epcc

Hands on with Object Stores

Exercises

Adrian Jackson, Nicolau Manubens

1 Introduction

This sheet details the practical exercises you can undertake within this tutorial. It also describes the system we are using and how to access it. For the first exercises we provide pre-compiled applications for you to run, but you will get the chance to implement the source code for this application in the later exercises.

2 Using the system

For this tutorial we will be using the NEXTGenIO prototype system. You should have an account on this system already. To access the system you need to ssh through a gateway node (replacing `XX` in the `ngguestXX` string below with the number of the actual account you have been given):

```
ssh -J ngguestXX@gateway.epcc.ed.ac.uk ngguestXX@nextgenio-login1
```

To access that system you need to a provide a authentication code, you can request this by emailing a.jackson@epcc.ed.ac.uk to get this.

From here you can compile and submit jobs. We use the modules environment for controlling software such as compilers and libraries. You can see what software has been installed on the system using the following command

```
module avail
```

The system is configured with a login node separate from the compute nodes in the system. To get the initial compiler and MPI libraries setup on the system you should do the following when you log on:

```
module load compiler/2021.1.1
module load mkl
```

We use the Slurm batch system to access and enquire about the compute nodes. You can discover how many compute nodes there are, and what memory they have installed, using the following command:

```
sinfo
```

Or:

```
sinfo -N -l
```

Below is an example of a Slurm batch script we can use to run a job:

```
#!/bin/bash
```

```
#SBATCH --nodes=1
#SBATCH --time=01:00:00
#SBATCH --job-name=test_job
#SBATCH --nvram-option=1LM:1000
#SBATCH --cpus-per-task=1

mpirun -n 48 -ppn 48 test_job
```

To run a job on the system we use the `sbatch` command, i.e. (assuming the script above is called runtestjob.sh)

```
sbatch runtestjob.sh
```

You can `squeue` to see running jobs (`squeue -u $USER` will show only your jobs) and `scancel` to cancel a job.

The `mpirun` command in the script above is the MPI job launcher which runs the executable on the selected number of nodes. The default MPI library being used is the Intel MPI library (although you can use the OpenMPI library as well by swapping out the requisite modules). For the Intel MPI library there are a number of arguments we are passing, i.e.:

```
mpirun -n 48 -ppn 48 test_job
```

Here `-n 48` specifies the number of processes (or copies of you executable) to run. `-ppn 48` specifies how many processes to run on each node requested. The exercises we are doing in this practical will have submission scripts with mpirun already specified in them

We are using the Intel compiler setup on this machine. This means the C compiler is called `icc` or `mpicc` and the Fortran compiler is called `ifort` or `mpif90`.

On the NEXTGenIO prototype system we have a Lustre filesystem where your home directory is installed, at:

```
/home/nx04/nx04/$USER
```

3 Exploring filesystem performance

To get started on the system copy the following software into your home directory:

```
git clone https://github.com/adrianjhpc/ObjectStoreTutorial
```

Change to the `ObjectStoreTutorial/Exercises` directory, i.e.:

```
cd ObjectStoreTutorial/Exercises
```

You should be able unpack with the command:

`tar xf IOR.tar.gz`.

For IOR you need to go into the `ior` directory and type:

```
make
```

Then you can run a DAOS filesystem IOR benchmark using:

```
sbatch daos_filesystem_ior.sh
```

The aim of this exercise is to run the different IOR benchmarks and compare the performance that Ceph and DAOS are providing. IOR is a common I/O benchmark designed to explore the maximum bandwidth a filesystem can provide to a parallel programme. It uses MPI to run many workers (processes) at once, and reports back the total bandwidth achieved for bulk I/O operations. It can be configured in various ways, but we are looking at relatively large read and write sizes to investigate filesystem performance.

Currently the batch script is setup to run on two nodes, but you can vary this to investigate how well both filesystems scale. The total number of nodes available is 8, and each has 32 cores, so the most you would be able to run is 256 MPI processes.

We only have DAOS configured to provide a filesystem interface at the moment (through a FUSE mount called dfuse), Ceph has to be accessed by the librados library for IOR. For this initial exercise simply run the `daos_filesystem_ior.sh` batch file provided using varying number of nodes from 2 to 8 to see what read and write bandwidth you get.

4 DFS IOR

You can also experiment with direct DAOS usage from the IOR benchmark application we have already used in the exercises. It is configured to work both in a direct method (using the DFS IOR interface) as opposed to the FUSE mount that you have run previously.

Then you can run a DFS DAOS IOR benchmark using:

```
sbatch daos_dfs_ior.sh
```

This will run using DFS using 2 nodes, and you can compare with the filesystem performance you saw earlier. You can also compare to the batch script that was used for the FUSE and you will see we no longer require the commands to setup the FUSE mount for I/O, we can simply contact DAOS directly. Explore performance across a range of node counts, what performance do you get?

You can also try exploring different DAOS directory object classes. This is done by adding this parameter to the `ior` run line in your batch script:

```
--dfs.dir_oclass SX
```

Currently this is commented out in the batch script, but you can uncomment it and try various object classes, i.e. S1, S2, S4, and SX. The default (if you do not specify anything it will use S1). How does performance vary?

We also have created a DAOS pool that has not been configured for POSIX access. To use this you should change this line in your batch script:

```
export pool=default-pool
```

to:

```
export pool=libdaos-pool
```

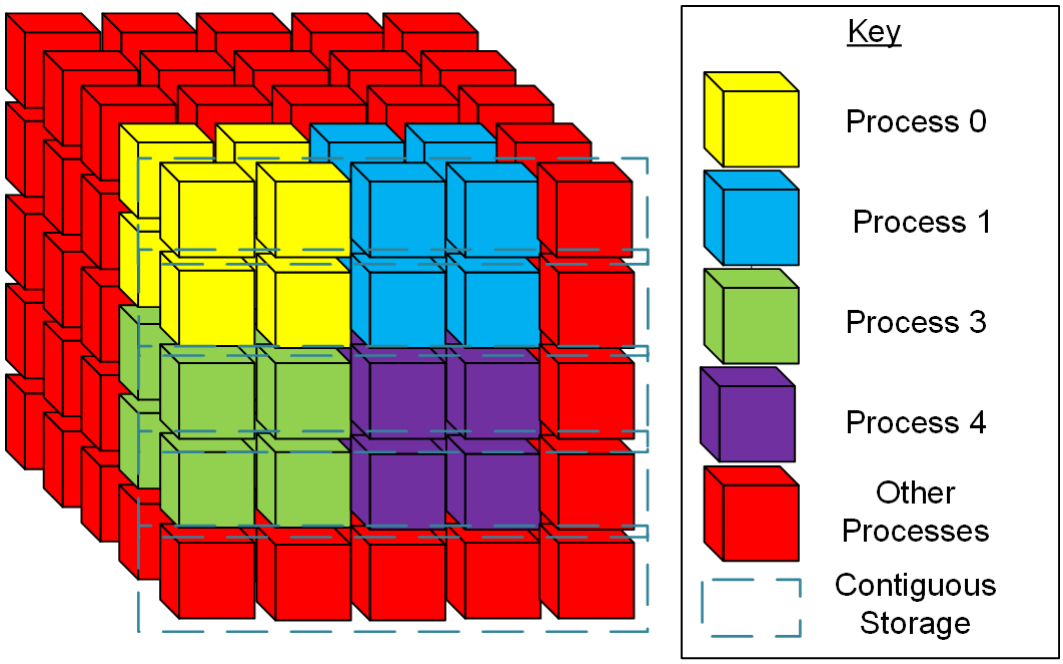
Try re-running your benchmarks with this pool, does it make a difference to the performance?

5 Exploring DAOS

There is a separate exercise sheet/presentation in the GitHub repository for the tutorial, under the `ObjectStoreTutorial/Exercises/DAOS/Handout` directory. You can follow this for the DAOS API practical exercises.

6 Porting an application to object storage technologies

To give you experience of what is required to port a larger application to an object store we have provided you with an application that uses HDF5 to store data to a file. It follows a relatively common pattern for file I/O for large scale parallel applications, i.e. each process has it's own portion of a share dataset that it writes to a common file, an example of this is illustrated in this figure:



The question is; how do you map this pattern from a single file to an object store. There are a range of options, from creating a key-value pair (or object) for each element in the array, to creating a single array for all the data shared by all processes. What you actually do depends on how you'd like to be able to index/share/search that data in the future. If you simply need to find the full dataset in the future than you could create a single array, although not all object stores give good performance for a single large array (i.e. Ceph might be limited in how big this array could be). You can also create an array per process, and then indexes that track those to enable searching/using the full dataset. You may also want to create an array/object per row or per column of data, or across some other decomposition that makes sense for your application.

If you know a bit about HDF5 you will be able to see that the current application we have provided uses a hyper-slab approach, which is writing the individual parts of data held by each process into a single shared dataset, ignoring the edges of the data held locally (data used for MPI halos) and ensuring it ends up in a single file.

We have a number of choices when porting this approach to an object store. DAOS supports using an array object, where each process writes their part of the array into the object, ending up with the same situation as the HDF5 hyperslab, albeit as an object rather than a file. It is possible to do this in Ceph but it is a bit more involved.

Another approach would be to write a single array or object per process, but then create a set of keys or objects that let you identify each part of the data and search/query them. This is possible in both Ceph and DAOS. For this practical we have provided you with three different skeleton files that you can use to implement these approaches, as well as the original HDF5 implementation:

- `hdf5.c`: The original HDF5 file
- `ceph.c`: A file containing the setup code and outline for implementing storage in Ceph
- `daos_individual.c`: The equivalent code and instructions for implementing this using the DAOS api but with an array per process
- `daos_array.c`: Code to do the same but using a single DAOS array for all process.

There are also batch scripts (the files ending in `*.sh`) and a `Makefile` to build them all. Your task is to take one (or more) of the Ceph or DAOS examples and complete the code to get it to write the dataset to the object store(s). The code is commented with what needs to be added (functions and function arguments) and you can build and run them through the Slurm batch system.

This task will involve a bit of reading the code and looking up the function call syntax for the file you have chosen, but you can ask the instructors or email/message us if you have any questions. We will release sample solutions for each of these after the tutorial has finished.

Thanks for attending and we hope it was useful!