|epcc|

THE UNIVERSITY OF EDINBURGH

MPI-IO Exercises

David Henty

1 Introduction

The aim of this exercise is to write a simple MPI-IO program that reads a file using a block decomposition on a 2D grid of processes. You are provided with a simple working template that you extend throughout the exercise. Before updating the template, it is important that you spend some time looking at the main program to ensure that you understand how the sizes of the arrays are defined, and how the processes coordinates in the 2D grid are stored.

2 Setup

First copy the file `mpiio.tar` file from the ARCHER course web page to your workspace directory on ARCHER (`/work/y14/y14/guestXX/`) then unpack it:

```
user@archer> tar xvf mpiio.tar
x mpiio/C, 0 bytes, 0 tape blocks
x mpiio/C/cinput0480x0216.dat, 414720 bytes, 810 tape blocks
...
```

3 Master IO

You should work in either the C or F subdirectories.

Type `make` to compile the code (and a utility program for viewing data files), and submit to the batch system to run on a single process as follows:

```
user@archer> qsub -q RXXXXXX mpiio.pbs
```

The argument to `-q` is the the reserved course queue ID, which will be provided for you by the instructor.

You can monitor the progress of your job using the `qstat` command; when it finishes, standard output will appear in a file of the form `mpiio.pbs.oXXXXXX`; any errors will be in `mpiio.pbs.eXXXXXX`.

You can view the input file as follows (for C or Fortran respectively):

```
user@archer> ./cioview cinput0480x0216.dat
user@archer> ./fioview finput0480x0216.dat
```

The output file has "`_00`" appended to it because it was written by rank 0 — view it as follows

```
user@archer> ./cioview coutput0480x0216_00.dat
user@archer> ./fioview foutput0480x0216_00.dat
```

You should take a look at the code in the main programs (`mpiio.c` and `mpiio.f90`) and check that you understand what is going on. You *should not* spend a lot of time looking at the routines in `ioutils.c` and `ioutils.f90` — all you need is a basic understanding of what they do.

- `initpgrid(pcoords, nxproc, nyproc)` uses MPI cartesian topologies to work out the positions of all the processes in a process grid of size `nxproc` × `nyproc`.
- `initarray(buf, M, N)` initialises the `buf` array to some default value which is chosen to show up as grey in the image viewer.
- `createfilename(filename, basename, M, N, rank)` creates file names which include the size of the array and the rank of the process, which is useful for debugging purposes. The file name is constructed in the format `basenameMxN_rank.dat`, with the trailing process identifier being omitted if `rank` is negative.
- `ioread(filename, buf, n)` is a serial routine that reads `n` single-precision floating-point numbers from the file `filename` into the array `buf`.
- `iowrite` is almost identical to `ioread` except that it writes data instead of reading it.

The program defines a large array `buf` which holds the entire file on the master. The smaller array `x` only holds local data. The template program simply copies a subsection of the `buf` array to `x`.

4 Parallel Runs

To run codes in parallel you must do two things:

- edit the values of `XPROCS` and `YPROCS` in `mpiio.c`, or `xprocs` and `yprocs` in `mpiio.f90`, to specify the dimensions of the process grid;
- change the argument to `aprun` in the PBS batch file to match.

The dimensions of the process grid are specified at compile time to make the allocation of 2D arrays of the correct size easier in C (where we use static array allocation for simplicity). At run time, the program checks that it is running on the correct number of processes.

The exercise now proceeds as follows:

1. Edit the main program to use a 2×2 process grid, then recompile and run on 4 processes. Look at the 4 output files (note they have different names from the previous single process run as the local array size is now smaller). Check you understand the output images — you can view multiple images as follows:

   ```
   user@archer> ./cioview coutput0240x0108_*.dat
   user@archer> ./fioview foutput0240x0108_*.dat
   ```

2. Now use MPI to broadcast the `buf` array to every process and check you understand the output.
3. By using the process coordinates stored in the `pcoords` array, copy down the appropriate subarray from `buf` to `x` and check that you get the correct output for a number of different decompositions.
4. Now remove the broadcast call and distribute the data by having the master copy the appropriate data for each process from `buf` into `x` and sending it; every other process should receive this data into its local copy of `x`. Having done all the sends, the master can finally copy down its own subsection of `x`. Run the program and check that it works for a number of different decompositions.

5 Derived Datatypes

We will now eliminate all of the copying by using derived datatypes. You will have to change the send calls on the master so that they now use the new datatypes, but you should be able to leave the receive calls on the other processes unaltered.

1. Define a vector datatype which corresponds to the subarray on each process and use this to send the data. As this is a floating datatype, the master will have to specify the start of the send buffer appropriately for each send call.
2. Repeat the above using a subarray datatype. As this is a fixed datatype, the master will need to define a different type appropriate for each receiving process. However, you will be able to use the same send buffer for every send call.

6 MPI-IO

We will now eliminate both the large `buf` array and the explicit MPI send/receive calls by using MPI-IO. With MPI-IO, the appropriate data is read directly from file onto each process.

The derived datatypes you defined on the master for the previous exercise are exactly the same as are required by each process to define what data it wants to read from the file via `MPI_File_set_view`.

For floating vector types, we previously had a single type but specified a non-zero offset into the sending buffer, different for each process. In MPI-IO, the equivalent is for each process to specify a non-zero displacement into the file. The difference is that the displacement must now be specified in *bytes*. You can find out the size in bytes of a basic type (eg an `MPI_FLOAT`) by calling `MPI_Type_size`.

For fixed subarray types, we previously had multiple types (one for each receiving process) and had a zero offset for each sending buffer, ie we simply passed the start of the buffer to each send call. In MPI-IO, the equivalent is for every process to define an appropriate subarray based on its position in the 2D grid, but to specify a zero displacement in `MPI_File_set_view`.

6.1 Updating your program

Make the following changes to your program:

- remove the `buf` array
- open the file with `MPI_File_open`
- pass the correct combination of derived datatype and displacement to the `MPI_File_set_view` routine (see below)
- read the appropriate number of floats/reals from the file using `MPI_File_read_all`
- close the file with `MPI_File_close`

You should try both of the following choices for `MPI_File_set_view`:

1. Using a vector datatype and non-zero displacements.
2. Using subarray datatypes and zero displacement.

You must explicitly check the return codes from all calls to MPI-IO routines, and report an error if they differ from `MPI_SUCCESS`.

To check that your program is working correctly you should visualise the results as before. You should also run on a range of processor decompositions, eg: 4×1; 2×2; 1×4; 2×3; 3×2; ...

7 Extra Exercises

Here is a list of extra exercises you might like to try – note that they are in no particular order.

1. We have concentrated on parallel input as it is much easier to design an incremental practical example based on reading files rather than writing them. However, in practice, files are usually written much more frequently than they are read, so parallel output is more critical in terms of application performance.

   You should be able to use your existing derived types to write a single output file in parallel using `MPI_File_write_all`. If done correctly, this should be bit-identical to the original input file. Note that if you want to see the file as an image you will have to construct the file name using `createfilename` as the viewing routines require the image size to be encoded in the name. Remember to pass a rank value of -1 to this routine so that all processes use the same file name.

2. Use `MPI_Type_create_darray` to create a block decomposition. Now generalise to a block-cyclic decomposition and check that you get the correct results.

3. Define the `x` array to have a halo of depth 1. In order to read data directly into the core of this array, you will have to define a derived datatype that corresponds to the array without its halo. You can then use this datatype in `MPI_File_read`. You should ensure that you update the call to `initarray` so that the whole of `x` is initialised including the halos. Run the program and check that the core data is received as expected, and that the halos are left untouched.