Extension of MPI parallel I/O API

Artur Malinowski and Paweł Czarnul

October 26, 2015

Abstract

This document describes modifications that are proposed to the MPI IO functions in order to provide support for persistent memory. These will be implemented as wrappers and thus will be independent from particular MPI implementations.

1 Introduction

MPI IO provides parallel I/O support for MPI [1]. Many MPI IO implementations have been proposed, the main focus is usually performance. The aim of this project is to improve I/O operations performance, but unlike typical solutions (e.g. ROMIO – highly optimized IO implementation used in MPICH, OpenMPI and many more), the idea behind the project is to use advantages of persistent memory – high capacity with acceptable speed and fast byte level operations.

The proposed MPI IO extensions can work in two modes: distributed cache and pmem aware fs. The particular mode needs to be selected when calling `MPI_File_open`. Provided `MPI_Info` entry with key `pmem_io_mode` can get one of two values:

- `PMEM_IO_AWARE_FS`,
- `PMEM_IO_DISTRIBUTED_CACHE`.

`PMEM_IO_AWARE_FS` mode is used only within a single node. The node operates on a file stored in local persistent memory, persistent memory is managed by pmem aware file system (e.g., ext4 + DAX). The maximum size of the file is limited by the size of pmem. The main advantage of this approach – compared to unmodified MPI – is introducing fast byte operations on a file.

`PMEM_IO_DISTRIBUTED_CACHE` does not require a file to be stored in persistent memory – it is especially convenient for files located in a distributed file system (e.g., NFS, orange-fs, Lustre). In this mode, all of the nodes that operate on a file share parts of persistent memory in order to create a distributed cache. Size of the file is limited to the sum of all

	Project	**Optimized MPI API for persistent memory**
1	Author	Pawel Czarnul
	Version	0.1
	Modification date	December 6, 2014
	Description	Template
2	Author	Artur Malinowski
	Version	0.2
	Modification date	February 11, 2015
	Description	Initial description
3	Author	Artur Malinowski and Paweł Czarnul
	Version	0.3
	Modification date	February 16, 2015
	Description	Updates and extensions.
4	Author	Artur Malinowski
	Version	0.4
	Modification date	June 06, 2015
	Description	New approach to distributed cache mode.
5	Author	Paweł Czarnul
	Version	0.4
	Modification date	June 15, 2015
	Description	Updates of the description.
6	Author	Artur Malinowski
	Version	0.5
	Modification date	October 25, 2015
	Description	Final updates.

pmem devices capacities in a cluster. On the other hand this approach introduces inter-node communication and an overhead related to providing cache consistency.

	`PMEM_IO_AWARE_FS`	`PMEM_IO_DISTRIBUTED_CACHE`
maximum file size	size of local pmem	sum of all pmem sizes in cluster nodes
file accessible to all nodes	no	yes
inter-node communication required	no	yes
overhead	no overhead	cache consistency related

1.1 General assumptions

According to the MPI 3.0 documentation, *a view defines the current set of data visible and accessible from an open file.* Extensions proposed in this document omit the idea of views, or – in other words – each process's view allows to access the whole file. Lack of views does not limit the MPI IO functionality.

2 PMEM_IO_AWARE_FS

2.1 MPI_File_open

The function is responsible for file creation (if needed) and mapping it into memory. Memory mapping is transparent from a developer's point of view, but required in order to provide fast byte read/write operations.

2.2 MPI_File_close

With `libpmem` library, a file could be closed immediately after mapping it into memory. Thus, `MPI_FILE_CLOSE`, instead of closing the file, synchronizes its state and calls `munmap` function.

2.3 MPI_File_set_size

The function needed to change the size of a file. In the backend, it may be required to call `munmap`, resize the file, and map it into memory again.

2.4 MPI_File_sync

Synchronizes the state of a file.

2.5 MPI_File_read_at

Returns a copy of a requested part of a file.

2.6 MPI_File_write_at

Writes data into pmem mapped memory area.

2.7 MPI_File_read_at_all

Collective version of `MPI_File_read_at` (all processes must call the function).

2.8 MPI_File_write_at_all

Collective version of `MPI_File_write_at` (all processes must call the function).

3 PMEM_IO_DISTRIBUTED_CACHE

3.1 Introduction to the second version

The solution presented in this paper is a second version improved in terms of performance for compute-intensive applications in a cluster. Architecture of the first approach was based on local cache managers controlled by a cache dispatcher. After execution of batch of tests performed on the first implementation, it turned out that a single instance of a dispatcher was a bottleneck that decreased scalability for compute-intensive codes. The main idea behind the new approach is to remove cache dispatcher entity and put more focus on scalability of the solution.

3.2 Architecture

The distributed cache mode requires a specific system architecture. All computing nodes must have access to a remote file using a distributed file system. Each node must be equipped with its own pmem storage.

A file is split into n continuous parts, where n is a number of nodes. Each part is managed by single cache manager. From the very beginning, the cache manager stores the whole part of a file in persistent memory, which enables fast read/write access. Each process knows exactly which cache manager holds the data, so no additional entity (like cache dispatcher in previous solution) is required.

3.3 Assumptions

3.3.1 Runnable entities

The Cache Managers should run independently and transparently. Three strategies were considered to achieve this goal: implementing it as a set of processes, a set of threads or using an active messages pattern. Although implementation using processes does not require any additional API than MPI, such a solution would be slower than more lightweight threads. On the other hand, threads require an additional library for thread management and synchronization which affects portability. Deployment of the third option, active messages, would require an additional library or non-trivial implementation [2], that is not related to the subject of this extension.

In order to maximize the performance and portability, POSIX Threads is the library used to create a thread environment. Thread synchronization is also implemented using POSIX standards (e.g. mutexes, conditional variables). The extension also requires that MPI provides `MPI_THREAD_MULTIPLE` support.

3.3.2 Resizing of a file

Storing the whole part of a file in a continuous block of persistent memory improves read/write performance. On the other hand, it makes difficult to resize a file. The extension is designed to work with files with a constant size (e.g maps, grids, 3D-models).

3.3.3 Network throughput and latency

Inter-node communication introduced with `PMEM_IO_DISTRIBUTED_CACHE` mode consists of data and meta-data transmission. While large data transmission requires high throughput, small data chunks and meta-data transmission should benefit from low latency network. Thus, InfiniBand is suggested.

3.4 MPI_File_open

The first task of `MPI_File_Open` in a distributed cache mode is starting a single cache manager per each node. The function is also responsible for opening the file, splitting it into parts and passing to the cache manager. The cache manager responsibilities include:

- allocating memory for cache,
- initializing the cache with complete data,
- starting listening for processes' requests.

It is also important that a file handler returned by `MPI_File_open` is not compatible with unextended MPI functions.

3.5 MPI_File_close

The logic behind this routine is in the opposite to `MPI_File_Open` – before the file is closed, the data is synchronized and written to the disk.

Each thread is responsible for freeing its resources and finishing its execution.

3.6 MPI_File_sync

The function causes synchronization between the cache and the file system. Each Cache Manager is responsible for writing its own data.

3.7 MPI_File_read_at

Reading a fragment of a file may require merging the data from several cache managers. The function calls cache managers one by one until the output buffer is filled.

3.8 MPI_File_write_at

The routine is similar to the one reading a file – it sends requests to each cache manager that holds the data. Saved data is stored in a cache until `MPI_File_sync` or `MPI_File_close` is executed.

3.9 MPI_File_read_at_all

Collective version of `MPI_File_read_at` (all processes must call the function).

3.10 MPI_File_write_at_all

Collective version of `MPI_File_write_at` (all processes must call the function).

References

[1] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0*, September 2012.

[2] Xin Zhao, Darius Buntinas, Judicael Zounmevo, James Dinan, David Goodell, Pavan Balaji, Rajeev Thakur, Ahmad Afsahi, and William Gropp. Toward Asynchronous and MPI-Interoperable Active Messages. In Balaji, P and Epema, D and Fahringer, T, editor, *PROCEEDINGS OF THE 2013 13TH IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER, CLOUD AND GRID COMPUTING (CCGRID 2013)*, pages 87–94, 2013.