local buffer and how they are placed in the file. When the processes are arranged as a \( X \times Y \) grid where \( X \) is less than \( Y \), the global data are split into horizontal stripes; this allows the data points with consecutive indexes in the local array have bigger chance to be located at consecutive addresses in the file, and allows each process to access as consecutive pieces of data in the global file as possible so a smaller number of larger contiguous reads or writes are ensured (shown in Figure 5). By contrast, the last few cases divide the global data into vertical stripes that are less conformable to the memory indexing in Fortran and a larger number of non-contiguous IO accesses are required.

The performances from various approaches to the block distribution are also compared. In general, the MPI-IO read of the Subarray approach is faster than that of the other two block approaches which also fluctuate in the read bandwidth between decompositions. The reason of the fluctuation is unclear, but different results of approaches can be
explained according to the implementation of each. Although the same IO pattern is described by the three approaches, Darray-block and Indexed-block are much more general and require more lines of code to implement, and MPI-IO may also require more complicated indexing during the data transfer between the global file and the local buffer on each process, and so Darray-block and Indexed-block result in poorer performance in read. By contrast to reads, MPI-IO produces almost the same write bandwidth for all block approaches, as an indication that MPI-IO write can cope well with the complexity of the implementations of these distributions.

Figure 14 Read performance of Subarray, Darray-block, Indexed-block approaches of block distribution, over 1024 processes with different process decompositions; defstriped.
5.3 Summary

Subarray is simple to do, but it is unable to describe complicated distributions, for example, block-cyclic distribution. This can be achieved by the Darray-cyclic and Indexed-cyclic approaches that will be introduced in Chapter 6, and they are also suited to describe simpler cases.

The process decomposition plays an important role in IO performance when the global array is distributed across processes in a block manner. It determines to what extent the processes can access the data of contiguous memory locations in the global file. Processes arranged in a way close to a vertical stripe, the data array local to each process each process has a more favourable memory layout in Fortran, and each process can access more data of consecutive addresses in the global file to achieve better performances.

Although the same data layout is generated by different approaches of block distribution over a fixed process decomposition, the complexity of the implementation affects the IO performance. For write, the Subarray approach is faster than the Darray-block and Indexed-block approaches, but for read, they are almost identical.
Chapter 6

Effects of Process Decompositions with Block-cyclic Distribution

The effect of block distribution on MPI-IO is discussed in Chapter 5, and in this chapter we extend the work to block-cyclic distribution which is implemented by two approaches in the project. They are named as Darray-cyclic and Indexed-cyclic according to which MPI derived datatype is used to describe which parts of the global data are visible on the calling process (the file view as mentioned in Section 5.1). In fact, block distribution is a special case of block-cyclic – block uses the local size $MpxNp$, but block-cyclic by default uses $16x16$ in the project.

In this chapter, we firstly explain how the Darray-cyclic and Index-cyclic approaches are programmed, followed by investigating how much is the IO performance dependent on how the processes are decomposed. This is discussed in two aspects: fixing the block size while the decomposition is variable, and by contrast, using various block sizes to describe the $32x32$ decomposition. Block sizes used in the $32x32$ decomposition are $512x640$, $256x320$, $128x160$, $64x80$, $32x40$, $16x20$, $8x10$ and $4x5$. The Darray-block/Indexed-block is the special case of Darray-cyclic/Indexed-cyclic with block size $512x640$ - the dimensions of the data array local to each process.

6.1 Approaches of Block-cyclic Distribution

The block-cyclic distribution splits up the global array to processes in a round-robin fashion. Which parts of the global data are distributed into a process is dependent on not only the process decomposition but also the block size, as shown in Figure 6. With the same decomposition and the same block size for distribution, the Darray-cyclic and the Indexed-cyclic produce the same pattern for MPI-IO.

6.1.1 Darray-cyclic Approach

Like the Darray-block approach, the logic of the Darray-cyclic is also implemented in the createDarrayType() subroutine. The file view and buffer view is individually produced by the MPI derived Darray and Subarray datatype by the corresponding MPI routine, but the arguments are passed with different values.

For file view, the arguments passed to the MPI_Type_create_darray() are given as:
For file view,

\[
\text{array\_of\_g\_sizes}() = (M, N) \\
\text{array\_of\_distrib}() = (\text{MPI\_DISTRIBUTE\_CYCLIC,_MPI\_DISTRIBUTE\_CYCLIC}) \\
\text{array\_of\_dargs}() = \text{aiFileDargs}() \\
\text{array\_of\_psizes}() = \text{dims}()
\]

**Table 7 The file view implemented by the Darray-cyclic approach.**

The \text{MPI\_DISTRIBUTE\_CYCLIC} denotes a (block-)cyclic distribution. \textit{dims} is obtained from the \text{MPI\_Cart\_get()} . \text{aiFileDargs} is the block size that is equivalent to the HPF layout \text{ARRAY(CYCLIC(n))} [10][12], where \( n \) is the block size in each dimension. For the block size given as 16x16, \text{aiFileDargs} is determined as below:

\[
\text{aiFileDargs}(1) = \begin{cases} 
16, & \text{if } M < 16 \\
M, & \text{otherwise}
\end{cases}
\]

\[
\text{aiFileDargs}(2) = \begin{cases} 
16, & \text{if } N < 16 \\
N, & \text{otherwise}
\end{cases}
\]

**Table 8 How the block size is determined in the Darray-cyclic approach.**

For buffer view, the arguments passed to the \text{MPI\_Type\_create\_subarray()} are given as:

\[
\text{array\_of\_sizes}() = (M + 2, N + 2) \\
\text{array\_of\_subsizes}() = \text{aiBuffSubSizes}() + 2 \\
\text{array\_of\_starts}() = 0
\]

**Table 9 The buffer view implemented by the Darray-cyclic approach.**

Note that the \text{array\_of\_starts} is set to (0, 0) rather than (1, 1), because the starting coordinate in each dimension is set in the \text{buf(1,1)} in the \text{MPI\_File\_read\_all()} and \text{MPI\_File\_write\_all()} call described in Section 3.3.3.4.1. \text{aiBuffSubSizes} is a two-element array determined as below:
\[
aiBuffSubSizes(t) = \text{count}(t) \times \text{aiFileDargs}(t) + q(t) \times \text{aiFileDargs}(t) + rr(t)
\]

Each of \text{count}, q, rr is a two-element array determined in a way that suffices:

\[
gsizes(t) = \text{count}(t) \times \text{aiFileDargs}(t) \times \text{dims}(t) + p \times \text{aiFileDargs}(t) + \text{mod}(gsizes(t), \text{aiFileDargs}(t))
\]

\[
gsizes(t) = (M/N,)
\]

\[
q = \begin{cases} 
1, & \text{if } (\text{coords} < p) \\
0, & \text{otherwise}
\end{cases}
\]

\[
rr = \begin{cases} 
\text{mod}(gsizes, \text{aiFileDargs}), & \text{if } (\text{coords} == p - 1) \\
0, & \text{otherwise}
\end{cases}
\]

Table 10 How aiBuffSubSizes is determined in the Darray-cyclic approach.

The code is complicated because it considers the general case that each process may have different local volume.

The \text{dims} denotes the dimensions of the process grid, and the \text{coords} is the coordinate of the process in the process grid.

For example, a global volume is 480x216, the process decomposition is 3x3, the block size is 16x16, then the variables are:

\[
M=480, N=216
\]

\[
dims=(/3, 3/)
\]

\[
\text{aiFileDargs}=(/16, 16/)
\]

\[
Mp=M/dims(1)=160, Np=N/dims(2)=72
\]

\[
gsizes(1)=480=10x16x3+0x16+0; \text{ therefore, count}(1)=10, p(1)=0
\]

\[
q(1)=0, rr(1)=0 \text{ for all process ranks}
\]

\[
gsizes(2)=216=4x16x3+1x16+8; \text{ therefore, count}(2)=4, p(2)=1
\]

\[
q(2)=1 \text{ for processes which suffices coords}(2)<1, \text{ otherwise } q(2)=0
\]

\[
rr(2)=8 \text{ for processes which suffice coords}(2)=1, \text{ otherwise } rr(2)=0
\]

Table 11 An example of the Darray-cyclic approach.
6.1.2 Indexed-cyclic Approach

Like the Indexed-block approach, the logic of the Indexed-cyclic is also implemented in the `createIndexedType()` subroutine. The file view and buffer view is individually produced by the MPI Indexed and Subarray derived datatype by the corresponding MPI routine, but the arguments are passed with different values.

Indexed-cyclic approach generates the buffer view in the same way as the Darray-cyclic approach does, but for file view, the arguments passed to the `MPI_Type_indexed()` are given as below:

```
count = aiNumBlocks(1)×aiNumBlocks(2)
array_of_blocklengths(1: count) = aiBlocklengths(1: count)
array_of_displacements(1: count) = aiDisplacements(1: count)
```

Table 12 The file view implemented by the Indexed-cyclic approach.

The `aiNumBlocks(1)` is the `count(1)`, and the `aiNumBlocks(2)` is the `aiBuffSubSizes(2)` described in Table 10 in Section 6.1.1.

The `aiBlocklengths` and `aiDisplacements` are determined as below:

```
aiBlocklengths(:,1) = aiFileDargs(1)
x=1
do jj=coords(2)×aiFileDargs(2),iN-1,aiFileDargs(2)×dims(2)
   upperBound = jj+aiFileDargs(2)-1
   if (upperBound >= N) then
      upperBound = N-1
   end if
   do j=jj, upperBound
      do i=coords(1)×aiFileDargs(1),iM-1,aiFileDargs(1)×dims(1)
         aiDisplacements(x) = j×M+i
         if ((i+aiFileDargs(1)) > M) then
            aiBlocklengths(x) = M-i
         end if
         x = x + 1
      end do ! i
   end do ! j
end do ! jj
```

Table 13 How aiBlocklengths and aiDisplacements are determined in the Indexed-cyclic approach.
Each of $aiFileDargs$, $coords$, $dims$, $M$, $N$ is of the same meaning as described in the Darray-cyclic approach in Section 6.1.1, and also has the same value.

6.2 Experiment Results

The code for creating Darray-cyclic and Indexed-cyclic is very complicated, but it is verified correct in Section 3.4.

The experiments are carried out on 1024 processes, MPI-IO is used with the default striping on Lustre. The first experiment in Section 6.2.1 discusses the effect of block sizes used in block-cyclic distribution. The second experiment in Section 6.2.2 is aimed to answer how much the IO performance differs with various approaches that implement block-cyclic distribution.

6.2.1 Various Block Sizes for 32x32 Decomposition

Figure 16 illustrates the parallel MPI-IO IO performances where the 1024 processes are arranged as a 32x32 grid. This experiment discusses how the block sizes applied to Darray-cyclic and Indexed-cyclic approaches affect the IO performances. Stripe count is set to 4. The result of the Subarray approach is added to compare with that of the approaches of block-cyclic distribution with block size $512x640$ that is exactly the same as the local volume.

We can see in Figure 16 that MPI-IO produces very close read and write performances for Darray-cyclic and Indexed-cyclic approaches. We can also observe that the read and write bandwidths of the Darray-cyclic and Indexed-cyclic peak at around 2.5 GiB/s on the $512x640$, slightly less than that of the Subarray approach (2.55 GiB/s for read and 2.76 GiB/s for write). The bandwidths degrade with decreasing block sizes, where MPI-IO would be expected to rearrange the non-contiguous data to ensure that it performs a small number of large contiguous reads/writes. The degradation in performance could be because of the extra time needed to do all the data rearrangement.
Figure 16 Comparison of IO performance over 1024 processes arranged in a 32x32 process grid, with Darray-cyclic and Indexed-cyclic approaches using various block sizes; defstriped. For Subarray approach (block size 512x640; not shown in the figure), the read rate sits at 2.55 GiB/s and the write rate sits at 2.76 GiB/s.

6.2.2 Fixed Block Size for Various Decompositions

Figure 17 and Figure 18 show the parallel MPI-IO read and write performances of various decompositions on 1024 processes, with the default striping for Lustre. Data is distributed in a block-cyclic manner described by Darray-cyclic and Indexed-cyclic approaches where the block size is fixed to 16x16, and the results are compared with IO performance produced with data distributed in a block manner described by Subarray approach.

The Subarray approach gives much better IO performance than the Darray-cyclic and Indexed-cyclic approaches do. Note that how the data is distributed in a block manner differs from that in a block-cyclic manner, according to Section 2.5. Subarray approach is simple while Darray-cyclic and Indexed-cyclic approaches are complicated to achieve so Subarray approach outperforms the other two block-cyclic approaches.

It is also interesting that although Indexed-cyclic approach requires more lines of code to describe both file view and buffer view than Darray-cyclic approach does, the figure obtained shows MPI-IO can cope with the complexity for Darray-cyclic as well as for Indexed-cyclic and thus produces very similar performance with both block-cyclic approaches.
Figure 17 Read performance of different process decompositions over 1024 processes, using Subarray, Darray-cyclic, and Indexed-cyclic approaches; defstriped. Subarray is block distribution; the others are block-cyclic distribution with block size 16x16.

Figure 18 Write performance of different process decompositions over 1024 processes, using Subarray, Darray-cyclic, and Indexed-cyclic approaches; defstriped. Subarray is block distribution; the others are block-cyclic distribution with block size 16x16.
6.3 Summary

The Darray-cyclic and Indexed-cyclic approach produce almost the same MPI-IO IO performance. For a fixed decomposition, smaller block sizes require larger volumes to for storing distribution information, the more overhead could be resulted from MPI-IO rearranging the data to ensure a small number of large operations, and as a result, the performance degrades.

The Subarray approach is faster than Darray-cyclic and Indexed-cyclic for several reasons: it is simpler to implementation, its block size is the local volume, larger than the $16\times16$ used in the approaches of the cyclic distribution that requires more memory for implementation and MPI-IO could need more data rearrangement to perform a small number of large contiguous reads/writes.
Chapter 7

Effects of IO Libraries

The MPI-IO, as well as the NetCDF, provides abstraction that enables parallel applications to achieve good performance and hides the detailed interactions between IO clients and IO servers as from the applications. Performance comes from the rearrangement of data that ensure a small number of large contiguous reads/writes.

In the project, however, MPI-IO works on raw binaries but NetCDF works on netCDF dataset. They also have different ways to describe the file view and buffer view. Furthermore, NetCDF is built on top of MPI-IO. All these differences motivate us to investigate how they work internally and the effects they have on the IO performance.

In this section we discuss how much difference MPI-IO and NetCDF make in IO performance. We focus on the performance achieved by increasing number of processes up to 1024 with the simplest decomposition, we are also interested in the impact of various process decompositions over 1024 processes. Furthermore, we will also discuss where each IO library is spending its time. The background information and the use of the IO libraries are described in Section 2.2 and Section 3.3.3.4.

7.1 Various numbers of processes

Figure 19 shows the IO bandwidths of MPI-IO and NetCDF libraries over different numbers of processes, ranging from 1 to 1024 processes. The Subarray approach is used to describe the buffer view and file view. Stripe count is set to 4.

In general, both libraries achieve higher performance with large numbers of processes although there is a degradation in performance (except MPI-IO write) on 128 processes. It is interesting that NetCDF has poorer serial read performance than MPI-IO read does, but it delivers higher bandwidths with increasing process count. Although MPI-IO triples the write bandwidth on more than one thousand processes compared with the serial write, NetCDF collective write does not benefit from the large process count as much as MPI-IO collective write does.
Figure 19 IO Performance of different IO libraries over different process counts, using Subarray approach; defstriped.

7.2 Various Decompositions over 1024 Processes

Figure 20 compares the IO bandwidth of two libraries over 1024 processes, with different process decompositions. The Subarray approach is used to describe the buffer view and file view. Stripe count is set to 4.

Similar to MPI-IO read and write, the NetCDF read has better performance at the first half decompositions, in which MPI-IO write can achieve 3 GiB/s and both reads can achieve 2.5 GiB/s. The result indicates that NetCDF could also attempt to rearrange the IO data to ensure a small number of large contiguous reads, and the overhead increases at the second half decompositions. However, NetCDF write performance staying around 700 MiB/s until it drops to below 500 MiB/s at $1024 \times 1$ decomposition. In general, the MPI-IO write is better than NetCDF write, but in the $512 \times 2$ and $1024 \times 1$ decompositions, it’s a reversed situation.
Figure 20 IO performance of different IO libraries over 1024 processes with different decompositions using Subarray approach.

### 7.3 Time on IO Routines

As observed in Section 7.1 and Section 7.2, no matter how process count or decomposition varies, there is little change in NetCDF write bandwidth. The reason is unclear, but we can try to understand by look into the IO routines.

The MPI-IO and NetCDF routines utilised in the benchmark are described in Section 3.3.3.4. These routines are used in the project-created, self-explanatory `mpiread()`, `mpiwrite()`, `netcdfread()` and `netcdfwrite()` to parallelise IO operations. In this section we look further into where time is spent by each IO library. To reduce the complexity of the figures, we only focus on the above-mentioned functions and the calls that account for the major time allocation in the functions.

As expected, Figure 21 shows that `MPI_File_read_all()` takes almost the same time to be executed as `mpiread()` does, taking up above 90% that of `mpiread()`. The same results can be seen in all the three striping settings. Similarly, Figure 22 shows the same results that most execution time of `mpiwrite()` is spent on `MPI_File_write_all()`. We can also see this circumstance in Figure 23 that `netcdfread()` takes most of execution time on `nf90_get_var()`.
Figure 21 Read time allocation for MPI-IO, measured on various numbers of processes and different striping settings, Subarray approach.

Figure 22 Write time allocation for MPI-IO, measured on various numbers of processes and different striping settings, Subarray approach.
Figure 23 Read time allocation for NetCDF, measured on various numbers of processes and different striping settings, Subarray approach.

Figure 24 Write time allocation for NetCDF, measured on various numbers of processes and different striping settings, Subarray approach.

However, Figure 24 shows that `nf90_put_var()` still takes up the majority of the execution time of `netcdfwrite()` in the small test cases, `nf90_close()` shows its increasingly significant role in the time consumption of `netcdfwrite()`. The circumstance is more considerable in the striped setting on more than 64 processes.
where \texttt{nf90\_close()} takes more execution time than \texttt{nf90\_put\_var()} does. We do not understand why \texttt{nf90\_close()} spends the most time on large process counts with the use of full striping, but the interesting experiment results indicate that NetCDF does not cope well with large number of processes neither with large OST counts.

7.4 Summary

In general, with default file striping, MPI-IO read/write NetCDF read are faster with increasing process counts and the performance saturate at around 256 processes that take full advantage of 4 OSTs. NetCDF can parallelise reads as well as MPI-IO does, but NetCDF write bandwidth stays almost unchanged even on more than one thousand processes. This indicates that NetCDF write could not cope well with large numbers of processes.

MPI-IO read/write and NetCDF read also perform better on the simpler process decompositions over 1024 processes, but NetCDF write remains almost unchanged. This shows that NetCDF read could rearrange the data to ensure a small number of large contiguous reads, but NetCDF could not.

The reason of why NetCDF write does not cope well with large process counts or simple decompositions over 1024 processes is unclear. The measurement of each IO routine shows that NetCDF write spends most time on \texttt{nf90\_put\_var()} on small process counts, but \texttt{nf90\_close()} tends to overtake \texttt{nf90\_put\_var()} in terming of time spent, with increasing process counts. With full striping, the time spent on \texttt{nf90\_close()} completely overtakes that on \texttt{nf90\_put\_var()} on more than 64 processes. The overall write rate for NetCDF is almost constant; the measurement on IO routines shows how the weight of write time changes from \texttt{nf90\_put\_var()} to \texttt{nf90\_close()} with increasing process counts, and shows that faster \texttt{nf90\_put\_var()} does not always result in decreased write time.
Chapter 8

Effects of File Striping

The Lustre file system allows a file to be split up across OSTs that enables to parallelise the requests of multiple IO clients accessing a single file. Parallelism in IO libraries comes from information such as the file view and buffer view on each process that enables multiple processes to access data of non-overlapping memory locations concurrently. However, MPI-IO or NetCDF could not achieve good bandwidth on a very large process count (for example, 1024) as processes may contend for a small number of IO services. IO libraries should rely on the cooperation of proper file striping to ensure scalability.

In this chapter, we investigate how file striping could affect IO performance of both MPI-IO and NetCDF. As IO performance can be improved by collective and parallel IO accesses, we further compare the performance of collective IO and non-collective IO to explain why collective IO is better and how it achieves good bandwidth.

8.1 Effects of Striping

The section discusses how much MPI-IO and NetCDF benefit from striping on Lustre file system on ARCHER. This experiment benchmarks over different numbers of processes, each of which uses Subarray approach to describe the data layout of its local array and the portions of the global data available to itself. The results of the experiment are shown in Figure 25 (read bandwidth) and Figure 26 (write bandwidth). Figure 27 is plotted from Figure 26 with the data of defstriped and striped mpiwrite removed to zoom in the data that sit at the bottom of Figure 26.

As can be seen, when the Lustre file system is configured to no striping (unstriped), both the throughputs of MPI-IO read and write sit at around 600 MiB/s and NetCDF at 300 MiB/s. There is a slight increase in both read and write rates at 4 processes, but both rates saturate at around 16 processes.

As expected, the bandwidths produced by defstriped and striped are basically the same as that we have seen by unstriped at the low number of processes. defstriped can easily improve the MPI-IO read bandwidth by a factor of 3, reaching 2 GiB/s, and improve the NetCDF read bandwidth more than 5 times the serial bandwidth at 2.2 GiB/s. But both read rates saturate around 256 processes. The improvement can be seen on MPI-IO and NetCDF writes, where the throughputs triple and double the serial IO bandwidth, at 2.8 GiB/s and 750 MiB/s respectively. MPI-IO write saturate it bandwidth after 128 processes, and NetCDF after 64 processes.
With striped setting, the bandwidth can increase by up to a factor of 11 for MPI-IO read and 9 for NetCDF read, at 11 GiB/s and 7.5 GiB/s on 512 processes. Although the improvements reduce on 1024 processes, we also can see the bandwidth achieves 6.5 times and 6 times the serial IO for MPI-IO read and NetCDF read at 6.6 GiB/s and 4.3 GiB/s respectively. NetCDF write increases the bandwidth up to a maximum of 900 MiB/s on 1024 processes, an increase of a factor of 2, while the full striping effect on NetCDF write is not as significant as on MPI-IO write, which achieves 10 GiB/s on 1024 processes, with an improvement of a factor of 12 compared to the serial write.

The results of this experiment shows an important point to achieve good IO performance: not only processes perform collective IO operations in parallel, but also the Lustre file system needs to be configured to stripe a single file across multiple OSTs. However, it is noteworthy that doing IO with a few thousand processes, both MPI-IO read and write can benefit significantly from striping, although NetCDF read can also enjoy the benefit from simply setting a higher level of striping, NetCDF write seems not.

![Figure 25 Read performance affected by difference numbers of processes and different striping settings, Subarray approach.](image)
8.2 Effects of Collective and Non-collective IO

In this section, we discuss how much difference collective and non-collective IO make with different process decompositions. To investigate the effects of non-collective IO, we need to replace the call `MPI_File_real_all()` with `MPI_File_read()`, and
replace MPI_File_write_all() with MPI_File_write(). Likewise, to do non-collective IO using NetCDF library, the argument nf90_collective passed to nf90_var_par_access() is simply replaced by nf90_independent.

The measurement is carried out on 1024 processes with various process decompositions, using MPI-IO and NetCDF libraries, Subarray approach of distribution and default striping.

Table 14 compares collective and non-collective IO bandwidth of MPI-IO. In general, non-collective is extremely poor in both read and write rates [11], with the exception of read for very simple decompositions such as 1x1024 and 2x512. Non-collective read rate is three times and 1.5 times as much as non-collective read rate on 1x1024 and 2x512 decompositions, respectively; however, in general, we can see a degradation in IO rates when changing collective IO to non-collective IO. Both non-collective read and write shows a rapid degradation on more difficult decomposition and the performance of both is extremely poor, at only 200 MiB/s for read and 2.5 MiB/s for write on 32x32 decomposition. Table 15 compares NetCDF non-collective and collective IO performances. Similar to MPI-IO, non-collective read is faster than collective read on 1x1024 and 2x512 decompositions, and a rapid degradation in both non-collective read and write rates is also presented. We only measure for the first few decompositions as non-collective IO is very slow, a single execution of MPI-IO or NetCDF read/write pair on 1024 processes cannot be completed in 3 hours.

<table>
<thead>
<tr>
<th>Data Decompositions</th>
<th>Collective (MiB/s)</th>
<th>Non-collective (MiB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mpiread</td>
<td>mpiwrite</td>
</tr>
<tr>
<td></td>
<td>mpiread</td>
<td>mpiwrite</td>
</tr>
<tr>
<td>1x1024</td>
<td>2016.42</td>
<td>2664.97</td>
</tr>
<tr>
<td></td>
<td>6360.19</td>
<td>219.78</td>
</tr>
<tr>
<td>2x512</td>
<td>2004.20</td>
<td>2586.59</td>
</tr>
<tr>
<td></td>
<td>3262.06</td>
<td>65.76</td>
</tr>
<tr>
<td>4x256</td>
<td>2169.85</td>
<td>2632.96</td>
</tr>
<tr>
<td></td>
<td>1898.60</td>
<td>20.61</td>
</tr>
<tr>
<td>32x32</td>
<td>2249.58</td>
<td>2415.91</td>
</tr>
<tr>
<td></td>
<td>199.67</td>
<td>2.46</td>
</tr>
</tbody>
</table>

Table 14 IO bandwidth of collective and non-collective MPI-IO over 1024 processes, measured with different process decompositions, Subarray approach; defstriped.

<table>
<thead>
<tr>
<th>Data Decompositions</th>
<th>Collective (MiB/s)</th>
<th>Non-collective (MiB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>netcdfread</td>
<td>netcdfwrite</td>
</tr>
<tr>
<td></td>
<td>netcdfread</td>
<td>netcdfwrite</td>
</tr>
<tr>
<td>1x1024</td>
<td>1983.93</td>
<td>893.59</td>
</tr>
<tr>
<td></td>
<td>6257.03</td>
<td>227.02</td>
</tr>
<tr>
<td>2x512</td>
<td>1880.03</td>
<td>813.75</td>
</tr>
<tr>
<td></td>
<td>3478.58</td>
<td>46.23</td>
</tr>
<tr>
<td>4x256</td>
<td>1856.49</td>
<td>787.20</td>
</tr>
<tr>
<td></td>
<td>1768.76</td>
<td>22.95</td>
</tr>
<tr>
<td>32x32</td>
<td>1927.36</td>
<td>774.42</td>
</tr>
<tr>
<td></td>
<td>225.4</td>
<td>2.46</td>
</tr>
</tbody>
</table>
The reason of why non-collective IO performance is so poor can be explained by looking into the MPI-IO reporting shown in Table 16 that collects the statistics of collective and non-collective MPI read on 1024 processes with 1x1024 decomposition, using Subarray approach and default file striping. The reporting is turned on by setting `MPICH_MPIIO_STATS=1` in the PBS scripts. We can see that the benchmark performs 1024 collective reads in collective mode but no collective reads in non-collective mode, on the contrary, it performs 1024 independent reads. We can see that the good performance of collective IO also comes from the aggregators that perform a small number of large reads (1280), reads are performed in the opposite manner in non-collective mode so it brings poor performance.

<table>
<thead>
<tr>
<th>Collective MPI-IO Read</th>
<th>Non-collective MPI-IO Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPIIO read access patterns for <code>./defstriped/big16384x20480.dat</code></td>
<td>MPIIO read access patterns for <code>./defstriped/big16384x20480.dat</code></td>
</tr>
<tr>
<td>independent reads = 0</td>
<td>independent reads = 1024</td>
</tr>
<tr>
<td>collective reads = 1024</td>
<td>collective reads = 0</td>
</tr>
<tr>
<td>independent readers = 0</td>
<td>independent readers = 1024</td>
</tr>
<tr>
<td>aggregators = 4</td>
<td>aggregators = 4</td>
</tr>
<tr>
<td>stripe count = 4</td>
<td>stripe count = 4</td>
</tr>
<tr>
<td>stripe size = 1048576</td>
<td>stripe size = 1048576</td>
</tr>
<tr>
<td>system reads = 1280</td>
<td>system reads = 20480</td>
</tr>
<tr>
<td>stripe sized reads = 0</td>
<td>stripe sized reads = 0</td>
</tr>
<tr>
<td>total bytes for reads = 1342177280 = 1280 MiB = 1 GiB</td>
<td>total bytes for reads = 1342177280 = 1280 MiB = 1 GiB</td>
</tr>
<tr>
<td>ave system read size = 1048576</td>
<td>ave system read size = 65536</td>
</tr>
<tr>
<td>number of read gaps = 0</td>
<td>number of read gaps = 0</td>
</tr>
<tr>
<td>ave read gap size = NA</td>
<td>ave read gap size = NA</td>
</tr>
</tbody>
</table>

See "Optimizing MPI I/O on Cray XE Systems" S-0013-20 for explanations.
8.3 Summary

The unstriped does not scale well, both MPI-IO and NetCDF perform parallel IO as well as a serial IO does. Although a slightly improvement in read and write performance at 4 processes, both rates saturate at around 16 processes.

MPI-IO can increase its read rate threefold with defstriped and at least 6.5 times with striped on a thousand processes, reaching 2.2 GiB/s and 6.6 GiB/s, respectively. MPI-IO write rate can triple with defstriped on 128 processes and achieve a significantly 12-time improvement with striped on 1024 processes, with write rate reaching 2.8 GiB/s and 10 GiB/s respectively. NetCDF read rate also benefit from striping, it increases fivefold with defstriped on 256 processes and at least sixfold with striped on 1024 processes, where the read rate is up to 2.2 GiB/s and 4.5 GiB/s respectively. However, NetCDF write appears not to take the advantage of striping well, the write rate can only double with striped on 1024 processes to reach 900 MiB/s that is very close to the 750 MiB/s achieved with defstriped on 128 processes.

In general, non-collective IO performance is extremely poor except the read for very simple decompositions such as 1x1024 and 2x512. It degrades rapidly on difficult process decompositions, and can only reach a few hundreds of MiB/s on read and less than 3 MiB/s on write for both MPI-IO and NetCDF.

The statistics in MPI-IO reporting shows performance of collective IO comes from multiple processes doing IO concurrently.

The comparison of MPI-IO statistics between collective and non-collective mode shows that the performance of collective IO comes from a small number of large IO operations that are performed concurrently.
Chapter 9

Conclusions

In the project, a benchmark is designed to investigate parallel IO performance on a large 2D dataset (16384x20480) on ARCHER in various aspects: different process counts, various process decompositions over a fixed process count, distributions with various implementations, IO libraries and file stripings on Lustre file system. These can be set by command arguments.

The experiments carried out in the project shows that good IO performance is achievable on ARCHER. With MPI-IO library, the write rate can reach 10’s GiB/s on 1024 processes, and the read rate can also reach 10’s GiB/s on 512 processes and 6.4 GiB/s on 1024 processes. With NetCDF library, the read and write rates can achieve 4.5 GiB/s and 900 MiB/s on 1024 processes respectively.

To be more specifically, the experiment results are discussed here:

1. Applications tend to not scale with unstriped, but can scale to 128 to 256 processes with defstriped and scale to 512 to 1024 processes with striped.
2. Fixing the process counts with various decompositions, applications can achieve best performance when processes are arranged in vertical stripes that allows the data with consecutive memory locations in the local buffer to have more chance to be also consecutive in the global file.
3. With the same decomposition, the data layout of the local array and in the global file described using block distribution can achieve better performance than that described using cyclic-block distribution, due to the complexity of algorithm.
4. Block distribution implemented by the Subarray approach is simpler and thus faster in terms of IO rates than that by the Darray-block and Indexed-block approaches.
5. The IO performance from block-cyclic distribution can significantly degrade with decreasing block sizes, where IO libraries increase overheads of IO data rearrangement prior/post to data transferring between the local array and the global file.
6. Non-collective IO is proven to be inefficient in general as independent operations on each processes tend to exhaust the IO servers with a large number of small reads/writes. The exceptions happen to the read for very simple decompositions such 1x1024 and 2x512.

To achieve good IO performance on ARCHER, we would recommend that:

1. IO needs to be performed in a collective manner.
2. The process decomposition and distribution must be chosen in a way that a small number of large contiguous reads and writes is ensured.

3. The implementation of distribution should be simple so the overhead of IO data rearrangement done by IO libraries before/after transferring between the local memory and the global file is minimised.

4. To ensure the scalability, the Lustre files system must be configured in a way that the processes assigned as aggregators can match the OSTs available so the aggregative bandwidth is obtained.

9.1 Reflection

The main goals of the project were set to look at different decompositions and investigate how parallel distributions affect IO rates, to compare IO rates on different machines, and to introduce unstructured patterns to the benchmark. If time allows, the project should select a real application and improve its IO performance, look at other parallel libraries, and look at asynchronous IO to see whether or not it benefits to IO rates.

However, around of the 50% of the tasks do not follow the plan (Figure 28) created at the start due to project change, the limitation of hardware, and time spent on writing the report:

1. The project does not study BlueGene/Q due to the unavailability. Although the MPI-IO IO performance is also measured on the UK-RDF DAC, NetCDF is not buildable on it, and yet it does not have enough cores to investigate if it can scale to one thousand processes. So task 6 is completed.

2. Asynchronous IO in task 8 is not done, the project changes to compare the performance of parallel IO in collective mode and non-collective mode.

3. MPI-IO and NetCDF libraries in task are investigated, but the project decides not proceed with HDF5 and focuses on the effects of various data distributions and their implementations.

4. The project does not improve the IO of a real application (task 7), but the conclusions given can be the guidance.

5. Unstructured data patterns mentioned in task 5 is not carried out, due to time spent on writing the report. The implementation of block-cyclic distribution in the benchmark code can work on general cases so the application to unstructured data patterns will be straightforward.
The tasks that have not been done or completed can be considered the future work.

1. Investigating IO performance on BlueGene/Q should be easier to accomplish if there is no extra effort or subtle changes to make the benchmark code buildable.
2. The benchmark could also be expanded with HDF5 and the other libraries.
3. Asynchronous IO could be introduced to the benchmark, which requires to study how it works on MP-IO and NetCDF to expand the code. A new command argument is also required to specify the IO mode.
4. The switch between collective mode and non-collective mode is hard-coded and needs to change code and recompile. This could be addressed by determined the mode dynamically according to the IO mode given to the benchmark.
5. Benchmarking unstructured data patterns could be complicated to achieve, the implementation of block-cyclic distribution in the benchmark could be a basis for the future work.
Appendix A

Provided Source

The supplied benchmark is written in Fortran and uses Crayfn as the compiler on ARCHER front-end. The key files are:

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>benchmark.f90</td>
<td>The main source file and the entry point of the benchmark.</td>
</tr>
<tr>
<td>comm.f90</td>
<td>The wrappers of MPI(-IO) APIs to create/end communicators, create derived datatypes, perform IO, etc.</td>
</tr>
<tr>
<td>netcdf.f90</td>
<td>The wrappers of NetCDF APIs to do IO.</td>
</tr>
<tr>
<td>Makefile</td>
<td>Make file.</td>
</tr>
<tr>
<td>benchmark.pbs</td>
<td>PBS scripts to submit jobs.</td>
</tr>
<tr>
<td>benchmark.sh</td>
<td>Bash scripts to run qsub command.</td>
</tr>
</tbody>
</table>

Table 17 Key Files of the benchmark
Appendix B

Running the Benchmark

B1. Scripts for Job Submission

The benchmark.sh and benchmark.pbs in the source package are provided to submit jobs on ARCHER. To submit jobs, simply typing:

> sh benchmark.sh

The details of the scripts are described as below.

# benchmark.sh

The Bash scripts can be used to loop over an array of process counts and runs the `qsub` command for each, and this can be done by setting the `nprocs` variable. For example,

```bash
for nprocs in 1 4 16 64
do
    NNODES=$(((nprocs+24-1)/24))
    qsub -l select=$NNODES -v NPROC="$nprocs" benchmark.pbs
    sleep 0.01
done
```

This submits 4 jobs to run the benchmark over 1, 4, 16, and 64 processes respectively.

# benchmark.pbs

The PBS scripts can set environment variables, command arguments, and run the benchmark. Important variables to determine which value to set to a command argument are explained as below:

The following are variables to define the values set to the command arguments.

1. IO libraries
   - `MPIIO=1`
   - `NETCDF=2`
2. Approaches of distribution
   - `SUBARRAY=1`
   - `DARRAY_CYCLIC=2`
   - `DARRAY_BLOCK=3`
   - `INDEXED_CYCLIC=4`
   - `INDEXED_BLOCK=5`
3. File striping; strings to define striping settings used in the project
The following variables must be set as an array so PBS scripts can loop over to benchmark in a single job.

1. Determine process decompositions; this sets the process counts in the first dimension in the process grid, for example, the job runs over 1024 processes with 1x1024, 2x512 and 4x256 decompositions:
   \[ PSIZESM = (1 \ 2 \ 4) \]
   The example lets the benchmark determine the decomposition automatically:
   \[ PSIZESM = (0) \]
2. Determine block sizes of each dimension for cyclic distribution; the example selects 16x16:
   \[ BSIZESM = (16) \]
   \[ BSIZESN = (16) \]
3. Determine IO libraries for both read and write; the example benchmarks over MPI-IO read/write and then over NetCDF read/write
   \[ INLIBS = ($MPIIO \ $NETCDF) \]
   \[ OUTLIBS = ($MPIIO \ $NETCDF) \]
4. Determine the approaches of distributions for read and write respectively; the example benchmarks over three approaches in a single job:
   \[ INDISTRIBS = ($SUBARRAY \ $DARRAY_BLOCK \ $INDEXED_BLOCK) \]
   \[ OUTDISTRIBS = ($SUBARRAY \ $DARRAY_CYCLIC \ $DARRAY_BLOCK) \]
5. Determine file stripings; the example benchmarks over three settings:
   \[ STRIPINGS = (unstriped \ defstriped \ striped) \]

**B2. Examples to Reproduce Experiments**

In the project we prepare three subdirectories, unstriped, defstriped, and striped, and configure to no striping (1 stripe), default striping (4 stripes), and full striping, respectively. Place the input file in each subdirectory, and specify the input file name in the PBS scripts, as described in B1. The benchmark will loop over STRIPINGS in a single job.

Below exemplify how to reproduce the experiments conducted in the project.

1. To benchmark over various process counts, using Subarray approach and defstriped (Chapter 4)
   (1) In benchmark.sh, modify the numbers that \[ nprocs \] will loop over:

```bash
unstriped
defstriped
striped
```
(2) In benchmark.pbs, set the following variables
   \( \text{PSIZESM}=(0) \)
   \( \text{INLIBS}=(\$\text{MPIIO}) \)
   \( \text{OUTLIBS}=(\$\text{MPIIO}) \)
   \( \text{INDISTRIBS}=(\$\text{SUBARRAY}) \)
   \( \text{OUTDISTRIBS}=(\$\text{SUBARRAY}) \)
   \( \text{STRIPINGS}=(\text{defstriped}) \)
   \( \text{RUNS}=20 \)

2. To benchmark over 1024 processes with various decompositions, using Subarray approach and \text{defstriped} (Chapter 5)
   (1) In benchmark.sh, modify to make \textit{nprocs} loop over 1024 only:
      \( \text{for nprocs in 1024} \)
   (2) In benchmark.pbs, set the following variables
      \( \text{PSIZESM}=(1 2 4 8 16 32 64 128 256 512 1024) \)
      \( \text{INLIBS}=(\$\text{MPIIO}) \)
      \( \text{OUTLIBS}=(\$\text{MPIIO}) \)
      \( \text{INDISTRIBS}=(\$\text{SUBARRAY}) \)
      \( \text{OUTDISTRIBS}=(\$\text{SUBARRAY}) \)
      \( \text{STRIPINGS}=(\text{defstriped}) \)
      \( \text{RUNS}=20 \)

3. To benchmark for MPI-IO over 1024 processes arranged in a 32x32 process grid, using various block sizes for block-cyclic distribution and \text{defstriped} (Section 6.2.1)
   (1) In benchmark.sh, modify to make \textit{nprocs} loop over 1024 only:
      \( \text{for nprocs in 1024} \)
   (2) In benchmark.pbs, set the following variables
      \( \text{PSIZESM}=(32) \)
      \( \text{BSIZESM}=(512 256 128 64 32 16 8 4) \)
      \( \text{BSIZESN}=(640 320 160 80 40 20 10 5) \)
      \( \text{INLIBS}=(\$\text{MPIIO}) \)
      \( \text{OUTLIBS}=(\$\text{MPIIO}) \)
      \( \text{INDISTRIBS}=(\text{DARRAY_CYCLIC} \text{INDEXED_CYCLIC}) \)
      \( \text{OUTDISTRIBS}=(\text{DARRAY_CYCLIC} \text{INDEXED_CYCLIC}) \)
      \( \text{STRIPINGS}=(\text{defstriped}) \)
      \( \text{RUNS}=20 \)

4. To benchmark for MPI-IO over 1024 processes with various decompositions, using 16x16 block size for block-cyclic distribution and \text{defstriped} (Section 6.2.2)
   (1) In benchmark.sh, modify to make \textit{nprocs} loop over 1024 only:
      \( \text{for nprocs in 1024} \)
   (2) In benchmark.pbs, set the following variables
      \( \text{PSIZESM}=(0) \)
      \( \text{BSIZESM}=(16) \)
      \( \text{BSIZESN}=(16) \)
      \( \text{INLIBS}=(\$\text{MPIIO}) \)
OUTLIBS=(SMPIIO)
INDISTRIBUT=(SDARRAY_CYCLIC$INDEXED_CYCLIC)
OUTDISTRIBUT=(SDARRAY_CYCLIC$INDEXED_CYCLIC)
STRIPINGS=(defstriped)
RUNS=20

5. To benchmark for MPI-IO and NetCDF over various process counts, using Subarray approach and defstriped (Section 7.1)
   (1) In benchmark.sh, modify the numbers that nprocs will loop over:
       for nprocs in 1 4 16 64 128 256 512 1024
   (2) In benchmark.pbs, set the following variables
       PSIZESM=(0)
       INLIBS=(SMPIIO $NETCDF)
       OUTLIBS=(SMPIIO $NETCDF)
       INDISTRIBUT=(SSUBSARRAY)
       OUTDISTRIBUT=(SSUBSARRAY)
       STRIPINGS=(defstriped)
       RUNS=20

   Experiment 6 and 7 are reproduced with the same settings.

6. To benchmark for MPI-IO and NetCDF over 1024 processes with various decompositions, using Subarray approach and defstriped (Section 7.2)

7. Time each IO routines (Section 7.3)
   (1) In benchmark.sh, modify to make nprocs loop over 1024 only:
       for nprocs in 1 1024
   (2) In benchmark.pbs, set the following variables
       PSIZESM=(1 2 4 8 16 32 64 128 256 512 1024)
       INLIBS=(SMPIIO $NETCDF)
       OUTLIBS=(SMPIIO $NETCDF)
       INDISTRIBUT=(SSUBSARRAY)
       OUTDISTRIBUT=(SSUBSARRAY)
       STRIPINGS=(defstriped)
       RUNS=20

8. To benchmark for MPI-IO and NetCDF over various process counts, using Subarray approach and 3 striping settings (Section 8.1)
   (1) In benchmark.sh, modify the numbers that nprocs will loop over:
       for nprocs in 1 4 16 64 128 256 512 1024
   (2) In benchmark.pbs, set the following variables
       PSIZESM=(0)
       INLIBS=(SMPIIO $NETCDF)
       OUTLIBS=(SMPIIO $NETCDF)
       INDISTRIBUT=(SSUBSARRAY)
       OUTDISTRIBUT=(SSUBSARRAY)
       STRIPINGS=(unstriped defstriped striped)
       RUNS=20
9. To benchmark for non-collective MPI-IO and NetCDF over 1024 processes with various decompositions, using Subarray approach and defstriped (Section 8.2)

(1) In comm.f90, modify code as described in Section 8.2 to use non-collective IO routines. Recompile the benchmark.

(2) In benchmark.sh, modify to make nprocs loop over 1024 only:

   for nprocs in 1024

(3) In benchmark.pbs, set the following variables

   PSIZEM=(1 2 4 8 16 32 64 128 256 512 1024)
   INLIBS=(MPIIO $NETCDF)
   OUTLIBS=(MPIIO $NETCDF)
   INDISTRIBS=(SUBSARRAY)
   OUTDISTRIBS=(SUBSARRAY)
   STRIPINGS=(defstriped)
   RUNS=20
References


