# Implementation notes for MPI parallel I/O API extension using persistent memory

Artur Malinowski and Paweł Czarnul

October 26, 2015

**Abstract**

This document provides information about implementation i.e. prerequisites, information about compilation, running the example and code organization.

## 1 Prerequisites

In order to compile and run the created extensions, the following two software packages need to be installed: an MPI implementation and the NVM Library. There are many existing MPI implementations and any one of these may be used as long as it provides the `MPI_THREAD_MULTIPLE` thread support level. This allows any thread to call MPI functions without limitations. The NVM Library can be downloaded from the NVM Library project page `http://pmem.io/nvml`. As the NVM Library actually consists of five separate libraries, not all of these have to be installed, because the code uses only features of the libpmem library and only this one is needed.

The created extensions may work on any Linux filesystem that is supported by the libpmem library, but in order to achieve the best performance, similarly to the libpmem library, it is recommended to use it on top of direct access storage (DAX). DAX can be downloaded from the project github repository at `https://github.com/01org/prd`. Installation instructions for DAX can be also found at the above Internet address.

## 2 Compilation

In order to compile the project the Makefile that can be found alongside the code can be used. Before compiling one needs to provide a path to an existing MPI C compiler wrapper `mpicc`. In order to set this path one needs to edit the Makefile and change the `CC` variable to point to a proper `mpicc` compiler wrapper.

During compilation one can also set an application logging level. It can be specified by setting (using `-D` compiler option) one of the following defines in `CFLAGS`:

| | **Project** | **Optimized MPI API for persistent memory** |
|---|---|---|
| 1 | Author | Pawel Czarnul |
| | Version | 0.1 |
| | Modification date | December 16, 2014 |
| | Description | Template |
| 2 | Author | Piotr Dorożyński |
| | Version | 0.2 |
| | Modification date | March 8, 2015 |
| | Description | Prerequisites and Compilation. |
| 3 | Author | Artur Malinowski |
| | Version | 0.2 |
| | Modification date | July 12, 2015 |
| | Description | Usage, Example and Code organization. |
| 4 | Author | Paweł Czarnul |
| | Version | 0.21 |
| | Modification date | July 14, 2015 |
| | Description | Updates. |
| 5 | Author | Artur Malinowski |
| | Version | 0.3 |
| | Modification date | October 10, 2015 |
| | Description | Changes related to failure recovery. |
| 6 | Author | Artur Malinowski |
| | Version | 0.4 |
| | Modification date | October 15, 2015 |
| | Description | Final updates. |

`_LOG_ERROR` – log only error messages,

`_LOG_INFO` – log error and information messages,

`_LOG_DEBUG` – log all error, information and debug messages.

After all of the above has been set the code may be compiled by running:

```
make
```

It will compile the library code as well as the example described in Section 4.

# 3 Usage

The extension can be used similarly to the basic MPI IO API, however, additional `MPI_Info` parameters must be set:

- MPI_PMEM_INFO_KEY_MODE

  The parameter determines extension mode. Possible values are available behind defines:

  - PMEM_IO_DISTRIBUTED_CACHE
  - PMEM_IO_AWARE_FS

- MPI_PMEM_INFO_KEY_PMEM_PATH

  Absolute path of a directory on pmem device where the cache files would be stored. The path needs to be same for each node. The user who runs the application requires read/write permission to the path. The size of free memory on the pmem device must be equal or greater than the size of a file divided by number of nodes.

## 3.1 Fail_recovery

Fail recovery ensures consistency by buffering data in a file on a pmem device before writing into cache. In case a write procedure ends with success, the recovery buffer is removed. In case of a failure, there is a possibility to reopen the cache and retry the write operation. Figure 1 presents lifecycle of the cache and the recovery buffer.

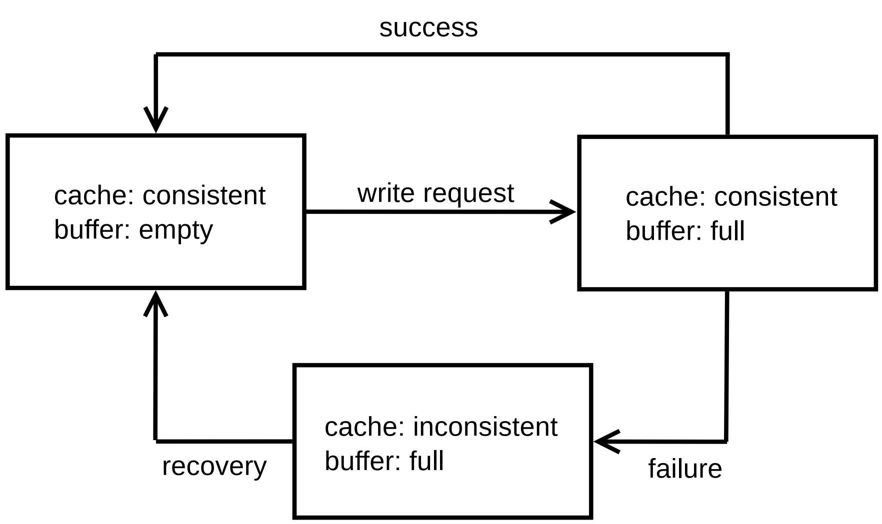


Figure 1: Diagram of possible cache and recovery buffer states

Parameters of `MPI_Info` used in fail recovery:

- MPI_PMEM_INFO_KEY_FAILURE_RECOVERY – Parameter that turns on support for failure recovery. Expected value: `true`.

- `MPI_PMEM_INFO_KEY_DO_FAILURE_RECOVERY` – Opening file with this parameter set to `true` results in reopening cache instead of creating a new one, and flushing all of the unfinished write operations.
- `MPI_PMEM_INFO_KEY_DO_FAILURE_RECOVERY_PATH` – Parameter required when `MPI_PMEM_INFO_KEY_DO_FAILURE_RECOVERY` is set to `true`. It provides a path to the directory the cache is stored in.

# 4 Examples

## 4.1 Perf_test

The source code includes a single example that can be configured using parameters described below. The example is a simple performance test with multiple command line options Running it without arguments triggers display of basic information about usage. The algorithm behind this test is as follows:

1. read from a file at random location,
2. perform some dummy computations,
3. write into a file at random location.

Command line arguments explanation:

- extension – decides whether extension is on (value 1) or off (value 2); useful to compare with unextended MPI IO,
- pmem aware – extension mode, 0 for distributed cache, 1 for pmem aware fs,
- test case – id of a test case, only 1 is supported,
- pmem path – absolute path of a directory on pmem device,
- file path – path of a file, all nodes must have access to file,
- file size – size of a file in MB,
- chunk size – size of single data chunk in bytes, that is read or written in the first and the third part of the test algorithm,
- iterations – number of algorithm iterations.

## 4.2 Fail_recovery

These two examples show how to use the fail recovery mode. The first, `Fail_safe`, is an example of turning on the support for failure recovery in case of unexpected application shutdown. The second, `Fail_recovery`, shows the process of reopening existing cache and restoring its consistency.

`Fail_safe` command line arguments:

- pmem path – absolute path of a directory on pmem device,
- file path – path to a test file (the file will be created by program), all nodes must have access to the file.

`Fail_recovery` command line arguments:

- pmem path – absolute path of a directory on pmem device,
- file path – path to a test file (the file will be created by program), all nodes must have access to the file,
- cache directory path – absolute path to the directory the cache is stored in; the directory is located on a path provided by `pmem path`; name of the directory consists of constant string `mpi_io_pmem` followed by timestamp followed by random string.

# 5 Code organization

Source code files description:

- `file_io_pmem`

  MPI IO extension conststs of two modes: distributed cache and pmem aware fs. Functions inluded in the file call correct wrappers according to the mode set while opening a file.

- `file_io_distributed_cache`

  The crucial part of the distributed cache mode – wrappers for open/close/read/write MPI IO functions. Wrappers do not operate on pmem directly, in order to provide file access they communicate with the cache manager.

- `cache_manager`

  A set of functions directly responsible for management of cache in distributed cache mode. Routines include initialization, deinitialization and `cache manager thread function` that operates on pmem resources and handles all of the supported MPI IO requests.

- `file_io_pmem_aware`

  Wrappers analogous to `file_io_distributed_cache`, but used with pmem aware fs mode. Functions operate directly on the file.

- `pmem_datatypes`

  A set of datatypes useful in the project.

- `mpi_node_rank`

  Function that allows to determine the rank of a process within a single node. It is used in order to provide a single instance of the cache manager for each node in a cluster. The algorithm is based on Markus Wittmann *MPI Node-Local Rank determination* blog entry[1].

- `failure_recovery`

  Three functions that are responsible for failure recovery in distributed cache mode. In case of a failure, the solution is able to restore the cache and finalize all of the unfinished write operations.

- `logger`, `messages`, `util`

  A couple of utils that are common for all of the project source code files.

[1] https://blogs.fau.de/wittmann/2013/02/mpi-node-local-rank-determination/