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 are using the Google (GCP) cloud. We have created a cloud virtual machine (VM) that will act as a login node and batch system host for compute VMs, along with Ceph and Parallelstore (Google's deployment of DAOS) storage systems that can be used from the login node and the compute VMs. You will need to provide us with an email address and then we will send you an SSH key, username, and IP address for the system. You can then access the system as follows (**note: the username you use will be different from below, this will be emailed to you along with the ssh key**):

```
ssh -AX -I ssh_key tu001@35.188.183.150
```

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 openmpi/v4.1.x
```

We use the Slurm batch system to access and enquire about the compute nodes. You can discover how many compute nodes there are 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 --job-name=test_job
#SBATCH --output=test.%A.out
#SBATCH --error=test.%A.err
#SBATCH --tasks-per-node=32
#SBATCH --cpus-per-task=1
#SBATCH --time=00:05:00
#SBATCH --nodes=2
```

```
srun hostname
```

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 `srun` 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 OpenMPI library. We are specifying the number of processes we want to run using the Slurm sbatch configuration. The batch script above runs 32 processes per node and is requesting 2 nodes, meaning it will run the application on 64 processes spread across 2 nodes. If you want to vary the number of processes you run an application on you can change the number of nodes, i.e this line:

```
#SBATCH --nodes=2
```

Or you can change the number of tasks per node, i.e. this line:

```
#SBATCH --tasks-per-node=32
```

A combination of these will specify the overall number of processes to use.

2.1 Available Hardware

Currently we have 48 compute nodes configured in this GCP setup. They are n2-highcpu-32 VM types, configured with Intel IceLake processes each with 32 threads/16 physical cores. Each node has 32 GiB of memory and a 50Gbps network interface.

Alongside this we have a DAOS/ParallelStore configuration using 3 nodes with 6 TiB of NVMe storage and 150GB DRAM each. As this is configured with a redundancy factor (erasure coding of 2+1) we actually have 4 TiB of storage per node, or 12TiB overall in the DAOS. Ceph uses the same node configuration (i.e. 3 nodes with 6 TiB of storage per node) but is configured with a pool replication factor of 2, giving a total of 3 TiB storage per Ceph node or 9TiB overall.

You can try running jobs up to 8 compute nodes, or 256 processes in total.

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 Ceph IOR

The next exercise is to run IOR using the librados backend, an implementation that undertakes IOR using the Ceph object store library. We have provided a batch script to enable this:

```
sbatch ceph_object_ior.sh
```

It will write and read using librados and you can explore the same variation in node and/or process counts as you did for the DAOS filesystem configuration.

5 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?

6 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.

7 Exploring librados

A Ceph Storage Cluster system (i.e. Ceph Monitor plus Ceph OSDs plus Ceph Manager, also know as RADOS) has been deployed for this tutorial and made accessible from the Slurm login node and compute nodes. A Ceph user named “client.tutorial” has been created in the system, as well as a “default-pool” with a replication factor of 2.

A Ceph configuration file to access this system and the tutorial user keyring have been installed under `/opt/apps/ceph` in all nodes. The “ceph” and “rados” command-line tools are installed as well, and librados and headers can be found under `/usr/lib64` and `/usr/include/rados`, respectively.

You can use the command-line tools first to explore the system. These commands need to know where to look for the Ceph configuration file and the Ceph user keyring, if not present in `/etc/ceph`, as well as what Ceph user to attempt the commands as – by default “client.admin” is used.

We could specify these in every command-line call via the “-c”, “-k”, and “-n” arguments, or alternatively define the CEPH_ARGS variable, as follows, so that these arguments are added automatically to all subsequent calls.

```
export CEPH_ARGS="-c /opt/apps/ceph/ceph.conf -n client.tutorial"
```

Note that the keyring file location is not specified because it is already provided in the Ceph configuration file. A summary of the Ceph Storage Cluster status can be retrieved as follows.

```
ceph -s
```

This reports the Cluster ID, health status, number of daemons up or down, number of pools, placement groups and objects in the system, and usage statistics. The pools in a system can be listed as follows.

```
ceph osd pool ls
```

The “ceph” command-line tool provides many other functions to check cluster, daemon, and device status, as well as to perform administrative tasks.

The “rados” command-line tool wraps librados, enabling simple librados tasks without the need to write and compile code using the library. This tool also requires configuration and detects the CEPH_ARGS environment variable if specified. Objects in a pool can be listed as follows.

```
rados ls --pool default-pool --namespace tu004
```

Note that, if --namespace is not provided, the “default” namespace will be used. To list objects in all namespaces, the --all option can be specified instead of --namespace.

The command “rados put” can be used to write data from a file into a new RADOS object, and “rados get” to read data from an object into a file, as follows.

```
echo "helloworld" > test.txt

rados put object_name test.txt --pool default-pool --namespace tu004

rados ls --pool default-pool --namespace tu004

rados get object_name test_read.txt --pool default-pool --namespace tu004

cat test_read.txt

rm test.txt test_read.txt
```

RADOS Omaps can also be used via the “rados” tool, as in the following example.

```
rados setomapval test_omap key1 val1 --pool=default-pool --namespace=tu004

rados getomapval test_omap key1 --pool=default-pool --namespace=tu004

rados listomapkeys test_omap --pool=default-pool --namespace=tu004
```

Use “rados rm” to remove objects.

```
rados rm object_name --pool default-pool --namespace tu004
```

To remove all objects in a namespace, you can use the following script.

```
rados ls --pool=default-pool --namespace=tu004 | xargs -I{} rados rm {} --
pool=default-pool --namespace=tu004
```

The “rados” command-line tool offers plenty of options, all documented in rados --help.

Nevertheless, programming your application with librados is the way to achieve best performance (as you can reuse cluster connections and I/O contexts) and have full access to features such as multi-operation transactions and asynchronous I/O.

To do so, you only need to include the librados headers in your C program, and write your code using librados, specifying the Ceph user name when creating the cluster handle and reading the configuration file as in the following example.

```
cat >> libradosapp.c << EOF

#include <librados.h>

int main() {

  int rc;

  rados_t cluster;

  rc = rados_create2(&cluster, "ceph", "client.tutorial", 0);

  if (rc < 0) {

    printf("rados_create2 failed\n");

    return 1;

  }

  rc = rados_conf_read_file(cluster, "/opt/apps/ceph/ceph.conf");

  if (rc < 0) {

    printf("rados_conf_read_file failed\n");

    return 1;

  }

  rc = rados_connect(cluster);

  if (rc < 0) {

    // ...

  }

  // ...

EOF
```

REMEMBER: DO NOT FORGET THAT, BEFORE TERMINATING EXECUTION,

- **CLUSTER CONNECTIONS MUST BE SHUT DOWN IF rados_connect SUCCEEDS**
- **IO CONTEXTS MUST BE DESTROYED IF rados_ioctx_create SUCCEEDS**
- **TRANSACTIONS MUST BE RELEASED IF rados_create_write/read_op SUCCEEDS**
- **OMAP ITERATORS MUST BE RELEASED IF rados_read_op_omap_get_* SUCCEEDS**

!!! EVEN IF ERRORS OCCUR DURING PROGRAM EXECUTION !!!

You can then compile the application, specifying the location of the librados headers and the requirement to link to librados, and run it.

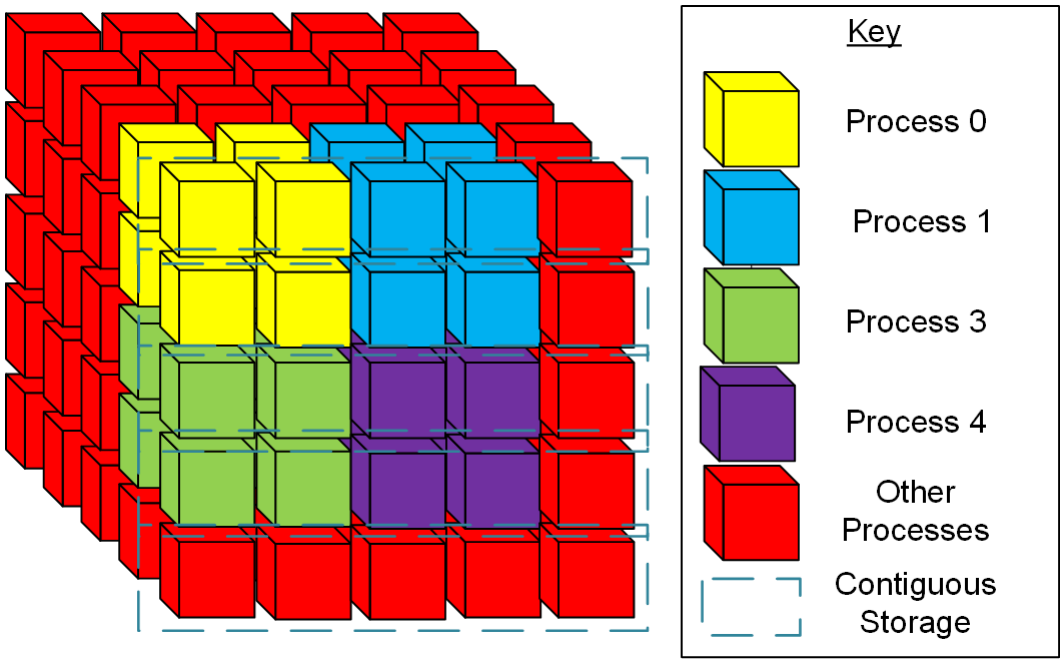
```
source /opt/intel/setvars.sh
```

```
mpicc libradosapp.c -I /usr/include/rados/ -lrados -o libradosapp

./libradosapp
```

8 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!