epcc

Persistent Memory Exercises

Adrian Jackson

1 Introduction

This exercise 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 to a gateway node (replacing `XX` in the `ngguestXX` string below with the number of the actual account you have been given):

```
ssh ngguestXX@hydra-vpn.epcc.ed.ac.uk
```

Once on this gateway node you need to ssh in to the NEXTGenIO system itself (if you want to open any windows, i.e. emacs, you'll need to add a –Y or –X flag to the command below):

```
ssh nextgenio-login2
```

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
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
```

The line `#SBATCH --nvram-option=1LM:1000` specifies the setup of the persistent memory (B-APM) required by the job. For this system this option is required otherwise your job will not run. The first part of the text after `--nvram-option=` specifies the platform mode required for the job, `1LM` specifies AppDirect mode and `2LM` specifies MemoryMode. The number after the platform mode (i.e. `:1000`) specifies how much B-APM memory is required for the specified platform mode (in this case the script is specifying AppDirect with 1TB of B-APM memory) .

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
```

On the compute nodes there are two further filesystems:

```
/mnt/pmem_fsdax0
/mnt/pmem_fsdax1
```

These correspond to the B-APM installed with the first processor in the node (pmem_fsdax0) and the second processor in the node (pmem_fsdax1). Accessing the B-APM attached to the processor your application is running on is faster than accessing the B-APM attached to the other processor in the node.

3 Running a basic program

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

```
/home/nx01/shared/pmtutorial/exercises/streams.tar.gz
/home/nx01/shared/pmtutorial/exercises/IOR.tar.gz
```

You should be able unpack both with the commands:

```
tar xf streams.tar.gz
tar xf IOR.tar.gz.
```

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

```
module load compiler

module load mpich/3.4.3

make
```

Then you can run a Lustre IOR benchmark using:

```
sbatch lustre_ior.sh
```

And you can run a fsdax IOR benchmark using:

```
sbatch fsdax_ior.sh
```

For the STREAMs benchmark you need to go into the streams directory and type:

```
make
```

Then you can run it using:

```
sbatch run_streams.sh
```

The aim of this exercise is to run the different IOR benchmarks and the STREAMs benchmarks.

4 NUMA-aware I/O

We should now have been able to run IOR using Lustre and also the filesystems mounted on the compute nodes (i.e. /mnt/pmem_fsdax0/), and STREAMS using DRAM. For this practical the exercise is to modify the IOR source code so that is can exploit both the filesystems on the compute node (i.e. /mnt/pmem_fsdax0 and /mnt/pmem_fsdax1), and ensure that processes running on socket 0 use pmem_fsdax0 and those running on socket 1 use pmem_fsdax1.

To compute which socket a process is running on you can use this code:

```
unsigned long GetProcessorAndCore(int *chip, int *core){

   unsigned long a,d,c;

 __asm__ volatile("rdtscp" : "=a" (a), "=d" (d), "=c" (c));

   *chip = (c & 0xFFF000)>>12;

   *core = c & 0xFFF;
```

```
   return ((unsigned long)a) | (((unsigned long)d) << 32);;

}
```

You can then use that, with some string manipulation, to create the string `/mnt/pmem_fsdax0` or `/mnt/pmem_fsdax1`. This code will need to be added into the file `IOR.c`, in the function `GetTestFileName`. Alter this to create the filename based on the socket number. If you struggle with this task there is an example of doing this in the `IOR-solution.tar.gz` code in the exercise directory.

5 DAOS IOR

You can also experiment with DAOS 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) or through a FUSE mount. We have provided both examples with submission scripts:

Then you can run a DFS DAOS IOR benchmark using:

```
sbatch daos_ior.sh
```

And you can run a FUSE DAOS IOR benchmark using:

```
sbatch daos_fuse_ior.sh
```

However, before you do either you will need to create a container on the DAOS storage system to put your data inside. The following command will setup a container for you:

```
srun -n 1 --nvram-options=1LM:1000 --reservation=nx01_59 daos container
create --pool=tutorial --label=$USER-container --type=POSIX
```

This should produce you output that looks like this:

```
  Container UUID : e241e056-5e16-4da4-b1f2-73bf96ed9425

  Container Label: adrianj-container

  Container Type : POSIX
```

You can check what containers are available in the DAOS pool you have access to with the following command:

```
srun -n 1 --nvram-options=1LM:1000 --reservation=nx01_59 daos pool list-
containers tutorial
```

Once you have your container you can update the `daos_ior.sh` and `daos_fuse_ior.sh` batch scripts to use that container. For the FUSE example (`daos_fuse_ior.sh`), the command is setup to use the container name, so you can change this line:

```
srun -N $NODES -n $NODES dfuse -m /tmp/$USER-fuse --disable-caching --pool
tutorial --cont ADD_CONTAINER_NAME_HERE --foreground &
```

To replace `ADD_CONTAINER_NAME_HERE` with the container name you have just created.

For the DFS API we are using the container ID for reference, so you can change this line:

```
srun --cpu-bind=core -N $NODES -n $NPROCS
/lustre/home/nx01/nx01/adrianj/ior/src/ior -v -a DFS -w -W -r -R -o
/test2.$SLURM_JOB_ID -b 1G -t 1m -C --dfs.pool cc290816-37e4-49ad-906c-
5f1389cca819 --dfs.cont ADD_CONTAINER_ID_HERE --dfs.oclass SX
```

To replace `ADD_CONTAINER_ID_HERE` with the UUID of the container you just created.

You should now be able to run both of these IOR scripts to use DAOS in different ways. You can try using different numbers of nodes (modify the `#SBATCH --nodes=1` line in the batch scripts) to see how performance changes. You can also try adding and removing the `-F` flag on the ior run line, which specifies if a file per process should be used, or a single shared file. This will give an idea of how well DAOS will handle larger scale parallel I/O to a single file, and how the different interface mechanisms cope with that.

6 Persistent Streams

For this exercise the goal is to change the STREAMS source code to utilise PMDK to create the memory allocations that STREAMS uses for its calculations, rather than (as currently happens) using DRAM. To exploit PMDK you will need to use the following functions and header files:

```
#include <libpmem.h>

pmem_map_file

pmem_persist

pmem_unmap
```

You will also need to load the PMDK module (`module load pmdk`) and add this to your batch script to compile and run your modified application.

There are different choices that can be made on when and where to persist data within the streams kernels, and which parts of the streams data to place in volatile and non-volatile memory. Try different persistent combinations and memory allocation locations to investigate impact on performance.

In more detail, you need to modify the file `streams_memory_task.c` in the STREAMS source code to change how memory is created (replacing the malloc and free calls to use PMDK functionality), and for safely adding persist function calls when the data transfers are taking place. You can initially start with a single persist function call (i.e. `pmem_persist`) after each benchmark loop.

Once you have implemented the above you can re-run and benchmark to see what performance you achieve, and you can then try moving the persist functions inside the loop so data is persisted after every write, to see what performance impact that has.

7 Correct Persisting

This is a more advanced Persistent Memory exercise, and lets you explore

To investigate correct persistence in applications create two application, one that writes data to persistent memory using PMDK and the other that reads the data. Ensure you persist memory at the end of the writing application and run your two applications so one consumes the data from the other and checks it is correct.

Once you have done that correctness check, remove the persistence in the write application, compile, and re-run your workflow (the two applications together). Can the reader now correctly obtain the data it is expecting?