Functional tests for MPI parallel I/O API extension using persistent memory

Artur Malinowski and Paweł Czarnul

February 7, 2016

Abstract

This document describes test cases for MPI parallel I/O API extension using persistent memory.

	Project	**Optimized MPI API for persistent memory**
1	Author	Paweł Czarnul
	Version	0.1
	Modification date	December 6, 2014
	Description	Template
2	Author	Artur Malinowski
	Version	0.2
	Modification date	August 9, 2015
	Description	Functional tests description.
3	Author	Artur Malinowski
	Version	0.3
	Modification date	August 15, 2015
	Description	More test cases, bugs fixed section
4	Author	Paweł Czarnul
	Version	0.3.1
	Modification date	August 16, 2015
	Description	Corrections
5	Author	Paweł Czarnul
	Version	0.4
	Modification date	August 25, 2015
	Description	Testing program parameters
6	Author	Artur Malinowski
	Version	0.5
	Modification date	February 7, 2016
	Description	Another method of exceptions handling

1 Testing program

After compilation of a project, testing routine should be available as executable at standard `PATH`. Running it without arguments triggers display of basic information about usage. The program should be executed using `mpirun` or `mpiexec` command. Required command line arguments:

- path on pmem - absolute path of a directory located on pmem device required to store testing files,
- path to directory shared by all of the nodes - testing using destributed cache mode requires single location that is available for all of the nodes (probably located on distributed file system).

2 Architecture of tests

For testing purposes no special testing framework was used, the program is written with MPI and C. Each test case is represented by a structure that consists of: a pointer to a test function, test case name, a flag that informs about the number of executed processes (single or multiple), and a flag that maps pmem io extension mode. Common data used by test routines is stored in a global context structure.

A `setjmp.h` C library allows to prepare assert functions that can be caught in the main routine. If a test is executed by a single process, assertion that failed stops the test immediately. Because of the MPI collective routines like `MPI_Barrier`, if the test case is executed by multiple processes, failed assertion does not stop the function – the function is executed until the end, then all of the error messages are gathered by the process with rank 0 and prepared to be displayed.

3 Tests

3.1 Common routines

Each test case begins with common routines:

1. test file creation,
2. opening the file with `MPI_File_open_pmem`,
3. checking if above steps finished successfully.

All of the tests are executed four times - with a test file size that equals 128B, 1MB, 10MB, and 100MB. Each function call that may return an error is checked during execution of a scenario. When the test case is finished, another common routine is executed:

1. closing the file with `MPI_File_close_pmem`,
2. checking if file operations have no impact on file size,
3. test file deletion.

3.2 Test cases

Id	01
Name	Read at, beginning of a file
Description	Read bytes from the beginning of a file. Check its correction.
Expected result	Function reads correct bytes.
Extension modes	pmem io fs aware, distributed cache
Number of processes	single

Id	02
Name	Read at, middle of a file
Description	Read bytes from the middle of a file. Check its correction.
Expected result	Function reads correct bytes.
Extension modes	pmem io fs aware, distributed cache
Number of processes	single

Id	03
Name	Read at, end of a file
Description	Read bytes from the end of a file. Check its correction.
Expected result	Function reads correct bytes.
Extension modes	pmem io fs aware, distributed cache
Number of processes	single

Id	04
Name	Write at, beginning of a file
Description	Write bytes into the beginning of a file. Check its correction.
Expected result	Function writes correct bytes.
Extension modes	pmem io fs aware, distributed cache
Number of processes	single

Id	05
Name	Write at, middle of a file
Description	Write bytes into the middle of a file. Check its correction.
Expected result	Function writes correct bytes.
Extension modes	pmem io fs aware, distributed cache
Number of processes	single

Id	06
Name	Write at, end of a file
Description	Write bytes into the end of a file. Check its correction.
Expected result	Function writes correct bytes.
Extension modes	pmem io fs aware, distributed cache
Number of processes	single

Id	07
Name	Read at, same part
Description	All of the processes read the same part of a file. Each process checks correction of the data.
Expected result	Function reads correct bytes.
Extension modes	distributed cache
Number of processes	multiple

Id	08
Name	Read at, overlapping parts
Description	Each process reads different part of a file, each part overlaps another. Each process checks correction of the data.
Expected result	Function reads correct bytes.
Extension modes	distributed cache
Number of processes	multiple

Id	09
Name	Read at, non-overlapping parts
Description	Each process reads different part of a file, parts are non-overlapping. Each process checks correction of the data.
Expected result	Function reads correct bytes.
Extension modes	distributed cache
Number of processes	multiple

Id	10
Name	Write at, same part
Description	All of the processes write into the same part of a file simultaneously. Each process checks correction of the data.
Expected result	Function writes correct bytes.
Extension modes	distributed cache
Number of processes	multiple

Id	11
Name	Write at, overlapping parts
Description	All of the processes write into the overlapping parts of a file one by one. Then, the first process checks correction of the data.
Expected result	Function writes correct bytes.
Extension modes	distributed cache
Number of processes	multiple

Id	12
Name	Write at, non-overlapping parts
Description	All of the processes write into the non-overlapping parts of a file simultaneously. Each process checks correction of the data.
Expected result	Function writes correct bytes.
Extension modes	distributed cache
Number of processes	multiple

Id	13
Name	Long sequence of read operations
Description	Multiple processes call `read_at` multiple times.
Expected result	Each function call reads correct bytes.
Extension modes	pmem io fs aware, distributed cache
Number of processes	multiple

Id	14
Name	Long sequence of write operations
Description	Multiple processes call `write_at` multiple times.
Expected result	With multiple, simultaneous `write_at` calls, final file content is unpredictable. The test is successful when it passes validations described in *Common routines* section.
Extension modes	pmem io fs aware, distributed cache
Number of processes	multiple

Id	15
Name	Long sequence of read/write operations
Description	Multiple processes call `read_at` and `write_at` multiple times.
Expected result	With multiple, simultaneous `write_at` calls, final file content is unpredictable. The test is successful when it passes validations described in *Common routines* section.
Extension modes	pmem io fs aware, distributed cache
Number of processes	multiple

Id	16
Name	File sync call
Description	Multiple processes call sequence of functions: `write_at`, `file_sync`, `read_at`, `write_at`.
Expected result	Data written into file is present on storage device after `MPI_File_sync` is called. Function `MPI_File_sync` does not close the file.
Extension modes	pmem io fs aware, distributed cache
Number of processes	multiple

4 Bugs fixed after testing

1. Closing a file caused increasing file size by 1B.
2. Function `file_close` sometimes failed because of incorrect file size value.
3. Some constants were missing.
4. Unnecessary `printf` in source code (previously used for debug purpose).