libSplash

User Manual

Last update: December 2, 2013

TU Dresden
Center for Information Services and
High Performance Computing (ZIH)
01062 Dresden
Germany

http://www.tu-dresden.de/zih

Helmholtz Zentrum Dresden Rossendorf
Bautzner Landstrasse 400
01328 Dresden
Germany

http://www.hzdr.de

Contents

Chapter 1

Introduction

1.1 About libSplash

libSplash is a combined project of the Center for Information Services and HPC (ZIH) of the Technical University of Dresden and the Helmholtz-Zentrum Dresden-Rossendorf (HZDR). The project aims at developing a HDF5-based I/O library for HPC simulations. It is created as an easy-to-use frontend for the standard HDF5 library with support for MPI processes in a cluster environment. While the standard HDF5 library provides detailed low-level control, libSplash simplifies tasks commonly found in large-scale HPC simulations, such as iterative computations and MPI distributed processes.

1.2 About this Manual

This manual describes the general ideas and usage modes of libSplash and its most important classes. For a detailed explanation of all available classes, interfaces and methods, please refer to the Doxygen HTML documentation. (Note: This manual may not be updated regularly!)

1.3 Installation

1.3.1 Requirements

Please see the file doc/INSTALL.md for details.

1.3.2 Compiling

Please see the file doc/INSTALL.md for details.

1.3.3 Linking

Please see the file doc/INSTALL.md for details.

1.4 Usage

To use libSplash in your application, the only include file necessary is `splash.h`. libSplash consists of two parts: the basic `DataCollector` interface and the extended `DomainCollector` interface.

`DataCollector` is the basic interface for most operations, such as accessing files as well as reading and writing Datasets and Attributes. This interface is implemented in the `SerialDataCollector` and `ParallelDataCollector` classes. `DomainCollector` extends this interface with operations on *Domains* which can represent the simulation area, a logical data field or a similar logical program structure. This is helpful to allow easy post-mortem access to the stored data from an analysis or visualization tool.

You can enable printing of verbose status messages by setting the environment variable `SPLASH_VERBOSE` to the required verbosity level.

Chapter 2

SerialDataCollector

2.1 Files

libSplash stores data in HDF5 files with the extension *.h5*. The filename structure is *(common name)_(mpi position).h5*. *common name* is the name chosen by the user for the libSplash files, e.g. 'simulation_data'. *mpi position* is the three-dimensional position of the MPI process creating this file, starting at (0, 0, 0). This format is chosen even if no MPI environment is used to create the files.

Example:

- If libSplash is used from a non-parallel program using one process, only the file *simulation_data_0_0_0.h5* is created.
- If libSplash is used by a MPI parallel program with 2x2 processes, the files *simulation_data_0_0_0.h5*, *simulation_data_1_0_0.h5*, *simulation_data_0_1_0.h5* and *simulation_data_1_1_0.h5* are created.

2.1.1 File Structure

Each libSplash file uses a similar internal file structure which is composed from groups (folders), datasets, attributes and references.

- **header** This group stores general information about this file and the creation context (e.g. the number of MPI processes participating in creating all related files).
- **data** This group stores the actual (simulation) data and their annotated attributes. Indexed sub-groups are used to reflect an iterative program pattern. In the following example, every 10th iteration is stored using libSplash.
  - **0** Iteration 0
    - dataset_A
    - dataset_B
  - **10** Iteration 10
    - dataset_A
    - dataset_B
  - ...
- **common**

2.1.2 Opening Files

Before data can be stored or read, files must be opened by calling `DataCollector::open`. This method requires the common part of the filename and an object of type `FileCreationAttr`. This object defines the file access type as well as the number of participating MPI processes, the MPI position of the calling process and further information. The following file access types are available:

- `FAT_CREATE` A new file is created. Any existing file with this name if overwritten. `FileCreationAttr` is used to determine the MPI position part of the filename.
- `FAT_WRITE` An existing file is opened for reading and writing. `FileCreationAttr` is used to determine the MPI position part of the filename. If the file does not exist, it is created. Otherwise, any write access to existing datasets will overwrite them.
- `FAT_READ` An existing file is opened in read-only mode. `FileCreationAttr` is used to determine the MPI position part of the filename. If the file does not exist, an exception is thrown.
- `FAT_READ_MERGED` All existing files belonging to a single multi-process run are opened simultaneously in read-only mode. Data from all files can be read transparently as if written to a single file.

2.1.3 Closing Files

After all file operations are finished and before opening or creating a new file, already opened files must be closed by a call to `DataCollector::close`. Otherwise, file information can be inconsistent and required data may not be stored properly.

2.2 Datasets

Datasets are the general way for storing user data, e.g. simulation results or intermediate systems states. They can be one-, two- or three-dimensional and each element can be a basic type (e.g. `int`) or structured type (i.e. a `struct`). Subclasses of the abstract class `CollectionType` are used to define the *Datatype* of a Dataset when storing data (or attributes to data). The range of available types can be easily extended by defining a new subclass (see *include/basetypes* for a list of existing types).

Datasets are stored in the *data* group of libSplash files and must be related to a specific index beneath this group.

2.2.1 Writing

To write data, use any of the `DataCollector::write` methods. They require the used datatype, the number of dimensions (`ndims`, 1-3), the size of the actual data, buffers and offsets in each dimension as a `Dimensions` object, the name for the dataset and a pointer holding the data. Any existing dataset in this group with the same is name is overwritten.

Please note that the user is responsible to pass a correct `CollectionType` to any `write` call. Otherwise, user data may be interpreted incorrectly.

2.2.2 Reading

To read data, use any of the `DataCollector::read` methods. They work similar to `DataCollector::write` but do not require a `CollectionType` or `rank`, as these

information are implicitly given from the read dataset. The destination buffer for reading must be allocated by the user. However, `read` methods can be passed a `NULL` pointer to not read any data but only return the required dimensions of the destination buffer.

2.2.3 Appending

Appending data is possible only for one-dimensional datasets. It is achieved using any of the `DataCollector::append` methods. In contrast to writing a dataset, any existing data remains unchanged and new elements are appended at the end. If the dataset for appending does not exist, it is created.

2.2.4 Removing

Datasets as well as whole program iterations can be removed from a file using `DataCollector::remove`. However, it may depend on the linked HDF5 library if file size actually decreases.

2.3 Attributes

Attributes are annotations to Datasets which can be used to store meta information. Attributes can be of any `CollectionType` but must only contain a single element of that type (in contrast to Datasets, which are multi-dimensional arrays). At the moment, it is not possible to annotate Attributes at groups.

2.3.1 Writing

Attributes are written using the `DataCollector::writeAttribute` method. It must be passed the location of the annotated Dataset (id and name) and the type and name of the Attribute.

2.3.2 Reading

To read an Attribute, use the `DataCollector::readAttribute` method. It must be passed the location of the annotated Dataset (id and name) and the name of the Attribute. If no Attribute with this name and location exists, an exception is thrown.

If the file has been opened for transparent merged read using `FAT_READ_MERGED`, additionally a MPI position can be defined to specify from which subfile to read the required Attribute. If this MPI position is set to `NULL`, the Attribute is read from the file with MPI position (0, 0, 0).

2.3.3 Global Attributes

Global Attributes are not specific to a single Dataset but belong to the whole file or to each subfile, respectively. To write and read Global Attributes, use `DataCollector::writeGlobalAttribute` and `DataCollector::readGlobalAttribute` methods. Reading and writing is similar to normal Attributes, including the optional MPI position when reading Global Attributes in `FAT_READ_MERGED` mode.

2.4 References

References are links to another Dataset within one HDF5 file. It can reference the whole Dataset as well as a user-defined subset, specified by *offset*, *count* and *stride*. After a reference is created using `DataCollector::createReference`, it can be accessed like a normal Dataset.

Chapter 3

DomainCollector

`DomainCollector` extends `SerialDataCollector` with domain management features. A domain is a logical view to data in memory in or files, in contrast to the physical/memory view. `DomainCollector` allows to efficiently read subdomains (subpartitions) from multi-process HDF5 files with entries annotated with domain information.

The following concept is used: Each process (of the MPI topology) annotates its local data with local and global domain information when writing. When reading from these files, data from all files can be accessed transparently as if in one single file. This global view uses the global domain information, created from all local subdomains.

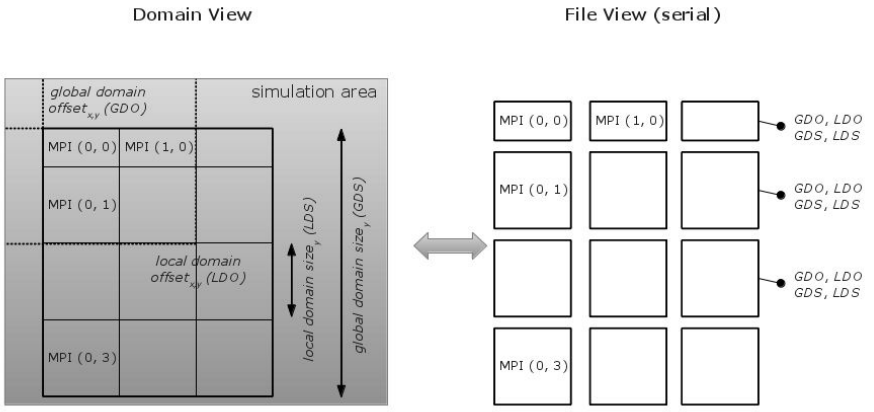

Figure 3.1: Multi-process domain- and file-view when using `DomainCollector` class.

Domain data can be of two types:

- **GridType** This type is used for data stored as 1-3-dimensional arrays, such as fields or volumes where each element has a specified position within the domain grid.
- **PolyType** This type is used for unordered 1-dimensional data, e.g. particles within a volume.

3.1 Writing Domains

To write Domain data, use any of the `DomainCollector::writeDomain` calls. Each requires information on the source data and buffer to read from, a name for the Dataset, the Datatype as well as Domain information: the stored domain offset and area and the type of data stored (GridType or PolyType).

3.2 Reading Domains

To read Domain data, use `DomainCollector::readDomain`. It must be passed the name and area of the requested Domain partition (sub-Domain). The Domain type and a `DataContainer` holding the read data are returned.

When reading Domain data, read calls to multiple HDF5 Datasets may be necessary if the requested sub-Domain spans multiple Datasets. Therefor, the returned DataContainer holds multiple `DomainData` objects. Each DomainData object stores a part of the requested data along with sub-Domain-specific information. If GridType data is read, only a single DomainData object may be returned as data from multiple Datasets can be transparently combined into one new array. Otherwise, when reading PolyType data, the DataContainer is likely to hold multiple DomainData objects, one for each physical read. These DomainData object can be queried successively or by their 1-3-dimensional index (`DataContainer::get` and `DataContainer::getIndex`). Additionally, all elements from all DomainData objects within one DataContainer can be queried continuously using `DataContainer::getNumElements` as well as `DataContainer::getElement`.

3.3 Appending Domains

Appending Domain data follows the same restrictions as appending normal Datasets (see 2.2.1).

Chapter 4

ParallelDomainCollector

`ParallelDomainCollector` extends `ParallelDataCollector` with domain management features, similarly to `DomainCollector`. However, only a single, parallel multi-process file is created which includes data from all MPI processes.

The following concept is used: Each process (of the MPI topology) annotates its local data with local and global domain information when writing. If global domain information is missing, it is derived from the local information of each process (if possible, see documentation).

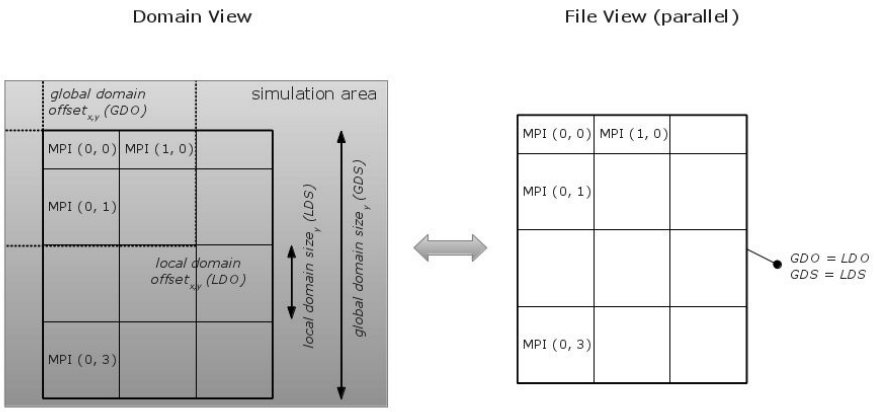

Figure 4.1: Multi-process domain- and file-view when using `ParallelDomainCollector` class.

Chapter 5

Misc

5.1 splashtools

splashtools is a convenience tool to check/modify/... HDF5 files created with libSplash. Features include:

- List all file entries.
- Transparently delete timesteps in all HDF5 files belonging to a single run.
- Check files for syntactic and semantic consistency.

Run `splashtools --help` for a complete list of all current features. **splashtool** supports both serial and parallel libSplash files.

5.2 Tests

The libSplash repository contains tests for self-testing the library. They can be found in the `tests` subdirectory. To build the tests, move to the `tests` subdirectory and execute `mkdir build; cd build; cmake ..; make`.
From the `tests` directory, all tests can be run using the `run_tests.sh` shell script. To build the tests, **cppunit** and **OpenMPI** must be installed.