PDC

Houjun Tang, Qiao Kang, Bin Dong, Quincey Koziol, Suren Byna,

Jul 19, 2023

GETTING STARTED

Proactive Data Containers (PDC) software provides an object-focused data management API, a runtime system with a set of scalable data object management services, and tools for managing data objects stored in the PDC system. The PDC API allows efficient and transparent data movement in complex memory and storage hierarchy. The PDC runtime system performs data movement asynchronously and provides scalable metadata operations to find and manipulate data objects. PDC revolutionizes how data is managed and accessed by using object-centric abstractions to represent data that moves in the high-performance computing (HPC) memory and storage subsystems. PDC manages extensive metadata to describe data objects to find desired data efficiently as well as to store information in the data objects.

More information and publications of PDC is available at https://sdm.lbl.gov/pdc

If you use PDC in your research, please use the following citation:

Byna, Suren, Dong, Bin, Tang, Houjun, Koziol, Quincey, Mu, Jingqing, Soumagne, Jerome, Vishwanath, Venkat, Warren, Richard, and Tessier, François. Proactive Data Containers (PDC) v0.1. Computer Software. https://github.com/hpc-io/pdc. USDOE. 11 May. 2017. Web. doi:10.11578/dc.20210325.1.

CHAPTER

ONE

GETTING STARTED

1.1 Installing PDC with Spack

Spack is a package manager for supercomputers, Linux, and macOS. More information about Spack can be found at: https://spack.io PDC and its dependent libraries can be installed with spack:

```
# Clone Spack
git clone -c feature.manyFiles=true https://github.com/spack/spack.git
# Install the latest PDC release version with Spack
./spack/bin/spack install pdc
```

If you run into issues with `libfabric` on macOS and some Linux distributions, you can enable all fabrics by installing PDC using:

```
spack install pdc ^libfabric fabrics=sockets,tcp,udp,rxm
```

1.2 Installing PDC from source code

We recommend using GCC 7 or a later version. Intel and Cray compilers also work.

1.2.1 Dependencies

The following dependencies need to be installed:

- MPI
- libfabric
- Mercury

PDC can use either MPICH or OpenMPI as the MPI library, if your system doesn't have one installed, follow MPICH Installers' Guide or Installing Open MPI

We provide detailed instructions for installing libfabric, Mercury, and PDC below.

Attention: Following the instructions below will record all the environmental variables needed to run PDC in the `$WORK_SPACE/pdc_env.sh` file, which can be used for future PDC runs with `source $WORK_SPACE/pdc_env.sh`.

Prepare Work Space and download source codes

Before installing the dependencies and downloading the code repository, we assume there is a directory created for your installation already, e.g. *$WORK_SPACE* and now you are in *$WORK_SPACE*.

```
export WORK_SPACE=/path/to/your/work/space
mkdir -p $WORK_SPACE/source
mkdir -p $WORK_SPACE/install

cd $WORK_SPACE/source
git clone git@github.com:ofiwg/libfabric.git
git clone git@github.com:mercury-hpc/mercury.git --recursive
git clone git@github.com:hpc-io/pdc.git

export LIBFABRIC_SRC_DIR=$WORK_SPACE/source/libfabric
export MERCURY_SRC_DIR=$WORK_SPACE/source/mercury
export PDC_SRC_DIR=$WORK_SPACE/source/pdc

export LIBFABRIC_DIR=$WORK_SPACE/install/libfabric
export MERCURY_DIR=$WORK_SPACE/install/mercury
export PDC_DIR=$WORK_SPACE/install/pdc

mkdir -p $LIBFABRIC_SRC_DIR
mkdir -p $MERCURY_SRC_DIR
mkdir -p $PDC_SRC_DIR

mkdir -p $LIBFABRIC_DIR
mkdir -p $MERCURY_DIR
mkdir -p $PDC_DIR

# Save the environment variables to a file
echo "export LIBFABRIC_SRC_DIR=$LIBFABRIC_SRC_DIR" > $WORK_SPACE/pdc_env.sh
echo "export MERCURY_SRC_DIR=$MERCURY_SRC_DIR" >> $WORK_SPACE/pdc_env.sh
echo "export PDC_SRC_DIR=$PDC_SRC_DIR" >> $WORK_SPACE/pdc_env.sh
echo "export LIBFABRIC_DIR=$LIBFABRIC_DIR" >> $WORK_SPACE/pdc_env.sh
echo "export MERCURY_DIR=$MERCURY_DIR" >> $WORK_SPACE/pdc_env.sh
echo "export PDC_DIR=$PDC_DIR" >> $WORK_SPACE/pdc_env.sh
```

From now on you can simply run the following commands to set the environment variables:

```
export WORK_SPACE=/path/to/your/work/space
source $WORK_SPACE/pdc_env.sh
```

Install libfabric

```
cd $LIBFABRIC_SRC_DIR
git checkout v1.18.0
./autogen.sh
./configure --prefix=$LIBFABRIC_DIR CC=mpicc CFLAG="-O2"
make -j && make install

# Test the installation
```

(continues on next page)

(continued from previous page)

```
make check

# Set the environment variables
export LD_LIBRARY_PATH="$LIBFABRIC_DIR/lib:$LD_LIBRARY_PATH"
export PATH="$LIBFABRIC_DIR/include:$LIBFABRIC_DIR/lib:$PATH"
echo 'export LD_LIBRARY_PATH=$LIBFABRIC_DIR/lib:$LD_LIBRARY_PATH' >> $WORK_SPACE/pdc_env.
↪sh
echo 'export PATH=$LIBFABRIC_DIR/include:$LIBFABRIC_DIR/lib:$PATH' >> $WORK_SPACE/pdc_
↪env.sh
```

Note: `CC=mpicc` may need to be changed to the corresponding compiler in your system, e.g. `CC=cc` or `CC=gcc`. On Perlmutter@NERSC, `--disable-efa --disable-sockets` should be added to the `./configure` command when compiling on login nodes.

Install Mercury

```
cd $MERCURY_SRC_DIR
# Checkout a release version
git checkout v2.2.0
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=$MERCURY_DIR -DCMAKE_C_COMPILER=mpicc -DBUILD_SHARED_
↪LIBS=ON \
      -DBUILD_TESTING=ON -DNA_USE_OFI=ON -DNA_USE_SM=OFF -DNA_OFI_TESTING_PROTOCOL=tcp ..
↪/
make -j && make install

# Test the installation
ctest

# Set the environment variables
export LD_LIBRARY_PATH="$MERCURY_DIR/lib:$LD_LIBRARY_PATH"
export PATH="$MERCURY_DIR/include:$MERCURY_DIR/lib:$PATH"
echo 'export LD_LIBRARY_PATH=$MERCURY_DIR/lib:$LD_LIBRARY_PATH' >> $WORK_SPACE/pdc_env.sh
echo 'export PATH=$MERCURY_DIR/include:$MERCURY_DIR/lib:$PATH' >> $WORK_SPACE/pdc_env.sh
```

Note: `CC=mpicc` may need to be changed to the corresponding compiler in your system, e.g. `-DCMAKE_C_COMPILER=cc` or `-DCMAKE_C_COMPILER=gcc`. Make sure the ctest passes. PDC may not work without passing all the tests of Mercury.

Install PDC

```
cd $PDC_SRC_DIR
git checkout develop
mkdir build
cd build
cmake -DBUILD_MPI_TESTING=ON -DBUILD_SHARED_LIBS=ON -DBUILD_TESTING=ON -DCMAKE_INSTALL_
↪PREFIX=$PDC_DIR \
      -DPDC_ENABLE_MPI=ON -DMERCURY_DIR=$MERCURY_DIR -DCMAKE_C_COMPILER=mpicc -DMPI_RUN_
↪CMD=mpiexec ../
make -j && make install

# Set the environment variables
export LD_LIBRARY_PATH="$PDC_DIR/lib:$LD_LIBRARY_PATH"
export PATH="$PDC_DIR/include:$PDC_DIR/lib:$PATH"
echo 'export LD_LIBRARY_PATH=$PDC_DIR/lib:$LD_LIBRARY_PATH' >> $WORK_SPACE/pdc_env.sh
echo 'export PATH=$PDC_DIR/include:$PDC_DIR/lib:$PATH' >> $WORK_SPACE/pdc_env.sh
```

Note: `-DCMAKE_C_COMPILER=mpicc -DMPI_RUN_CMD=mpiexec` may need to be changed to `-DCMAKE_C_COMPILER=cc -DMPI_RUN_CMD=srun` depending on your system environment.

Test Your PDC Installation

PDC's `ctest` contains both sequential and parallel/MPI tests, and can be run with the following in the *build* directory.

```
ctest
```

Note: If you are using PDC on an HPC system, e.g. Perlmutter@NERSC, `ctest` should be run on a compute node, you can submit an interactive job on Perlmutter: `salloc --nodes 1 --qos interactive --time 01:00:00 --constraint cpu --account=mxxxx`

1.2.2 Running PDC

If you have followed all the previous steps, `$WORK_SPACE/pdc_env.sh` sets all the environment variables needed to run PDC, and you only need to do the following once in each terminal session before running PDC.

```
export WORK_SPACE=/path/to/your/work/space
source $WORK_SPACE/pdc_env.sh
```

PDC is a typical client-server application. To run PDC, one needs to start the server processes first, and then the clients can be started and connected to the PDC servers automatically.

On Linux

Run 2 server processes in the background

```
mpiexec -np 2 $PDC_DIR/bin/pdc_server.exe &
```

Run 4 client processes that concurrently create 1000 objects and then create and query 1000 tags:

```
mpiexec -np 4 $PDC_DIR/share/test/bin/kvtag_add_get_scale 1000 1000 1000
```

On Perlmutter

Run 4 server processes, each on one compute node in the background:

```
srun -N 4 -n 4 -c 2 --mem=25600 --cpu_bind=cores $PDC_DIR/bin/pdc_server.exe &
```

Run 64 client processes that concurrently create 1000 objects and then create and query 100000 tags:

```
srun -N 4 -n 64 -c 2 --mem=25600 --cpu_bind=cores $PDC_DIR/share/test/bin/kvtag_add_get_
↪scale 100000 100000 100000
```

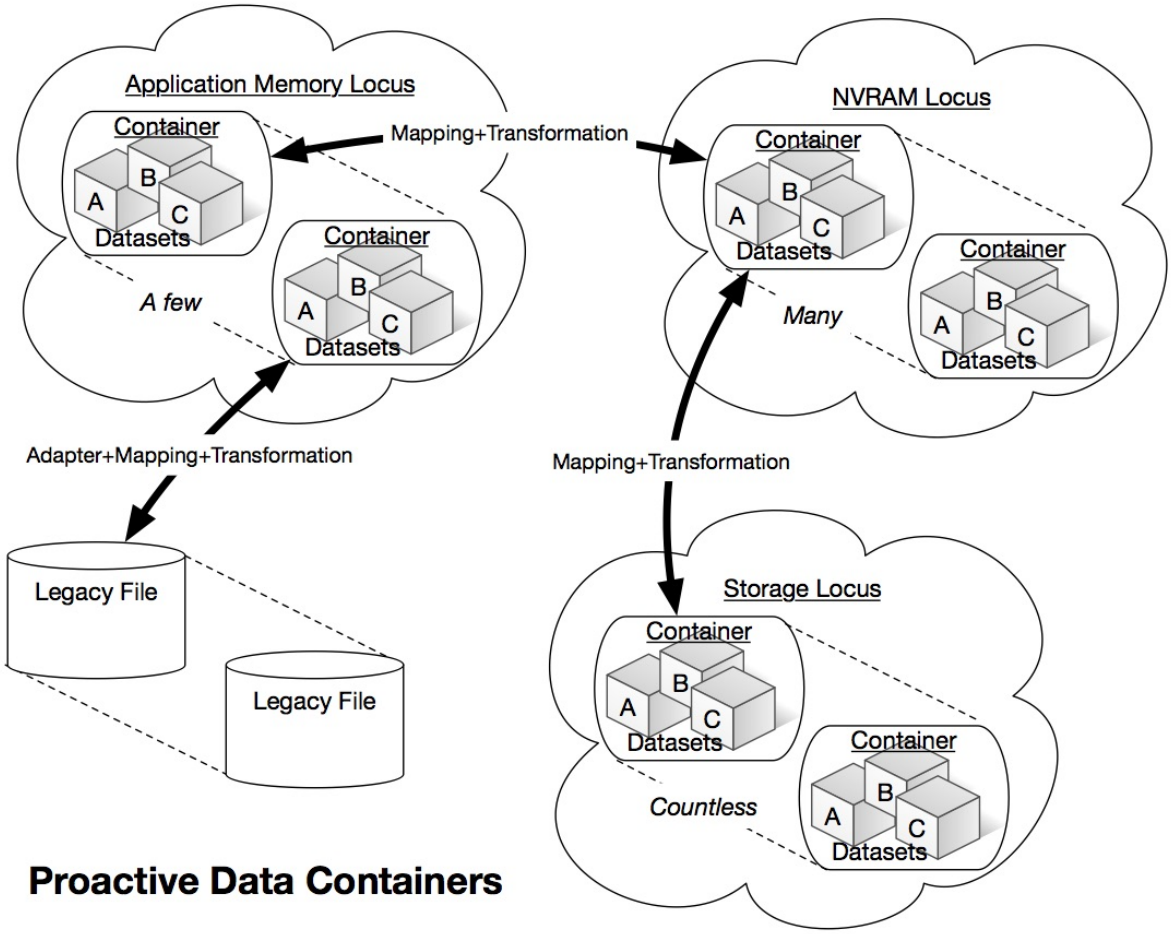
CHAPTER

TWO

INTRODUCTION

Emerging high performance computing (HPC) systems are expected to be deployed with an unprecedented level of complexity, due to a very deep system memory/storage hierarchy. This hierarchy is expected to range from CPU cache through several levels of volatile memory to non-volatile memory, traditional hard disks, and tape. Simple and efficient methods of data management and movement through this hierarchy is critical for scientific applications using exascale systems. Existing storage system and I/O (SSIO) technologies face severe challenges in dealing with these requirements. POSIX and MPI I/O standards that are the basis for existing I/O libraries and parallel file systems present fundamental challenges in the areas of scalable metadata operations, semantics-based data movement performance tuning, asynchronous operation, and support for scalable consistency of distributed operations.

Moving toward new paradigms for SSIO in the extreme-scale era, we propose to investigate novel object- based data abstractions and storage mechanisms that take advantage of the deep storage hierarchy and enable proactive automated performance tuning. In order to achieve these overarching goals, we propose a fundamental new data abstraction, called Proactive Data Containers (PDC). A PDC is a container within a locus of storage (memory, NVRAM, disk, etc.) that stores science data in an object-oriented manner. Managing data as objects enables powerful optimization opportunities for data movement and transformations. In this project, we will research: 1) formulation of object-oriented PDCs and their mapping in different levels of the exascale storage hierarchy; 2) efficient strategies for moving data in deep storage hierarchies using PDCs; 3) techniques for transforming and reorganizing data based on application requirements; and 4) novel analysis paradigms for enabling data transformations and user-defined analysis on data in PDCs. The intent of our research is to move the field of HPC SSIO in a direction where it may ultimately be possible to develop scientific applications without the need to perform cumbersome and inefficient tuning to optimize data movement on every system the application runs on.



PDCs will have an impact in many science areas, given the importance of the data management and I/O software stack in achieving science discoveries at scale. The foundations of the novel data management and storage paradigm approaches and formalisms proposed in this research are expected to be applicable to a broad range of scientific and engineering problems that utilize computational and experimental facilities for predictive understanding of physical processes through data analytics and visualization. The proposed techniques are expected to accelerate the crucial process of data-driven exploration and knowledge discovery. While we will work closely with a set of key DOE science applications in the areas of cosmology, climate, genomics, and high-energy density physics to evaluate our research, the proposed new I/O paradigm will be broadly applicable to all users of DOE HPC facilities.

CHAPTER

THREE

HDF5 VOL FOR PDC

3.1 Installing HDF5 VOL for PDC from source

The following instructions are for installing PDC on Linux and Cray machines. These instructions assume that PDC and its dependencies have all already been installed from source (libfabric and Mercury).

3.1.1 Install HDF5

If a local version of HDF5 is to be used, then the following can be used to install HDF5.

```
$ wget "https://www.hdfgroup.org/package/hdf5-1-12-1-tar-gz/?wpdmdl=15727&
↪refresh=612559667d6521629837670"
$ mv index.html?wpdmdl=15727&refresh=612559667d6521629837670 hdf5-1.12.1.tar.gz
$ tar zxf hdf5-1.12.1.tar.gz
$ cd hdf5-1.12.1
$ ./configure --prefix=$HDF5_DIR
$ make
$ make check
$ make install
$ make check-install
```

3.1.2 Building VOL-PDC

```
$ git clone https://github.com/hpc-io/vol-pdc.git
$ cd vol-pdc
$ mkdir build
$ cd build
$ cmake ../ -DHDF5_INCLUDE_DIR=$HDF5_INCLUDE_DIR -DHDF5_LIBRARY=$HDF5_LIBRARY -DBUILD_
↪SHARED_LIBS=ON -DHDF5_DIR=$HDF5_DIR
$ make
$ make install
```

To compile and run examples:

```
$ cd vol-pdc/examples
$ cmake .
$ make
$ mpirun -N 1 -n 1 -c 1 <pdc installation directory>/bin/pdc_server.exe &
```

(continues on next page)

(continued from previous page)

```
$ mpirun -N 1 -n 1 -c 1 ./h5pdc_vpicio test
$ mpirun -N 1 -n 1 -c 1 <pdc installation directory>/bin/close_server
```

3.2 VOL Function Notes

The following functions have been properly defined:

- H5VL_pdc_info_copy
- H5VL_pdc_info_cmp
- H5VL_pdc_info_free
- H5VL_pdc_info_to_str
- H5VL_pdc_str_to_info
- H5VL_pdc_file_create
- H5VL_pdc_file_open
- H5VL_pdc_file_close
- H5VL_pdc_file_specific
- H5VL_pdc_dataset_create
- H5VL_pdc_dataset_open
- H5VL_pdc_dataset_write
- H5VL_pdc_dataset_read
- H5VL_pdc_dataset_get
- H5VL_pdc_dataset_close
- H5VL_pdc_introspect_get_conn_cls
- H5VL_pdc_introspect_get_cap_flags
- H5VL_pdc_group_create
- H5VL_pdc_group_open
- H5VL_pdc_attr_create
- H5VL_pdc_attr_open
- H5VL_pdc_attr_read
- H5VL_pdc_attr_write
- H5VL_pdc_attr_get
- H5VL_pdc_object_open
- H5VL_pdc_object_get
- H5VL_pdc_object_specific

Any function not listed above is either not currently used by the VOL and therefore does nothing or is called by the VOL, but doesn't need to do anything relevant to the VOL and therefore does nothing.

CHAPTER

FOUR

HELLO PDC EXAMPLE

4.1 PDC Hello World

- pdc_init.c
- A PDC program starts with PDCinit and finishes with PDCclose.
- To a simple hello world program for PDC, use the following command.

```
make pdc_init
./run_test.sh ./pdc_init
```

- The script "run_test.sh" starts a server first. Then program "obj_get_data" is executed. Finally, the PDC servers are closed.
- Alternatively, the following command can be used for multile MPI processes.

```
make pdc_init
./mpi_test.sh ./pdc_init mpiexec 2 4
```

- The above command will start a server with 2 processes. Then it will start the application program with 4 processes. Finally, all servers are closed.
- On supercomputers, "mpiexec" can be replaced with "srun", "jsrun" or "aprun".

CHAPTER

FIVE

API DOCUMENTATION WITH EXAMPLES

5.1 PDC general APIs

- pdcid_t PDCinit(const char *pdc_name)
  - **Input:**
    - pdc_name is the reference for PDC class. Recommended use "pdc"
  - **Output:**
    - PDC class ID used for future reference.
  - All PDC client applications must call PDCinit before using it. This function will setup connections from clients to servers. A valid PDC server must be running.
  - For developers: currently implemented in pdc.c.
- perr_t PDCclose(pdcid_t pdcid)
  - **Input:**
    - PDC class ID returned from PDCinit.
  - **Output:**
    - SUCCEED if no error, otherwise FAIL.
  - This is a proper way to end a client-server connection for PDC. A PDCinit must correspond to one PDCclose.
  - For developers: currently implemented in pdc.c.
- perr_t PDC_Client_close_all_server()
  - **Output:**
    - SUCCEED if no error, otherwise FAIL.
  - Close all PDC servers that running.
  - For developers: see PDC_client_connect.c

5.2 PDC container APIs

- **pdcid_t PDCcont_create(const char *cont_name, pdcid_t cont_prop_id)**
  - **Input:**
    - cont_name: the name of container. e.g "c1", "c2"
    - cont_prop_id: property ID for inheriting a PDC property for container.
  - Output: pdc_id for future referencing of this container, returned from PDC servers.
  - Create a PDC container for future use.
  - For developers: currently implemented in pdc_cont.c. This function will send a name to server and receive an container id. This function will allocate necessary memories and initialize properties for a container.
- **pdcid_t PDCcont_create_col(const char *cont_name, pdcid_t cont_prop_id)**
  - **Input:**
    - cont_name: the name to be assigned to a container. e.g "c1", "c2"
    - cont_prop_id: property ID for inheriting a PDC property for container.
  - Output: pdc_id for future referencing.
  - Exactly the same as PDCcont_create, except all processes must call this function collectively. Create a PDC container for future use collectively.
  - For developers: currently implemented in pdc_cont.c.
- **pdcid_t PDCcont_open(const char *cont_name, pdcid_t pdc)**
  - **Input:**
    - cont_name: the name of container used for PDCcont_create.
    - pdc: PDC class ID returned from PDCinit.
  - **Output:**
    - error code. FAIL OR SUCCEED
  - Open a container. Must make sure a container named cont_name is properly created (registered by PDCcont_create at remote servers).
  - For developers: currently implemented in pdc_cont.c. This function will make sure the metadata for a container is returned from servers. For collective operations, rank 0 is going to broadcast this metadata ID to the rest of processes. A struct _pdc_cont_info is created locally for future reference.
- **perr_t PDCcont_close(pdcid_t id)**
  - **Input:**
    - container ID, returned from PDCcont_create.
    - cont_prop_id: property ID for inheriting a PDC property for container.
  - **Output:**
    - error code, SUCCEED or FAIL.
  - Correspond to PDCcont_open. Must be called only once when a container is no longer used in the future.

  - For developers: currently implemented in pdc_cont.c. The reference counter of a container is decremented. When the counter reaches zero, the memory of the container can be freed later.

- **struct pdc_cont_info *PDCcont_get_info(const char *cont_name)**
  - **Input:**
    - name of the container
  - **Output:**
    - Pointer to a new structure that contains the container information See container info (Get Container Info link)
    - Get container information
    - For developers: See pdc_cont.c. Use name to search for pdc_id first by linked list lookup. Make a copy of the metadata to the newly malloced structure.
- **perr_t PDCcont_persist(pdcid_t cont_id)**
  - **Input:**
    - cont_id: container ID, returned from PDCcont_create.
  - **Output:**
    - error code, SUCCEED or FAIL.
  - Make a PDC container persist.
  - For developers, see pdc_cont.c. Set the container life field PDC_PERSIST.
- **perr_t PDCprop_set_cont_lifetime(pdcid_t cont_prop, pdc_lifetime_t cont_lifetime)**
  - **Input:**
    - cont_prop: Container property pdc_id
    - cont_lifetime: See container life time (Get container life time link)
  - **Output:**
    - error code, SUCCEED or FAIL.
  - Set container life time for a property.
  - For developers, see pdc_cont.c.
- **pdcid_t PDCcont_get_id(const char *cont_name, pdcid_t pdc_id)**
  - **Input:**
    - cont_name: Name of the container
    - pdc_id: PDC class ID, returned by PDCinit
  - **Output:**
    - container ID
  - Get container ID by name. This function is similar to open.
  - For developers, see pdc_client_connect.c. It will query the servers for container information and create a container structure locally.
- **perr_t PDCcont_del(pdcid_t cont_id)**
  - **Input:**

        * cont_id: container ID, returned from PDCcont_create.
    - **Output:**
        * error code, SUCCEED or FAIL.
    - Delete a container
    - For developers: see pdc_client_connect.c. Need to send RPCs to servers for metadata update.
- **perr_t PDCcont_put_tag(pdcid_t cont_id, char *tag_name, void *tag_value, psize_t value_size)**
    - **Input:**
        * cont_id: Container ID, returned from PDCcont_create.
        * tag_name: Name of the tag
        * tag_value: Value to be written under the tag
        * value_size: Number of bytes for the tag_value (tag_size may be more informative)
    - **Output:**
        * error code, SUCCEED or FAIL.
    - Record a tag_value under the name tag_name for the container referenced by cont_id.
    - For developers: see pdc_client_connect.c. Need to send RPCs to servers for metadata update.
- **perr_t PDCcont_get_tag(pdcid_t cont_id, char *tag_name, void **tag_value, psize_t *value_size)**
    - **Input:**
        * cont_id: Container ID, returned from PDCcont_create.
        * tag_name: Name of the tag
        * value_size: Number of bytes for the tag_value (tag_size may be more informative)
    - **Output:**
        * tag_value: Pointer to the value to be read under the tag
        * error code, SUCCEED or FAIL.
    - Retrieve a tag value to the memory space pointed by the tag_value under the name tag_name for the container referenced by cont_id.
    - For developers: see pdc_client_connect.c. Need to send RPCs to servers for metadata retrival.
- **perr_t PDCcont_del_tag(pdcid_t cont_id, char *tag_name)**
    - **Input:**
        * cont_id: Container ID, returned from PDCcont_create.
        * tag_name: Name of the tag
    - **Output:**
        * error code, SUCCEED or FAIL.
    - Delete a tag for a container by name
    - For developers: see pdc_client_connect.c. Need to send RPCs to servers for metadata update.
- **perr_t PDCcont_put_objids(pdcid_t cont_id, int nobj, pdcid_t *obj_ids)**
    - **Input:**

        * cont_id: Container ID, returned from PDCcont_create.
        * nobj: Number of objects to be written
        * obj_ids: Pointers to the object IDs
    - **Output:**
        * error code, SUCCEED or FAIL.
    - Put an array of objects to a container.
    - For developers: see pdc_client_connect.c. Need to send RPCs to servers for metadata update.
- perr_t PDCcont_get_objids(pdcid_t cont_id ATTRIBUTE(unused), int *nobj ATTRIBUTE(unused), pdcid_t **obj_ids ATTRIBUTE(unused)) TODO:
- **perr_t PDCcont_del_objids(pdcid_t cont_id, int nobj, pdcid_t *obj_ids)**
    - **Input:**
        * cont_id: Container ID, returned from PDCcont_create.
        * nobj: Number of objects to be deleted
        * obj_ids: Pointers to the object IDs
    - **Output:**
        * error code, SUCCEED or FAIL.
    - Delete an array of objects to a container.
    - For developers: see pdc_client_connect.c. Need to send RPCs to servers for metadata update.

5.3 PDC object APIs

- **pdcid_t PDCobj_create(pdcid_t cont_id, const char *obj_name, pdcid_t obj_prop_id)**
    - **Input:**
        * cont_id: Container ID, returned from PDCcont_create.
        * obj_name: Name of objects to be created
        * obj_prop_id: Property ID to be inherited from.
    - **Output:**
        * Local object ID
    - Create a PDC object.
    - For developers: see pdc_obj.c. This process need to send the name of the object to be created to the servers. Then it will receive an object ID. The object structure will inherit attributes from its container and input object properties.
- **PDCobj_create_mpi(pdcid_t cont_id, const char *obj_name, pdcid_t obj_prop_id, int rank_id, MPI_Comm comm)**
    - **Input:**
        * cont_id: Container ID, returned from PDCcont_create.
        * obj_name: Name of objects to be created
        * rank_id: Which rank ID the object is placed to

        * comm: MPI communicator for the rank_id
    - **Output:**
        * Local object ID
    - Create a PDC object at the rank_id in the communicator comm. This function is a colllective operation.
    - For developers: see pdc_mpi.c. If rank_id equals local process rank, then a local object is created. Otherwise we create a global object. The object metadata ID is broadcasted to all processes if a global object is created using MPI_Bcast.

- **pdcid_t PDCobj_open(const char *obj_name, pdcid_t pdc)**
    - **Input:**
        * obj_name: Name of objects to be created
        * pdc: PDC class ID, returned from PDCInit
    - **Output:**
        * Local object ID
    - Open a PDC ID created previously by name.
    - For developers: see pdc_obj.c. Need to communicate with servers for metadata of the object.

- **perr_t PDCobj_close(pdcid_t obj_id)**
    - **Input:**
        * obj_id: Local object ID to be closed.
    - **Output:**
        * error code, SUCCEED or FAIL.
    - Close an object. Must do this after open an object.
    - For developers: see pdc_obj.c. Dereference an object by reducing its reference counter.

- **struct pdc_obj_info *PDCobj_get_info(pdcid_t obj)**
    - **Input:**
        * obj_name: Local object ID
    - **Output:**
      *object information see object information (insert link to object information)
    - Get a pointer to a structure that describes the object metadata.
    - For developers: see pdc_obj.c. Pull out local object metadata by ID.

- **pdcid_t PDCobj_put_data(const char *obj_name, void *data, uint64_t size, pdcid_t cont_id)**
    - **Input:**
        * obj_name: Name of object
        * data: Pointer to data memory
        * size: Size of data
        * cont_id: Container ID of this object
    - **Output:**
        * Local object ID created locally with the input name

  - Write data to an object.
  - For developers: see pdc_client_connect.c. Nedd to send RPCs to servers for this request. (TODO: change return value to perr_t)

- **perr_t PDCobj_get_data(pdcid_t obj_id, void *data, uint64_t size)**
  - **Input:**
    - obj_id: Local object ID
    - size: Size of data
  - **Output:**
    - data: Pointer to data to be filled
    - error code, SUCCEED or FAIL.
  - Read data from an object.
  - For developers: see pdc_client_connect.c. Use PDC_obj_get_info to retrieve name. Then forward name to servers to fulfill requests.
- **perr_t PDCobj_del_data(pdcid_t obj_id)**
  - **Input:**
    - obj_id: Local object ID
  - **Output:**
    - error code, SUCCEED or FAIL.
  - Delete data from an object.
  - For developers: see pdc_client_connect.c. Use PDC_obj_get_info to retrieve name. Then forward name to servers to fulfill requests.
- **perr_t PDCobj_put_tag(pdcid_t obj_id, char *tag_name, void *tag_value, psize_t value_size)**
  - **Input:**
    - obj_id: Local object ID
    - tag_name: Name of the tag to be entered
    - tag_value: Value of the tag
    - value_size: Number of bytes for the tag_value
  - **Output:**
    - error code, SUCCEED or FAIL.
  - Set the tag value for a tag
  - For developers: see pdc_client_connect.c. Need to use PDC_add_kvtag to submit RPCs to the servers for metadata update.
- **perr_t PDCobj_get_tag(pdcid_t obj_id, char *tag_name, void **tag_value, psize_t *value_size)**
  - **Input:**
    - obj_id: Local object ID
    - tag_name: Name of the tag to be entered
  - **Output:**

        * tag_value: Value of the tag
        * value_size: Number of bytes for the tag_value
        * error code, SUCCEED or FAIL.
    - Get the tag value for a tag
    - For developers: see pdc_client_connect.c. Need to use PDC_get_kvtag to submit RPCs to the servers for metadata update.
- **perr_t PDCobj_del_tag(pdcid_t obj_id, char *tag_name)**
    - **Input:**
        * obj_id: Local object ID
        * tag_name: Name of the tag to be entered
    - **Output:**
        * error code, SUCCEED or FAIL.
    - Delete a tag.
    - For developers: see pdc_client_connect.c. Need to use PDCtag_delete to submit RPCs to the servers for metadata update.

5.4 PDC region APIs

5.5 PDC property APIs

5.6 PDC query APIs

- **pdc_query_t *PDCquery_create(pdcid_t obj_id, pdc_query_op_t op, pdc_var_type_t type, void *value)**
    - **Input:**
        * obj_id: local PDC object ID
        * op: one of the followings, see PDC query operators (Insert PDC query operators link)
        * type: one of PDC basic types, see PDC basic types (Insert PDC basic types link)
        * value: constraint value
    - **Output:**
        * a new query structure, see PDC query structure (PDC query structure link)
    - Create a PDC query.
    - For developers, see pdc_query.c. The constraint field of the new query structure is filled with the input arguments. Need to search for the metadata ID using object ID.
- **void PDCquery_free(pdc_query_t *query)**
    - **Input:**
        * query: PDC query from PDCquery_create
    - Free a query structure.

  - For developers, see pdc_client_server_common.c.

- **void PDCquery_free_all(pdc_query_t *root)**
  - **Input:**
    - root: root of queries to be freed
  - **Output:**
    - error code, SUCCEED or FAIL.
  - Free all queries from a root.
  - For developers, see pdc_client_server_common.c. Recursively free left and right branches.
- **pdc_query_t *PDCquery_and(pdc_query_t *q1, pdc_query_t *q2)**
  - **Input:**
    - q1: First query
    - q2: Second query
  - **Output:**
    - A new query after and operator.
  - Perform the and operator on the two PDC queries.
  - For developers, see pdc_query.c
- **pdc_query_t *PDCquery_or(pdc_query_t *q1, pdc_query_t *q2)**
  - **Input:**
    - q1: First query
    - q2: Second query
  - **Output:**
    - A new query after or operator.
  - Perform the or operator on the two PDC queries.
  - For developers, see pdc_query.c
- **perr_t PDCquery_sel_region(pdc_query_t *query, struct pdc_region_info *obj_region)**
  - **Input:**
    - query: Query to select the region
    - obj_region: An object region
  - **Output:**
    - error code, SUCCEED or FAIL.
  - Select a region for a PDC query.
  - For developers, see pdc_query.c. Set the region pointer of the query structure to the obj_region pointer.
- **perr_t PDCquery_get_selection(pdc_query_t *query, pdc_selection_t *sel)**
  - **Input:**
    - query: Query to get the selection
  - **Output:**

        * sel: PDC selection defined as the following. This selection describes the query shape, see PDC selection structure (Insert link to PDC selection structure)
        * error code, SUCCEED or FAIL.
    - Get the selection information of a PDC query.
    - For developers, see pdc_query.c and PDC_send_data_query in pdc_client_connect.c. Copy the selection structure received from servers to the sel pointer.

- **perr_t PDCquery_get_nhits(pdc_query_t *query, uint64_t *n)**
    - **Input:**
        * query: Query to calculate the number of hits
    - **Output:**
        * n: number of hits
        * error code, SUCCEED or FAIL.
    - Get the number of hits for a PDC query
    - For developers, see pdc_query.c and PDC_send_data_query in pdc_client_connect.c. Copy the selection structure received from servers to the sel pointer.

- **perr_t PDCquery_get_data(pdcid_t obj_id, pdc_selection_t *sel, void *obj_data)**
    - **Input:**
        * obj_id: The object for query
        * sel: Selection of the query, query_id is inside it.
    - **Output:**
        * obj_data: Pointer to the data memory filled with query data.
    - Retrieve data from a PDC query for an object.
    - For developers, see pdc_query.c and PDC_Client_get_sel_data in pdc_client_connect.c.

- **perr_t PDCquery_get_histogram(pdcid_t obj_id)**
    - **Input:**
        * obj_id: The object for query
    - **Output:**
        * error code, SUCCEED or FAIL.
    - Retrieve histogram from a query for a PDC object.
    - For developers, see pdc_query.c. This is a local operation that does not really do anything.

- **void PDCselection_free(pdc_selection_t *sel)**
    - **Input:**
        * sel: Pointer to the selection to be freed.
    - **Output:**
        * None
    - Free a selection structure.
    - For developers, see pdc_client_connect.c. Free the coordinates.

- **void PDCquery_print(pdc_query_t *query)**
  - **Input:**
    - query: the query to be printed
  - **Output:**
    - None
  - Print the details of a PDC query structure.
  - For developers, see pdc_client_server_common.c.
- **void PDCselection_print(pdc_selection_t *sel)**
  - **Input:**
    - sel: the PDC selection to be printed
  - **Output:**
    - None
  - Print the details of a PDC selection structure.
  - For developers, see pdc_client_server_common.c.

5.7 PDC hist APIs

- **pdc_histogram_t *PDC_gen_hist(pdc_var_type_t dtype, uint64_t n, void *data)**
  - **Input:**
    - dtype: One of the PDC basic types see PDC basic types (Insert link to PDC basic types)
    - n: number of values with the basic types.
    - data: pointer to the data buffer.
  - **Output:**
    - a new PDC histogram structure (Insert link to PDC histogram structure)
  - Generate a PDC histogram from data. This can be used to optimize performance.
  - For developers, see pdc_hist_pkg.c
- **pdc_histogram_t *PDC_dup_hist(pdc_histogram_t *hist)**
  - **Input:**
    - hist: PDC histogram structure (Insert link to PDC histogram structure)
  - **Output:**
    - a copied PDC histogram structure (Insert link to PDC histogram structure)
  - For developers, see pdc_hist_pkg.c
- **pdc_histogram_t *PDC_merge_hist(int n, pdc_histogram_t **hists)**
  - **Input:**
    - hists: an array of PDC histogram structure to be merged (Insert link to PDC histogram structure)
  - **Output:**

        * A merged PDC histogram structure (Insert link to PDC histogram structure)
    - Merge multiple PDC histograms into one
    - For developers, see pdc_hist_pkg.c
- **void PDC_free_hist(pdc_histogram_t *hist)**
    - **Input:**
        * hist: the PDC histogram structure to be freed (Link to Histogram structure)
    - **Output:**
        * None
    - Delete a histogram
    - For developers, see pdc_hist_pkg.c, free structure's internal arrays.
- **void PDC_print_hist(pdc_histogram_t *hist)**
    - **Input:**
        * hist: the PDC histogram structure to be printed (Insert link to histogram structure)
    - **Output:**
        * None
    - Print a PDC histogram's information. The counter for every bin is displayed.
    - For developers, see pdc_hist_pkg.c.

5.8 PDC Data types

5.9 Basic types

```
typedef enum {
        PDC_UNKNOWN      = -1, /* error                                  */
        PDC_INT          = 0,  /* integer types                          */
        PDC_FLOAT        = 1,  /* floating-point types                   */
        PDC_DOUBLE       = 2,  /* double types                           */
        PDC_CHAR         = 3,  /* character types                        */
        PDC_COMPOUND     = 4,  /* compound types                         */
        PDC_ENUM         = 5,  /* enumeration types                      */
        PDC_ARRAY        = 6,  /* Array types                            */
        PDC_UINT         = 7,  /* unsigned integer types                 */
        PDC_INT64        = 8,  /* 64-bit integer types                   */
        PDC_UINT64       = 9,  /* 64-bit unsigned integer types          */
        PDC_INT16        = 10,
        PDC_INT8         = 11,
        NCLASSES         = 12  /* this must be last                      */
} pdc_var_type_t;
```

5.10 Histogram structure

```
typedef struct pdc_histogram_t {
        pdc_var_type_t dtype;
    int            nbin;
    double         incr;
    double        *range;
    uint64_t      *bin;
} pdc_histogram_t;
```

5.11 Container info

```
struct pdc_cont_info {
        /*Inherited from property*/
    char                  *name;
    /*Registered using PDC_id_register */
    pdcid_t                local_id;
    /* Need to register at server using function PDC_Client_create_cont_id */
    uint64_t               meta_id;
};
```

5.12 Container life time

```
typedef enum {
        PDC_PERSIST,
        PDC_TRANSIENT
} pdc_lifetime_t;
```

5.13 Object property public

```
struct pdc_obj_prop *obj_prop_pub {
    /* This ID is the one returned from PDC_id_register . This is a property ID*/
    pdcid_t           obj_prop_id;
    /* object dimensions */
    size_t            ndim;
    uint64_t         *dims;
    pdc_var_type_t    type;
};
```

5.14 Object property

```
struct _pdc_obj_prop {
        /* Suffix _pub probably means public attributes to be accessed. */
    struct pdc_obj_prop *obj_prop_pub {
        /* This ID is the one returned from PDC_id_register . This is a property ID*/
        pdcid_t           obj_prop_id;
        /* object dimensions */
        size_t            ndim;
        uint64_t         *dims;
        pdc_var_type_t    type;
    };
    /* This ID is returned from PDC_find_id with an input of ID returned from PDC init.
     * This is true for both object and container.
     * I think it is referencing the global PDC engine through its ID (or name). */
    struct _pdc_class   *pdc{
        char        *name;
        pdcid_t     local_id;
    };
    /* The following are created with NULL values in the PDC_obj_create function. */
    uint32_t            user_id;
    char               *app_name;
    uint32_t            time_step;
    char               *data_loc;
    char               *tags;
    void               *buf;
    pdc_kvtag_t        *kvtag;

    /* The following have been added to support of PDC analysis and transforms.
       Will add meanings to them later, they are not critical. */
    size_t            type_extent;
    uint64_t          locus;
    uint32_t          data_state;
    struct _pdc_transform_state transform_prop{
        _pdc_major_type_t storage_order;
        pdc_var_type_t    dtype;
        size_t            ndim;
        uint64_t          dims[4];
        int               meta_index; /* transform to this state */
    };
};
```

5.15 Object info

```
struct pdc_obj_info  {
        /* Directly coped from user argument at object creation. */
    char                  *name;
    /* 0 for location = PDC_OBJ_LOAL.
     * When PDC_OBJ_GLOBAL = 1, use PDC_Client_send_name_recv_id to retrieve ID. */
    pdcid_t                meta_id;
```

(continues on next page)

(continued from previous page)

```
    /* Registered using PDC_id_register */
    pdcid_t                 local_id;
    /* Set to 0 at creation time. *
    int                     server_id;
    /* Object property. Directly copy from user argument at object creation. */
    struct pdc_obj_prop     *obj_pt;
};
```

5.16 Object structure

```
struct _pdc_obj_info {
    /* Public properties */
    struct pdc_obj_info     *obj_info_pub {
        /* Directly copied from user argument at object creation. */
        char                    *name;
        /* 0 for location = PDC_OBJ_LOAL.
        * When PDC_OBJ_GLOBAL = 1, use PDC_Client_send_name_recv_id to retrieve ID. */
        pdcid_t                 meta_id;
        /* Registered using PDC_id_register */
        pdcid_t                 local_id;
        /* Set to 0 at creation time. *
        int                     server_id;
        /* Object property. Directly copy from user argument at object creation. */
        struct pdc_obj_prop     *obj_pt;
    };
    /* Argument passed to obj create*/
    _pdc_obj_location_t     location enum {
        /* Either local or global */
        PDC_OBJ_GLOBAL,
        PDC_OBJ_LOCAL
    }
    /* May be used or not used depending on which creation function called. */
    void                    *metadata;
    /* The container pointer this object sits in. Copied*/
    struct _pdc_cont_info   *cont;
    /* Pointer to object property. Copied*/
    struct _pdc_obj_prop    *obj_pt;
    /* Linked list for region, initialized with NULL at create time.*/
    struct region_map_list *region_list_head {
        pdcid_t                 orig_reg_id;
        pdcid_t                 des_obj_id;
        pdcid_t                 des_reg_id;
        /* Double linked list usage*/
        struct region_map_list *prev;
        struct region_map_list *next;
    };
};
```

5.17 Region info

```
struct pdc_region_info {
        pdcid_t                 local_id;
        struct _pdc_obj_info *obj;
        size_t                  ndim;
        uint64_t               *offset;
        uint64_t               *size;
        bool                    mapping;
        int                     registered_op;
        void                   *buf;
};
```

5.18 Access type

```
typedef enum { PDC_NA=0, PDC_READ=1, PDC_WRITE=2 }
```

5.19 Query operators

```
typedef enum {
    PDC_OP_NONE = 0,
    PDC_GT      = 1,
    PDC_LT      = 2,
    PDC_GTE     = 3,
    PDC_LTE     = 4,
    PDC_EQ      = 5
} pdc_query_op_t;
```

5.20 Query structures

```
typedef struct pdc_query_t {
    pdc_query_constraint_t *constraint{
            pdcid_t             obj_id;
            pdc_query_op_t      op;
            pdc_var_type_t      type;
            double              value;   // Use it as a generic 64bit value
            pdc_histogram_t     *hist;

            int                 is_range;
            pdc_query_op_t      op2;
            double              value2;

            void                *storage_region_list_head;
            pdcid_t             origin_server;
            int                 n_sent;
```

(continues on next page)

(continued from previous page)

```
            int                 n_recv;
}
    struct pdc_query_t      *left;
    struct pdc_query_t      *right;
    pdc_query_combine_op_t  combine_op;
    struct pdc_region_info *region;             // used only on client
    void                   *region_constraint;  // used only on server
    pdc_selection_t        *sel;
} pdc_query_t;
```

5.21 Selection structure

```
typedef struct pdcquery_selection_t {
pdcid_t  query_id;
size_t   ndim;
uint64_t nhits;
uint64_t *coords;
uint64_t coords_alloc;
} pdc_selection_t;
```

5.22 Developers notes

- This note is for developers. It helps developers to understand the code structure of PDC code as fast as possible.
- PDC internal data structure
  - **Linkedlist**
    - Linkedlist is an important data structure for managing PDC IDs.
    - Overall. An PDC instance after PDC_Init() has a global variable pdc_id_list_g. See pdc_interface.h

```
struct PDC_id_type {
PDC_free_t                  free_func;         /* Free function for object's␣
↪of this type    */
PDC_type_t                  type_id;           /* Class ID for the type    ␣
↪                */
        //    const                      PDCID_class_t *cls;/* Pointer to ID␣
↪class                      */
unsigned                    init_count;        /* # of times this type has␣
↪been initialized  */
unsigned                    id_count;          /* Current number of IDs␣
↪held                */
pdcid_t                     nextid;            /* ID to use for the next␣
↪atom               */
DC_LIST_HEAD(_pdc_id_info)  ids;               /* Head of list of IDs      ␣
↪                */
};
```

(continues on next page)

(continued from previous page)

```
struct pdc_id_list {
struct PDC_id_type *PDC_id_type_list_g[PDC_MAX_NUM_TYPES];
};
struct pdc_id_list *pdc_id_list_g;
```

* pdc_id_list_g is an array that stores the head of linked list for each types.
* The _pdc_id_info is defined as the followng in pdc_id_pkg.h.

```
struct _pdc_id_info {
pdcid_t             id;             /* ID for this info                */
hg_atomic_int32_t   count;          /* ref. count for this atom        */
void                *obj_ptr;       /* pointer associated with the atom */
PDC_LIST_ENTRY(_pdc_id_info) entry;
};
```

* obj_ptr is the pointer to the item the ID refers to.
* See pdc_linkedlist.h for implementations of search, insert, remove etc. operations

– **ID**

* ID is important for managing different data structures in PDC.
* e.g Creating objects or containers will return IDs for them

– **pdcid_t PDC_id_register(PDC_type_t type, void *object)**

* This function maintains a linked list. Entries of the linked list is going to be the pointers to the objects. Every time we create an object ID for object using some magics. Then the linked list entry is going to be put to the beginning of the linked list.
* type: One of the followings

```
typedef enum {
        PDC_BADID        = -1,  /* invalid Type                ␣
↪   */
        PDC_CLASS        = 1,   /* type ID for PDC             ␣
↪   */
        PDC_CONT_PROP    = 2,   /* type ID for container property ␣
↪   */
        PDC_OBJ_PROP     = 3,   /* type ID for object property ␣
↪   */
        PDC_CONT         = 4,   /* type ID for container       ␣
↪   */
        PDC_OBJ          = 5,   /* type ID for object          ␣
↪   */
        PDC_REGION       = 6,   /* type ID for region          ␣
↪   */
        PDC_NTYPES       = 7    /* number of library types, MUST BE LAST! ␣
↪   */
} PDC_type_t;
```

* Object: Pointer to the class instance created (bad naming, not necessarily a PDC object).

– **struct _pdc_id_info *PDC_find_id(pdcid_t idid);**

    * Use ID to get struct _pdc_id_info. For most of the times, we want to locate the object pointer inside the structure. This is linear search in the linked list.
    * idid: ID you want to search.

- PDC core classes.
  - **Property**
    * Property in PDC serves as hint and metadata storage purposes.
    * Different types of object has different classes (struct) of properties.
    * See pdc_prop.c, pdc_prop.h and pdc_prop_pkg.h for details.
  - **Container**
    * Container property

```
struct _pdc_cont_prop {
/* This class ID is returned from PDC_find_id with an input of ID returned␣
↪from PDC init. This is true for both object and container.
*I think it is referencing the global PDC engine through its ID (or name).␣
↪*/
        struct _pdc_class *pdc{
        /* PDC class instance name*/
        char        *name;
        /* PDC class instance ID. For most of the times, we only have 1 PDC␣
↪class instance. This is like a global variable everywhere.*/
        pdcid_t     local_id;
};
/* This ID is the one returned from PDC_id_register . This is a property ID␣
↪type.
 * Some kind of hashing algorithm is used to generate it at property create␣
↪time*/
 pdcid_t           cont_prop_id;
/* Not very important */          pdc_lifetime_t    cont_life;
};
```

    * Container structure (pdc_cont_pkg.h and pdc_cont.h)

```
struct _pdc_cont_info {
struct pdc_cont_info    *cont_info_pub {
        /*Inherited from property*/
        char                    *name;
        /*Registered using PDC_id_register */
        pdcid_t                 local_id;
        /* Need to register at server using function PDC_Client_create_cont_
↪id */
        uint64_t                meta_id;
};
/* Pointer to container property.
* This struct is copied at create time.*/
struct _pdc_cont_prop   *cont_pt;
};
```

  - Object
    * Object property See Object Property

* Object structure (pdc_obj_pkg.h and pdc_obj.h) See Object Structure

CHAPTER

SIX

PDC TOOLS

6.1 Build Instructions

```
$ cd tools
$ cmake .
$ make
```

6.2 Commands

6.2.1 pdc_ls

Takes in a directory containing PDC metadata checkpoints or an individual metadata checkpoint file and outputs information on objects saved in the checkpoint(s).

Usage: `./pdc_ls <checkpoint> <additional arguments>`:

Arguments:

- `-json <filename>`: save the output to a specified file <filename> in json format.
- `-n <objectname>`: only display objects with a specific object name <objectname>. Regex matching of object names is supported.
- `-i <objectid>`: only display objects with a specific object ID <objectid>. Regex matching of object IDs is supported
- `-ln`: list out all object names as an additional field in the output.
- `-li`: list out all object IDs as an additional field in the output.
- `-s`: display summary statistics (number of objects found, number of containers found, number of regions found) as an additional field in the output.

Examples:

```
$ ./pdc_ls pdc_tmp -n id.*
[INFO] File [pdc_tmp/metadata_checkpoint.0] last modified at: Fri Mar 11 14:19:13 2022

{
        "cont_id: 1000000":     [{
                        "obj_id":       1000007,
                        "app_name":     "VPICIO",
```

(continues on next page)

(continued from previous page)

```
                        "obj_name":     "id11",
                        "user_id":      0,
                        "tags": "ag0=1",
                        "data_type":    "PDC_INT",
                        "num_dims":     1,
                        "dims": [8388608],
                        "time_step":    0,
                        "region_list_info":     [{
                                        "storage_loc":  "/user/pdc_data/1000007/server0/
↪s0000.bin",
                                        "offset":       33554432,
                                        "num_dims":     1,
                                        "start":        [0],
                                        "count":        [8388608],
                                        "unit_size":    4,
                                        "data_loc_type":        "PDC_NONE"
                                }]
                }, {
                        "obj_id":       1000008,
                        "app_name":     "VPICIO",
                        "obj_name":     "id22",
                        "user_id":      0,
                        "tags": "ag0=1",
                        "data_type":    "PDC_INT",
                        "num_dims":     1,
                        "dims": [8388608],
                        "time_step":    0,
                        "region_list_info":     [{
                                        "storage_loc":  "/user/pdc_data/1000008/server0/
↪s0000.bin",
                                        "offset":       33554432,
                                        "num_dims":     1,
                                        "start":        [0],
                                        "count":        [8388608],
                                        "unit_size":    4,
                                        "data_loc_type":        "PDC_NONE"
                                }]
                }]
}
```

```
$ ./pdc_ls pdc_tmp -n obj-var-p.* -ln -li
[INFO] File [pdc_tmp/metadata_checkpoint.0] last modified at: Fri Mar 11 14:19:13 2022

{
        "cont_id: 1000000":     [{
                        "obj_id":       1000004,
                        "app_name":     "VPICIO",
                        "obj_name":     "obj-var-pxx",
                        "user_id":      0,
                        "tags": "ag0=1",
                        "data_type":    "PDC_FLOAT",
                        "num_dims":     1,
```

(continues on next page)

(continued from previous page)


```
                        "dims": [8388608],
                        "time_step":    0,
                        "region_list_info":     [{
                                        "storage_loc":  "/user/pdc_data/1000004/server0/
↪s0000.bin",
                                        "offset":       33554432,
                                        "num_dims":     1,
                                        "start":        [0],
                                        "count":        [8388608],
                                        "unit_size":    4,
                                        "data_loc_type":        "PDC_NONE"
                                }]
                }, {
                        "obj_id":       1000005,
                        "app_name":     "VPICIO",
                        "obj_name":     "obj-var-pyy",
                        "user_id":      0,
                        "tags": "ag0=1",
                        "data_type":    "PDC_FLOAT",
                        "num_dims":     1,
                        "dims": [8388608],
                        "time_step":    0,
                        "region_list_info":     [{
                                        "storage_loc":  "/user/pdc_data/1000005/server0/
↪s0000.bin",
                                        "offset":       33554432,
                                        "num_dims":     1,
                                        "start":        [0],
                                        "count":        [8388608],
                                        "unit_size":    4,
                                        "data_loc_type":        "PDC_NONE"
                                }]
                }, {
                        "obj_id":       1000006,
                        "app_name":     "VPICIO",
                        "obj_name":     "obj-var-pzz",
                        "user_id":      0,
                        "tags": "ag0=1",
                        "data_type":    "PDC_FLOAT",
                        "num_dims":     1,
                        "dims": [8388608],
                        "time_step":    0,
                        "region_list_info":     [{
                                        "storage_loc":  "/user/pdc_data/1000006/server0/
↪s0000.bin",
                                        "offset":       33554432,
                                        "num_dims":     1,
                                        "start":        [0],
                                        "count":        [8388608],
                                        "unit_size":    4,
                                        "data_loc_type":        "PDC_NONE"
                                }]
```


(continues on next page)

(continued from previous page)

```
                }],
        "all_obj_names":        ["obj-var-pxx", "obj-var-pyy", "obj-var-pzz"],
        "all_obj_ids":  [1000004, 1000005, 1000006]
}
```

6.2.2 pdc_import

Takes in file containing line separated paths to HDF5 files and converts those HDF5 files to a PDC checkpoint.

Usage: `./pdc_ls <file_list> <additional arguments>`:

Arguments:

- `-a <appname>`: Uses the specified <appname> as application name when creating PDC objects.
- `-o`: Specifies whether or not to overwrite pre-existing PDC objects when writing a PDC object that already exists.

Examples:

```
$ srun -N 1 -n 1 -c 2 --mem=25600 --cpu_bind=cores --gres=craynetwork:1 --overlap /path/
↪to/pdc_server.exe &
==PDC_SERVER[0]: using [./pdc_tmp/] as tmp dir, 1 OSTs, 1 OSTs per data file, 0% to BB
==PDC_SERVER[0]: using ofi+tcp
==PDC_SERVER[0]: without multi-thread!
==PDC_SERVER[0]: Read cache enabled!
==PDC_SERVER[0]: Successfully established connection to 0 other PDC servers
==PDC_SERVER[0]: Server ready!

$ srun -N 1 -n 1 -c 2 --mem=25600 --cpu_bind=cores --gres=craynetwork:1 --overlap ./pdc_
↪import file_names_list
==PDC_CLIENT: PDC_DEBUG set to 0!
==PDC_CLIENT[0]: Found 1 PDC Metadata servers, running with 1 PDC clients
==PDC_CLIENT: using ofi+tcp
==PDC_CLIENT[0]: Client lookup all servers at start time!
==PDC_CLIENT[0]: using [./pdc_tmp] as tmp dir, 1 clients per server
Running with 1 clients, 1 files
Importer 0: I will import 1 files
Importer 0: [../../test.h5]
Importer 0: processing [../../test.h5]
Importer 0: Created container [/]

==PDC_SERVER[0]: Checkpoint file [./pdc_tmp/metadata_checkpoint.0]
Import 8 datasets with 1 ranks took 0.93 seconds.
```

6.2.3 pdc_export

Converts PDC metadata checkpoint to a file of specified format. Currently only HDF5 is supported.

Usage: `./pdc_ls <checkpoint> <additional arguments>`:

Arguments:

- `-f <format>`: Uses the specified export <format>. Currently only supports HDF5 exports.

Examples:

```
$ srun -N 1 -n 1 -c 2 --mem=25600 --cpu_bind=cores --gres=craynetwork:1 --overlap /path/
↪to/pdc_server.exe &
==PDC_SERVER[0]: using [./pdc_tmp/] as tmp dir, 1 OSTs, 1 OSTs per data file, 0% to BB
==PDC_SERVER[0]: using ofi+tcp
==PDC_SERVER[0]: without multi-thread!
==PDC_SERVER[0]: Read cache enabled!
==PDC_SERVER[0]: Successfully established connection to 0 other PDC servers
==PDC_SERVER[0]: Server ready!

$ srun -N 1 -n 1 -c 2 --mem=25600 --cpu_bind=cores --gres=craynetwork:1 --overlap ./pdc_
↪export pdc_tmp
==PDC_CLIENT: PDC_DEBUG set to 0!
==PDC_CLIENT[0]: Found 1 PDC Metadata servers, running with 1 PDC clients
==PDC_CLIENT: using ofi+tcp
==PDC_CLIENT[0]: Client lookup all servers at start time!
==PDC_CLIENT[0]: using [./pdc_tmp] as tmp dir, 1 clients per server
[INFO] File [pdc_tmp/metadata_checkpoint.0] last modified at: Mon May  9 06:17:18 2022

POSIX read from file offset 117478480, region start = 0, region size = 8388608
POSIX read from file offset 130024208, region start = 0, region size = 8388608
POSIX read from file offset 130057104, region start = 0, region size = 8388608
POSIX read from file offset 130056720, region start = 0, region size = 8388608
POSIX read from file offset 130023696, region start = 0, region size = 8388608
POSIX read from file offset 130056592, region start = 0, region size = 8388608
POSIX read from file offset 130056720, region start = 0, region size = 8388608
POSIX read from file offset 130023696, region start = 0, region size = 8388608
```

Warning: PDC tools currently does not support compound data types and will have unexpected behavior when attempting to work with compound data types.

CHAPTER

SEVEN

DEVELOPER NOTES

7.1 How to implement an RPC?

This section covers how to implement a simple RPC from client to server. If you call an RPC on the client side, the server should be able to get the argument you passed from the client and execute the corresponding server RPC function.

A concrete example is `PDC_region_transfer_wait_all`. Mercury transfers at the client side are implemented in `pdc_client_connect.c`. The name of the function we are using in this example is `transfer_request_wait_all`. For each component mentioned next, replace `transfer_request_wait_all` with your function name. This section will not discuss the design of `transfer_request_wait_all` but rather point out where the Mercury components are and how they interact.

Firstly, in `pdc_client_connect.c`, search for `transfer_request_wait_all_register_id_g`. Create another variable by replacing `transfer_request_wait_all` with your function name. Secondly, search for `client_send_transfer_request_wait_all_rpc_cb`, and do the same text copy and replacement. This is the callback function on the client side when the RPC is finished on the server side. For most cases, this function loads the server return arguments to a structure and returns the values to the client RPC function. There is also some error checking. Then, search for `PDC_transfer_request_wait_all_register(*hg_class)` and `PDC_Client_transfer_request_wait_all`, and do text copy and replacement for both. This function is the entry point of the mercury RPC call. It contains argument loading, which has the variable name `in`' This RPC creates a mercury bulk transfer inside it. `HG_Create` and `HG_Bulk_create` are unnecessary if your mercury transfer does not transfer variable-sized data. `HG_Forward` has an argument `client_send_transfer_request_wait_all_rpc_cb`. The return values from the callback function are placed in `transfer_args`.

In file `pdc_client_connect.h`, search for `_pdc_transfer_request_wait_all_args`, do the text copy and replacement. This structure is the structure for returning values from client call back function `client_send_transfer_request_wait_all_rpc_cb` to client RPC function `PDC_Client_transfer_request_wait_all`. For most cases, an error code is sufficient. For other cases, like creating some object IDs, you must define the structure accordingly. Do not forget to load data in `_pdc_transfer_request_wait_all_args`. Search for `PDC_Client_transfer_request_wait_all`, and make sure you register your client connect entry function in the same way.

In file `pdc_server.c`, search for `PDC_transfer_request_wait_all_register(hg_class_g);`, make a copy, and replace the `transfer_request_wait_all` part with your function name (your function name has to be defined and used consistently throughout all these copy and replacement). In the file `pdc_client_server_common.h`, search for `typedef struct transfer_request_wait_all_in_t`. This is the structure used by a client passing its argument to the server side. You can define whatever you want that is fixed-sized inside this structure. If you have variable-sized data, it can be passed through mercury bulk transfer. The handle is `hg_bulk_t local_bulk_handle`. `typedef struct transfer_request_wait_all_out_t` is the return argument from the server to the client after the server RPC is finished. Next, search for `hg_proc_transfer_request_wait_all_in_t`. This function defines how arguments are transferred through mercury. Similarly, `hg_proc_transfer_request_wait_all_in_t` is the other way around. Next, search for `struct transfer_request_wait_all_local_bulk_args`. This structure is useful when a bulk transfer is used. Using this function, the server passes its variables from the RPC call to the bulk

transfer callback function. Finally, search for `PDC_transfer_request_wait_all_register`. For all these structures and functions, you should copy and replace `transfer_request_wait_all` with your own function name.

In file `pdc_client_server_common.c`, search for `PDC_FUNC_DECLARE_REGISTER(transfer_request_wait_all)` and `HG_TEST_THREAD_CB(transfer_request_wait_all)`, do text copy and function name replacement. `pdc_server_region_request_handler.h` is included directly in `pdc_client_server_common.c`. The server RPC of `transfer_request_wait_all` is implemented in `pdc_server_region_request_handler.h`. However, it is possible to put it directly in the `pdc_client_server_common.c`.

Let us open `pdc_server_region_request_handler.h`. Search for `HG_TEST_RPC_CB(transfer_request_wait_all, handle)`. This function is the entry point for the server RPC function call. `transfer_request_wait_all_in_t` contains the arguments you loaded previously from the client side. If you want to add more arguments, return to `pdc_client_server_common.h` and modify it correctly. `HG_Bulk_create` and `HG_Bulk_transfer` are the mercury bulk function calls. When the bulk transfer is finished, `transfer_request_wait_all_bulk_transfer_cb` is called.

After a walk-through of `transfer_request_wait_all`, you should have learned where different components of a mercury RPC should be placed and how they interact with each other. You can trace other RPC by searching their function names. If you miss things that are not optional, the program will likely hang there forever or run into segmentation faults.

7.2 PDC Server Metadata Overview

PDC metadata servers, a subset of PDC servers, store metadata for PDC classes such as objects and containers. PDC data server, also a subset of PDC servers (potentially overlapping with PDC metadata server), manages data from users. Such management includes server local caching and I/O to the file system. Both PDC metadata and data servers have some local metadata.

7.2.1 PDC Metadata Structure

PDC metadata is held in server memories. When servers are closed, metadata will be checkpointed into the local file system. Details about the checkpoint will be discussed in the metadata implementation section.

PDC metadata consists of three major parts at the moment:

- Metadata stored in the hash tables at the metadata server: stores persistent properties for PDC containers and PDC objects. When objects are created, these metadata are registered at the metadata server using mercury RPCs.
- Metadata query class at the metadata server: maps an object region to a data server, so clients can query for this information to access the corresponding data server. It is only used by dynamic region partition strategy
- Object regions stored at the data server: this includes file names and region chunking information inside the object file on the file system.

7.2.2 Metadata Operations at Client Side

In general, PDC object metadata is initialized when an object is created. The metadata stored at the metadata server is permanent. When clients create the objects, a PDC property is used as one of the arguments for the object creation function. Metadata for the object is set by using PDC property APIs. Most of the metadata are not subject to any changes. Currently, we support setting/getting object dimensions using object API.

7.3 PDC Metadata Management Strategy

This section discusses the metadata management approaches of PDC. First, we briefly summarize how PDC managed metadata in the past. Then, we propose new infrastructures for metadata management.

7.3.1 Managing Metadata and Data by the Same Server

Historically, a PDC server manages both metadata and data for objects it is responsible for. A client forwards I/O requests to the server computed based on MPI ranks statically. If a server is located on the same node as the client, the server will be chosen with a higher priority. This design can achieve high I/O parallelism if the I/O workloads from all clients are well-balanced. In addition, communication contention is minimized because servers are dedicated to serving disjoint subsets of clients.

However, this design has two potential drawbacks. The first disadvantage is supporting general I/O access. For clients served by different PDC servers, accessing overlapping regions is infeasible. Therefore, this design is specialized in applications with a non-overlapping I/O pattern. The second disadvantage is a lack of dynamic load-balancing mechanisms. For example, some applications use a subset of processes for processing I/O. A subset of servers may stay idle because the clients mapped to them are not sending I/O requests.

7.3.2 Separate Metadata Server from Data Server

Metadata service processes are required for distributed I/O applications with a one-sided communication design. When a client attempts to modify or access an object, metadata provides essential information such as object dimensions and the data server rank that contains the regions of interest. A PDC client generally does not have the runtime global metadata information. As a result, the first task is to obtain the essential metadata of the object from the correct metadata server.

Instead of managing metadata and data server together, we can separate the metadata management from the region I/O. A metadata server stores and manages all attributes related to a subset of PDC objects. A PDC server can be both a metadata and data server. However, the metadata and data can refer to different sets of objects.

This approach's main advantage is that the object regions' assignment to data servers becomes flexible. When an object is created, the name of the object maps to a unique metadata server. In our implementation, we adopt string hash values for object names and modulus operations to achieve this goal. The metadata information will be registered at the metadata server. Later, when other clients open the object, they can use the object's name to locate the same metadata server.

When a client accesses regions of an object, the metadata server informs the client of the corresponding data servers it should transfer its I/O requests. Metadata servers can map object regions to data servers in a few different methods.

7.3.3 Static Object Region Mappings

A metadata server can partition the object space evenly among all data servers. For high-dimensional objects, it is possible to define block partitioning methods similar to HDF5s's chunking strategies.

The static object region partitioning can theoretically achieve optimal parallel performance for applications with a balanced workload. In addition, static partitioning determines the mapping from object regions to data servers at object create/open time. No additional metadata management is required.

7.3.4 Dynamic Object Region Mappings

For applications that access a subset of regions for different objects, some data servers can stay idle while the rest are busy fetching or storing data for these regions concentrated around coordinates of interest. Dynamic object partitioning allows metadata servers to balance data server workloads in runtime. The mapping from object regions to the data server is determined at the time of starting region transfer request time. Partitioning object regions dynamically increases the complexity of metadata management. For example, a read from one client 0 after a write from another client 1 on overlapping regions demands metadata support. Client 0 has to locate the data server to which client 1 writes the region data using information from the metadata server. As a result, metadata servers must maintain up-to-date metadata of the objects they manage. There are a few options we can implement this feature.

Option 1: When a client attempts to modify object regions, the client can also send the metadata of this transfer request to the metadata server. Consequently, the metadata server serving for the modified objects always has the most up-to-date metadata.

Advantage: No need to perform communications between the servers (current strategy) Disadvantage: The metadata server can be a bottleneck because the number of clients accessing the server may scale up quickly.

Option 2: When a data server receives region transfer requests from any client, the data server forwards the corresponding metadata to the metadata server of the object.

Advantage: The number of servers is less than the number of clients, so we are reducing the chance of communication contention Disadvantage: Server-to-server RPC infrastructures need to be put in place.

Option 3: Similar to Option 2, but the data server will modify a metadata file. Later, a metadata server always checks the metadata file for metadata information updates.

Advantage: No communications are required if a metadata file is used. Disadvantage: Reading metadata files may take some time. If multiple servers are modifying the same metadata file, how should we proceed?

The following table summarizes the communication of the three mapping methods from clients to types of PDC servers when different PDC functions are called.

	Static Object Mapping	Dynamic Object Mapping & Static Region Mapping	Dynamic Object Mapping & Dynamic Region Mapping
`PDC_obj_create`	Client - Metadata Server	Client - Metadata Server	Client - Metadata Server
`PDC_obj_open`	Client - Metadata Server	Client - Metadata Server	Client - Metadata Server
`PDC_region_transfer_start`	Client - Data Server	Client - Data Server	Client - Data Server
`PDC_region_transfer_start`	Client - Data Server	Client - Data Server	Client - Metadata Server (Option 1)
`PDC_region_transfer_start`	Client - Data Server	Client - Data Server	Data Server - Metadata Server (Option 2)
`PDC_region_transfer_wait`	Data Server - Client (PDC_READ)	Data Server - Client (PDC_READ)	Data Server - Client (PDC_READ)

7.3.5 PDC Metadata Management Implementation

This section discusses how object metadata is implemented in the PDC production library. The following figure illustrates the flow of object metadata for different object operations. We label the 4 types of metadata in bold.

Create Metadata

Metadata for an object is created by using a PDC property. PDC property is created using client API `PDCprop_create(pdc_prop_type_t type, pdcid_t pdc_id)`. After a property instance is created, it is possible to set elements in this property using object property APIs. An alternative way is to use `pdcid_t PDCprop_obj_dup(pdcid_t prop_id)`, which copies all the existing entries in a property to a new object instance.

Binding Metadata to Object

Metadata is attached to an object at the object creation time. `PDCobj_create(pdcid_t cont_id, const char *obj_name, pdcid_t obj_prop_id)` is the prototype for binding an object property when an object is created.

Register Object Metadata at Metadata Server

Once an object is created locally at a client, the object metadata is sent to the corresponding metadata server based on the hash value computed from the object name. Internally, search for `typedef struct pdc_metadata_t {...} pdc_metadata_t;` in the `pdc_client_server_common.h` file. This data structure contains essential metadata about the object, such as its dimension and name.

Retrieve Metadata from Metadata Server

Object metadata can be obtained from the metadata server when clients open an object using the prototype `pdcid_t PDCobj_open(const char *obj_name, pdcid_t pdc)`. The client contacts the corresponding metadata server to retrieve data from the data type `pdc_metadata_t` stored at the server.

Object Metadata at Client

The current implementation stores metadata at the client in two separate places due to historical reasons. Both places can be accessed from the data type `struct _pdc_obj_info*`, which is a data type defined in `pdc_obj_pkg.h`.

We can generally use `struct _pdc_id_info *PDC_find_id(pdcid_t obj_id)` to locate the object info pointer `obj`. Then, `(struct _pdc_obj_info )(obj->obj_ptr)` allows use to obtain the `struct _pdc_obj_info` structure. We call this pointer `obj_info_ptr`. The first piece of local metadata, denoted as metadata buffer, is stored in `obj_info_ptr->metadata`. This value is a pointer that represents `pdc_metadata_t`. Its content matches the values stored at the metadata server side exactly. For object create, we copy the data from the pointer to the server memory using mercury RPCs. For object open, we copy from server memory to client memory.

The second piece of local metadata, denoted as object public property, is stored in `obj_info_ptr->obj_pt`, which has type `struct pdc_obj_prop` defined in the `pdc_prop.h` file. The values in this data type are copied from the first piece. This metadata data type contains essential information, such as object dims and region partition types.

Metadata at Data Server

Details about the data server will not be discussed in this section. In general, a data server takes inputs (both metadata and data for an object) from clients and processes them accordingly. It is not supposed to store metadata information for objects. However, it is responsible for storing the locations of data in the file system, including path and offset for regions.

If server cache is enabled, object dimension is stored by the server cache infrastructure when an object is registered for the first time. Object dimension is not used anywhere unless the I/O mode is set to be canonical file order storage. Currently, this mode does not allow clients to change object dimension, so it is not subject to metadata update, which is discussed in the following subsection.

Object Metadata Update

Object metadata is defined before creating an object. At the early stage of PDC, we did not plan to change any of the metadata after an object was created. However, it may be necessary to do this in the future. For example, sometimes applications want to change the sizes of PDC objects along different dimensions. An example is implemented as `perr_t PDCobj_set_dims(pdcid_t obj_id, int ndim, uint64_t *dims)`. This function can change object dimensions in runtime. As mentioned earlier, we need to update the metadata in three places. Two places are at the client side, and the other place is at the metadata server.

Object Region Metadata

Region metadata is required for dynamic region partitioning. Dynamic region partitioning strategy at the metadata server assigns data server IDs for regions in runtime. The file `pdc_server_region_transfer_metadata_query.c` implements the assignments of data server ID for individual regions. For dynamic region partition and local region partition strategies, a metadata server receives client region transfer requests. The metadata server returns a data server ID to the client so the client can send data to the corresponding data server. Details about how the client connects to the metadata server will be discussed in the implementation of the region transfer request.

Metadata Checkpoint

When PDC servers are closed, metadata stored by metadata servers is saved to the file system. Later, when users restart the servers, essential metadata are read back to the memory of the metadata server. In general, client applications should not be aware of any changes if servers are closed and restarted. This subsection layout the data format of PDC metadata when they are checkpointed.

Implementation of server checkpoint is in the function `PDC_Server_checkpoint`, and the corresponding restart is in the function `PDC_Server_restart(char *filename)`. The source file is `pdc_server.c`.

There are four categories of metadata to be checkpointed. One category is concatenated after another seamlessly. We demonstrate the first three categories of metadata in the following figures. Before each bracket, an integer value will indicate the number of repetitions for contents in the brackets. Contents after the bracket will start from the next byte after the last repetition for contents in the bracket. The last category is managed by an independent module `pdc_server_region_transfer_metadata_query.c`. The content of the metadata is subject to future changes.

Region metadata checkpoint is placed at the end of the server checkpoint file, right after the last byte of data server region. Function `transfer_request_metadata_query_checkpoint(char **checkpoint, uint64_t *checkpoint_size)` in `pdc_server_region_transfer_metadata_query.c` file handles the wrapping of region metadata.

7.3.6 Region Transfer Request at Client

This section describes how the region transfer request module in PDC works. The region transfer request module is the core of PDC I/O. From the client's point of view, some data is written to regions of objects through transfer request APIs. PDC region transfer request module arranges how data is transferred from clients to servers and how data is stored at servers.

PDC region: A PDC object abstracts a multi-dimensional array. The current implementation supports up to 3D. A PDC region can be used to access a subarray of the object. A PDC region describes the offsets and lengths to access a multi-dimensional array. Its prototype for creation is `PDCregion_create(psize_t ndims, uint64_t *offset, uint64_t *size)`. The input values to this create function will be copied into PDC internal memories, so it is safe to free the pointers later.

Region Transfer Request Create and Close

Region transfer request create function has prototype `PDCregion_transfer_create(void *buf, pdc_access_t access_type, pdcid_t obj_id, pdcid_t local_reg, pdcid_t remote_reg)`. The function takes a contiguous data buffer as input. Content in this data buffer will be stored in the region described by `remote_reg` for objects with `obj_id`. Therefore, `remote_reg` has to be contained in the dimension boundaries of the object. The transfer request create function copies the region information into the transfer request's memory, so it is safe to immediately close both `local_reg` and `remote_reg` after the create function is called.

`local_reg` describes the shape of the data buffer, aligning to the object's dimensions. For example, if "`local_reg` is a 1D region, the start index of the buf to be stored begins at the `offset[0]` of the `local_reg`, with a size of `size[0]`. Recall that `offset` and `size` are the input argument. If `local_reg` has dimensions larger than 1, then the shape of the data buffer is a subarray described by `local_reg` that aligns with the boundaries of object dimensions. In summary, `local_reg` is analogous to HDF5's memory space. `remote_reg` is parallel to HDF5's data space for data set I/O operations.

`PDCregion_transfer_close(pdcid_t transfer_request_id)` is used to clean up the internal memories associated with the "`transfer_request_id`.

Both create and close functions are local memory operations, so no mercury modules will be involved.

Region Transfer Request Start

Starting a region transfer request function will trigger the I/O operation. Data will be transferred from client to server using the `pdc_client_connect` module. `pdc_client_connect` module is a middleware layer that transfers client data to a designated server and triggers a corresponding RPC at the server side. In addition, the RPC transfer also allows data transfer by argument. Variables transferred by argument are fixed-sized. For variable-sized variables, mercury bulk transfer is used to transfer a contiguous memory buffer. Region transfer request start APIs: To transfer metadata and data with the pdc_client_connect module, the `region_transfer_request.c` file contains mechanisms to wrap request data into a contiguous buffer. There are two ways to start a transfer request. The first prototype is `PDCregion_transfer_start(pdcid_t transfer_request_id)`. This function starts a single transfer request specified by its ID. The second way is to use the aggregated prototype `PDCregion_transfer_start_all(pdcid_t *transfer_request_id, int size)`. This function can start multiple transfer requests. It is recommended to use the aggregated version when multiple requests can start together because it allows both client and server to aggregate the requests and achieve better performance.

For the 1D local region, `PDCregion_transfer_start` passes the pointer pointing to the `offset[0] * unit` location of the input buffer to the `pdc_client_connect` module. User data will be copied to a new contiguous buffer for higher dimensions using subregion copy based on local region shape. This implementation is in the static function `pack_region_buffer`. The new memory buffer will be passed to the `pdc_client_conenct` module.

This memory buffer passed to the pdc_client_connect module is registered with mercury bulk transfer. If it is a read operation, the bulk transfer is a pull operation. Otherwise, it is a push operation. Remote region information and some other relevant metadata are transferred using mercury RPC arguments. Once the pdc_client_connect module receives a return code and remote transfer request ID from the designated data server, PDCregion_transfer_start will cache the remote transfer request ID and exit.

PDCregion_transfer_start can be interpreted as PDCregion_transfer_start_all with the size argument set to 1, though the implementation is optimized. PDCregion_transfer_start_all performs aggregation of mercury bulk transfer whenever it is possible. Firstly, the function splits the read and write requests. Write requests are handled before the read requests. Wrapping region transfer requests to internal transfer packages: For each of the write requests, it is converted into one or more instances of the structure described by pdc_transfer_request_start_all_pkg defined in pdc_region_transfer.c. This structure contains the data buffer to be transferred, remote region shapes, and a data server rank to be transferred to. PDCregion_transfer_start_all implements the package translation in the static function prepare_start_all_requests.

As mentioned earlier in the metadata implementation, an abstract region for an object can be partitioned in different ways. There are four types of partitions: Object static partitioning, region static partitioning, region dynamic partitioning, and node-local region placement. PDCprop_set_obj_transfer_region_type(pdcid_t obj_prop, pdc_region_partition_t region_partition) allows users to set the partition method before creating an object on the client side. Different partitioning strategies have differences in the target data server rank when a transfer request is packed into pdc_transfer_request_start_all_pkg (s). We describe them separately.

For the object static partitioning strategy, the input transfer request is directly packed into pdc_transfer_request_start_all_pkg using a one-to-one mapping. The data server rank is determined at the object create/open time.

For dynamic region partitioning or node-local placement, the static function static perr_t register_metadata (in pdc_region_transfer.c) contacts the metadata server. The metadata server dynamically selects a data server for the input region transfer request based on the current system status. If local region placement is selected, metadata servers choose the data server on the same node (or as close as possible) of the client rank that transferred this request. If dynamic region partitioning is selected, the metadata server picks the data server currently holding the minimum number of bytes of data. The metadata server holds the region to data server mapping in its metadata region query system pdc_server_region_transfer_metadata_query.c. Metadata held by this module will be permanently stored in the file system as part of the metadata checkpoint file at the PDC server close time. After retrieving the correct data server ID, one pdc_transfer_request_start_all_pkg is created. The only difference in creating pdc_transfer_request_start_all_pkg compared with the object static partitioning strategy is how the data server ID is retrieved.

For the static region partitioning strategy, a region is equally partitioned across all data servers. As a result, one region transfer request generates the number of pdc_transfer_request_start_all_pkg equal to the total number of PDC servers. This implementation is in the static function static_region_partition in the pdc_region_transfer_request.c file.

Sending internal transfer request packages from client to server: For an aggregated region transfer request start all function call, two arrays of pdc_transfer_request_start_all_pkg are created as described in the previous subsection depending on the partitioning strategies. One is for PDC_WRITE, and the other is for PDC_READ. This section describes how pdc_region_transfer_request.c implements the communication from client to transfer. The core implementation is in the static function PDC_Client_start_all_requests.

Firstly, an array of pdc_transfer_request_start_all_pkg is sorted based on the target data server ID. Then, dor adjacent pdc_transfer_request_start_all_pkg that sends to the same data server ID, these packages are packed into a single contiguous memory buffer using the static function PDC_Client_pack_all_requests. This memory buffer is passed to the pdc_client_connect layer for mercury transfer.

Region transfer request wait: Region transfer request start does not guarantee the finish of data communication or I/O at the server by default. To make sure the input memory buffer is reusable or deletable, a wait function can be used. The wait function is also called implicitly when the object is closed, or special POSIX semantics is set ahead of time when the object is created.

Region Transfer Request Wait

Similar to the start case, the wait API has single and aggregated versions `PDCregion_transfer_start` and `PDCregion_transfer_start_all`. It is possible to wait for more than one request using the aggregated version.

The implementation of the wait all operation is similar to the implementation of the start all request. Firstly, packages defined by the structure `PDCregion_transfer_wait_all` are created. `PDCregion_transfer_wait_all` only contains the remote region transfer request ID and data server ID. These packages are sorted based on the data server ID. Region transfer requests to the same data server are packed into a contiguous buffer and sent through the PDC client connect module.

Region transfer request wait client control: As mentioned earlier, the region transfer request start all function packs many data packages into the same contiguous buffer, and passes this buffer to the PDC client connect layer for mercury transfer. This buffer can be shared by more than one region transfer request. This buffer can only be freed once wait operations are called on all these requests (not necessarily in a single wait operation call).

When a wait operation is called on a subset of these requests, we reduce the reference counter of the buffer. This reference counter is a pointer stored by the structure `pdc_transfer_request`. In terms of implementation, `pdc_transfer_request` stores an array of reference counter pointers and an array of data buffer pointers. Both arrays have the same size, forming a one-to-one mapping. Each of the data buffer pointers points to an aggregated memory buffer that this region transfer request packs some of its metadata/data into. When the aggregated buffer is created, the corresponding reference counter is set to be the number of region transfer requests that store the reference counter/data buffer pointers. As a result, when all of these region transfer requests have waited, the reference counter becomes zero, and the data buffer can be freed.

7.3.7 Region Transfer Request at Server

The region transfer request module at the server side is implemented in the `server/pdc_server_region` directory. This section describes how a data server is implemented at the server side.

Server Region Transfer Request RPC

At the PDC server side, `pdc_client_server_common.c` contains all the RPCs' entrances from client calls. `pdc_server_region_request_handler.h` contains all the RPCs' related to region transfer requests. The source code is directly included in the `pdc_client_server_common.c`. `HG_TEST_RPC_CB(transfer_request, handle)` and `HG_TEST_RPC_CB(transfer_request_all, handle)` are the server RPCs for region transfer request start and region transfer request start all functions called at the client side. `HG_TEST_RPC_CB(transfer_request_wait, handle)` and `HG_TEST_RPC_CB(transfer_request_wait_all, handle)` are the server RPCs for region transfer request wait and region transfer request wait all.

All functions containing `cb` at the end refer to the mercury bulk transfer callback functions. Mercury bulk transfer is used for transferring variable-sized data from client to server. The bulk transfer argument is passed through mercury RPC augment when server RPC is triggered. This argument is used by `HG_Bulk_create` and `HG_Bulk_transfer` to initiate data transfer from client to server. Once the transfer is finished, the mercury bulk transfer function triggers the call back function (one of the arguments passed to `HG_Bulk_transfer`) and processes the data received (or sent to the client).

Server Nonblocking Control

By design, the region transfer request start does not guarantee the finish of data transfer or server I/O. In fact, this function should return to the application as soon as possible. Data transfer and server I/O can occur in the background so that client applications can take advantage of overlapping timings between application computations and PDC data management.

Server Region Transfer Request Start

When server RPC for region transfer request start is triggered, it immediately starts the bulk transfer by calling the mercury bulk transfer functions. In addition, the region transfer request received by the data server triggers a register function `PDC_transfer_request_id_register` implemented `pdc_server_region_transfer.c`. This function returns a unique remote region transfer ID. This remote ID is returned to the client for future reference, so the wait operation can know which region transfer request should be finished on the data server side.

Then, `PDC_commit_request` is called for request registration. This operation pushes the metadata for the region transfer request to the end of the data server's linked list for temporary storage.

Finally, the server RPC returns a finished code to the client so that the client can return to the application immediately.

Server Region Transfer Request Wait

The request wait RPC on the server side receives a client's remote region transfer request ID. The RPC returns to the client when this request's local data server I/O is finished.

The implementation uses the `PDC_check_request` function in the `pdc_server_region_transfer.c` file. This function returns two possibilities. One possible return value is `PDC_TRANSFER_STATUS_COMPLETE`. In this case, the wait function can immediately return to the client. Another possibility is `PDC_TRANSFER_STATUS_PENDING`. This flag means that the local server I/O has not finished yet, so this RPC function will not return to the client. Instead, the mercury handle is binded to the structure `pdc_transfer_request_status` (defined in `pdc_server_region_transfer.h`) that stores the metadata of the region transfer request (search by its ID) within the function `PDC_check_request`.

When the region transfer request callback function for this region transfer is triggered, and the I/O operations are finished, the callback function calls `PDC_finish_request` to trigger the return of the wait mercury handle binded to the region transfer request. If a mercury handler is not found, `PDC_finish_request` sets the flag of `pdc_transfer_request_status` for the region transfer request to be `PDC_TRANSFER_STATUS_COMPLETE`, so a wait request called later can immediately return as described before. Server region transfer request aggregated mode: the server acquired a contiguous memory buffer through mercury bulk transfer for aggregated region transfer request start and wait calls. This contiguous memory buffer contains packed request metadata/data from the client side. These requests are parsed. For each of the requests, we process them one at a time. The processing method is described in the previous section.

7.3.8 Server Region Storage

PDC is a data management library. I/O is part of its service. Therefore, I/O operation is critical for data persistence. The function `PDC_Server_transfer_request_io` in the `pdc_server_region_transfer.c` file implements the core I/O function. There are two I/O modes for PDC. In general, one PDC object is stored in one file per data server.

Storage by File Offset

I/O by file only works for objects with fixed dimensions. Clients are not allowed to modify object dimensions by any means. When a region is written to an object, the region is translated into arrays of offsets and offset lengths based on the region shape using list I/O. Therefore, a region has fixed offsets to be placed on the file.

Storage by Region

I/O by region is a special feature of the PDC I/O management system. Writing a region to an object will append the region to the end of a file. If the same region is read back again sometime later, it only takes a single POSIX `lseek` and I/O operation to complete either write or read.

However, when a new region is written to an object, it is necessary to scan all the previously written regions to check for overlapping. The overlapping areas must be updated accordingly. If the new region is fully contained in any previously stored regions, it is unnecessary to append it to the end of the file.

I/O by region will store repeated bytes when write requests contain overlapping parts. In addition, the region update mechanism generates extra I/O operations. This is one of its disadvantages. Optimization for region search (as R trees) in the future can relieve this problem.

CHAPTER

EIGHT

CONTAINERS

Functions

static perr_t **PDC_cont_close**(struct _pdc_cont_info *cp)

perr_t **PDC_cont_init**()

container initialization

Returns

Non-negative on success/Negative on failure

pdcid_t **PDCcont_create**(const char *cont_name, pdcid_t cont_prop_id)

Create a container.

Parameters

- **cont_name** – [IN] Name of the container
- **cont_create_prop** – [IN] ID of container property returned by PDCprop_create(PDC_CONT_CREATE)

Returns

Container id on success/Zero on failure

pdcid_t **PDCcont_create_col**(const char *cont_name, pdcid_t cont_prop_id)

Create a container, used when all ranks are trying to create the same container.

Parameters

- **cont_name** – [IN] Name of the container
- **cont_prop_id** – [IN] ID of container property returned by PDCprop_create(PDC_CONT_CREATE)

Returns

Container id on success/Zero on failure

pdcid_t **PDC_cont_create_local**(pdcid_t pdc, const char *cont_name, uint64_t cont_meta_id)

Create a container locally.

Parameters

- **pdc** – [IN] PDC ID
- **cont_name** – [IN] Name of the container
- **cont_meta_id** – [out] Metadata id of container

Returns

Container id on success/Zero on failure

perr_t **PDC_cont_list_null**()

Check if container list is empty.

Parameters
pdc_id – [IN] ID of the PDC

Returns
SUCCEED if empty/FAIL if not empty

perr_t **PDCcont_close**(pdcid_t id)

Close a container.

Parameters
cont_id – [IN] Container id, returned by PDCcont_open(pdcid_t pdc_id, const char *cont_name)

Returns
Non-negative on success/Negative on failure

perr_t **PDC_cont_end**()

PDC container finalize.

Returns
Non-negative on success/Negative on failure

pdcid_t **PDCcont_open**(const char *cont_name, pdcid_t pdc)

Open a container.

Parameters

- **cont_name** – [IN] Name of the container
- **pdc_id** – [IN] ID of pdc

Returns
Container id on success/Zero on failure

pdcid_t **PDCcont_open_col**(const char *cont_name, pdcid_t pdc)

Open a container collectively.

Parameters

- **cont_name** – [IN] Name of the container
- **pdc_id** – [IN] ID of pdc

Returns
Container id on success/Zero on failure

struct _pdc_cont_info ***PDC_cont_get_info**(pdcid_t cont_id)

Return a container property.

Parameters
cont_name – [IN] Name of the container

Returns
Container struct on success/NULL on failure

struct pdc_cont_info ***PDCcont_get_info**(const char *cont_name)

Return a container property.

Parameters
cont_name – [IN] Name of the container

Returns
Container struct on success/NULL on failure

cont_handle *PDCcont_iter_start()

Iterate over containers within a PDC.

Returns
Pointer to cont_handle struct/NULL on failure

pbool_t PDCcont_iter_null(cont_handle *chandle)

Check if container handle is pointing to NULL.

Parameters
chandle – [IN] Pointer to cont_handle struct, returned by PDCcont_iter_start(pdcid_t pdc_id)

Returns
1 on success/0 on failure

cont_handle *PDCcont_iter_next(cont_handle *chandle)

Move to the next container within a PDC.

Parameters
chandle – [IN] Pointer to cont_handle struct, returned by PDCcont_iter_start(pdcid_t pdc_id)

Returns
Pointer to cont_handle struct/NULL on failure

struct pdc_cont_info *PDCcont_iter_get_info(cont_handle *chandle)

Retrieve container information.

Parameters
chandle – [IN] A cont_handle struct, returned by PDCcont_iter_start(pdcid_t pdc_id)

Returns
Pointer to a PDC_cont_info struct/NULL on failure

perr_t PDCcont_persist(pdcid_t cont_id)

Persist a transient container.

Parameters
cont_id – [IN] ID of the container, returned by PDCcont_open(pdcid_t pdc_id, const char *cont_name)

Returns
Non-negative on success/Negative on failure

perr_t PDCprop_set_cont_lifetime(pdcid_t cont_prop, pdc_lifetime_t cont_lifetime)

Set container lifetime.

Parameters

- **cont_create_prop** – [IN] ID of container property, returned by PDCprop_create(PDC_CONT_CREATE)
- **cont_lifetime** – [IN] container lifetime (enum type), PDC_PERSIST or PDC_TRANSIENT

Returns
Non-negative on success/Negative on failure

CHAPTER

NINE

OBJECTS

Functions

static perr_t **PDC_obj_close**(struct _pdc_obj_info *op)

perr_t **PDC_obj_init**()

PDC object initialization.

Returns

Non-negative on success/Negative on failure

pdcid_t **PDCobj_create**(pdcid_t cont_id, const char *obj_name, pdcid_t obj_prop_id)

Create an object.

Parameters

- **cont_id** – [IN] ID of the container
- **obj_name** – [IN] Name of the object
- **obj_create_prop** – [IN] ID of object property, returned by PDCprop_create(PDC_OBJ_CREATE)

Returns

Object id on success/Zero on failure

static perr_t **PDC_Client_attach_metadata_to_local_obj**(const char *obj_name, uint64_t obj_id, uint64_t cont_id, uint32_t data_server_id, pdc_region_partition_t region_partition, pdc_consistency_t consistency, struct _pdc_obj_info *obj_info)

Copy the metadata to local object property, internal function.

Parameters

- **obj_name[IN]** – Object name
- **obj_id[IN]** – Object ID (global, obtained from metadata server)
- **cont_id[IN]** – Container id (global, obtained from metadata server)
- **dataserver_id[IN]** – Data server id (global, obtained from metadata server)
- **region_partition[IN]** – region_partition for the object
- **consistency[IN]** – consistency for the object
- **obj_info[IN]** – Object property

Returns
Non-negative on success/Negative on failure

pdcid_t **PDC_obj_create**(pdcid_t cont_id, const char *obj_name, pdcid_t obj_prop_id, _pdc_obj_location_t location)

Create an object.

Parameters

- **cont_id** – [IN] ID of the container
- **obj_name** – [IN] Name of the object
- **obj_create_prop** – [IN] ID of object property, returned by PDCprop_create(PDC_OBJ_CREATE)
- **location** – [IN] PDC_OBJ_GLOBAL/PDC_OBJ_LOCAL

Returns
Object id on success/Negative on failure

perr_t **PDC_obj_list_null**()

Check if object list is empty.

Returns
SUCCEED if empty/FAIL if not empty

perr_t **PDCobj_flush_start**(pdcid_t obj_id)

Force write-back of the object from its cache.

This function does not guarantee the finish of flushing. Only useful when server cache is enabled.

Parameters
obj_id – [IN] ID of the object

Returns
Non-negative on success/Negative on failure

perr_t **PDCobj_flush_all_start**()

Force write-back of all objects from its cache.

This function does not guarantee the finish of flushing. Only useful when server cache is enabled.

Parameters
obj_id – [IN] ID of the object

Returns
Non-negative on success/Negative on failure

perr_t PDCobj_flush_start (pdcid_t obj_id __attribute__((unused)))

perr_t **PDCobj_close**(pdcid_t obj_id)

Close an object.

Parameters
obj_id – [IN] ID of the object

Returns
Non-negative on success/Negative on failure

perr_t **PDC_obj_end**()

PDC object finalize.

Returns
Non-negative on success/Negative on failure

static pdcid_t **PDCobj_open_common**(const char *obj_name, pdcid_t pdc, int is_col)

pdcid_t **PDCobj_open**(const char *obj_name, pdcid_t pdc)

Open an object within a container.

Parameters

- **pdc_id** – [IN] ID of pdc
- **obj_name** – [IN] Name of the object

Returns
Object id on success/Zero on failure

pdcid_t **PDCobj_open_col**(const char *obj_name, pdcid_t pdc)

Open an object within a container collectively.

Parameters

- **pdc_id** – [IN] ID of pdc
- **obj_name** – [IN] Name of the object

Returns
Object id on success/Zero on failure

obj_handle ***PDCobj_iter_start**(pdcid_t cont_id)

Iterate over objects in a container.

Parameters
cont_id – [IN] Container ID, returned by PDCobj_open(pdcid_t pdc_id, const char *cont_name)

Returns
A pointer to object handle struct on success/NULL on failure

pbool_t **PDCobj_iter_null**(obj_handle *ohandle)

Check if object handle is pointing to NULL.

Parameters
ohandle – [IN] A obj_handle struct, returned by PDCobj_iter_start(pdcid_t cont_id)

Returns
1 on success/0 on failure

obj_handle ***PDCobj_iter_next**(obj_handle *ohandle, pdcid_t cont_id)

Iterate the next object.

Parameters

- **ohandle** – [IN] A obj_handle struct, returned by PDCobj_iter_start(pdcid_t cont_id)
- **cont_id** – [IN] ID of the container

Returns
A pointer to object handle struct on success/Zero on failure

struct pdc_obj_info *PDCobj_iter_get_info(obj_handle *ohandle)

Get object information.

Parameters
ohandle – [IN] A pointer to obj_handle struct, returned by PDCobj_iter_start(pdcid_t cont_id)

Returns
Pointer to a pdc_obj_info struct on success/NULL on failure

perr_t PDCprop_set_obj_user_id(pdcid_t obj_prop, uint32_t user_id)

Set object user ID.

Parameters

- **obj_prop** – [IN] ID of object property, returned by PDCprop_create(PDC_OBJ_CREATE)
- **user_id** – [IN] User id

Returns
Non-negative on success/Negative on failure

perr_t PDCprop_set_obj_app_name(pdcid_t obj_prop, char *app_name)

Set object application name.

Parameters

- **obj_prop** – [IN] ID of object property, returned by PDCprop_create(PDC_OBJ_CREATE)
- **app_name** – [IN] Application name

Returns
Non-negative on success/Negative on failure

perr_t PDCprop_set_obj_time_step(pdcid_t obj_prop, uint32_t time_step)

Set object time step.

Parameters

- **obj_prop** – [IN] ID of object property, returned by PDCprop_create(PDC_OBJ_CREATE)
- **time_step** – [IN] Time step

Returns
Non-negative on success/Negative on failure

perr_t PDCprop_set_obj_data_loc(pdcid_t obj_prop, char *loc)

Set object data location.

Parameters

- **obj_prop** – [IN] ID of object property, returned by PDCprop_create(PDC_OBJ_CREATE)
- **app_name** – [IN] Data location path

Returns
Non-negative on success/Negative on failure

perr_t PDCprop_set_obj_tags(pdcid_t obj_prop, char *tags)

Set object tag.

Parameters

- **obj_prop** – [IN] ID of object property, returned by PDCprop_create(PDC_OBJ_CREATE)
- **tags** – [IN] Tags

Returns
Non-negative on success/Negative on failure

perr_t **PDCprop_set_obj_dims**(pdcid_t obj_prop, PDC_int_t ndim, uint64_t *dims)
Set object dimension.

Parameters

- **obj_prop** – [IN] ID of object property, returned by PDCprop_create(PDC_OBJ_CREATE)
- **ndim** – [IN] Number of dimensions
- **dims** – [IN] Size of each dimension

Returns
Non-negative on success/Negative on failure

perr_t **PDCprop_set_obj_type**(pdcid_t obj_prop, pdc_var_type_t type)
Set object type.

Parameters

- **obj_prop** – [IN] ID of object property, returned by PDCprop_create(PDC_OBJ_CREATE)
- **type** – [IN] Object variable type (enum type), i.e. PDC_int_t, PDC_float_t

Returns
Non-negative on success/Negative on failure

perr_t **PDCprop_set_obj_transfer_region_type**(pdcid_t obj_prop, pdc_region_partition_t region_partition)
Set object transfer partitioning.

Parameters

- **obj_prop** – [IN] ID of object property, returned by PDCprop_create(PDC_OBJ_CREATE)
- **type** – [IN] Object transfer partitioning method (enum type), i.e. PDC_OBJ_STATIC, PDC_REGION_STATIC, PDC_REGION_DYNAMIC

Returns
Non-negative on success/Negative on failure

perr_t **PDCprop_set_obj_consistency_semantics**(pdcid_t obj_prop, pdc_consistency_t consistency)
Set object consistency semantics.

Parameters

- **obj_prop** – [IN] ID of object property, returned by PDCprop_create(PDC_OBJ_CREATE)
- **consistency** – [IN] Consistency semantics required e.g., PDC_CONSISTENCY_DEFAULT, PDC_CONSISTENCY_POSIX, etc

Returns
Non-negative on success/Negative on failure

perr_t **PDCprop_set_obj_buf**(pdcid_t obj_prop, void *buf)
Set an object buffer.

Parameters

- **obj_prop** – [IN] ID of object property, returned by PDCprop_create(PDC_OBJ_CREATE)
- **buf** – [IN] Starting point of object storage

Returns
Non-negative on success/Negative on failure

perr_t **PDCobj_set_dims**(pdcid_t obj_id, int ndim, uint64_t *dims)

Reset obj dimension.

Parameters

- **obj_id** – [IN] ID of object,
- **ndim** – [IN] number of dimensions, this one must match existing record.
- **dims** – [IN] new dimensions to be set

Returns

Non-negative on success/Negative on failure

perr_t **PDCobj_get_dims**(pdcid_t obj_id, int *ndim, uint64_t **dims)

Get obj dimension.

Parameters

- **obj_id** – [IN] ID of object,
- **ndim** – [OUT] number of dimensions, this one must match existing record.
- **dims** – [OUT] new dimensions to be set

Returns

Non-negative on success/Negative on failure

void ****PDCobj_buf_retrieve**(pdcid_t obj_id)

get obj dimension

Parameters

- **obj_id** – [IN] ID of object,
- **ndim** – [IN] number of object dimensions
- **dims** – [IN] object dimensions, in a newly malloced buffer.

Returns

Non-negative on success/Negative on failure

struct _pdc_obj_info ***PDC_obj_get_info**(pdcid_t obj_id)

Get object information.

Parameters

obj_id – [IN] ID of the object

Returns

Pointer to _pdc_obj_info struct on success/Null on failure

perr_t **PDC_free_obj_info**(struct _pdc_obj_info *obj)

Free object information.

Parameters

obj_id – [IN] ID of the object

Returns

Non-negative on success/Negative on failure

struct pdc_obj_info ***PDCobj_get_info**(pdcid_t obj_id)

Get object information.

Parameters

obj_name – [IN] Name of the object

Returns
Pointer to pdc_obj_info struct on success/Null on failure

perr_t **PDCobj_del**(pdcid_t obj_id)

Parameters
obj_id – [IN] Object ID

Returns
Non-negative on success/Negative on failure

Functions

pdcid_t **PDCobj_create_mpi**(pdcid_t cont_id, const char *obj_name, pdcid_t obj_prop_id, int rank_id, MPI_Comm comm)

Create an object.

Parameters

- **cont_id** – [IN] ID of the container
- **obj_name** – [IN] Name of the object
- **obj_create_prop** – [IN] ID of object property, returned by PDCprop_create(PDC_OBJ_CREATE)
- **rank_id** – [IN] MPI process rank

Returns
Object ID on success/Negative on failure

perr_t **PDCobj_encode**(pdcid_t obj_id, pdcid_t *meta_id)

pdcid_t **PDCobj_decode**(pdcid_t obj_id, pdcid_t meta_id)

CHAPTER

TEN

PREPERTIES

Functions

static perr_t **pdc_prop_cont_close**(struct _pdc_cont_prop *cp)

static perr_t **pdc_prop_obj_close**(struct _pdc_obj_prop *cp)

perr_t **PDC_prop_init**()

PDC container and object property initialization.

Returns
Non-negative on success/Negative on failure

pdcid_t **PDCprop_create**(pdc_prop_type_t type, pdcid_t pdcid)

Create PDC property.

Parameters

- **type** – [IN] PDC property creation type (enum type), PDC_CONT_CREATE or PDC_OBJ_CREATE
- **id** – [IN] ID of the PDC

Returns
PDC property id on success/Zero on failure

pdcid_t **PDCprop_obj_dup**(pdcid_t prop_id)

Quickly create object property from an existing object property Share the same property with the existing one other than data_loc, data type, buf, which are data related.

Parameters

- **type** – [IN] PDC property creation type (enum type), PDC_CONT_CREATE or PDC_OBJ_CREATE
- **id** – [IN] ID of the PDC

Returns
PDC property id on success/Zero on failure

perr_t **PDC_prop_cont_list_null**()

Check if container property list is empty.

Returns
SUCCEED if empty/FAIL if not empty

perr_t **PDC_prop_obj_list_null**()

Check if object property list is empty.

Returns
SUCCEED if empty/FAIL if not empty

perr_t **PDCprop_close**(pdcid_t id)

Close property.

Parameters
id – [IN] ID of the property

Returns
Non-negative on success/Negative on failure

perr_t **PDC_prop_end**()

PDC container and object property finalize.

Returns
Non-negative on success/Negative on failure

struct _pdc_cont_prop ***PDCcont_prop_get_info**(pdcid_t cont_prop)

Get container property infomation.

Parameters
prop_id – [IN] ID of the property

Returns
Pointer to _pdc_cont_prop struct/Null on failure

struct pdc_obj_prop ***PDCobj_prop_get_info**(pdcid_t obj_prop)

Get object property infomation.

Parameters
prop_id – [IN] ID of the object property

Returns
Pointer to pdc_obj_prop struct/Null on failure

struct _pdc_obj_prop ***PDC_obj_prop_get_info**(pdcid_t obj_prop)

Get object property infomation.

Parameters
prop_id – [IN] ID of the object property

Returns
Pointer to _pdc_obj_prop struct/Null on failure

perr_t **PDC_obj_prop_free**(struct _pdc_obj_prop *cp)

Parameters
cp – [IN] Object property

Returns
Non-negative on success/Negative on failure

CHAPTER

ELEVEN

REGIONS

Functions

static perr_t **pdc_region_close**(struct pdc_region_info *op)

static perr_t **pdc_transfer_request_close**()

perr_t **PDC_region_init**()

PDC region initialization.

Returns
Non-negative on success/Negative on failure

perr_t **PDC_transfer_request_init**()

PDC region transfer instance initialization.

Returns
Non-negative on success/Negative on failure

perr_t **PDC_region_list_null**()

Check if region list is empty.

Returns
SUCCEED if empty/FAIL if not empty

perr_t **PDCregion_close**(pdcid_t region_id)

Close a region.

Parameters
region_id – [IN] ID of the object

Returns
Non-negative on success/Negative on failure

perr_t **PDC_region_end**()

PDC region finalize.

Returns
Non-negative on success/Negative on failure

pdcid_t **PDCregion_create**(psize_t ndims, uint64_t *offset, uint64_t *size)

Create a region.

Parameters

- **ndims** – [IN] Number of dimensions
- **offset** – [IN] Offset of each dimension

- **size** – [IN] Size of each dimension

Returns
Object id on success/Zero on failure

perr_t **PDCbuf_obj_map**(void *buf, pdc_var_type_t local_type, pdcid_t local_reg, pdcid_t remote_obj, pdcid_t remote_reg)

Map an application buffer to an object.

Parameters

- **buf** – [IN] Start point of an application buffer
- **local_type** – [IN] Data type of data in memory
- **local_reg** – [IN] ID of the source region
- **remote_obj** – [IN] ID of the target object
- **remote_reg** – [IN] ID of the target region

Returns
Non-negative on success/Negative on failure

struct pdc_region_info ***PDCregion_get_info**(pdcid_t reg_id)

Get region information.

Parameters

- **reg_id** – [IN] ID of the region
- **obj_id** – [IN] ID of the object

Returns
Pointer to pdc_region_info struct on success/Null on failure

perr_t **PDCbuf_obj_unmap**(pdcid_t remote_obj_id, pdcid_t remote_reg_id)

Unmap all regions within the object from a buffer (write unmap)

Parameters

- **remote_obj_id** – [IN] ID of the target object
- **remote_reg_id** – [IN] ID of the target region

Returns
Non-negative on success/Negative on failure

perr_t **PDCreg_obtain_lock**(pdcid_t obj_id, pdcid_t reg_id, pdc_access_t access_type, pdc_lock_mode_t lock_mode)

Obtain the region lock.

Parameters

- **obj_id** – [IN] ID of the object
- **reg_id** – [IN] ID of the region
- **access_type** – [IN] Region access type: READ or WRITE
- **lock_mode** – [IN] Lock mode of the region: BLOCK or NOBLOCK

Returns
Non-negative on success/Negative on failure

perr_t **PDCreg_release_lock**(pdcid_t obj_id, pdcid_t reg_id, pdc_access_t access_type)

Release the region lock.

Parameters

- **obj_id** – [IN] ID of the object
- **reg_id** – [IN] ID of the region
- **access_type** – [IN] Region access type

Returns

Non-negative on success/Negative on failure

Typedefs

typedef struct *pdc_transfer_request* **pdc_transfer_request**

typedef struct *pdc_transfer_request_start_all_pkg* **pdc_transfer_request_start_all_pkg**

typedef struct *pdc_transfer_request_wait_all_pkg* **pdc_transfer_request_wait_all_pkg**

Functions

static int **sort_by_data_server_start_all**(const void *elem1, const void *elem2)

static int **sort_by_metadata_server_start_all**(const void *elem1, const void *elem2)

static int **sort_by_data_server_wait_all**(const void *elem1, const void *elem2)

pdcid_t **PDCregion_transfer_create**(void *buf, pdc_access_t access_type, pdcid_t obj_id, pdcid_t local_reg, pdcid_t remote_reg)

perr_t **PDCregion_transfer_close**(pdcid_t transfer_request_id)

static perr_t **attach_local_transfer_request**(struct _pdc_obj_info *p, pdcid_t transfer_request_id)

static perr_t **remove_local_transfer_request**(struct _pdc_obj_info *p, pdcid_t transfer_request_id)

static perr_t **static_region_partition**(char *buf, int ndim, uint64_t unit, pdc_access_t access_type, uint64_t *obj_dims, uint64_t *offset, uint64_t *size, int set_output_buf, int *n_data_servers, uint32_t **data_server_ids, uint64_t ***sub_offsets, uint64_t ***output_offsets, uint64_t ***output_sizes, char ***output_buf)

static perr_t **pack_region_buffer**(char *buf, uint64_t *obj_dims, size_t total_data_size, int local_ndim, uint64_t *local_offset, uint64_t *local_size, size_t unit, pdc_access_t access_type, char **new_buf)

static perr_t **set_obj_server_bufs**(*pdc_transfer_request* *transfer_request)

static perr_t **pack_region_metadata_query**(*pdc_transfer_request_start_all_pkg* **transfer_request, int size, char **buf_ptr, uint64_t *total_buf_size_ptr)

static perr_t **unpack_region_metadata_query**(char *buf, *pdc_transfer_request_start_all_pkg* **transfer_request_input, *pdc_transfer_request_start_all_pkg* **transfer_request_head_ptr, *pdc_transfer_request_start_all_pkg* **transfer_request_end_ptr, int *size_ptr)

static perr_t **register_metadata**(*pdc_transfer_request_start_all_pkg* **transfer_request_input, int input_size, uint8_t is_write, *pdc_transfer_request_start_all_pkg* ***transfer_request_output_ptr, int *output_size_ptr)

static int **prepare_start_all_requests**(pdcid_t *transfer_request_id, int size, *pdc_transfer_request_start_all_pkg* ***write_transfer_request_ptr, *pdc_transfer_request_start_all_pkg* ***read_transfer_request_ptr, int *write_size_ptr, int *read_size_ptr, pdcid_t **posix_transfer_request_id_ptr, int *posix_size_ptr)

This function prepares lists of read and write requests separately for start_all function.

The lists are sorted in terms of data_server_id. We pack data from user buffer to contiguous buffers. Static partitioning requires having at most n_data_servers number of contiguous regions.

static int **finish_start_all_requests**(*pdc_transfer_request_start_all_pkg* **write_transfer_request, *pdc_transfer_request_start_all_pkg* **read_transfer_request, int write_size, int read_size)

static perr_t **PDC_Client_pack_all_requests**(int n_objs, *pdc_transfer_request_start_all_pkg* **transfer_requests, pdc_access_t access_type, char **bulk_buf_ptr, size_t *total_buf_size_ptr, char **read_bulk_buf)

static perr_t **PDC_Client_start_all_requests**(*pdc_transfer_request_start_all_pkg* **transfer_requests, int size)

perr_t **PDCregion_transfer_start_all**(pdcid_t *transfer_request_id, int size)

static int **sorted_array_unions**(const int **array, const int *input_size, int n_arrays, int **out_array, int *out_size)

Input: Sorted arrays Output: A single array that is sorted, and the union of sorted arrays.

perr_t **PDCregion_transfer_start**(pdcid_t transfer_request_id)

Start a region transfer from local region to remote region for an object on buf.

Parameters

- **buf** – [IN] Start point of an application buffer
- **obj_id** – [IN] ID of the target object
- **data_type** – [IN] Data type of data in memory
- **local_reg** – [IN] ID of the source region
- **remote_reg** – [IN] ID of the target region

Returns

Non-negative on success/Negative on failure

static perr_t **release_region_buffer**(char *buf, uint64_t *obj_dims, int local_ndim, uint64_t *local_offset, uint64_t *local_size, size_t unit, pdc_access_t access_type, int bulk_buf_size, char *new_buf, char **bulk_buf, int **bulk_buf_ref, char **read_bulk_buf)

perr_t **PDCregion_transfer_status**(pdcid_t transfer_request_id, pdc_transfer_status_t *completed)

perr_t **PDCregion_transfer_wait_all**(pdcid_t *transfer_request_id, int size)

perr_t **PDCregion_transfer_wait**(pdcid_t transfer_request_id)

struct **pdc_transfer_request**

Public Members

pdcid_t **obj_id**

uint32_t **data_server_id**

uint32_t **metadata_server_id**

uint64_t ***metadata_id**

pdc_access_t **access_type**

pdc_var_type_t **mem_type**

size_t **unit**

char ***buf**

char ****read_bulk_buf**

char ***new_buf**

char ****bulk_buf**

int ****bulk_buf_ref**

pdc_region_partition_t **region_partition**

pdc_consistency_t **consistency**

int **n_obj_servers**

uint32_t ***obj_servers**

uint64_t ****output_offsets**

uint64_t **sub_offsets

uint64_t **output_sizes

char **output_buf

int local_region_ndim

uint64_t *local_region_offset

uint64_t *local_region_size

int remote_region_ndim

uint64_t *remote_region_offset

uint64_t *remote_region_size

uint64_t total_data_size

int obj_ndim

uint64_t *obj_dims

struct _pdc_obj_info *obj_pointer

struct **pdc_transfer_request_start_all_pkg**

Public Members

uint32_t data_server_id

pdc_transfer_request *transfer_request

uint64_t *remote_offset

uint64_t *remote_size

int index

char *buf

struct *pdc_transfer_request_start_all_pkg* ***next**

struct **pdc_transfer_request_wait_all_pkg**

Public Members

uint64_t **metadata_id**

uint32_t **data_server_id**

int **index**

pdc_transfer_request ***transfer_request**

struct *pdc_transfer_request_wait_all_pkg* ***next**

CHAPTER

TWELVE

QUERY

Functions

pdc_query_t *PDCquery_create(pdcid_t obj_id, pdc_query_op_t op, pdc_var_type_t type, void *value)

Create a PDC query.

Parameters

- **obj_id** – [IN] *********
- **op** – [IN] *********
- **type** – [OUT] *********
- **value** – [OUT] *********

Returns

****

perr_t PDCquery_sel_region(pdc_query_t *query, struct pdc_region_info *obj_region)

Parameters

- **query** – [IN] *********
- **obj_region** – [IN] Object region information

Returns

Non-negative on success/Negative on failure

pdc_query_t *PDCquery_and(pdc_query_t *q1, pdc_query_t *q2)

Parameters

- **query1** – [IN] *********
- **query2** – [IN] *********

Returns

******

pdc_query_t *PDCquery_or(pdc_query_t *q1, pdc_query_t *q2)

Parameters

- **query1** – [IN] *********
- **query2** – [IN] *********

Returns
******

perr_t **PDCquery_get_nhits**(pdc_query_t *query, uint64_t *n)

Parameters

- **query** – [IN] *********
- **n** – [IN] *********

Returns
Non-negative on success/Negative on failure

perr_t **PDCquery_get_selection**(pdc_query_t *query, pdc_selection_t *sel)

Parameters

- **query** – [IN] *********
- **sel** – [IN] *********

Returns
Non-negative on success/Negative on failure

perr_t **PDCquery_get_data**(pdcid_t obj_id, pdc_selection_t *sel, void *obj_data)

Parameters

- **obj_id** – [IN] Object ID
- **sel** – [IN] *********
- **obj_data** – [IN] ********

Returns
Non-negative on success/Negative on failure

perr_t **PDCquery_get_histogram**(pdcid_t obj_id)

Parameters
obj_id – [IN] Object ID

Returns
Non-negative on success/Negative on failure

CHAPTER

THIRTEEN

ANALYSIS

Functions

void *PDC_Server_get_region_data_ptr(pdcid_t object_id)

static int iterator_init(pdcid_t objectId, pdcid_t reg_id, int blocks, struct _pdc_iterator_info *iter)

pdcid_t PDCobj_data_block_iterator_create(pdcid_t obj_id, pdcid_t reg_id, int contig_blocks)

Create a PDC block iterator (general form)

Parameters

- **obj_id** – [IN] PDC object ID
- **reg_id** – [IN] PDC region ID
- **contig_blocks** – [IN] How many rows or columns the iterator should return

Returns

Iterator id on success/Zero on failure

pdcid_t PDCobj_data_iter_create(pdcid_t obj_id, pdcid_t reg_id)

Create a PDC iterator (basic form which internally calls the block version with contig_blocks = 1)

Parameters

- **obj_id** – [IN] PDC object ID
- **reg_id** – [IN] PDC region ID

Returns

Iterator id on success/Zero on failure

char *PDC_get_argv0_()

Returns

****

char *PDC_get_realpath(char *fname, char *app_path)

Parameters

- **fname** – [IN] Path of working directory
- **app_path** – [IN] Name of the application

Returns

****

perr_t **PDCobj_analysis_register**(char *func, pdcid_t iterIn, pdcid_t iterOut)

Register a function to be invoked at registration time and/or when data buffers referenced by the input iterator is modified.

Parameters

- **func** – [IN] String containing the [libraryname:]function to be registered. (default library name = "libpdcanalysis")
- **iterIn** – [IN] PDC iterator id containing the input data (may be a NULL iterator == 0).
- **dims** – [IN] PDC iterator id containing the output data (may be a NULL iterator == 0).

Returns

Non-negative on success/Negative on failure NOTES: In the use case where null iterators are supplied, the function will always be invoked on the data server. Because there are no data references, the supplied function will only be called at registration time. This can be used to invoke special functions such as the reading of data from files (HDF5, etc.) or to start monitoring or timing functions. One could for example provide an API to start or stop profilers, take data snapshots, etc.

size_t **PDCobj_data_getSliceCount**(pdcid_t iter)

Parameters

iter – [IN] The number iteration

Returns

****

size_t **PDCobj_data_getNextBlock**(pdcid_t iter, void **nextBlock, size_t *dims)

Use a PDC data iterator to retrieve object data for PDC in-locus Analysis.

Parameters

- **iter** – [IN] PDC iterator id
- **nextBlock** – [OUT] Pointer to a contiguous block of memory that contains object data
- **dims** – [IN/OUT] A pointer to an array (2D) of rows/columns

Returns

Total number of elements contained by the datablock.

perr_t **PDC_analysis_end**()

Returns

Non-negative on success/Negative on failure

perr_t **PDC_iterator_end**()

Returns

Non-negative on success/Negative on failure

Variables

static char *default_pdc_analysis_lib = "libpdcanalysis.so"

Functions

static inline int compare_gt(int *a, int b)

perr_t PDC_Server_instantiate_data_iterator (obj_data_iterator_in_t *in ATTRIBUTE(unused), obj_data_iterator_out_t *out ATTRIBUTE(unused))

void * PDC_Server_get_ftn_reference (char *ftn ATTRIBUTE(unused))

int PDC_get_analysis_registry (struct _pdc_region_analysis_ftn_info ***registry ATTRIBUTE(unused))

static int pdc_analysis_registry_init_(size_t newSize)

static int pdc_transform_registry_init_(size_t newSize)

int pdc_analysis_registry_finalize_()

int check_analysis (pdc_obj_transform_t op_type ATTRIBUTE(unused), struct pdc_region_info *dest_region)

int PDC_add_analysis_ptr_to_registry_(struct _pdc_region_analysis_ftn_info *ftn_infoPtr)

Parameters
ftn_infoPtr – [IN] *********

Returns
*****

int PDCiter_get_nextId(void)

Returns
****

int PDC_check_transform(pdc_obj_transform_t op_type, struct pdc_region_info *dest_region)

Parameters

- op_type – [IN] The type of transformation
- dest_region – [IN] The struct pointing to destination region

Returns
****

int PDC_get_transforms(struct _pdc_region_transform_ftn_info ***registry)

Parameters
registry – [IN] *******

Returns
******

int **PDC_add_transform_ptr_to_registry_**(struct _pdc_region_transform_ftn_info *ftn_infoPtr)

Parameters
op_type – [IN] **********

Returns
****

int **PDC_update_transform_server_meta_index**(int client_index, int meta_index)

Parameters

- **client_index** – [IN] The index of the client
- **meta_index** – [IN] **********

Returns

void **PDC_set_execution_locus**(_pdc_loci_t locus_identifier)

Parameters
locus_identifier – [IN] **********

_pdc_loci_t **PDC_get_execution_locus**()

Returns

int **PDC_get_ftnPtr_**(const char *ftn, const char *loadpath, void **ftnPtr)

Parameters

- **ftn** – [IN] *******
- **loadpath** – [IN] *******
- **ftnPtr** – [IN] *******

Returns
*****

HG_TEST_RPC_CB(analysis_ftn, handle)

HG_TEST_RPC_CB(obj_data_iterator, handle)

HG_TEST_THREAD_CB(obj_data_iterator)

hg_id_t **PDC_obj_data_iterator_register**(hg_class_t *hg_class)

void **PDC_free_analysis_registry**()

void **PDC_free_transform_registry**()

void **PDC_free_iterator_cache**()

Variables

size_t **analysis_registry_size** = 0

size_t **transform_registry_size** = 0

hg_atomic_int32_t **registered_transform_ftn_count_g**

int ***i_cache_freed** = NULL

size_t **iterator_cache_entries** = CACHE_SIZE

hg_atomic_int32_t **i_cache_index**

hg_atomic_int32_t **i_free_index**

_pdc_loci_t **execution_locus** = UNKNOWN

hg_thread_pool_t ***hg_test_thread_pool_g**

hg_thread_pool_t ***hg_test_thread_pool_fs_g**

struct _pdc_region_analysis_ftn_info ****pdc_region_analysis_registry** = NULL

struct _pdc_region_transform_ftn_info ****pdc_region_transform_registry** = NULL

Functions

static hg_return_t **client_register_iterator_rpc_cb**(const struct hg_cb_info *info)

static hg_return_t **client_register_analysis_rpc_cb**(const struct hg_cb_info *info)

static hg_return_t **client_register_transform_rpc_cb**(const struct hg_cb_info *info)

perr_t **PDC_free_obj_info**(struct _pdc_obj_info *obj)

perr_t **PDC_Client_send_iter_recv_id**(pdcid_t iter_id, pdcid_t *meta_id)

Registers an iterator.

Parameters

- **iter_id** – [IN] The ID of iteration
- **meta_id** – [OUT] Metadata ID

Returns

Non-negative on success/Negative on failure

perr_t **PDC_Client_register_obj_analysis**(struct _pdc_region_analysis_ftn_info *thisFtn, const char *func, const char *loadpath, pdcid_t in_local, pdcid_t out_local, pdcid_t in_meta, pdcid_t out_meta)

Parameters

- **iterthisFtn_id** – [IN] *****
- **func** – [IN] *****
- **loadpath** – [IN] *****
- **in_local** – [IN] *****
- **out_local** – [OUT] *****
- **in_meta** – [IN] *****
- **out_meta** – [OUT] *****

Returns

Non-negative on success/Negative on failure

perr_t PDC_Client_register_region_transform (const char *func, const char *loadpath, pdcid_t src_region_id ATTRIBUTE(unused), pdcid_t dest_region_id, pdcid_t obj_id, int start_state, int next_state, int op_type, int when, int client_index)

Parameters

- **func** – [IN] Name of the function
- **loadpath** – [IN] Name of the path
- **src_region_id** – [IN] ID of source region
- **dest_region_id** – [IN] ID of destination region
- **obj_id** – [IN] ID of the object
- **start_state** – [IN] Start start
- **next_state** – [IN] Next state
- **op_type** – [IN] Operation type
- **when** – [IN] When to start transformation
- **client_index** – [IN] Client index

Returns

Non-negative on success/Negative on failure

Variables

static hg_context_t ***send_context_g** = NULL

static int **work_todo_g** = 0

hg_id_t **analysis_ftn_register_id_g**

hg_id_t **transform_ftn_register_id_g**

hg_id_t **server_transform_ftn_register_id_g**

hg_id_t **object_data_iterator_register_id_g**

Functions

perr_t **PDC_sample_min_max**(pdc_var_type_t dtype, uint64_t n, void *data, double sample_pct, double *min, double *max)

Parameters

- **dtype** – [IN] Data type
- **n** – [IN] *******
- **data** – [IN] *******
- **sample_pct** – [IN] *******
- **min** – [IN] *******
- **max** – [IN] *******

Returns
Non-negative on success/Negative on failure

double **ceil_power_of_2**(double number)

double **floor_power_of_2**(double number)

pdc_histogram_t ***PDC_create_hist**(pdc_var_type_t dtype, int nbin, double min, double max)
Insert the metdata received from client to the hash table.

Parameters

- **dtype** – [IN] Data type
- **nbin** – [IN] ********
- **min** – [IN] *******
- **max** – [IN] *******

Returns
****

perr_t **PDC_hist_incr_all**(pdc_histogram_t *hist, pdc_var_type_t dtype, uint64_t n, void *data)
Insert the metdata received from client to the hash table.

Parameters

- **hist** – [IN] ********
- **dtype** – [IN] Data type
- **n** – [IN] ********
- **data** – [IN] ********

Returns
Non-negative on success/Negative on failure

pdc_histogram_t *PDC_gen_hist(pdc_var_type_t dtype, uint64_t n, void *data)

Parameters

- **obj_name** – [IN] Data type
- **n** – [IN] *******
- **data** – [IN] *******

Returns
****

void PDC_free_hist(pdc_histogram_t *hist)

Parameters
hist – [IN] ********

void PDC_print_hist(pdc_histogram_t *hist)

Parameters
hist – [IN] ********

pdc_histogram_t *PDC_dup_hist(pdc_histogram_t *hist)

Parameters
hist – [IN] ********

Returns
****

pdc_histogram_t *PDC_merge_hist(int n, pdc_histogram_t **hists)

Parameters

- **n** – [IN] ********
- **hist** – [IN] ********

Returns
****

void PDC_copy_hist(pdc_histogram_t *to, pdc_histogram_t *from)

Parameters

- **to** – [IN] ********
- **hist** – [IN] ********

CHAPTER

FOURTEEN

TRANSFORMATION

Functions

perr_t **PDCobj_transform_register**(char *func, pdcid_t obj_id, int current_state, int next_state, pdc_obj_transform_t op_type, pdc_data_movement_t when)

Register a function to be invoked at a specified point during execution to transform the supplied data.

Parameters

- **func** – [IN] String containing the [libraryname:]function to be registered. (default library name = "libpdctransforms")
- **obj_id** – [IN] PDC object id containing the input data.
- **current_state** – [IN] State/Sequence ID to identify when the transform can take place.
- **next_state** – [IN] State/Sequence ID after the transform is complete (should be +1 or -1).
- **op_type** – [IN] An enumerated ID specifying an operation type that invokes the transform.
- **when** – [IN] An enumerated ID specifying when/where a transform is invoked. (examples for data movement: DATA_OUT, DATA_IN)

Returns

Non-negative on success/Negative on failure

perr_t **PDCbuf_map_transform_register**(char *func, void *buf, pdcid_t src_region_id, pdcid_t dest_object_id, pdcid_t dest_region_id, int current_state, int next_state, pdc_data_movement_t when)

Register a function to be invoked as a result of having mapped two regions.

The specfied transform function is invoked as a result of data movement between src and dest.

Parameters

- **func** – [IN] String containing the [libraryname:]function to be registered. (default library name = "libpdctransforms")
- **src_region_id** – [IN] PDC region id of the data mapping source.
- **dest_region_id** – [IN] PDC region id of the data mapping destination (target).
- **current_state** – [IN] State/Sequence ID to identify when the transform can take place.
- **next_state** – [IN] State/Sequence ID after the transform is complete (should be +1 or -1).
- **when** – [IN] An enumerated ID specifying when/where a transform is invoked. (examples for data movement: DATA_OUT, DATA_IN)

Returns
Non-negative on success/Negative on failure

perr_t **PDCbuf_io_transform_register** (char *func ATTRIBUTE(unused), void *buf ATTRIBUTE(unused), pdcid_t src_region_id ATTRIBUTE(unused), int current_state ATTRIBUTE(unused), int next_state ATTRIBUTE(unused), pdc_data_movement_t when ATTRIBUTE(unused))

perr_t **PDC_transform_end**()

To end PDC transform.

Returns
Non-negative on success/Negative on failure

Variables

int **pdc_client_mpi_rank_g**

int **pdc_client_mpi_size_g**

static char ***default_pdc_transforms_lib** = "libpdctransforms.so"

CHAPTER

FIFTEEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

R

S

T

U

W