|epcc|

MPI-I/O Exercises

Adrian Jackson

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.

There are two ways you can complete this exercise. One is to try implementing some of the different MPI-I/O functionality and running the code to test performance. The other is running the solutions and investigating the differences in performance between the different versions that have been implemented

We have already implemented versions of the code that:

- Read the file in on the master and broadcast the data to all processes (`mpiio_bcast.c|f90`)
- Read the file in on the master and send the appropriate chunks of data to each processes(`mpiio_ssend.c|f90`)
- Using serial I/O each processes reads its section of the file (`mpiio_individual.c|f90`)

There are then a number of versions to be fully implemented:

- Read the file in on the master and use an MPI vector datatype to send chunks of data to each processes(`mpiio_master_vector.c|f90`)
- Read the file in on the master and use an MPI subarray datatype to send chunks of data to each processes(`mpiio_master_subarray.c|f90`)
- Each process use an MPI vector datatype to to read in its portion of the file using MPI-I/O(`mpiio_read_vector.c|f90`)
- Each process use an MPI subarray datatype to to read in its portion of the file using MPI-I/O(`mpiio_read_subarray.c|f`
- Each process use an MPI vector datatype to to read in its portion of the file using collective MPI-I/O(`mpiio_readall_vector.c|f90`)
- Each process use an MPI subarray datatype to to read in its portion of the file using collective MPI-I/O(`mpiio_readall_subarray.c|f90`)

For the versions that have not been fully implemented the majority of the code is in place, with areas to be completed marked with:

```
! ADD CODE IN HERE
or
// ADD CODE IN HERE
```

The code to read in input parameters, setup datatypes, and time the I/O, is already implemented for you.

2 Setup

First copy the file `mpiio.tar` file your workspace directory on ARCHER (`/work/d131/d131/ngioXX/`) then unpack it:

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

3 Development and Running

You should work in either the C or F subdirectories.

Type `./compile.sh` 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 RNUMBER mpiio_xxxxx.pbs
```

The argument to `-q` is the the reserved queue ID, which is only available to the guest accounts, and only for the duration of the tutorial. There is a batch file for each version of the code (i.e. `mpiio_bcast.pbs`).

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_bcast.pbs.oXXXXXX`; any errors will be in `mpiio.pbs.eXXXXXX`.

The batch script is setup to run the application three times, once using a 4GB file that has been striped across 4 stripes, once a 4GB file that has been striped across all available stripes, and one using a 40GB stripe that has been striped across all available stripes.

The application should print out the mean, minimum, and maximum run time across all the processes used. The batch scripts are setup to use one node (24 cores). This can be changed for different experiments.

The executable expects a number of arguments (as you will see if you look in one of the batch files). They are as follows (using the `mpiio_bcast` executable as an example):

```
mpiio_bcast inputfile nx ny xprocs yprocs
```

Where `inputfile` is the name of the data file to be read, `nx` and `ny` specify the size of the data in the input file, and `xprocs` and `yprocs` specify how processes should be arranged in a 2d grid when running (`xprocs` is the number of processes in the x direction, and `yprocs` the number of processes in the y direction).

Note, `xprocs` $*$ `yprocs` must equal the number of MPI processes being used, and `nx` my divide exactly by `xprocs`. Likewise for `ny` and `yprocs`.

For versions of the program where the master reads in the whole array, 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`. For versions of the program where each process reads their own section of the data the `buf` array is omitted and on the `x` array defined.

It should be evident that for large scale programs the master I/O approach is not practical as it is not hard to get to data set sizes where the full dataset cannot fit into the memory in a single node.

If you decide to do the coding parts of this practical use the `C` or `F` directories. If you're more interested in just running the applications and looking at performance use the `Csol` or `Fsol` directories.

When running try different numbers of nodes/cores. You can do this by changing the `#PBS -lselect=1` line, which specifies the number of nodes to use, and the `aprun -n 24` parameter which specifies the number of MPI processes to used (24 in this example). Remember, when changing the number of MPI processes to use you will also need to change the `xprocs` and `yprocs` variables so $xprocs * yprocs = n$, ensuring `xprocs` still exactly divides `nx` and `x=yprocs` still exactly divides `ny`

In the `ioutils.c|f90` file(s) the following routines are implemented:

- `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.